Character encoding in .NET

03/09/2020 • 17 minutes to read •

In this article

This article provides an introduction to character encoding systems that are used by .NET. The article explains how the String, Char, Rune, and StringInfo types work with Unicode, UTF-16, and UTF-8.

The term *character* is used here in the general sense of *what a reader perceives as a single display element*. Common examples are the letter "a", the symbol "@", and the emoji "🐂". Sometimes what looks like one character is actually composed of multiple independent display elements, as the section on grapheme clusters explains.

The string and char types

An instance of the string class represents some text. A `string` is logically a sequence of 16-bit values, each of which is an instance of the char struct. The string.Length property returns the number of `char` instances in the `string` instance.

The following sample function prints out the values in hexadecimal notation of all the `char` instances in a `string`:

C#

Copy

```
void PrintChars(string s)
{
    Console.WriteLine($"\"{s}\".Length = {s.Length}");
    for (int i = 0; i < s.Length; i++)
    {
```

```csharp
        Console.WriteLine($"s[{i}] = '{s[i]}' ('\\u{(int)s[i]:x4}')");
    }
    Console.WriteLine();
}
```

Pass the string "Hello" to this function, and you get the following output:

C#

```csharp
PrintChars("Hello");
```

Output

```
"Hello".Length = 5
s[0] = 'H' ('\u0048')
s[1] = 'e' ('\u0065')
s[2] = 'l' ('\u006c')
s[3] = 'l' ('\u006c')
s[4] = 'o' ('\u006f')
```

Each character is represented by a single `char` value. That pattern holds true for most of the world's languages. For example, here's the output for two Chinese characters that sound like *nǐ hǎo* and mean *Hello*:

C#

```csharp
PrintChars("你好");
```

Output

```
"你好".Length = 2
s[0] = '你' ('\u4f60')
s[1] = '好' ('\u597d')
```

However, for some languages and for some symbols and emoji, it takes two `char` instances to represent a single character. For example, compare the characters and `char` instances in the word that means *Osage* in the Osage language:

C#

```csharp
PrintChars("𐓏𐓘𐓻𐓘𐓻𐓟 𐒻𐓟");
```

Output

```
"𐓏𐓘𐓻𐓘𐓻𐓟 𐒻𐓟".Length = 17
s[0] = '�' ('\ud801')
s[1] = '�' ('\udccf')
s[2] = '�' ('\ud801')
s[3] = '�' ('\udcd8')
s[4] = '�' ('\ud801')
s[5] = '�' ('\udcfb')
s[6] = '�' ('\ud801')
s[7] = '�' ('\udcd8')
s[8] = '�' ('\ud801')
s[9] = '�' ('\udcfb')
s[10] = '�' ('\ud801')
s[11] = '�' ('\udcdf')
s[12] = ' ' ('\u0020')
s[13] = '�' ('\ud801')
s[14] = '�' ('\udcbb')
s[15] = '�' ('\ud801')
s[16] = '�' ('\udcdf')
```

In the preceding example, each character except the space is represented by two `char` instances.

A single Unicode emoji is also represented by two `char`s, as seen in the following example showing an ox emoji:

```
"🐂".Length = 2
s[0] = '�' ('\ud83d')
s[1] = '�' ('\udc02')
```

These examples show that the value of `string.Length`, which indicates the number of `char` instances, doesn't necessarily indicate the number of displayed characters. A single `char` instance by itself doesn't necessarily represent a character.

The `char` pairs that map to a single character are called *surrogate pairs*. To understand how they work, you need to understand Unicode and UTF-16 encoding.

Unicode code points

Unicode is an international encoding standard for use on various platforms and with various languages and scripts.

The Unicode Standard defines over 1.1 million [code points](). A code point is an integer value that can range from 0 to `U+10FFFF` (decimal 1,114,111). Some code points are assigned to letters, symbols, or emoji. Others are assigned to actions that control how text or characters are displayed, such as advance to a new line. Many code points are not yet assigned.

