Best Practices for Using Strings in the .NET Framework

.NET Framework 4.5 8 out of 11 rated this helpful

The .NET Framework provides extensive support for developing localized and globalized applications, and makes it easy to apply the conventions of either the current culture or a specific culture when performing common operations such as sorting and displaying strings. But sorting or comparing strings is not always a culture-sensitive operation. For example, strings that are used internally by an application typically should be handled identically across all cultures. When culturally independent string data, such as XML tags, HTML tags, user names, file paths, and the names of system objects, are interpreted as if they were culture-sensitive, application code can be subject to subtle bugs, poor performance, and, in some cases, security issues.

This topic examines the string sorting, comparison, and casing methods in the .NET Framework, presents recommendations for selecting an appropriate string-handling method, and provides additional information about string-handling methods. It also examines how formatted data, such as numeric data and date and time data, is handled for display and for storage.

This topic contains the following sections:

Recommendations for String Usage

When you develop with the .NET Framework, follow these simple recommendations when you use strings:

- Use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type StringComparison.
- Use StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for comparisons as your safe default for culture-agnostic string matching.
- Use comparisons with StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase for better performance.
- Use string operations that are based on StringComparison.CurrentCulture when you display output to the user.
- Use the non-linguistic StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase values instead of string operations based on CultureInfo.InvariantCulture when the comparison is linguistically irrelevant (symbolic, for example).
- Use the String.ToUpperInvariant method instead of the String.ToLowerInvariant method when you normalize strings for comparison.
- Use an overload of the String.Equals method to test whether two strings are equal.
- Use the String.Compare and String.CompareTo methods to sort strings, not to check for equality.
- Use culture-sensitive formatting to display non-string data, such as numbers and dates, in a user interface. Use formatting with the invariant culture to persist non-string data in string form.

Avoid the following practices when you use strings:

- Do not use overloads that do not explicitly or implicitly specify the string comparison rules for string operations.
- Do not use string operations based on StringComparison.InvariantCulture in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.
- Do not use an overload of the String.Compare or CompareTo method and test for a return value of zero to determine whether two strings are equal.
- Do not use culture-sensitive formatting to persist numeric data or date and time data in string form.

Back to top

Specifying String Comparisons Explicitly

Most of the string manipulation methods in the.NET Framework are overloaded. Typically, one or more overloads accept default settings, whereas others accept no defaults and instead define the precise way in which strings are to be compared or manipulated. Most of the methods that do not rely on defaults include a parameter of type StringComparison, which is an enumeration that explicitly specifies rules for string comparison by culture and case. The following table describes the StringComparison enumeration members.

StringComparison member	Description
CurrentCulture	Performs a case-sensitive comparison using the current culture.
CurrentCultureIgnoreCase	Performs a case-insensitive comparison using the current culture.
InvariantCulture	Performs a case-sensitive comparison using the invariant culture.
InvariantCultureIgnoreCase	Performs a case-insensitive comparison using the invariant culture.
Ordinal	Performs an ordinal comparison.
OrdinalIgnoreCase	Performs a case-insensitive ordinal comparison.

For example, the IndexOf method, which returns the index of a substring in a String object that matches either a character or a string, has nine overloads:

- IndexOf(Char) , IndexOf(Char, Int32), and IndexOf(Char, Int32, Int32), which by default perform an ordinal (case-sensitive and culture-insensitive) search for a character in the string.
- IndexOf(String) , IndexOf(String, Int32), and IndexOf(String, Int32, Int32), which by default perform a case-sensitive and culture-sensitive search for a substring in the string.
- IndexOf(String, StringComparison) , IndexOf(String, Int32, StringComparison), and IndexOf(String, Int32, Int32, StringComparison), which include a parameter of type StringComparison that allows the form of the comparison to be specified.

