Strings in Visual Basic

Visual Studio 2015

This section describes the basic concepts behind using strings in Visual Basic.

In This Section

Introduction to Strings in Visual Basic
: Lists topics that describe the basic concepts behind using strings in Visual Basic.

How to: Create Strings Using a StringBuilder in Visual Basic
: Demonstrates how to efficiently create a long string from many smaller strings.

How to: Search Within a String (Visual Basic)
: Demonstrates how to determine the index of the first occurrence of a substring.

Converting Between Strings and Other Data Types in Visual Basic
: Lists topics that describe how to convert strings into other data types.

Validating Strings in Visual Basic
: Lists topics that discuss how to validate strings.

Walkthrough: Encrypting and Decrypting Strings in Visual Basic
: Demonstrates how to encrypt and decrypt strings by using the cryptographic service provider version of the Triple Data Encryption Standard algorithm.

See Also

Visual Basic Language Features

Introduction to Strings in Visual Basic

Visual Studio 2015

This section describes the basic concepts behind using strings in Visual Basic.

In This Section

String Basics in Visual Basic
: Introduces the basic concepts behind using strings and string variables.

Types of String Manipulation Methods in Visual Basic
: Introduces several different ways to analyze and manipulate strings.

How Culture Affects Strings in Visual Basic
: Discusses how Visual Basic uses culture information to perform string conversions and comparisons.

See Also

Strings in Visual Basic

String Basics in Visual Basic

Visual Studio 2015

The **String** data type represents a series of characters (each representing in turn an instance of the **Char** data type). This topic introduces the basic concepts of strings in Visual Basic.

String Variables

An instance of a string can be assigned a literal value that represents a series of characters. For example:

VB

```
Dim MyString As String
MyString = "This is an example of the String data type"
```

A **String** variable can also accept any expression that evaluates to a string. Examples are shown below:

VB

```
Dim OneString As String
Dim TwoString As String
OneString = "one, two, three, four, five"
' Evaluates to "two".
TwoString = OneString.Substring(5, 3)
OneString = "1"
' Evaluates to "11".
TwoString = OneString & "1"
```

Any literal that is assigned to a **String** variable must be enclosed in quotation marks (""). This means that a quotation mark within a string cannot be represented by a quotation mark. For example, the following code causes a compiler error:

VB

```
Dim myString As String

' This line would cause an error.
' myString = "He said, "Look at this example!""
```

This code causes an error because the compiler terminates the string after the second quotation mark, and the remainder of the string is interpreted as code. To solve this problem, Visual Basic interprets two quotation marks in a string literal as one quotation mark in the string. The following example demonstrates the correct way to include a quotation mark in a string:

VB

```
' The value of myString is: He said, "Look at this example!"
```

```
myString = "He said, ""Look at this example!"""
```

In the preceding example, the two quotation marks preceding the word `Look` become one quotation mark in the string. The three quotation marks at the end of the line represent one quotation mark in the string and the string termination character.

String literals can contain multiple lines:

VB

```
Dim x = "hello
world"
```

The resulting string contains newline sequences that you used in your string literal (vbcr, vbcrlf, etc.). You no longer need to use the old workaround:

VB

```
Dim x = <xml><![CDATA[Hello
World]]></xml>.Value
```

Characters in Strings

A string can be thought of as a series of **Char** values, and the **String** type has built-in functions that allow you to perform many manipulations on a string that resemble the manipulations allowed by arrays. Like all array in .NET Framework, these are zero-based arrays. You may refer to a specific character in a string through the **Chars** property, which provides a way to access a character by the position in which it appears in the string. For example:

VB

```
Dim myString As String = "ABCDE"
Dim myChar As Char
' The value of myChar is "D".
myChar = myString.Chars(3)
```

In the above example, the **Chars** property of the string returns the fourth character in the string, which is `D`, and assigns it to `myChar`. You can also get the length of a particular string through the **Length** property. If you need to perform multiple array-type manipulations on a string, you can convert it to an array of **Char** instances using the **ToCharArray** function of the string. For example:

VB

```
Dim myString As String = "abcdefghijklmnop"
Dim myArray As Char() = myString.ToCharArray
```

The variable `myArray` now contains an array of **Char** values, each representing a character from `myString`.

The Immutability of Strings

A string is *immutable*, which means its value cannot be changed once it has been created. However, this does not prevent you from assigning more than one value to a string variable. Consider the following example:

VB

```
Dim myString As String = "This string is immutable"
myString = "Or is it?"
```

Here, a string variable is created, given a value, and then its value is changed.

More specifically, in the first line, an instance of type **String** is created and given the value `This string is immutable`. In the second line of the example, a new instance is created and given the value `Or is it?`, and the string variable discards its reference to the first instance and stores a reference to the new instance.

Unlike other intrinsic data types, **String** is a reference type. When a variable of reference type is passed as an argument to a function or subroutine, a reference to the memory address where the data is stored is passed instead of the actual value of the string. So in the previous example, the name of the variable remains the same, but it points to a new and different instance of the **String** class, which holds the new value.

See Also

Introduction to Strings in Visual Basic
String Data Type (Visual Basic)
Char Data Type (Visual Basic)
Basic String Operations in the .NET Framework

Types of String Manipulation Methods in Visual Basic

Visual Studio 2015

There are several different ways to analyze and manipulate your strings. Some of the methods are a part of the Visual Basic language, and others are inherent in the **String** class.

Visual Basic Language and the .NET Framework

Visual Basic methods are used as inherent functions of the language. They may be used without qualification in your code. The following example shows typical use of a Visual Basic string-manipulation command:

VB

```
Dim aString As String = "SomeString"
Dim bString As String
' Assign "meS" to bString.
bString = Mid(aString, 3, 3)
```

In this example, the **Mid** function performs a direct operation on `aString` and assigns the value to `bString`.

For a list of Visual Basic string manipulation methods, see String Manipulation Summary (Visual Basic).

Shared Methods and Instance Methods

You can also manipulate strings with the methods of the **String** class. There are two types of methods in **String**: *shared* methods and *instance* methods.

Shared Methods

A shared method is a method that stems from the **String** class itself and does not require an instance of that class to work. These methods can be qualified with the name of the class (**String**) rather than with an instance of the **String** class. For example:

VB

```
Dim aString As String = String.Copy("A literal string")
```

In the preceding example, the String.Copy method is a static method, which acts upon an expression it is given and assigns the resulting value to `bString`.

Instance Methods

Instance methods, by contrast, stem from a particular instance of **String** and must be qualified with the instance name. For example:

VB

```
Dim aString As String = "A String"
Dim bString As String
' Assign "String" to bString.
bString = aString.Substring(2, 6)
```

In this example, the String.Substring method is a method of the instance of **String** (that is, `aString`). It performs an operation on `aString` and assigns that value to `bString`.

For more information, see the documentation for the String class.

See Also

Introduction to Strings in Visual Basic

How Culture Affects Strings in Visual Basic

Visual Studio 2015

This Help page discusses how Visual Basic uses culture information to perform string conversions and comparisons.

When to Use Culture-Specific Strings

Typically, you should use culture-specific strings for all data presented to and read from users, and use culture-invariant strings for your application's internal data.

For example, if your application asks users to enter a date as a string, it should expect users to format the strings according to their culture, and the application should convert the string appropriately. If your application then presents that date in its user interface, it should present it in the user's culture.

However, if the application uploads the date to a central server, it should format the string according to one specific culture, to prevent confusion between potentially different date formats.

Culture-Sensitive Functions

All of the Visual Basic string-conversion functions (except for the **Str** and **Val** functions) use the application's culture information to make sure that the conversions and comparisons are appropriate for the culture of the application's user.

The key to successfully using string-conversion functions in applications that run on computers with different culture settings is to understand which functions use a specific culture setting, and which use the current culture setting. Notice that the application's culture settings are, by default, inherited from the culture settings of the operating system. For more information, see Asc, AscW, Chr, ChrW, Format, Hex, Oct, and Type Conversion Functions (Visual Basic).