Here are some examples of code point assignments, with links to Unicode charts in which they appear:

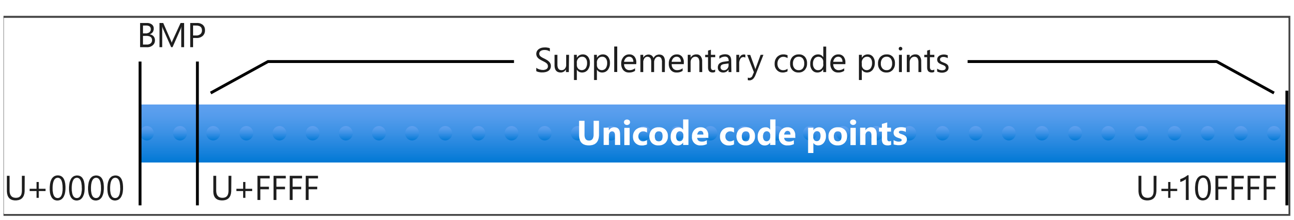
Decimal	Hex	Example	Description
10	`U+000A`	N/A	LINE FEED
65	`U+0061`	a	LATIN SMALL LETTER A
562	`U+0232`	Ȳ	LATIN CAPITAL LETTER Y WITH MACRON
68,675	`U+10C43`	𐱃	OLD TURKIC LETTER ORKHON AT
127,801	`U+1F339`	🌹	ROSE emoji

Code points are customarily referred to by using the syntax `U+xxxx`, where `xxxx` is the hex-encoded integer value.

Within the full range of code points there are two subranges:

- The **Basic Multilingual Plane (BMP)** in the range `U+0000..U+FFFF`. This 16-bit range provides 65,536 code points, enough to cover the majority of the world's writing systems.
- **Supplementary code points** in the range `U+10000..U+10FFFF`. This 21-bit range provides more than a million additional code points that can be used for less well-known languages and other purposes such as emojis.

The following diagram illustrates the relationship between the BMP and the supplementary code points.



UTF-16 code units

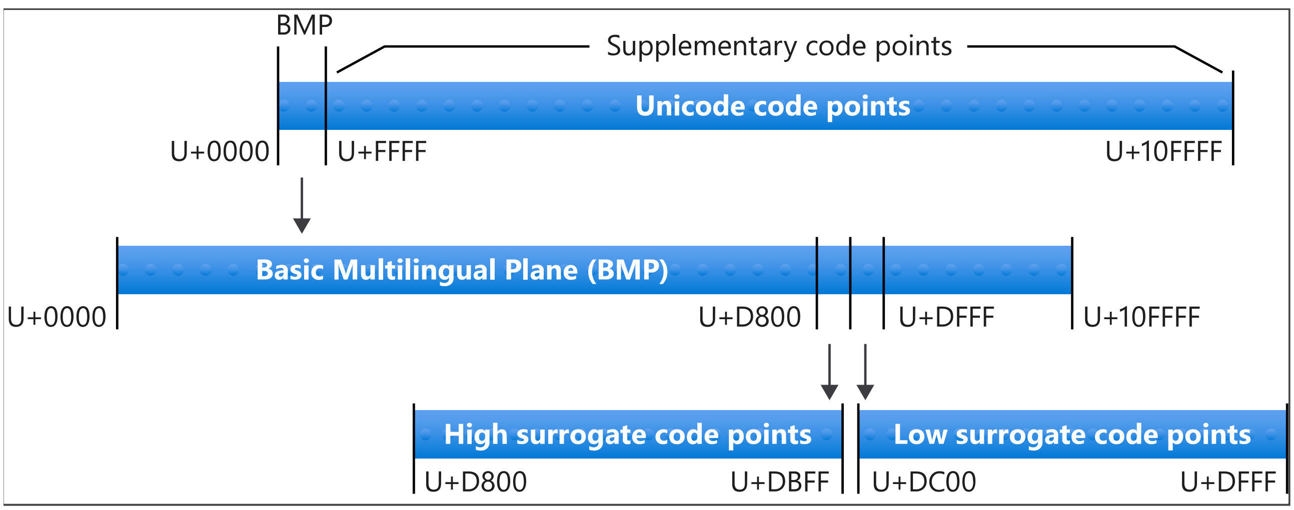
16-bit Unicode Transformation Format ([UTF-16]) is a character encoding system that uses 16-bit *code units* to represent Unicode code points. .NET uses UTF-16 to encode the text in a `string`. A `char` instance represents a 16-bit code unit.