We recommend that you select an overload that does not use default values, for the following reasons:

- Some overloads with default parameters (those that search for a Char in the string instance) perform an ordinal comparison, whereas others (those that search for a string in the string instance) are culture-sensitive. It is difficult to remember which method uses which default value, and easy to confuse the overloads.
- The intent of the code that relies on default values for method calls is not clear. In the following example, which relies on defaults, it is difficult to know whether the developer actually intended an ordinal or a linguistic comparison of two strings, or whether a case difference between protocol and "http" might cause the test for equality to return **false**.

C#

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http")) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

In general, we recommend that you call a method that does not rely on defaults, because it makes the intent of the code unambiguous. This, in turn, makes the code more readable and easier to debug and maintain. The following example addresses the questions raised about the previous example. It makes it clear that ordinal comparison is used and that differences in case are ignored.

C#

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase)) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

Back to top

The Details of String Comparison

String comparison is the heart of many string-related operations, particularly sorting and testing for equality. Strings sort in a determined order: If "my" appears before "string" in a sorted list of strings, "my" must compare less than or equal to "string". Additionally, comparison implicitly defines equality. The comparison operation returns zero for strings it deems equal. A good interpretation is that neither string is less than the other. Most meaningful operations involving strings include one or both of these procedures: comparing with another string, and executing a well-defined sort operation.

However, evaluating two strings for equality or sort order does not yield a single, correct result; the outcome depends on the criteria used to compare the strings. In particular, string comparisons that are ordinal or that are based on the casing and sorting conventions of the current culture or the invariant culture (a locale-agnostic culture based on the English language) may produce different results.

String Comparisons that Use the Current Culture

One criterion involves using the conventions of the current culture when comparing strings. Comparisons that are based on the current culture use the thread's current culture or locale. If the culture is not set by the user, it defaults to the setting in the **Regional Options** window in Control Panel. You should always use comparisons that are based on the current culture when data is linguistically relevant, and when it reflects culture-sensitive user interaction.

However, comparison and casing behavior in the .NET Framework changes when the culture changes. This happens when an application executes on a computer that has a different culture than the computer on which the application was developed, or when the executing thread changes its culture. This behavior is intentional, but it remains non-obvious to many developers. The following example illustrates differences in sort order between the U.S. English ("en-US") and Swedish ("sv-SE") cultures. Note that the words "ångström", "Windows", and "Visual Studio" appear in different positions in the sorted string arrays.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
```

```
    {
      string[] values= { "able", "ångström", "apple", "Æble",
                         "Windows", "Visual Studio" };
      Array.Sort(values);
      DisplayArray(values);

      // Change culture to Swedish (Sweden).
      string originalCulture = CultureInfo.CurrentCulture.Name;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
      Array.Sort(values);
      DisplayArray(values);

      // Restore the original culture.
      Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);
    }

    private static void DisplayArray(string[] values)
    {
      Console.WriteLine("Sorting using the {0} culture:",
                        CultureInfo.CurrentCulture.Name);
      foreach (string value in values)
        Console.WriteLine("   {0}", value);

