How to use character encoding classes in .NET

Article • 09/15/2021 • 43 minutes to read

This article explains how to use the classes that .NET provides for encoding and decoding text by using various encoding schemes. The instructions assume you have read Introduction to character encoding in .NET.

Encoders and decoders

.NET provides encoding classes that encode and decode text by using various encoding systems. For example, the UTF8Encoding class describes the rules for encoding to, and decoding from, UTF-8. .NET uses UTF-16 encoding (represented by the UnicodeEncoding class) for `string` instances. Encoders and decoders are available for other encoding schemes.

Encoding and decoding can also include validation. For example, the UnicodeEncoding class checks all `char` instances in the surrogate range to make sure they're in valid surrogate pairs. A fallback strategy determines how an encoder handles invalid characters or how a decoder handles invalid bytes.

> ⚠ **Warning**
>
> .NET encoding classes provide a way to store and convert character data. They should not be used to store binary data in string form. Depending on the encoding used, converting binary data to string format with the encoding classes can introduce unexpected behavior and produce inaccurate or corrupted data. To convert binary data to a string form, use the **Convert.ToBase64String** method.

All character encoding classes in .NET inherit from the System.Text.Encoding class, which is an abstract class that defines the functionality common to all character encodings. To access the individual encoding objects implemented in .NET, do the following:

- Use the static properties of the Encoding class, which return objects that represent the standard character encodings available in .NET (ASCII, UTF-7, UTF-8, UTF-16,

and UTF-32). For example, the Encoding.Unicode property returns a
UnicodeEncoding object. Each object uses replacement fallback to handle strings
that it cannot encode and bytes that it cannot decode. For more information, see
Replacement fallback.

- Call the encoding's class constructor. Objects for the ASCII, UTF-7, UTF-8, UTF-16,
  and UTF-32 encodings can be instantiated in this way. By default, each object uses
  replacement fallback to handle strings that it cannot encode and bytes that it
  cannot decode, but you can specify that an exception should be thrown instead.
  For more information, see Replacement fallback and Exception fallback.

- Call the Encoding(Int32) constructor and pass it an integer that represents the
  encoding. Standard encoding objects use replacement fallback, and code page and
  double-byte character set (DBCS) encoding objects use best-fit fallback to handle
  strings that they cannot encode and bytes that they cannot decode. For more
  information, see Best-fit fallback.

- Call the Encoding.GetEncoding method, which returns any standard, code page, or
  DBCS encoding available in .NET. Overloads let you specify a fallback object for
  both the encoder and the decoder.

You can retrieve information about all the encodings available in .NET by calling the
Encoding.GetEncodings method. .NET supports the character encoding schemes listed in
the following table.

Encoding class	Description
ASCII	Encodes a limited range of characters by using the lower seven bits of a byte. Because this encoding only supports character values from `U+0000` through `U+007F`, in most cases it is inadequate for internationalized applications.
UTF-7	Represents characters as sequences of 7-bit ASCII characters. Non-ASCII Unicode characters are represented by an escape sequence of ASCII characters. UTF-7 supports protocols such as email and newsgroup. However, UTF-7 is not particularly secure or robust. In some cases, changing one bit can radically alter the interpretation of an entire UTF-7 string. In other cases, different UTF-7 strings can encode the same text. For sequences that include non-ASCII characters, UTF-7 requires more space than UTF-8, and encoding/decoding is slower. Consequently, you should use UTF-8 instead of UTF-7 if possible.

Encoding class	Description
UTF-8	Represents each Unicode code point as a sequence of one to four bytes. UTF-8 supports 8-bit data sizes and works well with many existing operating systems. For the ASCII range of characters, UTF-8 is identical to ASCII encoding and allows a broader set of characters. However, for Chinese-Japanese-Korean (CJK) scripts, UTF-8 can require three bytes for each character, and can cause larger data sizes than UTF-16. Sometimes the amount of ASCII data, such as HTML tags, justifies the increased size for the CJK range.
UTF-16	Represents each Unicode code point as a sequence of one or two 16-bit integers. Most common Unicode characters require only one UTF-16 code point, although Unicode supplementary characters (U+10000 and greater) require two UTF-16 surrogate code points. Both little-endian and big-endian byte orders are supported. UTF-16 encoding is used by the common language runtime to represent Char and String values, and it is used by the Windows operating system to represent `WCHAR` values.
UTF-32	Represents each Unicode code point as a 32-bit integer. Both little-endian and big-endian byte orders are supported. UTF-32 encoding is used when applications want to avoid the surrogate code point behavior of UTF-16 encoding on operating systems for which encoded space is too important. Single glyphs rendered on a display can still be encoded with more than one UTF-32 character.
ANSI/ISO encoding	Provides support for a variety of code pages. On Windows operating systems, code pages are used to support a specific language or group of languages. For a table that lists the code pages supported by .NET, see the Encoding class. You can retrieve an encoding object for a particular code page by calling the Encoding.GetEncoding(Int32) method. A code page contains 256 code points and is zero-based. In most code pages, code points 0 through 127 represent the ASCII character set, and code points 128 through 255 differ significantly between code pages. For example, code page 1252 provides the characters for Latin writing systems, including English, German, and French. The last 128 code points in code page 1252 contain the accent characters. Code page 1253 provides character codes that are required in the Greek writing system. The last 128 code points in code page 1253 contain the Greek characters. As a result, an application that relies on ANSI code pages cannot store Greek and German in the same text stream unless it includes an identifier that indicates the referenced code page.
Double-byte character set (DBCS) encodings	Supports languages, such as Chinese, Japanese, and Korean, that contain more than 256 characters. In a DBCS, a pair of code points (a double byte) represents each character. The Encoding.IsSingleByte property returns `false` for DBCS encodings. You can retrieve an encoding object for a particular DBCS by calling the Encoding.GetEncoding(Int32) method. When an application handles DBCS data, the first byte of a DBCS character (the lead byte) is processed in combination with the