The **Str** (converts numbers to strings) and **Val** (converts strings to numbers) functions do not use the application's culture information when converting between strings and numbers. Instead, they recognize only the period (.) as a valid decimal separator. The culturally-aware analogues of these functions are:

- **Conversions that use the current culture.** The **CStr** and **Format** functions convert a number to a string, and the **CDbl** and **CInt** functions convert a string to a number.
- **Conversions that use a specific culture.** Each number object has a `ToString(IFormatProvider)` method that converts a number to a string, and a `Parse(String, IFormatProvider)` method that converts a string to a number. For example, the **Double** type provides the ToString(IFormatProvider) and Parse(String, IFormatProvider) methods.

For more information, see Str and Val.

Using a Specific Culture

Imagine that you are developing an application that sends a date (formatted as a string) to a Web service. In this case, your application must use a specific culture for the string conversion. To illustrate why, consider the result of using the date's ToString() method: If your application uses that method to format the date July 4, 2005, it returns "7/4/2005 12:00:00 AM" when run with the United States English (en-US) culture, but it returns "04.07.2005 00:00:00" when run with the German (de-DE) culture.

When you need to perform a string conversion in a specific culture format, you should use the **CultureInfo** class that is built into the .NET Framework. You can create a new **CultureInfo** object for a specific culture by passing the culture's name to the CultureInfo constructor. The supported culture names are listed in the CultureInfo class Help page.

Alternatively, you can get an instance of the *invariant culture* from the CultureInfo.InvariantCulture property. The invariant culture is based on the English culture, but there are some differences. For example, the invariant culture specifies a 24-hour clock instead of a 12-hour clock.

To convert a date to the culture's string, pass the CultureInfo object to the date object's ToString(IFormatProvider) method. For example, the following code displays "07/04/2005 00:00:00", regardless of the application's culture settings.

VB

```
Dim d As Date = #7/4/2005#
MsgBox(d.ToString(System.Globalization.CultureInfo.InvariantCulture))
```

Note

Date literals are always interpreted according to the English culture.

Comparing Strings

There are two important situations where string comparisons are needed:

- **Sorting data for display to the user.** Use operations based on the current culture so the strings sort appropriately.
- **Determining if two application-internal strings exactly match (typically for security purposes).** Use operations that disregard the current culture.

You can perform both types of comparisons with the Visual Basic StrComp function. Specify the optional *Compare* argument to control the type of comparison: **Text** for most input and output **Binary** for determining exact matches.

The **StrComp** function returns an integer that indicates the relationship between the two compared strings based on the sorting order. A positive value for the result indicates that the first string is greater than the second string. A negative result indicates the first string is smaller, and zero indicates equality between the strings.

VB

```
' Defines variables.
Dim TestStr1 As String = "ABCD"
Dim TestStr2 As String = "abcd"
```

```
Dim TestComp As Integer
' The two strings sort equally. Returns 0.
TestComp = StrComp(TestStr1, TestStr2, CompareMethod.Text)
' TestStr1 sorts before TestStr2. Returns -1.
TestComp = StrComp(TestStr1, TestStr2, CompareMethod.Binary)
' TestStr2 sorts after TestStr1. Returns 1.
TestComp = StrComp(TestStr2, TestStr1, CompareMethod.Binary)
```

You can also use the .NET Framework partner of the **StrComp** function, the String.Compare method. This is a static, overloaded method of the base string class. The following example illustrates how this method is used:

VB

```
Dim myString As String = "Alphabetical"
Dim secondString As String = "Order"
Dim result As Integer
result = String.Compare(myString, secondString)
```

For finer control over how the comparisons are performed, you can use additional overloads of the Compare method. With the String.Compare method, you can use the *comparisonType* argument to specify which type of comparison to use.

Value for *comparisonType* argument	Type of comparison	When to use
Ordinal	Comparison based on strings' component bytes.	Use this value when comparing: case-sensitive identifiers, security-related settings, or other non-linguistic identifiers where the bytes must match exactly.
OrdinalIgnoreCase	Comparison based on strings' component bytes. **OrdinalIgnoreCase** uses the invariant culture information to determine when two characters differ only in capitalization.	Use this value when comparing: case-insensitive identifiers, security-related settings, and data stored in Windows.
CurrentCulture or **CurrentCultureIgnoreCase**	Comparison based on the strings' interpretation in the current culture.	Use these values when comparing: data that is displayed to the user, most user input, and other data that requires linguistic interpretation.
InvariantCulture or **InvariantCultureIgnoreCase**	Comparison based on the strings' interpretation in the invariant culture. This is different than the **Ordinal** and **OrdinalIgnoreCase**, because the invariant culture treats characters outside its accepted range as equivalent invariant	Use these values only when comparing persisting data or displaying linguistically-relevant data that requires a fixed sort order.

	characters.	

Security Considerations

If your application makes security decisions based on the result of a comparison or case-change operation, then the operation should use the String.Compare method, and pass **Ordinal** or **OrdinalIgnoreCase** for the *comparisonType* argument.

See Also

CultureInfo
Introduction to Strings in Visual Basic
Type Conversion Functions (Visual Basic)

Nothing and Strings in Visual Basic

Visual Studio 2015

The Visual Basic runtime and the .NET Framework evaluate **Nothing** differently when it comes to strings.

Visual Basic Runtime and the .NET Framework

Consider the following example:

VB

```
Dim MyString As String = "This is my string"
Dim stringLength As Integer
' Explicitly set the string to Nothing.
MyString = Nothing
' stringLength = 0
stringLength = Len(MyString)
' This line, however, causes an exception to be thrown.
stringLength = MyString.Length
```

The Visual Basic runtime usually evaluates **Nothing** as an empty string (""). The .NET Framework does not, however, and throws an exception whenever an attempt is made to perform a string operation on **Nothing**.

See Also

Introduction to Strings in Visual Basic

Zero-based vs. One-based String Access in Visual Basic

Visual Studio 2015

This topic compares how Visual Basic and the .NET Framework provide access to the characters in a string. The .NET Framework always provides zero-based access to the characters in a string, whereas Visual Basic provides zero-based and one-based access, depending on the function.

One-Based

For an example of a one-based Visual Basic function, consider the **Mid** function. It takes an argument that indicates the character position at which the substring will start, starting with position 1. The .NET Framework String.Substring method takes an index of the character in the string at which the substring is to start, starting with position 0. Thus, if you have a string "ABCDE", the individual characters are numbered 1,2,3,4,5 for use with the **Mid** function, but 0,1,2,3,4 for use with the String.Substring method.

Zero-Based

For an example of a zero-based Visual Basic function, consider the **Split** function. It splits a string and returns an array containing the substrings. The .NET Framework String.Split method also splits a string and returns an array containing the substrings. Because the **Split** function and Split method return .NET Framework arrays, they must be zero-based.

See Also

Mid
Split
Substring
Split
Introduction to Strings in Visual Basic

How to: Create Strings Using a StringBuilder in Visual Basic

Visual Studio 2015

This example constructs a long string from many smaller strings using the StringBuilder class. The StringBuilder class is more efficient than the **&=** operator for concatenating many strings.

Example

The following example creates an instance of the StringBuilder class, appends 1,000 strings to that instance, and then returns its string representation.

VB

```
Private Function StringBuilderTest() As String
    Dim builder As New System.Text.StringBuilder
    For i As Integer = 1 To 1000
        builder.Append("Step " & i & vbCrLf)
    Next
    Return builder.ToString
End Function
```

See Also

- Using the StringBuilder Class in the .NET Framework
- &= Operator (Visual Basic)
- Strings in Visual Basic
- Creating New Strings in the .NET Framework
- Manipulating Strings in the .NET Framework
- Strings Sample

How to: Search Within a String (Visual Basic)

Visual Studio 2015

This example calls the IndexOf method on a String object to report the index of the first occurrence of a substring.