      Console.WriteLine();
    }
}
// The example displays the following output:
//       Sorting using the en-US culture:
//         able
//         Æble
//         ångström
//         apple
//         Visual Studio
//         Windows
//
//       Sorting using the sv-SE culture:
//         able
//         Æble
//         apple
//         Windows
//         Visual Studio
//         ångström
```

Case-insensitive comparisons that use the current culture are the same as culture-sensitive comparisons, except that they ignore case as dictated by the thread's current culture. This behavior may manifest itself in sort orders as well.

Comparisons that use current culture semantics are the default for the following methods:

- String.Compare overloads that do not include a StringComparison parameter.
- String.CompareTo overloads.
- The default String.StartsWith(String) method, and the String.StartsWith(String, Boolean, CultureInfo) method with a **null** CultureInfo parameter.
- The default String.EndsWith(String) method, and the String.EndsWith(String, Boolean, CultureInfo) method with a **null** CultureInfo parameter.
- String.IndexOf overloads that accept a String as a search parameter and that do not have a StringComparison parameter.
- String.LastIndexOf overloads that accept a String as a search parameter and that do not have a StringComparison parameter.

In any case, we recommend that you call an overload that has a StringComparison parameter to make the intent of the method call

clear.

Subtle and not so subtle bugs can emerge when non-linguistic string data is interpreted linguistically, or when string data from a particular culture is interpreted using the conventions of another culture. The canonical example is the Turkish-I problem.

For nearly all Latin alphabets, including U.S. English, the character "i" (\u0069) is the lowercase version of the character "I" (\u0049). This casing rule quickly becomes the default for someone programming in such a culture. However, the Turkish ("tr-TR") alphabet includes an "I with a dot" character "İ" (\u0130), which is the capital version of "i". Turkish also includes a lowercase "i without a dot" character, "ı" (\u0131), which capitalizes to "I". This behavior occurs in the Azerbaijani ("az") culture as well.

Therefore, assumptions made about capitalizing "i" or lowercasing "I" are not valid among all cultures. If you use the default overloads for string comparison routines, they will be subject to variance between cultures. If the data to be compared is non-linguistic, using the default overloads can produce undesirable results, as the following attempt to perform a case-insensitive comparison of the strings "file" and "FILE" illustrates.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
  public static void Main()
  {
    string fileUrl = "file";
    Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
    Console.WriteLine("Culture = {0}",
                      Thread.CurrentThread.CurrentCulture.DisplayName);
    Console.WriteLine("(file == FILE) = {0}",
                      fileUrl.StartsWith("FILE", true, null));
    Console.WriteLine();

    Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
    Console.WriteLine("Culture = {0}",
                      Thread.CurrentThread.CurrentCulture.DisplayName);
    Console.WriteLine("(file == FILE) = {0}",
                      fileUrl.StartsWith("FILE", true, null));
  }
}
// The example displays the following output:
//       Culture = English (United States)
//       (file == FILE) = True
//
//       Culture = Turkish (Turkey)
//       (file == FILE) = False
```

This comparison could cause significant problems if the culture is inadvertently used in security-sensitive settings, as in the following example. A method call such as IsFileURI("file:") returns **true** if the current culture is U.S. English, but **false** if the current culture is Turkish. Thus, on Turkish systems, someone could circumvent security measures that block access to case-insensitive URIs that begin with "FILE:".

C#

```
public static bool IsFileURI(String path)
{
  return path.StartsWith("FILE:", true, null);
}
```

In this case, because "file:" is meant to be interpreted as a non-linguistic, culture-insensitive identifier, the code should instead be written as shown in the following example.

C#

```
public static bool IsFileURI(string path)
{
  return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
}
```

Ordinal String Operations

Specifying the StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase value in a method call signifies a non-linguistic comparison in which the features of natural languages are ignored. Methods that are invoked with these StringComparison values base string operation decisions on simple byte comparisons instead of casing or equivalence tables that are parameterized by culture. In most cases, this approach best fits the intended interpretation of strings while making code faster and more reliable.

Ordinal comparisons are string comparisons in which each byte of each string is compared without linguistic interpretation; for example, "windows" does not match "Windows". This is essentially a call to the C runtime **strcmp** function. Use this comparison when the context dictates that strings should be matched exactly or demands conservative matching policy. Additionally, ordinal comparison is the fastest comparison operation because it applies no linguistic rules when determining a result.

Strings in the .NET Framework can contain embedded null characters. One of the clearest differences between ordinal and culture-sensitive comparison (including comparisons that use the invariant culture) concerns the handling of embedded null characters in a string. These characters are ignored when you use the String.Compare and String.Equals methods to perform culture-sensitive comparisons (including comparisons that use the invariant culture). As a result, in culture-sensitive comparisons, strings that contain embedded null characters can be considered equal to strings that do not.

Important

Although string comparison methods disregard embedded null characters, string search methods such as String.Contains, String.EndsWith, String.IndexOf, String.LastIndexOf, and String.StartsWith do not.