Encoding class	Description
	trail byte that immediately follows it. Because a single pair of double-byte code points can represent different characters depending on the code page, this scheme still does not allow for the combination of two languages, such as Japanese and Chinese, in the same data stream.

These encodings enable you to work with Unicode characters as well as with encodings that are most commonly used in legacy applications. In addition, you can create a custom encoding by defining a class that derives from Encoding and overriding its members.

.NET Core encoding support

By default, .NET Core does not make available any code page encodings other than code page 28591 and the Unicode encodings, such as UTF-8 and UTF-16. However, you can add the code page encodings found in standard Windows apps that target .NET to your app. For more information, see the CodePagesEncodingProvider topic.

Selecting an Encoding Class

If you have the opportunity to choose the encoding to be used by your application, you should use a Unicode encoding, preferably either UTF8Encoding or UnicodeEncoding. (.NET also supports a third Unicode encoding, UTF32Encoding.)

If you are planning to use an ASCII encoding (ASCIIEncoding), choose UTF8Encoding instead. The two encodings are identical for the ASCII character set, but UTF8Encoding has the following advantages:

- It can represent every Unicode character, whereas ASCIIEncoding supports only the Unicode character values between U+0000 and U+007F.

- It provides error detection and better security.

- It has been tuned to be as fast as possible and should be faster than any other encoding. Even for content that is entirely ASCII, operations performed with UTF8Encoding are faster than operations performed with ASCIIEncoding.

You should consider using ASCIIEncoding only for legacy applications. However, even

for legacy applications, UTF8Encoding might be a better choice for the following reasons (assuming default settings):

- If your application has content that is not strictly ASCII and encodes it with ASCIIEncoding, each non-ASCII character encodes as a question mark (?). If the application then decodes this data, the information is lost.

- If your application has content that is not strictly ASCII and encodes it with UTF8Encoding, the result seems unintelligible if interpreted as ASCII. However, if the application then uses a UTF-8 decoder to decode this data, the data performs a round trip successfully.

In a web application, characters sent to the client in response to a web request should reflect the encoding used on the client. In most cases, you should set the HttpResponse.ContentEncoding property to the value returned by the HttpRequest.ContentEncoding property to display text in the encoding that the user expects.

Using an Encoding Object

An encoder converts a string of characters (most commonly, Unicode characters) to its numeric (byte) equivalent. For example, you might use an ASCII encoder to convert Unicode characters to ASCII so that they can be displayed at the console. To perform the conversion, you call the Encoding.GetBytes method. If you want to determine how many bytes are needed to store the encoded characters before performing the encoding, you can call the GetByteCount method.