A single 16-bit code unit can represent any code point in the 16-bit range of the Basic Multilingual Plane. But for a code point in the supplementary range, two `char` instances are needed.

Surrogate pairs

The translation of two 16-bit values to a single 21-bit value is facilitated by a special range called the *surrogate code points*, from `U+D800` to `U+DFFF` (decimal 55,296 to 57,343), inclusive.

The following diagram illustrates the relationship between the BMP and the surrogate code points.



When a *high surrogate* code point (`U+D800..U+DBFF`) is immediately followed by a *low surrogate* code point (`U+DC00..U+DFFF`), the pair is interpreted as a supplementary code point by using the following formula:

Copy

```
code point = 0x10000 +
  ((high surrogate code point - 0xD800) * 0x0400) +
  (low surrogate code point - 0xDC00)
```

Here's the same formula using decimal notation:

Copy

```
code point = 65,536 +
  ((high surrogate code point - 55,296) * 1,024) +
  (low surrogate code point - 56,320)
```

A *high* surrogate code point doesn't have a higher number value than a *low* surrogate code point. The high surrogate code point is called "high" because it's used to calculate the higher-order 11 bits of the full 21-bit code point range. The low surrogate code point is used to calculate the lower-order 10 bits.

For example, the actual code point that corresponds to the surrogate pair `0xD83C` and `0xDF39` is computed as follows:

Copy

```
actual = 0x10000 + ((0xD83C - 0xD800) * 0x0400) + (0xDF39 - 0xDC00)
       = 0x10000 + (          0x003C  * 0x0400) +           0x0339
       = 0x10000 +                      0xF000  +           0x0339
       = 0x1F339
```

Here's the same calculation using decimal notation:

Copy

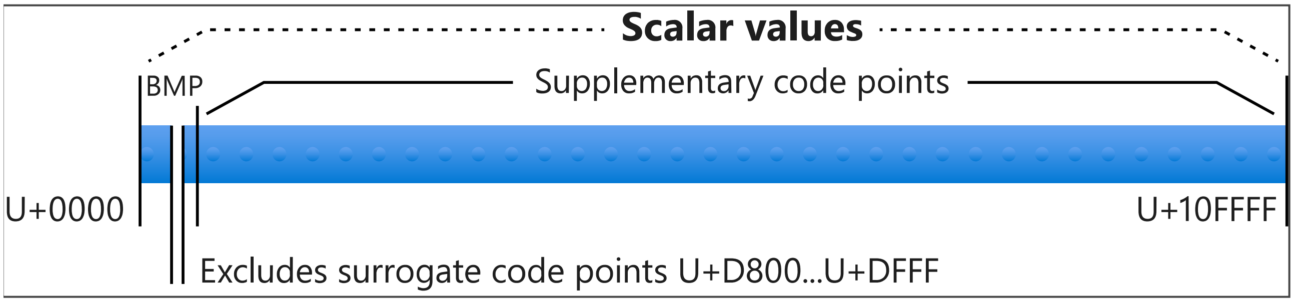
```
actual =  65,536 + ((55,356 - 55,296) * 1,024) + (57,145 - 56320)
       =  65,536 + (              60  * 1,024) +             825
       =  65,536 +                     61,440  +             825
       = 127,801
```

The preceding example demonstrates that `"\ud83c\udf39"` is the UTF-16 encoding of the `U+1F339 ROSE ('🌹')` code point mentioned earlier.