The following example performs a culture-sensitive comparison of the string "Aa" with a similar string that contains several embedded null characters between "A" and "a", and shows how the two strings are considered equal.

C#

```
using System;

public class Example
{
  public static void Main()
  {
    string str1 = "Aa";
    string str2 = "A" + new String('\u0000', 3) + "a";
    Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                      str1, ShowBytes(str1), str2, ShowBytes(str2));
    Console.WriteLine("   With String.Compare:");
    Console.WriteLine("      Current Culture: {0}",
                      String.Compare(str1, str2, StringComparison.CurrentCulture));
    Console.WriteLine("      Invariant Culture: {0}",
                      String.Compare(str1, str2, StringComparison.InvariantCulture));

    Console.WriteLine("   With String.Equals:");
    Console.WriteLine("      Current Culture: {0}",
                      String.Equals(str1, str2, StringComparison.CurrentCulture));
    Console.WriteLine("      Invariant Culture: {0}",
                      String.Equals(str1, str2, StringComparison.InvariantCulture));
  }

  private static string ShowBytes(string str)
  {
    string hexString = String.Empty;
    for (int ctr = 0; ctr < str.Length; ctr++)
```

```
        {
          string result = String.Empty;
          result = Convert.ToInt32(str[ctr]).ToString("X4");
          result = " " + result.Substring(0,2) + " " + result.Substring(2, 2);
          hexString += result;
        }
        return hexString.Trim();
      }
   }
   // The example displays the following output:
   //    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
   //       With String.Compare:
   //          Current Culture: 0
   //          Invariant Culture: 0
   //       With String.Equals:
   //          Current Culture: True
   //          Invariant Culture: True
```

However, the strings are not considered equal when you use ordinal comparison, as the following example shows.

C#

```
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                  str1, ShowBytes(str1), str2, ShowBytes(str2));
Console.WriteLine("   With String.Compare:");
Console.WriteLine("      Ordinal: {0}",
                  String.Compare(str1, str2, StringComparison.Ordinal));

Console.WriteLine("   With String.Equals:");
Console.WriteLine("      Ordinal: {0}",
                  String.Equals(str1, str2, StringComparison.Ordinal));
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Ordinal: 97
//       With String.Equals:
//          Ordinal: False
```

Case-insensitive ordinal comparisons are the next most conservative approach. These comparisons ignore most casing; for example, "windows" matches "Windows". When dealing with ASCII characters, this policy is equivalent to StringComparison.Ordinal, except that it ignores the usual ASCII casing. Therefore, any character in [A, Z] (\u0041-\u005A) matches the corresponding character in [a,z] (\u0061-\007A). Casing outside the ASCII range uses the invariant culture's tables. Therefore, the following comparison:

C#

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

is equivalent to (but faster than) this comparison:

C#

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal);
```

These comparisons are still very fast.

Note

The string behavior of the file system, registry keys and values, and environment variables is best represented by StringComparison.OrdinalIgnoreCase.

Both StringComparison.Ordinal and StringComparison.OrdinalIgnoreCase use the binary values directly, and are best suited for matching. When you are not sure about your comparison settings, use one of these two values. However, because they perform a byte-by-byte comparison, they do not sort by a linguistic sort order (like an English dictionary) but by a binary sort order. The results may look odd in most contexts if displayed to users.

Ordinal semantics are the default for String.Equals overloads that do not include a StringComparison argument (including the equality operator). In any case, we recommend that you call an overload that has a StringComparison parameter.

String Operations that Use the Invariant Culture

Comparisons with the invariant culture use the CompareInfo property returned by the static CultureInfo.InvariantCulture property. This behavior is the same on all systems; it translates any characters outside its range into what it believes are equivalent invariant characters. This policy can be useful for maintaining one set of string behavior across cultures, but it often provides unexpected results.

Case-insensitive comparisons with the invariant culture use the static CompareInfo property returned by the static CultureInfo.InvariantCulture property for comparison information as well. Any case differences among these translated characters are ignored.

Comparisons that use StringComparison.InvariantCulture and StringComparison.Ordinal work identically on ASCII strings. However, StringComparison.InvariantCulture makes linguistic decisions that might not be appropriate for strings that have to be interpreted as a set of bytes. The **CultureInfo.InvariantCulture.CompareInfo** object makes the Compare method interpret certain sets of characters as equivalent. For example, the following equivalence is valid under the invariant culture:

InvariantCulture: a + ° = å

The LATIN SMALL LETTER A character "a" (\u0061), when it is next to the COMBINING RING ABOVE character "+ "̊" (\u030a), is interpreted as the LATIN SMALL LETTER A WITH RING ABOVE character "å" (\u00e5). As the following example shows, this behavior differs from ordinal comparison.

C#

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.InvariantCulture) == 0);

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.Ordinal) == 0);
// The example displays the following output:
//    Equal sort weight of a° and å using InvariantCulture: True
//    Equal sort weight of a° and å using Ordinal: False
```