The following example uses a single byte array to encode strings in two separate operations. It maintains an index that indicates the starting position in the byte array for the next set of ASCII-encoded bytes. It calls the ASCIIEncoding.GetByteCount(String) method to ensure that the byte array is large enough to accommodate the encoded string. It then calls the ASCIIEncoding.GetBytes(String, Int32, Int32, Byte[], Int32) method to encode the characters in the string.

```C#
using System;
using System.Text;

public class Example
```

```csharp
{
    public static void Main()
    {
        string[] strings= { "This is the first sentence. ",
                            "This is the second sentence. " };
        Encoding asciiEncoding = Encoding.ASCII;

        // Create array of adequate size.
        byte[] bytes = new byte[49];
        // Create index for current position of array.
        int index = 0;

        Console.WriteLine("Strings to encode:");
        foreach (var stringValue in strings) {
            Console.WriteLine("   {0}", stringValue);

            int count = asciiEncoding.GetByteCount(stringValue);
            if (count + index >=  bytes.Length)
                Array.Resize(ref bytes, bytes.Length + 50);

            int written = asciiEncoding.GetBytes(stringValue, 0,
                                                 stringValue.Length,
                                                 bytes, index);

            index = index + written;
        }
        Console.WriteLine("\nEncoded bytes:");
        Console.WriteLine("{0}", ShowByteValues(bytes, index));
        Console.WriteLine();

        // Decode Unicode byte array to a string.
        string newString = asciiEncoding.GetString(bytes, 0, index);
        Console.WriteLine("Decoded: {0}", newString);
    }

    private static string ShowByteValues(byte[] bytes, int last )
    {
        string returnString = "   ";
        for (int ctr = 0; ctr <= last - 1; ctr++) {
            if (ctr % 20 == 0)
                returnString += "\n   ";
            returnString += String.Format("{0:X2} ", bytes[ctr]);
        }
        return returnString;
    }
}
// The example displays the following output:
//       Strings to encode:
//          This is the first sentence.
//          This is the second sentence.
```

```
//
//        Encoded bytes:
//
//            54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//            6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
//            73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
//
//        Decoded: This is the first sentence. This is the second sentence.
```

A decoder converts a byte array that reflects a particular character encoding into a set of characters, either in a character array or in a string. To decode a byte array into a character array, you call the Encoding.GetChars method. To decode a byte array into a string, you call the GetString method. If you want to determine how many characters are needed to store the decoded bytes before performing the decoding, you can call the GetCharCount method.

The following example encodes three strings and then decodes them into a single array of characters. It maintains an index that indicates the starting position in the character array for the next set of decoded characters. It calls the GetCharCount method to ensure that the character array is large enough to accommodate all the decoded characters. It then calls the ASCIIEncoding.GetChars(Byte[], Int32, Int32, Char[], Int32) method to decode the byte array.

```csharp
C#

using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings = { "This is the first sentence. ",
                           "This is the second sentence. ",
                           "This is the third sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;
      // Array to hold encoded bytes.
      byte[] bytes;
      // Array to hold decoded characters.
      char[] chars = new char[50];
      // Create index for current position of character array.
      int index = 0;

      foreach (var stringValue in strings) {
         Console.WriteLine("String to Encode: {0}", stringValue);
```

```
            // Encode the string to a byte array.
            bytes = asciiEncoding.GetBytes(stringValue);
            // Display the encoded bytes.
            Console.Write("Encoded bytes: ");
            for (int ctr = 0; ctr < bytes.Length; ctr++)
                Console.Write(" {0}{1:X2}",
                              ctr % 20 == 0 ? Environment.NewLine : "",
                              bytes[ctr]);
            Console.WriteLine();

            // Decode the bytes to a single character array.
            int count = asciiEncoding.GetCharCount(bytes);
            if (count + index >=  chars.Length)
                Array.Resize(ref chars, chars.Length + 50);

            int written = asciiEncoding.GetChars(bytes, 0,
                                                 bytes.Length,
                                                 chars, index);

            index = index + written;
            Console.WriteLine();
        }

        // Instantiate a single string containing the characters.
        string decodedString = new string(chars, 0, index - 1);
        Console.WriteLine("Decoded string: ");
        Console.WriteLine(decodedString);
    }
}
// The example displays the following output:
//    String to Encode: This is the first sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the second sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
//    65 6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the third sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    Decoded string:
//    This is the first sentence. This is the second sentence. This is the
third sentence.
```

The encoding and decoding methods of a class derived from Encoding are designed to

work on a complete set of data; that is, all the data to be encoded or decoded is supplied in a single method call. However, in some cases, data is available in a stream, and the data to be encoded or decoded may be available only from separate read operations. This requires the encoding or decoding operation to remember any saved state from its previous invocation. Methods of classes derived from Encoder and Decoder are able to handle encoding and decoding operations that span multiple method calls.

An Encoder object for a particular encoding is available from that encoding's Encoding.GetEncoder property. A Decoder object for a particular encoding is available from that encoding's Encoding.GetDecoder property. For decoding operations, note that classes derived from Decoder include a Decoder.GetChars method, but they do not have a method that corresponds to Encoding.GetString.