Example

VB

```
Dim SearchWithinThis As String = "ABCDEFGHIJKLMNOP"
Dim SearchForThis As String = "DEF"
Dim FirstCharacter As Integer = SearchWithinThis.IndexOf(SearchForThis)
```

Compiling the Code

This example requires:

- An **Imports** statement specifying the System namespace. For more information, see Imports Statement (.NET Namespace and Type).

Robust Programming

The IndexOf method reports the location of the first character of the first occurrence of the substring. The index is 0-based, which means the first character of a string has an index of 0.

If IndexOf does not find the substring, it returns -1.

The IndexOf method is case-sensitive and uses the current culture.

For optimal error control, you might want to enclose the string search in the **Try** block of a Try...Catch...Finally Statement (Visual Basic) construction.

See Also

IndexOf
Try...Catch...Finally Statement (Visual Basic)
Introduction to Strings in Visual Basic
Strings in Visual Basic

Converting Between Strings and Other Data Types in Visual Basic

Visual Studio 2015

This section describes how to convert strings into other data types.

In This Section

How to: Convert an Array of Bytes into a String in Visual Basic
: How to convert the bytes from a byte array into a string.

How to: Convert Strings into an Array of Bytes in Visual Basic
: How to convert a string into an array of bytes.

How to: Create a String from An Array of Char Values (Visual Basic)
: How to create the string "abcd" from individual characters.

How to: Convert Hexadecimal Strings to Numbers (Visual Basic)
: How to convert a hexadecimal string into an integer.

How to: Convert an Array of Bytes into a String in Visual Basic

Visual Studio 2015

This topic shows how to convert the bytes from a byte array into a string.

Example

This example uses the GetString method of the Encoding.Unicode encoding class to convert all the bytes from a byte array into a string.

VB

```
Private Function UnicodeBytesToString(
    ByVal bytes() As Byte) As String

    Return System.Text.Encoding.Unicode.GetString(bytes)
End Function
```

You can choose from several encoding options to convert a byte array into a string:

- Encoding.ASCII: Gets an encoding for the ASCII (7-bit) character set.
- Encoding.BigEndianUnicode: Gets an encoding for the UTF-16 format using the big-endian byte order.
- Encoding.Default: Gets an encoding for the system's current ANSI code page.
- Encoding.Unicode: Gets an encoding for the UTF-16 format using the little-endian byte order.
- Encoding.UTF32: Gets an encoding for the UTF-32 format using the little-endian byte order.
- Encoding.UTF7: Gets an encoding for the UTF-7 format.
- Encoding.UTF8: Gets an encoding for the UTF-8 format.

See Also

System.Text.Encoding
GetString
How to: Convert Strings into an Array of Bytes in Visual Basic

How to: Convert Strings into an Array of Bytes in Visual Basic

Visual Studio 2015

This topic shows how to convert a string into an array of bytes.

Example

This example uses the GetBytes method of the Encoding.Unicode encoding class to convert a string into an array of bytes.

VB

```
Private Function UnicodeStringToBytes(
    ByVal str As String) As Byte()

    Return System.Text.Encoding.Unicode.GetBytes(str)
End Function
```

You can choose from several encoding options to convert a string into a byte array:

- Encoding.ASCII: Gets an encoding for the ASCII (7-bit) character set.
- Encoding.BigEndianUnicode: Gets an encoding for the UTF-16 format using the big-endian byte order.
- Encoding.Default: Gets an encoding for the system's current ANSI code page.
- Encoding.Unicode: Gets an encoding for the UTF-16 format using the little-endian byte order.
- Encoding.UTF32: Gets an encoding for the UTF-32 format using the little-endian byte order.
- Encoding.UTF7: Gets an encoding for the UTF-7 format.
- Encoding.UTF8: Gets an encoding for the UTF-8 format.

See Also

System.Text.Encoding
GetBytes
How to: Convert an Array of Bytes into a String in Visual Basic

How to: Create a String from An Array of Char Values (Visual Basic)

Visual Studio 2015

This example creates the string "abcd" from individual characters.

Example

VB

```
Private Sub MakeStringFromCharacters()
    Dim characters() As Char = {"a"c, "b"c, "c"c, "d"c}
    Dim alphabet As New String(characters)
End Sub
```

Compiling the Code

This method has no special requirements.

The syntax `"a"c`, where a single `c` follows a single character in quotation marks, is used to create a character literal.

Robust Programming

Null characters (equivalent to `Chr(0)`) in the string lead to unexpected results when using the string. The null character will be included with the string, but characters following the null character will not be displayed in some situations.

See Also

String
Char Data Type (Visual Basic)
Data Types in Visual Basic

How to: Convert Hexadecimal Strings to Numbers (Visual Basic)

Visual Studio 2015

This example converts a hexadecimal string to an integer using the ToInt32 method.

To convert a hexadecimal string to a number

- Use the ToInt32 method to convert the number expressed in base-16 to an integer.

  The first argument of the ToInt32 method is the string to convert. The second argument describes what base the number is expressed in; hexadecimal is base 16.

  VB

```
' Assign the value 49153 to i.
Dim i As Integer = Convert.ToInt32("c001", 16)
```

See Also

Hex
ToInt32

Conversion.Hex Method (Object)

.NET Framework (current version)

Returns a string representing the hexadecimal value of a number.

Namespace: Microsoft.VisualBasic
Assembly: Microsoft.VisualBasic (in Microsoft.VisualBasic.dll)

Syntax

VB

```
Public Shared Function Hex (
        Number As Object
) As String
```

Parameters

Number
Type: System.Object
Required. Any valid numeric expression or **String** expression.

Return Value

Type: System.String
Returns a string representing the hexadecimal value of a number.

Exceptions

Exception	Condition
ArgumentNullException	*Number* is not specified.
ArgumentException	*Number* is not a numeric type.

Remarks

If *Number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *Number* is	Hex returns
Empty	Zero (0)
Any numeric value	Up to sixteen hexadecimal characters

You can represent hexadecimal numbers directly by preceding numbers in the proper range with **&H**. For example, `&H10` represents decimal 16 in hexadecimal notation.

Examples

This example uses the **Hex** function to return the hexadecimal value of a number.

VB

```
Dim TestHex As String
' Returns 5.
TestHex = Hex(5)
' Returns A.
TestHex = Hex(10)
' Returns 1CB.
TestHex = Hex(459)
```

Version Information

.NET Framework
Available since 1.1
Silverlight
Available since 2.0

See Also

Oct
ArgumentNullException
Hex Overload
Conversion Class
Microsoft.VisualBasic Namespace
Type Conversion Functions (Visual Basic)
How to: Convert Hexadecimal Strings to Numbers (Visual Basic)

Return to top

Convert.ToInt32 Method (String, Int32)

.NET Framework (current version)

Converts the string representation of a number in a specified base to an equivalent 32-bit signed integer.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

VB

```
Public Shared Function ToInt32 (
        value As String,
        fromBase As Integer
) As Integer
```

Parameters

value
Type: System.String
A string that contains the number to convert.

fromBase
Type: System.Int32
The base of the number in *value*, which must be 2, 8, 10, or 16.

Return Value

Type: System.Int32
A 32-bit signed integer that is equivalent to the number in *value*, or 0 (zero) if *value* is **null**.

Exceptions

Exception	Condition
ArgumentException	*fromBase* is not 2, 8, 10, or 16. -or- *value*, which represents a non-base 10 signed number, is prefixed with a negative sign.

ArgumentOutOfRangeException	*value* is String.Empty.
FormatException	*value* contains a character that is not a valid digit in the base specified by *fromBase*. The exception message indicates that there are no digits to convert if the first character in *value* is invalid; otherwise, the message indicates that *value* contains invalid trailing characters.
OverflowException	*value*, which represents a non-base 10 signed number, is prefixed with a negative sign. -or- *value* represents a number that is less than Int32.MinValue or greater than Int32.MaxValue.

Remarks

If *fromBase* is 16, you can prefix the number specified by the *value* parameter with "0x" or "0X".