When interpreting file names, cookies, or anything else where a combination such as "å" can appear, ordinal comparisons still offer the most transparent and fitting behavior.

On balance, the invariant culture has very few properties that make it useful for comparison. It does comparison in a linguistically relevant manner, which prevents it from guaranteeing full symbolic equivalence, but it is not the choice for display in any culture. One of the few reasons to use StringComparison.InvariantCulture for comparison is to persist ordered data for a cross-culturally identical display. For example, if a large data file that contains a list of sorted identifiers for display accompanies an application, adding to this list would require an insertion with invariant-style sorting.

Back to top

Choosing a StringComparison Member for Your Method Call

The following table outlines the mapping from semantic string context to a StringComparison enumeration member.

Data	**Behavior**	**Corresponding System.StringComparison value**
Case-sensitive internal identifiers. Case-sensitive identifiers in standards such as XML and HTTP. Case-sensitive security-related settings.	A non-linguistic identifier, where bytes match exactly.	Ordinal
Case-insensitive internal identifiers. Case-insensitive identifiers in standards such as XML and HTTP. File paths. Registry keys and values. Environment variables. Resource identifiers (for example, handle names). Case-insensitive security-related settings.	A non-linguistic identifier, where case is irrelevant; especially data stored in most Windows system services.	OrdinalIgnoreCase
Some persisted, linguistically relevant data. Display of linguistic data that requires a fixed sort order.	Culturally agnostic data that still is linguistically relevant.	InvariantCulture -or- InvariantCultureIgnoreCase
Data displayed to the user. Most user input.	Data that requires local linguistic customs.	CurrentCulture -or- CurrentCultureIgnoreCase

Back to top

Common String Comparison Methods in the .NET Framework

The following sections describe the methods that are most commonly used for string comparison.

String.Compare

Default interpretation: StringComparison.CurrentCulture.

As the operation most central to string interpretation, all instances of these method calls should be examined to determine whether strings should be interpreted according to the current culture, or dissociated from the culture (symbolically). Typically, it is the latter, and a StringComparison.Ordinal comparison should be used instead.

The System.Globalization.CompareInfo class, which is returned by the CultureInfo.CompareInfo property, also includes a Compare method that provides a large number of matching options (ordinal, ignoring white space, ignoring kana type, and so on) by means of the CompareOptions flag enumeration.

String.CompareTo

Default interpretation: StringComparison.CurrentCulture.

This method does not currently offer an overload that specifies a StringComparison type. It is usually possible to convert this method to the recommended String.Compare(String, String, StringComparison) form.

Types that implement the IComparable and IComparable<T> interfaces implement this method. Because it does not offer the option of a StringComparison parameter, implementing types often let the user specify a StringComparer in their constructor. The following example defines a FileName class whose class constructor includes a StringComparer parameter. This StringComparer object is then used in the FileName.CompareTo method.