The following example illustrates the difference between using the Encoding.GetString and Decoder.GetChars methods for decoding a Unicode byte array. The example encodes a string that contains some Unicode characters to a file, and then uses the two decoding methods to decode them ten bytes at a time. Because a surrogate pair occurs in the tenth and eleventh bytes, it is decoded in separate method calls. As the output shows, the Encoding.GetString method is not able to correctly decode the bytes and instead replaces them with U+FFFD (REPLACEMENT CHARACTER). On the other hand, the Decoder.GetChars method is able to successfully decode the byte array to get the original string.

```csharp
using System;
using System.IO;
using System.Text;

public class Example
{
    public static void Main()
    {
        // Use default replacement fallback for invalid encoding.
        UnicodeEncoding enc = new UnicodeEncoding(true, false, false);

        // Define a string with various Unicode characters.
        string str1 = "AB YZ 19 \uD800\udc05 \u00e4";
        str1 += "Unicode characters. \u00a9 \u010C s \u0062\u0308";
        Console.WriteLine("Created original string...\n");

        // Convert string to byte array.
```

```
        byte[] bytes = enc.GetBytes(str1);

        FileStream fs = File.Create(@".\characters.bin");
        BinaryWriter bw = new BinaryWriter(fs);
        bw.Write(bytes);
        bw.Close();

        // Read bytes from file.
        FileStream fsIn = File.OpenRead(@".\characters.bin");
        BinaryReader br = new BinaryReader(fsIn);

        const int count = 10;            // Number of bytes to read at a time.
        byte[] bytesRead = new byte[10]; // Buffer (byte array).
        int read;                        // Number of bytes actually read.
        string str2 = String.Empty;      // Decoded string.

        // Try using Encoding object for all operations.
        do {
            read = br.Read(bytesRead, 0, count);
            str2 += enc.GetString(bytesRead, 0, read);
        } while (read == count);
        br.Close();
        Console.WriteLine("Decoded string using
UnicodeEncoding.GetString()...");
        CompareForEquality(str1, str2);
        Console.WriteLine();

        // Use Decoder for all operations.
        fsIn = File.OpenRead(@".\characters.bin");
        br = new BinaryReader(fsIn);
        Decoder decoder = enc.GetDecoder();
        char[] chars = new char[50];
        int index = 0;                   // Next character to write in array.
        int written = 0;                 // Number of chars written to array.
        do {
            read = br.Read(bytesRead, 0, count);
            if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >=
chars.Length)
                Array.Resize(ref chars, chars.Length + 50);

            written = decoder.GetChars(bytesRead, 0, read, chars, index);
            index += written;
        } while (read == count);
        br.Close();
        // Instantiate a string with the decoded characters.
        string str3 = new String(chars, 0, index);
        Console.WriteLine("Decoded string using
UnicodeEncoding.Decoder.GetString()...");
        CompareForEquality(str1, str3);
    }
```

```csharp
    private static void CompareForEquality(string original, string decoded)
    {
        bool result = original.Equals(decoded);
        Console.WriteLine("original = decoded: {0}",
                          original.Equals(decoded, StringComparison.Ordinal));
        if (! result) {
            Console.WriteLine("Code points in original string:");
            foreach (var ch in original)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
            Console.WriteLine();

            Console.WriteLine("Code points in decoded string:");
            foreach (var ch in decoded)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//    Created original string...
//
//    Decoded string using UnicodeEncoding.GetString()...
//    original = decoded: False
//    Code points in original string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055
006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//    Code points in decoded string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055
006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//
//    Decoded string using UnicodeEncoding.Decoder.GetString()...
//    original = decoded: True
```

Choosing a Fallback Strategy

When a method tries to encode or decode a character but no mapping exists, it must implement a fallback strategy that determines how the failed mapping should be handled. There are three types of fallback strategies:

- Best-fit fallback

- Replacement fallback

- Exception fallback

> ⓘ **Important**
>
> The most common problems in encoding operations occur when a Unicode
> character cannot be mapped to a particular code page encoding. The most
> common problems in decoding operations occur when invalid byte sequences
> cannot be translated into valid Unicode characters. For these reasons, you should
> know which fallback strategy a particular encoding object uses. Whenever possible,
> you should specify the fallback strategy used by an encoding object when you
> instantiate the object.

Best-Fit Fallback

When a character does not have an exact match in the target encoding, the encoder can
try to map it to a similar character. (Best-fit fallback is mostly an encoding rather than a
decoding issue. There are very few code pages that contain characters that cannot be
successfully mapped to Unicode.) Best-fit fallback is the default for code page and
double-byte character set encodings that are retrieved by the
Encoding.GetEncoding(Int32) and Encoding.GetEncoding(String) overloads.

> ⚠ **Note**
>
> In theory, the Unicode encoding classes provided in .NET (**UTF8Encoding**,
> **UnicodeEncoding**, and **UTF32Encoding**) support every character in every character
> set, so they can be used to eliminate best-fit fallback issues.

Best-fit strategies vary for different code pages. For example, for some code pages, full-
width Latin characters map to the more common half-width Latin characters. For other
code pages, this mapping is not made. Even under an aggressive best-fit strategy, there
is no imaginable fit for some characters in some encodings. For example, a Chinese
ideograph has no reasonable mapping to code page 1252. In this case, a replacement
string is used. By default, this string is just a single QUESTION MARK (U+003F).

> ⓘ **Note**
>
> Best-fit strategies are not documented in detail. However, several code pages are documented at the **Unicode Consortium's** website. Please review the **readme.txt** file in that folder for a description of how to interpret the mapping files.