Unicode scalar values

The term [Unicode scalar value] refers to all code points other than the surrogate code points. In other words, a scalar value is any code point that is assigned a character or can be assigned a character in the future. "Character" here refers to anything that can be assigned to a code point, which includes such things as actions that control how text or characters are displayed.

The following diagram illustrates the scalar value code points.



The Rune type as a scalar value

Beginning with .NET Core 3.0, the [System.Text.Rune] type represents a Unicode scalar value. **`Rune` is not available in .NET Core 2.x or .NET Framework 4.x.**

The `Rune` constructors validate that the resulting instance is a valid Unicode scalar value, otherwise they throw an exception. The following example shows code that successfully instantiates `Rune` instances because the input represents valid scalar values:

C# Copy

```
Rune a = new Rune('a');
Rune b = new Rune(0x0061);
Rune c = new Rune('\u0061');
Rune d = new Rune(0x10421);
Rune e = new Rune('\ud801', '\udc21');
```

The following example throws an exception because the code point is in the surrogate range and isn't part of a surrogate pair:

C# Copy

```
Rune f = new Rune('\ud801');
```

The following example throws an exception because the code point is beyond the supplementary range:

C# Copy

```
Rune g = new Rune(0x12345678);
```

Rune usage example: changing letter case

An API that takes a `char` and assumes it is working with a code point that is a scalar value doesn't work correctly if the `char` is from a surrogate pair. For example, consider the following method that calls [Char.ToUpperInvariant]() on each char in a string:

C# Copy

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string ConvertToUpperBadExample(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    for (int i = 0; i < input.Length; i++) /* or 'foreach' */
    {
        builder.Append(char.ToUpperInvariant(input[i]));
    }
    return builder.ToString();
}
```

If the `input` string contains the lowercase Deseret letter `er` (𐑉), this code won't convert it to uppercase (𐐡). The code calls `char.ToUpperInvariant` separately on each surrogate code point, `U+D801` and `U+DC49`. But `U+D801` doesn't have enough information by itself to identify it as a lowercase letter, so `char.ToUpperInvariant` leaves it alone. And it handles `U+DC49` the same way. The result is that lowercase '𐑉' in the `input` string doesn't get converted to uppercase '𐐡'.

Here are two options for correctly converting a string to uppercase:

- Call [String.ToUpperInvariant]() on the input string rather than iterating `char`-by-`char`. The `string.ToUpperInvariant` method has access to both parts of each surrogate pair, so it can handle all Unicode code points correctly.

- Iterate through the Unicode scalar values as `Rune` instances instead of `char` instances, as shown in the following example. Since a `Rune` instance is a valid Unicode scalar value, it can be passed to APIs that expect to operate on a scalar value. For example, calling [Rune.ToUpperInvariant]() as shown in the following example gives correct results:

  C# Copy