C#

```
using System;

public class FileName : IComparable
{
  string fname;
  StringComparer comparer;

  public FileName(string name, StringComparer comparer)
  {
    if (String.IsNullOrEmpty(name))
      throw new ArgumentNullException("name");

    this.fname = name;

    if (comparer != null)
      this.comparer = comparer;
    else
      this.comparer = StringComparer.OrdinalIgnoreCase;
  }

  public string Name
  {
    get { return fname; }
  }

  public int CompareTo(object obj)
  {
    if (obj == null) return 1;

    if (! (obj is FileName))
      return comparer.Compare(this.fname, obj.ToString());
    else
      return comparer.Compare(this.fname, ((FileName) obj).Name);
  }
}
```

String.Equals

Default interpretation: StringComparison.Ordinal.

The String class lets you test for equality by calling either the static or instance Equals method overloads, or by using the static

equality operator. The overloads and operator use ordinal comparison by default. However, we still recommend that you call an overload that explicitly specifies the StringComparison type even if you want to perform an ordinal comparison; this makes it easier to search code for a certain string interpretation.

String.ToUpper and String.ToLower

Default interpretation: StringComparison.CurrentCulture.

You should be careful when you use these methods, because forcing a string to a uppercase or lowercase is often used as a small normalization for comparing strings regardless of case. If so, consider using a case-insensitive comparison.

The String.ToUpperInvariant and String.ToLowerInvariant methods are also available. ToUpperInvariant is the standard way to normalize case. Comparisons made using StringComparison.OrdinalIgnoreCase are behaviorally the composition of two calls: calling ToUpperInvariant on both string arguments, and doing a comparison using StringComparison.Ordinal.

Overloads are also available for converting to uppercase and lowercase in a specific culture, by passing a CultureInfo object that represents that culture to the method.

Char.ToUpper and Char.ToLower

Default interpretation: StringComparison.CurrentCulture.

These methods work similarly to the String.ToUpper and String.ToLower methods described in the previous section.

String.StartsWith and String.EndsWith

Default interpretation: StringComparison.CurrentCulture.

By default, both of these methods perform a culture-sensitive comparison.

String.IndexOf and String.LastIndexOf

Default interpretation: StringComparison.CurrentCulture.

There is a lack of consistency in how the default overloads of these methods perform comparisons. All String.IndexOf and String.LastIndexOf methods that include a Char parameter perform an ordinal comparison, but the default String.IndexOf and String.LastIndexOf methods that include a String parameter perform a culture-sensitive comparison.

If you call the String.IndexOf(String) or String.LastIndexOf(String) method and pass it a string to locate in the current instance, we recommend that you call an overload that explicitly specifies the StringComparison type. The overloads that include a Char argument do not allow you to specify a StringComparison type.

Back to top

Methods that Perform String Comparison Indirectly

Some non-string methods that have string comparison as a central operation use the StringComparer type. The StringComparer class includes six static properties that return StringComparer instances whose StringComparer.Compare methods perform the following types of string comparisons:

- Culture-sensitive string comparisons using the current culture. This StringComparer object is returned by the

StringComparer.CurrentCulture property.

- Case-insensitive comparisons using the current culture. This StringComparer object is returned by the StringComparer.CurrentCultureIgnoreCase property.
- Culture-insensitive comparisons using the word comparison rules of the invariant culture. This StringComparer object is returned by the StringComparer.InvariantCulture property.
- Case-insensitive and culture-insensitive comparisons using the word comparison rules of the invariant culture. This StringComparer object is returned by the StringComparer.InvariantCultureIgnoreCase property.
- Ordinal comparison. This StringComparer object is returned by the StringComparer.Ordinal property.
- Case-insensitive ordinal comparison. This StringComparer object is returned by the StringComparer.OrdinalIgnoreCase property.

Array.Sort and Array.BinarySearch

Default interpretation: StringComparison.CurrentCulture.

