Advanced C# String Techniques

Topics Covered

- String Data
- The String Class
  - Immutability
  - const, literal and verbatim strings
- String Equality
- String Methods
  - Concatenation
  - Split
  - Join
  - Searches
  - Modifiers
- String Formatting
- Reading / Writing XML

String Information

A great amount of computer code is dedicated to working with information embedded in strings. In addition to the written word (from phrases to entire books), we represent many kinds of data as strings. A few examples:

- DNA sequences of G, A, T or C representing genetic information
- CSV tabular text delimited with commas
- JSON Javascript Object Notation is a string representation of objects
- XML Extensible Markup Language is a string representation of complex objects
- SMILES A string representation of chemical structures
- MathML An XML representation of mathematical expressions
- HTML A string encoding a web page.

The String Class

```
[System.Runtime.InteropServices.ComVisible(true)]
public sealed class String : ICloneable, IComparable, IComparable<string>, IConvertible,
IEquatable<string>, System.Collections.Generic.IEnumerable<char>
```

String is a sealed class – it cannot be inherited.

Strings represent a sequence of zero or more unicode (2-byte) characters.

The maximum size of a string is 2 gigabytes – about 1 billion characters.

Unlike C++, C# strings can contain embedded NULL characters.

C# strings are *immutable*.

Immutability

C# strings are immutable. Once a string object is defined, the underlying character data never changes, though the syntax can make it appear otherwise:

```
string h = "Hello ", w = "World!";
h = h + w;
```

Here, we define a string variable h and initialize it with a string literal “Hello “. We then apply the concatenation operator and assign the result back to h.

The concatenation creates a brand new string. The original “Hello” object is now unreferenced and is subject to garbage collection.

None of the methods of the string class modify the underlying character sequence. They are like const methods in C++.

Another frequently used immutable C# type is DateTime.

const Strings

const strings are string objects that cannot be reassigned.

If your code contains the same literal string repeated many times, it is better to define it in just one place as a const string.

Unlike string variables, const strings can be used as case labels in a switch statement.

```
public const string HelloWorld = "Hello World!";

static void ConstStrings()
{
  // This is an error because a const is not a variable:
  HELLOWORLD = "Wassup World?";

  // But it is a string object.
  // We can use its methods/properties:
  int length = HelloWorld.Length;
  char W = HelloWorld[6];
  string Hell = HelloWorld.Substring(0, 4);
}
```

```
public const string APlus = "A+", A = "A", B = "B", C = "C", D = "D";

static int Score(string grade)
{
  switch(grade)
  {
    case APlus: return 100;
    case A: return 95;
    case B: return 85;
    case C: return 75;
    case D: return 65;
    default:  return 0;
  }
}
```

C# String Literals

We very often initialize string variables using string literals. C# string literals are enclosed in double quotes.

```
string fileName = "preferences.txt";

// Escape sequences
string tab = "\t", crlf = "\r\n";
Console.WriteLine(tab.Length);  // 1
string backSlash = "\\", nulls = "\0\0";
string twoDoubleQuotes = "\"\"";

// Unicode
string unicodeKing = "♔", rook = "♖";
string aceOfHearts = "🂱";
string aceOfHeartsToo = "\uF0B1";
```

Some characters like tabs, line feeds and double quotes require using an escape sequence that starts with '\'. Because of its role, the backslash character requires an escape sequence.

Unicode symbols can be embedded directly into string literals, or can be represented using the "\u" escape sequence followed by the character's hex code.

Verbatim @ Strings

Literal strings are pre-processed by the compiler to convert the escapes sequences to the equivalent unicode character. For verbatim strings, this pre-processing is disabled:

```
string filePath = @"d:\Larry\Curly\Moe\PieFace.png";
string myQuote = @"She said ""Hello Cupcakes 🧁"".";
Console.WriteLine(@"\uF0B1"); //  \uF0B1
string multiLine = @"This string
contains embedded
tabs and line returns.";
```

Double quote characters must be doubled in verbatim strings.