The following example uses code page 1252 (the Windows code page for Western European languages) to illustrate best-fit mapping and its drawbacks. The Encoding.GetEncoding(Int32) method is used to retrieve an encoding object for code page 1252. By default, it uses a best-fit mapping for Unicode characters that it does not support. The example instantiates a string that contains three non-ASCII characters - CIRCLED LATIN CAPITAL LETTER S (U+24C8), SUPERSCRIPT FIVE (U+2075), and INFINITY (U+221E) - separated by spaces. As the output from the example shows, when the string is encoded, the three original non-space characters are replaced by QUESTION MARK (U+003F), DIGIT FIVE (U+0035), and DIGIT EIGHT (U+0038). DIGIT EIGHT is a particularly poor replacement for the unsupported INFINITY character, and QUESTION MARK indicates that no mapping was available for the original character.

```csharp
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        // Get an encoding for code page 1252 (Western Europe character set).
        Encoding cp1252 = Encoding.GetEncoding(1252);

        // Define and display a string.
        string str = "\u24c8 \u2075 \u221e";
        Console.WriteLine("Original string: " + str);
        Console.Write("Code points in string: ");
        foreach (var ch in str)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

        Console.WriteLine("\n");

        // Encode a Unicode string.
        Byte[] bytes = cp1252.GetBytes(str);
        Console.Write("Encoded bytes: ");
        foreach (byte byt in bytes)
```

```
            Console.Write("{0:X2} ", byt);
        Console.WriteLine("\n");

        // Decode the string.
        string str2 = cp1252.GetString(bytes);
        Console.WriteLine("String round-tripped: {0}", str.Equals(str2));
        if (! str.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
        }
    }
}
// The example displays the following output:
//      Original string: Ⓢ ⁵ ∞
//      Code points in string: 24C8 0020 2075 0020 221E
//
//      Encoded bytes: 3F 20 35 20 38
//
//      String round-tripped: False
//      ? 5 8
//      003F 0020 0035 0020 0038
```

Best-fit mapping is the default behavior for an Encoding object that encodes Unicode data into code page data, and there are legacy applications that rely on this behavior. However, most new applications should avoid best-fit behavior for security reasons. For example, applications should not put a domain name through a best-fit encoding.

> ⓘ **Note**
>
> You can also implement a custom best-fit fallback mapping for an encoding. For more information, see the **Implementing a Custom Fallback Strategy** section.

If best-fit fallback is the default for an encoding object, you can choose another fallback strategy when you retrieve an Encoding object by calling the Encoding.GetEncoding(Int32, EncoderFallback, DecoderFallback) or Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) overload. The following section includes an example that replaces each character that cannot be mapped to code page 1252 with an asterisk (*).

```
C#
```

```csharp
using System;
```

```csharp
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding cp1252r = Encoding.GetEncoding(1252,
                                    new EncoderReplacementFallback("*"),
                                    new DecoderReplacementFallback("*"));

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

        Console.WriteLine();

        byte[] bytes = cp1252r.GetBytes(str1);
        string str2 = cp1252r.GetString(bytes);
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
        if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

Replacement Fallback

When a character does not have an exact match in the target scheme, but there is no appropriate character that it can be mapped to, the application can specify a replacement character or string. This is the default behavior for the Unicode decoder, which replaces any two-byte sequence that it cannot decode with REPLACEMENT_CHARACTER (U+FFFD). It is also the default behavior of the ASCIIEncoding class, which replaces each character that it cannot encode or decode with a question mark. The following example illustrates character replacement for the

Unicode string from the previous example. As the output shows, each character that cannot be decoded into an ASCII byte value is replaced by 0x3F, which is the ASCII code for a question mark.

C#

```csharp
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding enc = Encoding.ASCII;

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

        Console.WriteLine("\n");

        // Encode the original string using the ASCII encoder.
        byte[] bytes = enc.GetBytes(str1);
        Console.Write("Encoded bytes: ");
        foreach (var byt in bytes)
            Console.Write("{0:X2} ", byt);
        Console.WriteLine("\n");

        // Decode the ASCII bytes.
        string str2 = enc.GetString(bytes);
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
        if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 3F 20 3F
//
//       Round-trip: False
//       ? ? ?
```

```
//          003F 0020 003F 0020 003F
```

.NET includes the EncoderReplacementFallback and DecoderReplacementFallback classes, which substitute a replacement string if a character does not map exactly in an encoding or decoding operation. By default, this replacement string is a question mark, but you can call a class constructor overload to choose a different string. Typically, the replacement string is a single character, although this is not a requirement. The following example changes the behavior of the code page 1252 encoder by instantiating an EncoderReplacementFallback object that uses an asterisk (*) as a replacement string.

C#