```csharp
static string ConvertToUpper(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    foreach (Rune rune in input.EnumerateRunes())
    {
        builder.Append(Rune.ToUpperInvariant(rune));
    }
    return builder.ToString();
}
```

Other Rune APIs

The `Rune` type exposes analogs of many of the `char` APIs. For example, the following methods mirror static APIs on the `char` type:

- Rune.IsLetter
- Rune.IsWhiteSpace
- Rune.IsLetterOrDigit
- Rune.GetUnicodeCategory

To get the raw scalar value from a `Rune` instance, use the Rune.Value property.

To convert a `Rune` instance back to a sequence of `char`s, use Rune.ToString or the Rune.EncodeToUtf16 method.

Since any Unicode scalar value is representable by a single `char` or by a surrogate pair, any `Rune` instance can be represented by at most 2 `char` instances. Use Rune.Utf16SequenceLength to see how many `char` instances are required to represent a `Rune` instance.

For more information about the .NET `Rune` type, see the Rune API reference.

Grapheme clusters

What looks like one character might result from a combination of multiple code points, so a more descriptive term that is often used in place of "character" is grapheme cluster. The equivalent term in .NET is text element.

Consider the `string` instances "a", "á". "á", and "👩🏽‍🚒". If your operating system handles them as specified by the Unicode standard, each of these `string` instances appears as a single text element or grapheme cluster. But the last two are represented by more than one scalar value code point.

- The string "a" is represented by one scalar value and contains one `char` instance.
  - `U+0061 LATIN SMALL LETTER A`
- The string "á" is represented by one scalar value and contains one `char` instance.
  - `U+00E1 LATIN SMALL LETTER A WITH ACUTE`
- The string "á" looks the same as "á" but is represented by two scalar values and contains two `char` instances.
  - `U+0065 LATIN SMALL LETTER A`
  - `U+0301 COMBINING ACUTE ACCENT`
- Finally, the string "👩🏽‍🚒" is represented by four scalar values and contains seven `char` instances.
  - `U+1F469 WOMAN` (supplementary range, requires a surrogate pair)
  - `U+1F3FD EMOJI MODIFIER FITZPATRICK TYPE-4` (supplementary range, requires a surrogate pair)
  - `U+200D ZERO WIDTH JOINER`
  - `U+1F692 FIRE ENGINE` (supplementary range, requires a surrogate pair)

In some of the preceding examples - such as the combining accent modifier or the skin tone modifier - the code point does not display as a standalone element on the screen. Rather, it serves to modify the appearance of a text element that came before it. These examples show that it might take multiple scalar values to make up what we think of as a single "character," or "grapheme cluster."

To enumerate the grapheme clusters of a `string`, use the [StringInfo] class as shown in the following example. If you're familiar with Swift, the .NET `StringInfo` type is conceptually similar to [Swift's character type].

Example: count char, Rune, and text element instances

In .NET APIs, a grapheme cluster is called a *text element*. The following method demonstrates the differences between `char`, `Rune`, and text element instances in a `string`:

C# Copy

```
static void PrintTextElementCount(string s)
{
```

```
    Console.WriteLine(s);
    Console.WriteLine($"Number of chars: {s.Length}");
    Console.WriteLine($"Number of runes: {s.EnumerateRunes().Count()}");

    TextElementEnumerator enumerator = StringInfo.GetTextElementEnumerator(s);

    int textElementCount = 0;
    while (enumerator.MoveNext())
    {
        textElementCount++;
    }

    Console.WriteLine($"Number of text elements: {textElementCount}");
```

C#

```
PrintTextElementCount("á");
// Number of chars: 1
// Number of runes: 1
// Number of text elements: 1

PrintTextElementCount("á");
// Number of chars: 2
// Number of runes: 2
// Number of text elements: 1

PrintTextElementCount("👩🏽‍🚒");
// Number of chars: 7
// Number of runes: 4
// Number of text elements: 1
```

If you run this code in .NET Framework or .NET Core 3.1 or earlier, the text element count for the emoji shows `4`. That is due to a bug in the `StringInfo` class that is fixed in .NET 5.

Example: splitting string instances

When splitting `string` instances, avoid splitting surrogate pairs and grapheme clusters. Consider the following example of incorrect code, which intends to insert line breaks every 10 characters in a string:

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string InsertNewlinesEveryTencharsBadExample(string input)
{
```

```
    StringBuilder builder = new StringBuilder();

    // First, append chunks in multiples of 10 chars
    // followed by a newline.
    int i = 0;
    for (; i < input.Length - 10; i += 10)
    {
        builder.Append(input, i, 10);
        builder.AppendLine(); // newline
    }

    // Then append any leftover data followed by
    // a final newline.
    builder.Append(input, i, input.Length - i);
    builder.AppendLine(); // newline

    return builder.ToString();
}
```

Because this code enumerates `char` instances, a surrogate pair that happens to straddle a 10-`char` boundary will be split and a newline injected between them. This insertion introduces data corruption, because surrogate code points are meaningful only as pairs.