String Equality

String is a reference type, and the default behavior of the in/equality operators is to compare references. The String class defines custom operators so that == and != compare the character sequences:

```
public static bool operator == (string a, string b);
public static bool operator != (string a, string b);
```

If we generate two strings at runtime with the same character sequence and apply the == operator to them, the result is true. If we first cast them to object and then apply the == operator, the result is false because they are different objects:

```
string s1 = "Hello " + "World!";
string s2 = "Hel";
s2 += "lo World!";
Console.WriteLine(s1 == s2);  // True
object o1 = s1, o2 = s2;
Console.WriteLine(o1 == o2);  // False
```

String.Equals()

The String class overloads the Equals method to provide comparison options:

```
public static bool Equals(String a, String b, StringComparison comparisonType);
```

The third argument is a StringComparison enum with values allowing culture-sensitive or ordinal comparisons and case-insensitive comparisons.

```
string s1 = "Hello " + "World!";
string s2 = "Hel";
s2 += "lo World!";
string s3 = s2.ToLower();
bool test = string.Equals(s1, s3, StringComparison.Ordinal);  // false
test = string.Equals(s1, s3, StringComparison.OrdinalIgnoreCase); // true
```

Some string case-changes are culture-sensitive. This is a specialized and complex topic. You can learn more about it in the String.Equals documentation.

String Concatenation

We seen already the + and += operators used to concatenate strings. These are fine for limited use, but should not be used repeatedly.

In the first example on the left, we add a single character to a string 1000 times. In the process, we are creating 1000 new string objects of growing size, all of which are discarded except for the last. This incurs a serious performance penalty in memory and processor use.

The second example shows how to accomplish the same task using a StringBuilder. The string is constructed using a single memory allocation.

```
static void BadConcatenation()
{
  string s = "";
  Random rand = new Random();
  char getChar() => (char)rand.Next(32, 256);
  // Don't do this!
  for(int i=0;i<1000;++i)
  {
    // concatenate a character
    s += getChar();
  }
  Console.WriteLine(s.Length);  // 1000

  // Do this instead
  StringBuilder builder = new StringBuilder(1000);
  for(int i=0;i<1000;++i) builder.Append(getChar());
  s = builder.ToString();
  Console.WriteLine(s.Length);  // 1000
}
```

String.Concat

The String class defines 11 overloads of Concat. The 4 shown here are so versatile that you will not likely need the others:

```
public static String Concat(params String[] values);
public static String Concat(params object[] args);
public static String Concat<T>(IEnumerable<T> values);
public static String Concat(IEnumerable<String> values);
```

The Concat methods take a variable argument list or an enumeration of values and concatenate them into a single string which is returned by the method. The concatenation is accomplished in a memory-efficient manner even when a large enumeration is provided.

Joining Strings

Another operations similar to concatenation is joining strings. String.Join lets us create delimited strings. The overloads are similar to Concat, except the first argument is always a string separator:

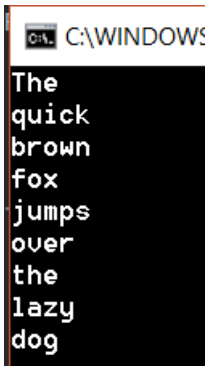
```
public static String Join<T>(String separator, IEnumerable<T> values);
public static String Join(String separator, IEnumerable<String> values);
public static String Join(String separator, params String[] value);
public static String Join(String separator, params object[] values);
```

```
IEnumerable<int> values = Enumerable.Range(1, 10);
string valueList = string.Join("+", values);
// 1+2+3+4+5+6+7+8+9+10
Console.WriteLine(string.Concat(valueList, "=", values.Sum()));
// 1+2+3+4+5+6+7+8+9+10=55
```

String.Split

String.Split allows us to break a string into fragments based on one or more delimiter characters or substrings. The delimiters are absent from the resulting substrings:

```
static void Split()
{
  const string sentence = "The quick brown fox jumps over the lazy dog.";
  string[] words = sentence.Split(' ', '.');
  foreach (string w in words) Console.WriteLine(w);
}
```



Comparing Strings

Compare methods are used for sorting. Compare(x, y) return -1 if x precedes y in the sort order, 1 if y precedes x, and 0 if they are equivalent.