When you store any data in a collection, or read persisted data from a file or database into a collection, switching the current culture can invalidate the invariants in the collection. The Array.BinarySearch method assumes that the elements in the array to be searched are already sorted. To sort any string element in the array, the Array.Sort method calls the String.Compare method to order individual elements. Using a culture-sensitive comparer can be dangerous if the culture changes between the time that the array is sorted and its contents are searched. For example, in the following code, storage and retrieval operate on the comparer that is provided implicitly by the **Thread.CurrentThread.CurrentCulture** property. If the culture can change between the calls to StoreNames and DoesNameExist, and especially if the array contents are persisted somewhere between the two method calls, the binary search may fail.

C#

```
// Incorrect.
string []storedNames;

public void StoreNames(string [] names)
{
  int index = 0;
  storedNames = new string[names.Length];

  foreach (string name in names)
  {
    this.storedNames[index++] = name;
  }

  Array.Sort(names); // Line A.
}

public bool DoesNameExist(string name)
{
  return (Array.BinarySearch(this.storedNames, name) >= 0);  // Line B.
}
```

A recommended variation appears in the following example, which uses the same ordinal (culture-insensitive) comparison method both to sort and to search the array. The change code is reflected in the lines labeled Line A and Line B in the two examples.

C#

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
  int index = 0;
```

```
    storedNames = new string[names.Length];

    foreach (string name in names)
    {
      this.storedNames[index++] = name;
    }

    Array.Sort(names, StringComparer.Ordinal);  // Line A.
  }

  public bool DoesNameExist(string name)
  {
    return (Array.BinarySearch(this.storedNames, name, StringComparer.Ordinal) >= 0);  // Line B.
  }
```

If this data is persisted and moved across cultures, and sorting is used to present this data to the user, you might consider using StringComparison.InvariantCulture, which operates linguistically for better user output but is unaffected by changes in culture. The following example modifies the two previous examples to use the invariant culture for sorting and searching the array.

C#

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
  int index = 0;
  storedNames = new string[names.Length];

  foreach (string name in names)
  {
    this.storedNames[index++] = name;
  }

  Array.Sort(names, StringComparer.InvariantCulture);  // Line A.
}

public bool DoesNameExist(string name)
{
  return (Array.BinarySearch(this.storedNames, name, StringComparer.InvariantCulture) >= 0);  // Line B.
}
```

Collections Example: Hashtable Constructor

Hashing strings provides a second example of an operation that is affected by the way in which strings are compared.

The following example instantiates a Hashtable object by passing it the StringComparer object that is returned by the StringComparer.OrdinalIgnoreCase property. Because a class StringComparer that is derived from StringComparer implements the IEqualityComparer interface, its GetHashCode method is used to compute the hash code of strings in the hash table.

C#

```
const int initialTableCapacity = 100;
Hashtable h;

public void PopulateFileTable(string directory)
{
  h = new Hashtable(initialTableCapacity,
            StringComparer.OrdinalIgnoreCase);
```

```
    foreach (string file in Directory.GetFiles(directory))
        h.Add(file, File.GetCreationTime(file));
  }

  public void PrintCreationTime(string targetFile)
  {
    Object dt = h[targetFile];
    if (dt != null)
    {
      Console.WriteLine("File {0} was created at time {1}.",
        targetFile,
        (DateTime) dt);
    }
    else
    {
      Console.WriteLine("File {0} does not exist.", targetFile);
    }
  }
```

Back to top

Displaying and Persisting Formatted Data

When you display non-string data such as numbers and dates and times to users, format them by using the user's cultural settings. By default, the String.Format method and the **ToString** methods of the numeric types and the date and time types use the current thread culture for formatting operations. To explicitly specify that the formatting method should use the current culture, you can call an overload of a formatting method that has a *provider* parameter, such as String.Format(IFormatProvider, String, Object[]) or DateTime.ToString(IFormatProvider), and pass it the CultureInfo.CurrentCulture property.