The potential for data corruption isn't eliminated if you enumerate `Rune` instances (scalar values) instead of `char` instances. A set of `Rune` instances might make up a grapheme cluster that straddles a 10-`char` boundary. If the grapheme cluster set is split up, it can't be interpreted correctly.

A better approach is to break the string by counting grapheme clusters, or text elements, as in the following example:

C# Copy

```
static string InsertNewlinesEveryTenTextElements(string input)
{
    StringBuilder builder = new StringBuilder();

    // Append chunks in multiples of 10 chars

    TextElementEnumerator enumerator =
StringInfo.GetTextElementEnumerator(input);

    int textElementCount = 0;
    while (enumerator.MoveNext())
    {
        builder.Append(enumerator.Current);
        if (textElementCount % 10 == 0 && textElementCount > 0)
```

```
        {
            builder.AppendLine(); // newline
        }
        textElementCount++;
    }

    // Add a final newline.
    builder.AppendLine(); // newline
    return builder.ToString();

}
```

As noted earlier, however, in implementations of .NET other than .NET 5, the `StringInfo` class might handle some grapheme clusters incorrectly.

UTF-8 and UTF-32

The preceding sections focused on UTF-16 because that's what .NET uses to encode `string` instances. There are other encoding systems for Unicode - UTF-8 and UTF-32. These encodings use 8-bit code units and 32-bit code units, respectively.

Like UTF-16, UTF-8 requires multiple code units to represent some Unicode scalar values. UTF-32 can represent any scalar value in a single 32-bit code unit.

Here are some examples showing how the same Unicode code point is represented in each of these three Unicode encoding systems:

Copy

```
Scalar: U+0061 LATIN SMALL LETTER A ('a')
UTF-8 : [ 61 ]           (1x  8-bit code unit  = 8 bits total)
UTF-16: [ 0061 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000061 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+0429 CYRILLIC CAPITAL LETTER SHCHA ('Щ')
UTF-8 : [ D0 A9 ]        (2x  8-bit code units = 16 bits total)
UTF-16: [ 0429 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000429 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+A992 JAVANESE LETTER GA ('ꦒ')
UTF-8 : [ EA A6 92 ]     (3x  8-bit code units = 24 bits total)
UTF-16: [ A992 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 0000A992 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+104CC OSAGE CAPITAL LETTER TSHA ('𐓌')
UTF-8 : [ F0 90 93 8C ]  (4x  8-bit code units = 32 bits total)
```

```
UTF-16: [ D801 DCCC ]    (2x 16-bit code units = 32 bits total)
UTF-32: [ 000104CC ]     (1x 32-bit code unit  = 32 bits total)
```

As noted earlier, a single UTF-16 code unit from a surrogate pair is meaningless by itself. In the same way, a single UTF-8 code unit is meaningless by itself if it's in a sequence of two, three, or four used to calculate a scalar value.

Endianness

In .NET, the UTF-16 code units of a string are stored in contiguous memory as a sequence of 16-bit integers (`char` instances). The bits of individual code units are laid out according to the endianness of the current architecture.

On a little-endian architecture, the string consisting of the UTF-16 code points `[ D801 DCCC ]` would be laid out in memory as the bytes `[ 0x01, 0xD8, 0xCC, 0xDC ]`. On a big-endian architecture that same string would be laid out in memory as the bytes `[ 0xD8, 0x01, 0xDC, 0xCC ]`.

Computer systems that communicate with each other must agree on the representation of data crossing the wire. Most network protocols use UTF-8 as a standard when transmitting text, partly to avoid issues that might result from a big-endian machine communicating with a little-endian machine. The string consisting of the UTF-8 code points `[ F0 90 93 8C ]` will always be represented as the bytes `[ 0xF0, 0x90, 0x93, 0x8C ]` regardless of endianness.