As with Equals, the String.Compare overloads allow us to specify whether to consider culture and case in the comparison.

```
// lower-case characters precede upper case
int result = string.Compare("a", "A");  //-1
result = string.Compare("a", "A",
  StringComparison.OrdinalIgnoreCase);  // 0
// numbers precede letters:
result = string.Compare("a", "1");      // 1
// Shorter strings come first:
result = string.Compare("a", "aa");     //-1
result = string.Compare("A", "aA");     //-1
```

String Search Methods

The most frequently used string search method is String.IndexOf. It has 8 overloads, of which 6 are shown:

```
public int IndexOf(char value, int startIndex);
public int IndexOf(char value);
public int IndexOf(String value);
public int IndexOf(String value, int startIndex);
public int IndexOf(String value, StringComparison comparisonType);
public int IndexOf(String value, int startIndex, StringComparison comparisonType);
```

IndexOf searches a string either for a character or for a substring. If the target is found, the methods return the index of the target; otherwise they return -1. We used two of these overloads in our WordSearcher:

```
private static int FindWordCount(string document, string searchText)
{
  int count = 0;
  int ndx = document.IndexOf(searchText);
  while(ndx >= 0)
  {
    count++;
    ndx = document.IndexOf(searchText, ndx + searchText.Length);
  }
  return count;
}
```

String Modifications

These methods all return a string which is in some way modified from the original:

Method	Description
Insert(int index, string value)	Returns a new string in which a specified string is inserted at a specified index position in this instance.
PadLeft, PadRight	Returns a string to which spaces or other characters have been added to the left or right to achieve a specified total string length.
Remove	Returns a new string from which one or more characters have been removed.
Replace	Returns a string for which a character or substring of the current string is replaced with a another specified character or string.
ToLower, ToLowerInvariant ToUpper, ToUpperInvariant	Returns a string in which all characters have been converted to lower or upper case using casing rules of the current or invariant culture.
TrimStart, TrimEnd, Trim	Returns a string from which the given character(s) have been removed from the start, end, or both start/end of a string.

Formatting Strings

We often need to embed variables in an output string. The variables can be of any type – int, floating, DateTime, user-defined, etc. String.Format uses format strings that specify how a variable is converted to a string – how many decimal points for doubles, whether to use comma separators for large integers, which parts of a DateTime to render, etc.

C# Version 6 added string interpolation, which provides a more readable and convenient syntax that the Format methods.

Interpolation Syntax →
Format Syntax →

```
string c1 = "Column 1", c2 = "Column 2", c3 = "Column 3";
Console.WriteLine($"{c1}\t{c2}\t{c3}");
Console.WriteLine("{0}\t{1}\t{2}", c1, c2, c3);
```

Interpolation and Format Examples

```
int radius = 12;
Console.WriteLine($"The circle's {nameof(radius)} is {radius}, so the diameter is {(2 * radius * Math.PI):F2}.");
Console.WriteLine("The circle's {0} is {1}, so the diameter is {2:F2}.", nameof(radius), radius, 2 * radius * Math.PI);
```

```
The circle's radius is 12, so the diameter is 75.40.
```

F2 specifies two decimal places.

With interpolation, formatting instructions and variables are grouped together.
With formatting, formatting instructions and variables are grouped separately.