Because the negative sign is not supported for non-base 10 numeric representations, the ToInt32(String, Int32) method assumes that negative numbers use two's complement representation. In other words, the method always interprets the highest-order binary bit of an integer (bit 31) as its sign bit. As a result, it is possible to write code in which a non-base 10 number that is out of the range of the Int32 data type is converted to an Int32 value without the method throwing an exception. The following example increments Int32.MaxValue by one, converts the resulting number to its hexadecimal string representation, and then calls the ToInt32(String, Int32) method. Instead of throwing an exception, the method displays the message, "0x80000000 converts to -2147483648."

VB

```
' Create a hexadecimal value out of range of the Integer type.
Dim value As String = Convert.ToString(CLng(Integer.MaxValue) + 1, 16)
' Convert it back to a number.
Try
   Dim number As Integer = Convert.ToInt32(value, 16)
   Console.WriteLine("0x{0} converts to {1}.", value, number)
Catch e As OverflowException
   Console.WriteLine("Unable to convert '0x{0}' to an integer.", value)
End Try
```

When performing binary operations or numeric conversions, it is always the responsibility of the developer to verify that a method is using the appropriate numeric representation to interpret a particular value. As the following example illustrates, you can ensure that the method handles overflows appropriately by first retrieving the sign of the numeric value before converting it to its hexadecimal string representation. Throw an exception if the original value was positive but the conversion back to an integer yields a negative value.

VB

```
' Create a hexadecimal value out of range of the Integer type.
Dim sourceNumber As Long = CLng(Integer.MaxValue) + 1
```

```
Dim isNegative As Boolean = (Math.Sign(sourceNumber) = -1)
Dim value As String = Convert.ToString(sourceNumber, 16)
Dim targetNumber As Integer
Try
   targetNumber = Convert.ToInt32(value, 16)
   If Not isNegative And ((targetNumber And &H80000000) <> 0) Then
      Throw New OverflowException()
   Else
      Console.WriteLine("0x{0} converts to {1}.", value, targetNumber)
   End If
Catch e As OverflowException
   Console.WriteLine("Unable to convert '0x{0}' to an integer.", value)
End Try
' Displays the following to the console:
'    Unable to convert '0x80000000' to an integer.
```

Version Information

Universal Windows Platform
Available since 8
.NET Framework
Available since 1.1
Portable Class Library
Supported in: portable .NET platforms
Silverlight
Available since 2.0
Windows Phone Silverlight
Available since 7.0
Windows Phone
Available since 8.1

See Also

ToInt32 Overload
Convert Class
System Namespace

Return to top

How to: Convert a String to an Array of Characters in Visual Basic

Visual Studio 2015

Sometimes it is useful to have data about the characters in your string and the positions of those characters within your string, such as when you are parsing a string. This example shows how you can get an array of the characters in a string by calling the string's ToCharArray method.

Example

This example demonstrates how to split a string into a **Char** array, and how to split a string into a **String** array of its Unicode text characters. The reason for this distinction is that Unicode text characters can be composed of two or more **Char** characters (such as a surrogate pair or a combining character sequence). For more information, see TextElementEnumerator and "The Unicode Standard" at http://www.unicode.org.

VB

```
Dim testString1 As String = "ABC"
' Create an array containing "A", "B", and "C".
Dim charArray() As Char = testString1.ToCharArray
```

Example

It is more difficult to split a string into its Unicode text characters, but this is necessary if you need information about the visual representation of a string. This example uses the SubstringByTextElements method to get information about the Unicode text characters that make up a string.

VB

```
' This string is made up of a surrogate pair (high surrogate
' U+D800 and low surrogate U+DC00) and a combining character
' sequence (the letter "a" with the combining grave accent).
Dim testString2 As String = ChrW(&HD800) & ChrW(&HDC00) & "a" & ChrW(&H300)

' Create and initialize a StringInfo object for the string.
Dim si As New System.Globalization.StringInfo(testString2)

' Create and populate the array.
Dim unicodeTestArray(si.LengthInTextElements) As String
For i As Integer = 0 To si.LengthInTextElements - 1
    unicodeTestArray(i) = si.SubstringByTextElements(i, 1)
Next
```

See Also

- Chars
- System.Globalization.StringInfo
- How to: Access Characters in Strings in Visual Basic
- Converting Between Strings and Other Data Types in Visual Basic
- Strings in Visual Basic

How to: Access Characters in Strings in Visual Basic

Visual Studio 2015

This example demonstrates how to use the Chars property to access the character at the specified location in a string.

Example

Sometimes it is useful to have data about the characters in your string and the positions of those characters within your string. You can think of a string as an array of characters (**Char** instances); you can retrieve a particular character by referencing the index of that character through the Chars property.

VB

```
Dim myString As String = "ABCDE"
Dim myChar As Char
' Assign "D" to myChar.
myChar = myString.Chars(3)
```

The *index* parameter of the Chars property is zero-based.

Robust Programming

The Chars property returns the character at the specified position. However, some Unicode characters can be represented by more than one character. For more information on how to work with Unicode characters, see How to: Convert a String to an Array of Characters in Visual Basic.

The Chars property throws an IndexOutOfRangeException exception if the *index* parameter is greater than or equal to the length of the string, or if it is less than zero

See Also

- Chars
- How to: Convert a String to an Array of Characters in Visual Basic
- Converting Between Strings and Other Data Types in Visual Basic
- Strings in Visual Basic

Validating Strings in Visual Basic

Visual Studio 2015

This section discusses how to validate strings in Visual Basic.

In This Section

How to: Validate File Names and Paths in Visual Basic
: How to determine whether a string represents a file name or path.

How to: Validate Strings That Represent Dates or Times (Visual Basic)
: How to determine whether a string represents a valid date.

Using Regular Expressions with the MaskedTextBox Control in Visual Basic
: Demonstrates how to convert simple regular expressions to work with the MaskedTextBox control.

Walkthrough: Validating That Passwords Are Complex (Visual Basic)
: How to determine whether a string has the characteristics of a strong password.

See Also

Strings in Visual Basic
MaskedTextBox Control (Windows Forms)

How to: Validate File Names and Paths in Visual Basic

Visual Studio 2015

This example returns a **Boolean** value that indicates whether a string represents a file name or path. The validation checks if the name contains characters that are not allowed by the file system.

Example

VB

```
Function IsValidFileNameOrPath(ByVal name As String) As Boolean
    ' Determines if the name is Nothing.
    If name Is Nothing Then
        Return False
    End If

    ' Determines if there are bad characters in the name.
    For Each badChar As Char In System.IO.Path.GetInvalidPathChars
        If InStr(name, badChar) > 0 Then
            Return False
        End If
    Next

    ' The name passes basic validation.
    Return True
End Function
```

This example does not check if the name has incorrectly placed colons, or directories with no name, or if the length of the name exceeds the system-defined maximum length. It also does not check if the application has permission to access the file-system resource with the specified name.

See Also

GetInvalidPathChars
Validating Strings in Visual Basic

How to: Validate Strings That Represent Dates or Times (Visual Basic)

Visual Studio 2015

The following code example sets a **Boolean** value that indicates whether a string represents a valid date or time.

Example

VB

```
Dim isValidDate As Boolean = IsDate("01/01/03")
Dim isValidTime As Boolean = IsDate("9:30 PM")
```

Compiling the Code

Replace `("01/01/03")` and `"9:30 PM"` with the date and time you want to validate. You can replace the string with another hard-coded string, with a **String** variable, or with a method that returns a string, such as **InputBox**.

Robust Programming

Use this method to validate the string before trying to convert the **String** to a **DateTime** variable. By checking the date or time first, you can avoid generating an exception at run time.

See Also

IsDate
InputBox
Validating Strings in Visual Basic

Using Regular Expressions with the MaskedTextBox Control in Visual Basic

Visual Studio 2015

This example demonstrates how to convert simple regular expressions to work with the MaskedTextBox control.