```csharp
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding cp1252r = Encoding.GetEncoding(1252,
                                    new EncoderReplacementFallback("*"),
                                    new DecoderReplacementFallback("*"));

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

        Console.WriteLine();

        byte[] bytes = cp1252r.GetBytes(str1);
        string str2 = cp1252r.GetString(bytes);
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
        if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//          Ⓢ ⁵ ∞
//          24C8 0020 2075 0020 221E
//          Round-trip: False
```

```
//        * * *
//        002A 0020 002A 0020 002A
```

> ⓘ **Note**
>
> You can also implement a replacement class for an encoding. For more information,
> see the **Implementing a Custom Fallback Strategy** section.

In addition to QUESTION MARK (U+003F), the Unicode REPLACEMENT CHARACTER
(U+FFFD) is commonly used as a replacement string, particularly when decoding byte
sequences that cannot be successfully translated into Unicode characters. However, you
are free to choose any replacement string, and it can contain multiple characters.

Exception Fallback

Instead of providing a best-fit fallback or a replacement string, an encoder can throw an
EncoderFallbackException if it is unable to encode a set of characters, and a decoder can
throw a DecoderFallbackException if it is unable to decode a byte array. To throw an
exception in encoding and decoding operations, you supply an
EncoderExceptionFallback object and a DecoderExceptionFallback object, respectively, to
the Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method. The
following example illustrates exception fallback with the ASCIIEncoding class.

```csharp
C#

using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding enc = Encoding.GetEncoding("us-ascii",
                                            new EncoderExceptionFallback(),
                                            new DecoderExceptionFallback());

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
```

```csharp
            Console.WriteLine("\n");

            // Encode the original string using the ASCII encoder.
            byte[] bytes = {};
            try {
                bytes = enc.GetBytes(str1);
                Console.Write("Encoded bytes: ");
                foreach (var byt in bytes)
                    Console.Write("{0:X2} ", byt);

                Console.WriteLine();
            }
            catch (EncoderFallbackException e) {
                Console.Write("Exception: ");
                if (e.IsUnknownSurrogate())
                    Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} 0x{1:X3} at index {2}.",
                                      Convert.ToUInt16(e.CharUnknownHigh),
                                      Convert.ToUInt16(e.CharUnknownLow),
                                      e.Index);
                else
                    Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                                      Convert.ToUInt16(e.CharUnknown),
                                      e.Index);
                return;
            }
            Console.WriteLine();

            // Decode the ASCII bytes.
            try {
                string str2 = enc.GetString(bytes);
                Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
                if (! str1.Equals(str2)) {
                    Console.WriteLine(str2);
                    foreach (var ch in str2)
                        Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

                    Console.WriteLine();
                }
            }
            catch (DecoderFallbackException e) {
                Console.Write("Unable to decode byte(s) ");
                foreach (byte unknown in e.BytesUnknown)
                    Console.Write("0x{0:X2} ");

                Console.WriteLine("at index {0}", e.Index);
            }
        }
    }
}
// The example displays the following output:
```