You can persist non-string data either as binary data or as formatted data. If you choose to save it as formatted data, you should call a formatting method overload that includes a *provider* parameter and pass it the CultureInfo.InvariantCulture property. The invariant culture provides a consistent format for formatted data that is independent of culture and machine. In contrast, persisting data that is formatted by using cultures other than the invariant culture has a number of limitations:

- The data is likely to be unusable if it is retrieved on a system that has a different culture, or if the user of the current system changes the current culture and tries to retrieve the data.
- The properties of a culture on a specific computer can differ from standard values. At any time, a user can customize culture-sensitive display settings. Because of this, formatted data that is saved on a system may not be readable after the user customizes cultural settings. The portability of formatted data across computers is likely to be even more limited.
- International, regional, or national standards that govern the formatting of numbers or dates and times change over time, and these changes are incorporated into Windows operating system updates. When formatting conventions change, data that was formatted by using the previous conventions may become unreadable.

The following example illustrates the limited portability that results from using culture-sensitive formatting to persist data. The example saves an array of date and time values to a file. These are formatted by using the conventions of the English (United States) culture. After the application changes the current thread culture to French (Switzerland), it tries to read the saved values by using the formatting conventions of the current culture. The attempt to read two of the data items throws a FormatException exception, and the array of dates now contains two incorrect elements that are equal to MinValue.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;
```

```
public class Example
{
   private static string filename = @".\dates.dat";

   public static void Main()
   {
      DateTime[] dates = { new DateTime(1758, 5, 6, 21, 26, 0),
                           new DateTime(1818, 5, 5, 7, 19, 0),
                           new DateTime(1870, 4, 22, 23, 54, 0),
                           new DateTime(1890, 9, 8, 6, 47, 0),
                           new DateTime(1905, 2, 18, 15, 12, 0) };
      // Write the data to a file using the current culture.
      WriteData(dates);
      // Change the current culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-CH");
      // Read the data using the current culture.
      DateTime[] newDates = ReadData();
      foreach (var newDate in newDates)
         Console.WriteLine(newDate.ToString("g"));
   }

   private static void WriteData(DateTime[] dates)
   {
      StreamWriter sw = new StreamWriter(filename, false, Encoding.UTF8);
      for (int ctr = 0; ctr < dates.Length; ctr++) {
         sw.Write("{0}", dates[ctr].ToString("g", CultureInfo.CurrentCulture));
         if (ctr < dates.Length - 1) sw.Write("|");
      }
      sw.Close();
   }

   private static DateTime[] ReadData()
   {
      bool exceptionOccurred = false;

      // Read file contents as a single string, then split it.
      StreamReader sr = new StreamReader(filename, Encoding.UTF8);
      string output = sr.ReadToEnd();
      sr.Close();

      string[] values = output.Split( new char[] { '|' } );
      DateTime[] newDates = new DateTime[values.Length];
      for (int ctr = 0; ctr < values.Length; ctr++) {
         try {
            newDates[ctr] = DateTime.Parse(values[ctr], CultureInfo.CurrentCulture);
         }
         catch (FormatException) {
            Console.WriteLine("Failed to parse {0}", values[ctr]);
            exceptionOccurred = true;
         }
      }
      if (exceptionOccurred) Console.WriteLine();
      return newDates;
   }
}
// The example displays the following output:
//       Failed to parse 4/22/1870 11:54 PM
//       Failed to parse 2/18/1905 3:12 PM
//
//       05.06.1758 21:26
//       05.05.1818 07:19
//       01.01.0001 00:00
//       09.08.1890 06:47
```

```
//       01.01.0001 00:00
//       01.01.0001 00:00
```

However, if you replace the CultureInfo.CurrentCulture property with CultureInfo.InvariantCulture in the calls to DateTime.ToString(String, IFormatProvider) and DateTime.Parse(String, IFormatProvider), the persisted date and time data is successfully restored, as the following output shows.

```
06.05.1758 21:26
05.05.1818 07:19
22.04.1870 23:54
08.09.1890 06:47
18.02.1905 15:12
```

Back to top

See Also

Other Resources
Manipulating Strings in the .NET Framework