Description of the Masking Language

The standard MaskedTextBox masking language is based on the one used by the **Masked Edit** control in Visual Basic 6.0 and should be familiar to users migrating from that platform.

The Mask property of the MaskedTextBox control specifies what input mask to use. The mask must be a string composed of one or more of the masking elements from the following table.

Masking element	Description	Regular expression element
0	Any single digit between 0 and 9. Entry required.	\d
9	Digit or space. Entry optional.	[\d]?
#	Digit or space. Entry optional. If this position is left blank in the mask, it will be rendered as a space. Plus (+) and minus (-) signs are allowed.	[\d+-]?
L	ASCII letter. Entry required.	[a-zA-Z]
?	ASCII letter. Entry optional.	[a-zA-Z]?
&	Character. Entry required.	[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]
C	Character. Entry optional.	[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]?
A	Alphanumeric. Entry optional.	\W
.	Culture-appropriate decimal placeholder.	Not available.
,	Culture-appropriate thousands placeholder.	Not available.
:	Culture-appropriate time separator.	Not available.

/	Culture-appropriate date separator.	Not available.
$	Culture-appropriate currency symbol.	Not available.
<	Converts all characters that follow to lowercase.	Not available.
>	Converts all characters that follow to uppercase.	Not available.
|	Undoes a previous shift up or shift down.	Not available.
\	Escapes a mask character, turning it into a literal. "\\" is the escape sequence for a backslash.	\
All other characters.	Literals. All non-mask elements will appear as themselves within MaskedTextBox.	All other characters.

The decimal (.), thousandths (,), time (:), date (/), and currency ($) symbols default to displaying those symbols as defined by the application's culture. You can force them to display symbols for another culture by using the FormatProvider property.

Regular Expressions and Masks

Although you can use regular expressions and masks to validate user input, they are not completely equivalent. Regular expressions can express more complex patterns than masks, but masks can express the same information more succinctly and in a culturally relevant format.

The following table compares four regular expressions and the equivalent mask for each.

Regular Expression	Mask	Notes
`\d{2}/\d{2}/\d{4}`	`00/00/0000`	The `/` character in the mask is a logical date separator, and it will appear to the user as the date separator appropriate to the application's current culture.
`\d{2}-[A-Z][a-z]{2}-\d{4}`	`00->L<LL-0000`	A date (day, month abbreviation, and year) in United States format in which the three-letter month abbreviation is displayed with an initial uppercase letter followed by two lowercase letters.
`(\(\d{3}\)-)?\d{3}-d{4}`	`(999)-000-0000`	United States phone number, area code optional. If the user does not wish to enter the optional characters, she can either enter spaces or place the mouse pointer directly at the position in the mask represented by the first 0.
`$\d{6}.00`	`$999,999.00`	A currency value in the range of 0 to 999999. The currency, thousandth, and decimal characters will be replaced at run-time with their culture-specific equivalents.

See Also

Mask
MaskedTextBox
Validating Strings in Visual Basic
MaskedTextBox Control (Windows Forms)

Walkthrough: Validating That Passwords Are Complex (Visual Basic)

Visual Studio 2015

This method checks for some strong-password characteristics and updates a string parameter with information about which checks the password fails.

Passwords can be used in a secure system to authorize a user. However, the passwords must be difficult for unauthorized users to guess. Attackers can use a *dictionary attack* program, which iterates through all of the words in a dictionary (or multiple dictionaries in different languages) and tests whether any of the words work as a user's password. Weak passwords such as "Yankees" or "Mustang" can be guessed quickly. Stronger passwords, such as "?You'L1N3vaFiNdMeyeP@sSWerd!", are much less likely to be guessed. A password-protected system should ensure that users choose strong passwords.

A strong password is complex (containing a mixture of uppercase, lowercase, numeric, and special characters) and is not a word. This example demonstrates how to verify complexity.

Example

Code

VB

```
''' <summary>Determines if a password is sufficiently complex.</summary>
''' <param name="pwd">Password to validate</param>
''' <param name="minLength">Minimum number of password characters.</param>
''' <param name="numUpper">Minimum number of uppercase characters.</param>
''' <param name="numLower">Minimum number of lowercase characters.</param>
''' <param name="numNumbers">Minimum number of numeric characters.</param>
''' <param name="numSpecial">Minimum number of special characters.</param>
''' <returns>True if the password is sufficiently complex.</returns>
Function ValidatePassword(ByVal pwd As String,
    Optional ByVal minLength As Integer = 8,
    Optional ByVal numUpper As Integer = 2,
    Optional ByVal numLower As Integer = 2,
    Optional ByVal numNumbers As Integer = 2,
    Optional ByVal numSpecial As Integer = 2) As Boolean

    ' Replace [A-Z] with \p{Lu}, to allow for Unicode uppercase letters.
    Dim upper As New System.Text.RegularExpressions.Regex("[A-Z]")
    Dim lower As New System.Text.RegularExpressions.Regex("[a-z]")
    Dim number As New System.Text.RegularExpressions.Regex("[0-9]")
    ' Special is "none of the above".
    Dim special As New System.Text.RegularExpressions.Regex("[^a-zA-Z0-9]")

    ' Check the length.
    If Len(pwd) < minLength Then Return False
```

```
        ' Check for minimum number of occurrences.
        If upper.Matches(pwd).Count < numUpper Then Return False
        If lower.Matches(pwd).Count < numLower Then Return False
        If number.Matches(pwd).Count < numNumbers Then Return False
        If special.Matches(pwd).Count < numSpecial Then Return False

        ' Passed all checks.
        Return True
    End Function

    Sub TestValidatePassword()
        Dim password As String = "Password"
        ' Demonstrate that "Password" is not complex.
        MsgBox(password & " is complex: " & ValidatePassword(password))

        password = "Z9f%a>2kQ"
        ' Demonstrate that "Z9f%a>2kQ" is not complex.
        MsgBox(password & " is complex: " & ValidatePassword(password))
    End Sub
```

Compiling the Code

Call this method by passing the string that contains that password.

This example requires:

- Access to the members of the System.Text.RegularExpressions namespace. Add an **Imports** statement if you are not fully qualifying member names in your code. For more information, see Imports Statement (.NET Namespace and Type).

Security

If you are moving the password across a network, you need to use a secure method for transferring data. For more information, see ASP.NET Web Application Security.

You can improve the accuracy of the `ValidatePassword` function by adding additional complexity checks:

- Compare the password and its substrings against the user's name, user identifier, and an application-defined dictionary. In addition, treat visually similar characters as equivalent when performing the comparisons. For example, treat the letters "l" and "e" as equivalent to the numerals "1" and "3".
- If there is only one uppercase character, make sure it is not the password's first character.
- Make sure that the last two characters of the password are letter characters.
- Do not allow passwords in which all the symbols are entered from the keyboard's top row.

See Also

Regex
ASP.NET Web Application Security

Walkthrough: Encrypting and Decrypting Strings in Visual Basic

Visual Studio 2015

This walkthrough shows you how to use the DESCryptoServiceProvider class to encrypt and decrypt strings using the cryptographic service provider (CSP) version of the Triple Data Encryption Standard (TripleDES) algorithm. The first step is to create a simple wrapper class that encapsulates the 3DES algorithm and stores the encrypted data as a base-64 encoded string. Then, that wrapper is used to securely store private user data in a publicly accessible text file.

You can use encryption to protect user secrets (for example, passwords) and to make credentials unreadable by unauthorized users. This can protect an authorized user's identity from being stolen, which protects the user's assets and provides non-repudiation. Encryption can also protect a user's data from being accessed by unauthorized users.

For more information, see Cryptographic Services.

Security Note

The Rijndael (now referred to as Advanced Encryption Standard [AES]) and Triple Data Encryption Standard (3DES) algorithms provide greater security than DES because they are more computationally intensive. For more information, see DES and Rijndael.