```
//        Ⓢ ⁵ ∞
//        24C8 0020 2075 0020 221E
//
//        Exception: Unable to encode 0x24C8 at index 0.
```

> ⓘ **Note**
>
> You can also implement a custom exception handler for an encoding operation. For
> more information, see the Implementing a Custom Fallback Strategy section.

The EncoderFallbackException and DecoderFallbackException objects provide the
following information about the condition that caused the exception:

- The EncoderFallbackException object includes an IsUnknownSurrogate method,
  which indicates whether the character or characters that cannot be encoded
  represent an unknown surrogate pair (in which case, the method returns `true`) or
  an unknown single character (in which case, the method returns `false`). The
  characters in the surrogate pair are available from the
  EncoderFallbackException.CharUnknownHigh and
  EncoderFallbackException.CharUnknownLow properties. The unknown single
  character is available from the EncoderFallbackException.CharUnknown property.
  The EncoderFallbackException.Index property indicates the position in the string at
  which the first character that could not be encoded was found.

- The DecoderFallbackException object includes a BytesUnknown property that
  returns an array of bytes that cannot be decoded. The
  DecoderFallbackException.Index property indicates the starting position of the
  unknown bytes.

Although the EncoderFallbackException and DecoderFallbackException objects provide
adequate diagnostic information about the exception, they do not provide access to the
encoding or decoding buffer. Therefore, they do not allow invalid data to be replaced or
corrected within the encoding or decoding method.

Implementing a Custom Fallback Strategy

In addition to the best-fit mapping that is implemented internally by code pages, .NET
includes the following classes for implementing a fallback strategy:

- Use EncoderReplacementFallback and EncoderReplacementFallbackBuffer to replace characters in encoding operations.

- Use DecoderReplacementFallback and DecoderReplacementFallbackBuffer to replace characters in decoding operations.

- Use EncoderExceptionFallback and EncoderExceptionFallbackBuffer to throw an EncoderFallbackException when a character cannot be encoded.

- Use DecoderExceptionFallback and DecoderExceptionFallbackBuffer to throw a DecoderFallbackException when a character cannot be decoded.

In addition, you can implement a custom solution that uses best-fit fallback, replacement fallback, or exception fallback, by following these steps:

1. Derive a class from EncoderFallback for encoding operations, and from DecoderFallback for decoding operations.

2. Derive a class from EncoderFallbackBuffer for encoding operations, and from DecoderFallbackBuffer for decoding operations.

3. For exception fallback, if the predefined EncoderFallbackException and DecoderFallbackException classes do not meet your needs, derive a class from an exception object such as Exception or ArgumentException.

Deriving from EncoderFallback or DecoderFallback

To implement a custom fallback solution, you must create a class that inherits from EncoderFallback for encoding operations, and from DecoderFallback for decoding operations. Instances of these classes are passed to the Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method and serve as the intermediary between the encoding class and the fallback implementation.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The EncoderFallback.MaxCharCount or DecoderFallback.MaxCharCount property, which returns the maximum possible number of characters that the best-fit, replacement, or exception fallback can return to replace a single character. For a custom exception fallback, its value is zero.

- The EncoderFallback.CreateFallbackBuffer or DecoderFallback.CreateFallbackBuffer method, which returns your custom EncoderFallbackBuffer or DecoderFallbackBuffer implementation. The method is called by the encoder when it encounters the first character that it is unable to successfully encode, or by the decoder when it encounters the first byte that it is unable to successfully decode.

Deriving from EncoderFallbackBuffer or DecoderFallbackBuffer

To implement a custom fallback solution, you must also create a class that inherits from EncoderFallbackBuffer for encoding operations, and from DecoderFallbackBuffer for decoding operations. Instances of these classes are returned by the CreateFallbackBuffer method of the EncoderFallback and DecoderFallback classes. The EncoderFallback.CreateFallbackBuffer method is called by the encoder when it encounters the first character that it is not able to encode, and the DecoderFallback.CreateFallbackBuffer method is called by the decoder when it encounters one or more bytes that it is not able to decode. The EncoderFallbackBuffer and DecoderFallbackBuffer classes provide the fallback implementation. Each instance represents a buffer that contains the fallback characters that will replace the character that cannot be encoded or the byte sequence that cannot be decoded.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The EncoderFallbackBuffer.Fallback or DecoderFallbackBuffer.Fallback method. EncoderFallbackBuffer.Fallback is called by the encoder to provide the fallback buffer with information about the character that it cannot encode. Because the character to be encoded may be a surrogate pair, this method is overloaded. One overload is passed the character to be encoded and its index in the string. The second overload is passed the high and low surrogate along with its index in the string. The DecoderFallbackBuffer.Fallback method is called by the decoder to provide the fallback buffer with information about the bytes that it cannot decode. This method is passed an array of bytes that it cannot decode, along with the index of the first byte. The fallback method should return `true` if the fallback buffer can supply a best-fit or replacement character or characters; otherwise, it should return `false`. For an exception fallback, the fallback method should throw an exception.

- The EncoderFallbackBuffer.GetNextChar or DecoderFallbackBuffer.GetNextChar

method, which is called repeatedly by the encoder or decoder to get the next character from the fallback buffer. When all fallback characters have been returned, the method should return U+0000.

- The EncoderFallbackBuffer.Remaining or DecoderFallbackBuffer.Remaining property, which returns the number of characters remaining in the fallback buffer.

- The EncoderFallbackBuffer.MovePrevious or DecoderFallbackBuffer.MovePrevious method, which moves the current position in the fallback buffer to the previous character.

- The EncoderFallbackBuffer.Reset or DecoderFallbackBuffer.Reset method, which reinitializes the fallback buffer.

If the fallback implementation is a best-fit fallback or a replacement fallback, the classes derived from EncoderFallbackBuffer and DecoderFallbackBuffer also maintain two private instance fields: the exact number of characters in the buffer; and the index of the next character in the buffer to return.

An EncoderFallback Example

An earlier example used replacement fallback to replace Unicode characters that did not correspond to ASCII characters with an asterisk (*). The following example uses a custom best-fit fallback implementation instead to provide a better mapping of non-ASCII characters.

The following code defines a class named `CustomMapper` that is derived from EncoderFallback to handle the best-fit mapping of non-ASCII characters. Its `CreateFallbackBuffer` method returns a `CustomMapperFallbackBuffer` object, which provides the EncoderFallbackBuffer implementation. The `CustomMapper` class uses a Dictionary<TKey,TValue> object to store the mappings of unsupported Unicode characters (the key value) and their corresponding 8-bit characters (which are stored in two consecutive bytes in a 64-bit integer). To make this mapping available to the fallback buffer, the `CustomMapper` instance is passed as a parameter to the `CustomMapperFallbackBuffer` class constructor. Because the longest mapping is the string "INF" for the Unicode character U+221E, the `MaxCharCount` property returns 3.