To use UTF-8 for transmitting text, .NET applications often use code like the following example:

C# Copy

```csharp
string stringToWrite = GetString();
byte[] stringAsUtf8Bytes = Encoding.UTF8.GetBytes(stringToWrite);
await outputStream.WriteAsync(stringAsUtf8Bytes, 0, stringAsUtf8Bytes.Length);
```

In the preceding example, the method Encoding.UTF8.GetBytes decodes the UTF-16 `string` back into a series of Unicode scalar values, then it re-encodes those scalar values into UTF-8 and places the resulting sequence into a `byte` array. The method Encoding.UTF8.GetString performs the opposite transformation, converting a UTF-8 `byte` array to a UTF-16 `string`.

⚠ Warning

Since UTF-8 is commonplace on the internet, it may be tempting to read raw bytes from the wire and to treat the data as if it were UTF-8. However, you should validate that it is indeed well-formed. A malicious client might submit ill-formed UTF-8 to your service. If you operate on that data as if it were well-formed, it could cause errors or security holes in your application. To validate UTF-8 data, you can use a method like `Encoding.UTF8.GetString`, which will perform validation while converting the incoming data to a `string`.

Well-formed encoding

A well-formed Unicode encoding is a string of code units that can be decoded unambiguously and without error into a sequence of Unicode scalar values. Well-formed data can be transcoded freely back and forth between UTF-8, UTF-16, and UTF-32.

The question of whether an encoding sequence is well-formed or not is unrelated to the endianness of a machine's architecture. An ill-formed UTF-8 sequence is ill-formed in the same way on both big-endian and little-endian machines.

Here are some examples of ill-formed encodings:

- In UTF-8, the sequence `[ 6C C2 61 ]` is ill-formed because `C2` cannot be followed by `61`.
- In UTF-16, the sequence `[ DC00 DD00 ]` (or, in C#, the string `"\udc00\udd00"`) is ill-formed because the low surrogate `DC00` cannot be followed by another low surrogate `DD00`.
- In UTF-32, the sequence `[ 0011ABCD ]` is ill-formed because `0011ABCD` is outside the range of Unicode scalar values.

In .NET, `string` instances almost always contain well-formed UTF-16 data, but that isn't guaranteed. The following examples show valid C# code that creates ill-formed UTF-16 data in `string` instances.

- An ill-formed literal:

C# Copy

```
const string s = "\ud800";
```

- A substring that splits up a surrogate pair:

C# Copy

```csharp
string x = "\ud83e\udd70"; // "🥰"
string y = x.Substring(1, 1); // "\udd70" standalone low surrogate
```

APIs like Encoding.UTF8.GetString never return ill-formed `string` instances. `Encoding.GetString` and `Encoding.GetBytes` methods detect ill-formed sequences in the input and perform character substitution when generating the output. For example, if Encoding.ASCII.GetString(byte[]) sees a non-ASCII byte in the input (outside the range U+0000..U+007F), it inserts a '?' into the returned `string` instance. Encoding.UTF8.GetString(byte[]) replaces ill-formed UTF-8 sequences with `U+FFFD REPLACEMENT CHARACTER ('�')` in the returned `string` instance. For more information, see the Unicode Standard, Sections 5.22 and 3.9.

The built-in `Encoding` classes can also be configured to throw an exception rather than perform character substitution when ill-formed sequences are seen. This approach is often used in security-sensitive applications where character substitution might not be acceptable.

C# Copy

```csharp
byte[] utf8Bytes = ReadFromNetwork();
UTF8Encoding encoding = new UTF8Encoding(encoderShouldEmitUTF8Identifier:
false, throwOnInvalidBytes: true);
string asString = encoding.GetString(utf8Bytes); // will throw if 'utf8Bytes'
is ill-formed
```

For information about how to use the built-in `Encoding` classes, see How to use character encoding classes in .NET.

See also

- String
- Char
- Rune

- Globalization and Localization