To create the encryption wrapper

1. Create the `Simple3Des` class to encapsulate the encryption and decryption methods.

   VB

   ```
   Public NotInheritable Class Simple3Des
   End Class
   ```

2. Add an import of the cryptography namespace to the start of the file that contains the `Simple3Des` class.

   VB

   ```
   Imports System.Security.Cryptography
   ```

3. In the `Simple3Des` class, add a private field to store the 3DES cryptographic service provider.

   VB

   ```
   Private TripleDes As New TripleDESCryptoServiceProvider
   ```

4. Add a private method that creates a byte array of a specified length from the hash of the specified key.

VB

```
Private Function TruncateHash(
    ByVal key As String,
    ByVal length As Integer) As Byte()

    Dim sha1 As New SHA1CryptoServiceProvider

    ' Hash the key.
    Dim keyBytes() As Byte =
        System.Text.Encoding.Unicode.GetBytes(key)
    Dim hash() As Byte = sha1.ComputeHash(keyBytes)

    ' Truncate or pad the hash.
    ReDim Preserve hash(length - 1)
    Return hash
End Function
```

5. Add a constructor to initialize the 3DES cryptographic service provider.

   The *key* parameter controls the `EncryptData` and `DecryptData` methods.

VB

```
Sub New(ByVal key As String)
    ' Initialize the crypto provider.
    TripleDes.Key = TruncateHash(key, TripleDes.KeySize \ 8)
    TripleDes.IV = TruncateHash("", TripleDes.BlockSize \ 8)
End Sub
```

6. Add a public method that encrypts a string.

VB

```
Public Function EncryptData(
    ByVal plaintext As String) As String

    ' Convert the plaintext string to a byte array.
    Dim plaintextBytes() As Byte =
        System.Text.Encoding.Unicode.GetBytes(plaintext)

    ' Create the stream.
    Dim ms As New System.IO.MemoryStream
    ' Create the encoder to write to the stream.
    Dim encStream As New CryptoStream(ms,
        TripleDes.CreateEncryptor(),
        System.Security.Cryptography.CryptoStreamMode.Write)

    ' Use the crypto stream to write the byte array to the stream.
    encStream.Write(plaintextBytes, 0, plaintextBytes.Length)
```

```
        encStream.FlushFinalBlock()

        ' Convert the encrypted stream to a printable string.
        Return Convert.ToBase64String(ms.ToArray)
    End Function
```

7. Add a public method that decrypts a string.

VB

```
    Public Function DecryptData(
        ByVal encryptedtext As String) As String

        ' Convert the encrypted text string to a byte array.
        Dim encryptedBytes() As Byte = Convert.FromBase64String(encryptedtext)

        ' Create the stream.
        Dim ms As New System.IO.MemoryStream
        ' Create the decoder to write to the stream.
        Dim decStream As New CryptoStream(ms,
            TripleDes.CreateDecryptor(),
            System.Security.Cryptography.CryptoStreamMode.Write)

        ' Use the crypto stream to write the byte array to the stream.
        decStream.Write(encryptedBytes, 0, encryptedBytes.Length)
        decStream.FlushFinalBlock()

        ' Convert the plaintext stream to a string.
        Return System.Text.Encoding.Unicode.GetString(ms.ToArray)
    End Function
```

The wrapper class can now be used to protect user assets. In this example, it is used to securely store private user data in a publicly accessible text file.

To test the encryption wrapper

1. In a separate class, add a method that uses the wrapper's `EncryptData` method to encrypt a string and write it to the user's My Documents folder.

VB

```
    Sub TestEncoding()
        Dim plainText As String = InputBox("Enter the plain text:")
        Dim password As String = InputBox("Enter the password:")

        Dim wrapper As New Simple3Des(password)
        Dim cipherText As String = wrapper.EncryptData(plainText)

        MsgBox("The cipher text is: " & cipherText)
        My.Computer.FileSystem.WriteAllText(
            My.Computer.FileSystem.SpecialDirectories.MyDocuments &
```

```
            "\cipherText.txt", cipherText, False)
    End Sub
```

2. Add a method that reads the encrypted string from the user's My Documents folder and decrypts the string with the wrapper's `DecryptData` method.

VB

```
    Sub TestDecoding()
        Dim cipherText As String = My.Computer.FileSystem.ReadAllText(
            My.Computer.FileSystem.SpecialDirectories.MyDocuments &
                "\cipherText.txt")
        Dim password As String = InputBox("Enter the password:")
        Dim wrapper As New Simple3Des(password)

        ' DecryptData throws if the wrong password is used.
        Try
            Dim plainText As String = wrapper.DecryptData(cipherText)
            MsgBox("The plain text is: " & plainText)
        Catch ex As System.Security.Cryptography.CryptographicException
            MsgBox("The data could not be decrypted with the password.")
        End Try
    End Sub
```

3. Add user interface code to call the `TestEncoding` and `TestDecoding` methods.

4. Run the application.

   When you test the application, notice that it will not decrypt the data if you provide the wrong password.

See Also

System.Security.Cryptography
DESCryptoServiceProvider
DES
TripleDES
Rijndael
Cryptographic Services

Cryptographic Services

.NET Framework (current version)

Public networks such as the Internet do not provide a means of secure communication between entities. Communication over such networks is susceptible to being read or even modified by unauthorized third parties. Cryptography helps protect data from being viewed, provides ways to detect whether data has been modified, and helps provide a secure means of communication over otherwise nonsecure channels. For example, data can be encrypted by using a cryptographic algorithm, transmitted in an encrypted state, and later decrypted by the intended party. If a third party intercepts the encrypted data, it will be difficult to decipher.

In the .NET Framework, the classes in the System.Security.Cryptography namespace manage many details of cryptography for you. Some are wrappers for the unmanaged Microsoft Cryptography API (CryptoAPI), while others are purely managed implementations. You do not need to be an expert in cryptography to use these classes. When you create a new instance of one of the encryption algorithm classes, keys are autogenerated for ease of use, and default properties are as safe and secure as possible.

This overview provides a synopsis of the encryption methods and practices supported by the .NET Framework, including the ClickOnce manifests, Suite B, and Cryptography Next Generation (CNG) support introduced in the .NET Framework 3.5.

This overview contains the following sections:

For additional information about cryptography and about Microsoft services, components, and tools that enable you to add cryptographic security to your applications, see the Win32 and COM Development, Security section of this documentation.

Cryptographic Primitives

In a typical situation where cryptography is used, two parties (Alice and Bob) communicate over a nonsecure channel. Alice and Bob want to ensure that their communication remains incomprehensible by anyone who might be listening.

Furthermore, because Alice and Bob are in remote locations, Alice must make sure that the information she receives from Bob has not been modified by anyone during transmission. In addition, she must make sure that the information really does originate from Bob and not from someone who is impersonating Bob.

Cryptography is used to achieve the following goals:

- Confidentiality: To help protect a user's identity or data from being read.
- Data integrity: To help protect data from being changed.
- Authentication: To ensure that data originates from a particular party.
- Non-repudiation: To prevent a particular party from denying that they sent a message.

To achieve these goals, you can use a combination of algorithms and practices known as cryptographic primitives to create a cryptographic scheme. The following table lists the cryptographic primitives and their uses.

Cryptographic primitive	**Use**
Secret-key encryption (symmetric cryptography)	Performs a transformation on data to keep it from being read by third parties. This type of encryption uses a single shared, secret key to encrypt and decrypt data.
Public-key encryption (asymmetric cryptography)	Performs a transformation on data to keep it from being read by third parties. This type of encryption uses a public/private key pair to encrypt and decrypt data.
Cryptographic signing	Helps verify that data originates from a specific party by creating a digital signature that is unique to that party. This process also uses hash functions.
Cryptographic hashes	Maps data from any length to a fixed-length byte sequence. Hashes are statistically unique; a different two-byte sequence will not hash to the same value.

Back to top

Secret-Key Encryption

Secret-key encryption algorithms use a single secret key to encrypt and decrypt data. You must secure the key from access by unauthorized agents, because any party that has the key can use it to decrypt your data or encrypt their own data, claiming it originated from you.