```
C#
```

```csharp
public class CustomMapper : EncoderFallback
{
    public string DefaultString;
    internal Dictionary<ushort, ulong> mapping;

    public CustomMapper() : this("*")
    {
    }

    public CustomMapper(string defaultString)
    {
        this.DefaultString = defaultString;

        // Create table of mappings
        mapping = new Dictionary<ushort, ulong>();
        mapping.Add(0x24C8, 0x53);
        mapping.Add(0x2075, 0x35);
        mapping.Add(0x221E, 0x49004E0046);
    }

    public override EncoderFallbackBuffer CreateFallbackBuffer()
    {
        return new CustomMapperFallbackBuffer(this);
    }

    public override int MaxCharCount
    {
        get { return 3; }
    }
}
```

The following code defines the `CustomMapperFallbackBuffer` class, which is derived from EncoderFallbackBuffer. The dictionary that contains best-fit mappings and that is defined in the `CustomMapper` instance is available from its class constructor. Its `Fallback` method returns `true` if any of the Unicode characters that the ASCII encoder cannot encode are defined in the mapping dictionary; otherwise, it returns `false`. For each fallback, the private `count` variable indicates the number of characters that remain to be returned, and the private `index` variable indicates the position in the string buffer, `charsToReturn`, of the next character to return.

```
C#
```

```csharp
public class CustomMapperFallbackBuffer : EncoderFallbackBuffer
{
```

```csharp
        int count = -1;                         // Number of characters to return
        int index = -1;                         // Index of character to return
        CustomMapper fb;
        string charsToReturn;

        public CustomMapperFallbackBuffer(CustomMapper fallback)
        {
            this.fb = fallback;
        }

        public override bool Fallback(char charUnknownHigh, char charUnknownLow,
    int index)
        {
            // Do not try to map surrogates to ASCII.
            return false;
        }

        public override bool Fallback(char charUnknown, int index)
        {
            // Return false if there are already characters to map.
            if (count >= 1) return false;

            // Determine number of characters to return.
            charsToReturn = String.Empty;

            ushort key = Convert.ToUInt16(charUnknown);
            if (fb.mapping.ContainsKey(key)) {
                byte[] bytes = BitConverter.GetBytes(fb.mapping[key]);
                int ctr = 0;
                foreach (var byt in bytes) {
                    if (byt > 0) {
                        ctr++;
                        charsToReturn += (char) byt;
                    }
                }
                count = ctr;
            }
            else {
                // Return default.
                charsToReturn = fb.DefaultString;
                count = 1;
            }
            this.index = charsToReturn.Length - 1;

            return true;
        }

        public override char GetNextChar()
        {
            // We'll return a character if possible, so subtract from the count of
```

```csharp
chars to return.
        count--;
        // If count is less than zero, we've returned all characters.
        if (count < 0)
            return '\u0000';

        this.index--;
        return charsToReturn[this.index + 1];
    }

    public override bool MovePrevious()
    {
        // Original: if count >= -1 and pos >= 0
        if (count >= -1) {
            count++;
            return true;
        }
        else {
            return false;
        }
    }

    public override int Remaining
    {
        get { return count < 0 ? 0 : count; }
    }

    public override void Reset()
    {
        count = -1;
        index = -1;
    }
}
```

The following code then instantiates the `CustomMapper` object and passes an instance of it to the Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) method. The output indicates that the best-fit fallback implementation successfully handles the three non-ASCII characters in the original string.

```csharp
C#

using System;
using System.Collections.Generic;
using System.Text;

class Program
{
```

```csharp
    static void Main()
    {
        Encoding enc = Encoding.GetEncoding("us-ascii", new CustomMapper(),
new DecoderExceptionFallback());

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        for (int ctr = 0; ctr <= str1.Length - 1; ctr++) {
            Console.Write("{0} ", Convert.ToUInt16(str1[ctr]).ToString("X4"));
            if (ctr == str1.Length - 1)
                Console.WriteLine();
        }
        Console.WriteLine();

        // Encode the original string using the ASCII encoder.
        byte[] bytes = enc.GetBytes(str1);
        Console.Write("Encoded bytes: ");
        foreach (var byt in bytes)
            Console.Write("{0:X2} ", byt);

        Console.WriteLine("\n");

        // Decode the ASCII bytes.
        string str2 = enc.GetString(bytes);
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
        if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
        }
    }
}
```

See also

- Introduction to character encoding in .NET
- Encoder
- Decoder
- DecoderFallback
- Encoding
- EncoderFallback
- Globalization and localization