Formatting and Curly Braces

Both interpolation and composite formatting use curly braces to delimit the formatted expressions. You must therefore exercise care when formatting strings which contain literal curly braces. In these cases the braces must be doubled up to generate the desired output. Curly braces used without the expected formatting expression will generate a FormatException.

```
Console.WriteLine($"I need to display data ({radius}) and actual curly braces like these: {{ }}");
Console.WriteLine("I need to display data ({0}) and actual curly braces like these: {{ }}", radius);
```

```
I need to display data (12) and actual curly braces like these: { }
```

Interpolation and Format Examples

```
double v = Math.Pow(Math.E, -5);
Console.WriteLine($"e to the -5th power is {v:e4}.");
Console.WriteLine("e to the -5th power is {0:e4}.", v);
v = Math.Sqrt(10);
Console.WriteLine($"The square root of 10 is {v:F3}.");
Console.WriteLine("The square roo if 100 is {0:F3}.", v);
long l = (long)(1e9 * v);
Console.WriteLine($"Multiply this by a billion and you get {l:N0}.");
Console.WriteLine("Multiply this by a billion and you get {0:N0}.", l);
```

```
e to the -5th power is 6.7379e-003.
The square root of 10 is 3.162.
Multiply this by a billion and you get 3,162,277,660.
```

e4 : scientific notation with 4 decimal places
N0: general numeric formatting with 0 decimal places.
This also gives us comma separators.

Reading XML

XML is a versatile language, and is a natural fit to OOP: Elements correspond to classes, and attributes to properties.

Here we use XElement in the constructor of our Translations class.

```
private const string LanguageAttribute = "language";
private const string WordElement = "Word";
private List<string> _words;
public Translations(XElement source)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (source.Name != nameof(Translations)) throw new ArgumentException("Wrong element.");
  if (!source.HasAttributes || source.FirstAttribute.Name.LocalName != "language")
    throw new ArgumentException($"{nameof(source)} must have a {LanguageAttribute} first.");
  _words = new List<string>();
  Language = source.FirstAttribute.Value;
  foreach(XElement e in source.Elements(WordElement))
  {
    _words.Add(e.Value);
  }
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<TranslationSet>
  <Translations language="English">
    <Word>Yes</Word>
    <Word>No</Word>
  </Translations>
  <Translations language="French">
    <Word>Oui</Word>
    <Word>Non</Word>
  </Translations>
  <Translations language="German">
    <Word>Ja</Word>
    <Word>Nein</Word>
  </Translations>
  <Translations language="Russian">
    <Word>да</Word>
    <Word>нет</Word>
  </Translations>
</TranslationSet>
```

Loading XML from a File

XElement.Load is a factory method that creates an XElement object from a file. It has overloads that affect whitespace handling.

```
XElement xml = XElement.Load("TranslationSet.xml");
List<Translations> txls = xml.Elements().Select(e => new Translations(e)).ToList();
Console.OutputEncoding = Encoding.UTF8; // required for Cyrillic characters
for(int nWord=0;nWord<2;++nWord)
{
  foreach(Translations tx in txls)
  {
    Console.WriteLine($"{tx.Language}: {tx[nWord]}");
  }
  Console.WriteLine();
}
```

XElement also has a static Parse method that creates and XElement from an xml string.

Writing XML

We can use the XElement constructors to create a named XElement, then use the Add method to give it attributes and child elements:

```
public XElement ToXml()
{
  XElement r = new XElement(nameof(Translations));
  r.Add(new XAttribute(LanguageAttribute, Language));
  foreach(string w in _words)
  {
    r.Add(new XElement(WordElement, w));
  }
  return r;
}
```

We can then create a container element to package our Translations, then use the Save method to save the xml to a file:

```
xml = new XElement("TranslationSet");
foreach(Translations tx in txls)
{
  xml.Add(tx.ToXml());
}
xml.Save("TranslationSetCopy.xml");
```

Topics Covered

- String Data
- The String Class
  - Immutability
  - const, literal and verbatim strings
- String Equality
- String Methods
  - Concatenation
  - Split
  - Join
  - Searches
  - Modifiers
- String Formatting
- Reading / Writing XML