Secret-key encryption is also referred to as symmetric encryption because the same key is used for encryption and decryption. Secret-key encryption algorithms are very fast (compared with public-key algorithms) and are well suited for performing cryptographic transformations on large streams of data. Asymmetric encryption algorithms such as RSA are limited mathematically in how much data they can encrypt. Symmetric encryption algorithms do not generally have those problems.

A type of secret-key algorithm called a block cipher is used to encrypt one block of data at a time. Block ciphers such as Data Encryption Standard (DES), TripleDES, and Advanced Encryption Standard (AES) cryptographically transform an

input block of *n* bytes into an output block of encrypted bytes. If you want to encrypt or decrypt a sequence of bytes, you have to do it block by block. Because *n* is small (8 bytes for DES and TripleDES; 16 bytes [the default], 24 bytes, or 32 bytes for AES), data values that are larger than *n* have to be encrypted one block at a time. Data values that are smaller than *n* have to be expanded to *n* in order to be processed.

One simple form of block cipher is called the electronic codebook (ECB) mode. ECB mode is not considered secure, because it does not use an initialization vector to initialize the first plaintext block. For a given secret key *k*, a simple block cipher that does not use an initialization vector will encrypt the same input block of plaintext into the same output block of ciphertext. Therefore, if you have duplicate blocks in your input plaintext stream, you will have duplicate blocks in your output ciphertext stream. These duplicate output blocks alert unauthorized users to the weak encryption used the algorithms that might have been employed, and the possible modes of attack. The ECB cipher mode is therefore quite vulnerable to analysis, and ultimately, key discovery.

The block cipher classes that are provided in the base class library use a default chaining mode called cipher-block chaining (CBC), although you can change this default if you want.

CBC ciphers overcome the problems associated with ECB ciphers by using an initialization vector (IV) to encrypt the first block of plaintext. Each subsequent block of plaintext undergoes a bitwise exclusive OR (**XOR**) operation with the previous ciphertext block before it is encrypted. Each ciphertext block is therefore dependent on all previous blocks. When this system is used, common message headers that might be known to an unauthorized user cannot be used to reverse-engineer a key.

One way to compromise data that is encrypted with a CBC cipher is to perform an exhaustive search of every possible key. Depending on the size of the key that is used to perform encryption, this kind of search is very time-consuming using even the fastest computers and is therefore infeasible. Larger key sizes are more difficult to decipher. Although encryption does not make it theoretically impossible for an adversary to retrieve the encrypted data, it does raise the cost of doing this. If it takes three months to perform an exhaustive search to retrieve data that is meaningful only for a few days, the exhaustive search method is impractical.

The disadvantage of secret-key encryption is that it presumes two parties have agreed on a key and IV, and communicated their values. The IV is not considered a secret and can be transmitted in plaintext with the message. However, the key must be kept secret from unauthorized users. Because of these problems, secret-key encryption is often used together with public-key encryption to privately communicate the values of the key and IV.

Assuming that Alice and Bob are two parties who want to communicate over a nonsecure channel, they might use secret-key encryption as follows: Alice and Bob agree to use one particular algorithm (AES, for example) with a particular key and IV. Alice composes a message and creates a network stream (perhaps a named pipe or network e-mail) on which to send the message. Next, she encrypts the text using the key and IV, and sends the encrypted message and IV to Bob over the intranet. Bob receives the encrypted text and decrypts it by using the IV and previously agreed upon key. If the transmission is intercepted, the interceptor cannot recover the original message, because he does not know the key. In this scenario, only the key must remain secret. In a real world scenario, either Alice or Bob generates a secret key and uses public-key (asymmetric) encryption to transfer the secret (symmetric) key to the other party. For more information about public-key encryption, see the next section.

The .NET Framework provides the following classes that implement secret-key encryption algorithms:

- AesManaged (introduced in the .NET Framework 3.5).
- DESCryptoServiceProvider.
- HMACSHA1 (This is technically a secret-key algorithm because it represents message authentication code that is calculated by using a cryptographic hash function combined with a secret key. See Hash Values, later in this topic.)

- RC2CryptoServiceProvider.
- RijndaelManaged.
- TripleDESCryptoServiceProvider.

Back to top

Public-Key Encryption

Public-key encryption uses a private key that must be kept secret from unauthorized users and a public key that can be made public to anyone. The public key and the private key are mathematically linked; data that is encrypted with the public key can be decrypted only with the private key, and data that is signed with the private key can be verified only with the public key. The public key can be made available to anyone; it is used for encrypting data to be sent to the keeper of the private key. Public-key cryptographic algorithms are also known as asymmetric algorithms because one key is required to encrypt data, and another key is required to decrypt data. A basic cryptographic rule prohibits key reuse, and both keys should be unique for each communication session. However, in practice, asymmetric keys are generally long-lived.

Two parties (Alice and Bob) might use public-key encryption as follows: First, Alice generates a public/private key pair. If Bob wants to send Alice an encrypted message, he asks her for her public key. Alice sends Bob her public key over a nonsecure network, and Bob uses this key to encrypt a message. Bob sends the encrypted message to Alice, and she decrypts it by using her private key. If Bob received Alice's key over a nonsecure channel, such as a public network, Bob is open to a man-in-the-middle attack. Therefore, Bob must verify with Alice that he has a correct copy of her public key.

During the transmission of Alice's public key, an unauthorized agent might intercept the key. Furthermore, the same agent might intercept the encrypted message from Bob. However, the agent cannot decrypt the message with the public key. The message can be decrypted only with Alice's private key, which has not been transmitted. Alice does not use her private key to encrypt a reply message to Bob, because anyone with the public key could decrypt the message. If Alice wants to send a message back to Bob, she asks Bob for his public key and encrypts her message using that public key. Bob then decrypts the message using his associated private key.

In this scenario, Alice and Bob use public-key (asymmetric) encryption to transfer a secret (symmetric) key and use secret-key encryption for the remainder of their session.

The following list offers comparisons between public-key and secret-key cryptographic algorithms:

- Public-key cryptographic algorithms use a fixed buffer size, whereas secret-key cryptographic algorithms use a variable-length buffer.
- Public-key algorithms cannot be used to chain data together into streams the way secret-key algorithms can, because only small amounts of data can be encrypted. Therefore, asymmetric operations do not use the same streaming model as symmetric operations.
- Public-key encryption has a much larger keyspace (range of possible values for the key) than secret-key encryption. Therefore, public-key encryption is less susceptible to exhaustive attacks that try every possible key.
- Public keys are easy to distribute because they do not have to be secured, provided that some way exists to verify the identity of the sender.
- Some public-key algorithms (such as RSA and DSA, but not Diffie-Hellman) can be used to create digital signatures

to verify the identity of the sender of data.

- Public-key algorithms are very slow compared with secret-key algorithms, and are not designed to encrypt large amounts of data. Public-key algorithms are useful only for transferring very small amounts of data. Typically, public-key encryption is used to encrypt a key and IV to be used by a secret-key algorithm. After the key and IV are transferred, secret-key encryption is used for the remainder of the session.

The .NET Framework provides the following classes that implement public-key encryption algorithms:

- DSACryptoServiceProvider
- RSACryptoServiceProvider
- ECDiffieHellman (base class)
- ECDiffieHellmanCng
- ECDiffieHellmanCngPublicKey (base class)
- ECDiffieHellmanKeyDerivationFunction (base class)
- ECDsaCng

RSA allows both encryption and signing, but DSA can be used only for signing, and Diffie-Hellman can be used only for key generation. In general, public-key algorithms are more limited in their uses than private-key algorithms.

Back to top

Digital Signatures

Public-key algorithms can also be used to form digital signatures. Digital signatures authenticate the identity of a sender (if you trust the sender's public key) and help protect the integrity of data. Using a public key generated by Alice, the recipient of Alice's data can verify that Alice sent it by comparing the digital signature to Alice's data and Alice's public key.

To use public-key cryptography to digitally sign a message, Alice first applies a hash algorithm to the message to create a message digest. The message digest is a compact and unique representation of data. Alice then encrypts the message digest with her private key to create her personal signature. Upon receiving the message and signature, Bob decrypts the signature using Alice's public key to recover the message digest and hashes the message using the same hash algorithm that Alice used. If the message digest that Bob computes exactly matches the message digest received from Alice, Bob is assured that the message came from the possessor of the private key and that the data has not been modified. If Bob trusts that Alice is the possessor of the private key, he knows that the message came from Alice.

Note
A signature can be verified by anyone because the sender's public key is common knowledge and is typically included in the digital signature format. This method does not retain the secrecy of the message; for the message to be secret, it must also be encrypted.

The .NET Framework provides the following classes that implement digital signature algorithms:

- DSACryptoServiceProvider
- RSACryptoServiceProvider
- ECDsa (base class)
- ECDsaCng

Back to top

Hash Values

Hash algorithms map binary values of an arbitrary length to smaller binary values of a fixed length, known as hash values. A hash value is a numerical representation of a piece of data. If you hash a paragraph of plaintext and change even one letter of the paragraph, a subsequent hash will produce a different value. If the hash is cryptographically strong, its value will change significantly. For example, if a single bit of a message is changed, a strong hash function may produce an output that differs by 50 percent. Many input values may hash to the same output value. However, it is computationally infeasible to find two distinct inputs that hash to the same value.

Two parties (Alice and Bob) could use a hash function to ensure message integrity. They would select a hash algorithm to sign their messages. Alice would write a message, and then create a hash of that message by using the selected algorithm. They would then follow one of the following methods:

- Alice sends the plaintext message and the hashed message (digital signature) to Bob. Bob receives and hashes the message and compares his hash value to the hash value that he received from Alice. If the hash values are identical, the message was not altered. If the values are not identical, the message was altered after Alice wrote it.

  Unfortunately, this method does not establish the authenticity of the sender. Anyone can impersonate Alice and send a message to Bob. They can use the same hash algorithm to sign their message, and all Bob can determine is that the message matches its signature. This is one form of a man-in-the-middle attack. See NIB: Cryptography Next Generation (CNG) Secure Communication Example for more information.

- Alice sends the plaintext message to Bob over a nonsecure public channel. She sends the hashed message to Bob over a secure private channel. Bob receives the plaintext message, hashes it, and compares the hash to the privately exchanged hash. If the hashes match, Bob knows two things:

  - The message was not altered.
  - The sender of the message (Alice) is authentic.

  For this system to work, Alice must hide her original hash value from all parties except Bob.

- Alice sends the plaintext message to Bob over a nonsecure public channel and places the hashed message on her publicly viewable Web site.

  This method prevents message tampering by preventing anyone from modifying the hash value. Although the message and its hash can be read by anyone, the hash value can be changed only by Alice. An attacker who wants to impersonate Alice would require access to Alice's Web site.

None of the previous methods will prevent someone from reading Alice's messages, because they are transmitted in plaintext. Full security typically requires digital signatures (message signing) and encryption.

The .NET Framework provides the following classes that implement hashing algorithms:

- HMACSHA1.
- MACTripleDES.
- MD5CryptoServiceProvider.
- RIPEMD160.
- SHA1Managed.
- SHA256Managed.
- SHA384Managed.
- SHA512Managed.
- HMAC variants of all of the Secure Hash Algorithm (SHA), Message Digest 5 (MD5), and RIPEMD-160 algorithms.
- CryptoServiceProvider implementations (managed code wrappers) of all the SHA algorithms.
- Cryptography Next Generation (CNG) implementations of all the MD5 and SHA algorithms.

Note

MD5 design flaws were discovered in 1996, and SHA-1 was recommended instead. In 2004, additional flaws were discovered, and the MD5 algorithm is no longer considered secure. The SHA-1 algorithm has also been found to be insecure, and SHA-2 is now recommended instead.

Back to top

Random Number Generation

Random number generation is integral to many cryptographic operations. For example, cryptographic keys need to be as random as possible so that it is infeasible to reproduce them. Cryptographic random number generators must generate output that is computationally infeasible to predict with a probability that is better than one half. Therefore, any method of predicting the next output bit must not perform better than random guessing. The classes in the .NET Framework use random number generators to generate cryptographic keys.

The RNGCryptoServiceProvider class is an implementation of a random number generator algorithm.

Back to top

ClickOnce Manifests

In the .NET Framework 3.5, the following cryptography classes let you obtain and verify information about manifest signatures for applications that are deployed using ClickOnce technology:

- The ManifestSignatureInformation class obtains information about a manifest signature when you use its VerifySignature method overloads.
- You can use the ManifestKinds enumeration to specify which manifests to verify. The result of the verification is one of the SignatureVerificationResult enumeration values.
- The ManifestSignatureInformationCollection class provides a read-only collection of ManifestSignatureInformation objects of the verified signatures.

In addition, the following classes provide specific signature information:

- StrongNameSignatureInformation holds the strong name signature information for a manifest.
- AuthenticodeSignatureInformation represents the Authenticode signature information for a manifest.
- TimestampInformation contains information about the time stamp on an Authenticode signature.
- TrustStatus provides a simple way to check whether an Authenticode signature is trusted.

Back to top

Suite B Support

The .NET Framework 3.5 supports the Suite B set of cryptographic algorithms published by the National Security Agency (NSA). For more information about Suite B, see the NSA Suite B Cryptography Fact Sheet.

The following algorithms are included:

- Advanced Encryption Standard (AES) algorithm with key sizes of 128, 192, , and 256 bits for encryption.
- Secure Hash Algorithms SHA-1, SHA-256, SHA-384, and SHA-512 for hashing. (The last three are generally grouped together and referred to as SHA-2.)
- Elliptic Curve Digital Signature Algorithm (ECDSA), using curves of 256-bit, 384-bit, and 521-bit prime moduli for signing. The NSA documentation specifically defines these curves, and calls them P-256, P-384, and P-521. This algorithm is provided by the ECDsaCng class. It enables you to sign with a private key and verify the signature with a public key.
- Elliptic Curve Diffie-Hellman (ECDH) algorithm, using curves of 256-bit, 384-bit, and 521-bit prime moduli for the key exchange and secret agreement. This algorithm is provided by the ECDiffieHellmanCng class.

Managed code wrappers for the Federal Information Processing Standard (FIPS) certified implementations of the AES,

SHA-256, SHA-384, and SHA-512 implementations are available in the new AesCryptoServiceProvider, SHA256CryptoServiceProvider, SHA384CryptoServiceProvider, and SHA512CryptoServiceProvider classes.

Back to top

Cryptography Next Generation (CNG) Classes

The Cryptography Next Generation (CNG) classes provide a managed wrapper around the native CNG functions. (CNG is the replacement for CryptoAPI.) These classes have "Cng" as part of their names. Central to the CNG wrapper classes is the CngKey key container class, which abstracts the storage and use of CNG keys. This class lets you store a key pair or a public key securely and refer to it by using a simple string name. The elliptic curve-based ECDsaCng signature class and the ECDiffieHellmanCng encryption class can use CngKey objects.

The CngKey class is used for a variety of additional operations, including opening, creating, deleting, and exporting keys. It also provides access to the underlying key handle to use when calling native functions directly.

The .NET Framework 3.5 also includes a variety of supporting CNG classes, such as the following:

- CngProvider maintains a key storage provider.
- CngAlgorithm maintains a CNG algorithm.
- CngProperty maintains frequently used key properties.

Back to top

Related Topics

Title	Description
.NET Framework Cryptography Model	Describes how cryptography is implemented in the base class library.
Walkthrough: Creating a Cryptographic Application	Demonstrates basic encryption and decryption tasks.
Configuring Cryptography Classes	Describes how to map algorithm names to cryptographic classes and map object identifiers to a cryptographic algorithm.