## **String and Text Handling: Working with Character Data**

Text manipulation is a fundamental aspect of almost every application. C# provides robust support for working with characters and strings, underpinned by the Unicode standard.

### **char: A Single Unicode Character**

In C#, the char type is an alias for the System.Char struct. It represents a single Unicode character, which is a 16-bit value.

| char c = 'A'; // A simple character char newLine = '\n'; // A special control character (newline) |
| --- |

**System.Char Static Methods:**

The System.Char struct provides a variety of useful static methods for character manipulation and categorization:

* **Case Conversion:** ToUpper(), ToLower().

| Console.WriteLine(char.ToUpper('c')); // C Console.WriteLine(char.IsWhiteSpace('\t')); // True |
| --- |

* Culture Sensitivity Caution: ToUpper() and ToLower() are culture-sensitive by default. This means their behavior can change based on the end user's locale settings. For example, in Turkish culture, char.ToUpper('i') results in 'İ' (a dotted capital I), not 'I'.  
  To avoid such locale-specific behavior, use the culture-invariant versions: ToUpperInvariant() and ToLowerInvariant(). These always follow English culture rules.

| Console.WriteLine(char.ToUpperInvariant('i')); // I |
| --- |

* Character Categorization: Methods like IsLetter(), IsDigit(), IsWhiteSpace(), IsPunctuation(), IsSymbol(), IsControl(), etc., allow you to check the type of a character.  
  For more granular categorization, GetUnicodeCategory() returns a UnicodeCategory enum, providing detailed information about a character's classification within the Unicode standard.

### **string: Immutable Sequences of Characters**

A C# string (aliasing System.String) is an **immutable** (unchangeable) sequence of char characters. Once a string object is created, its content cannot be altered. Any operation that appears to modify a string (e.g., Replace, Substring) actually returns a *new* string object.

**Constructing Strings:**

* **Literals:** The most common way.

| string s1 = "Hello"; string s2 = "First Line\r\nSecond Line"; // Escape sequences string s3 = @"C:\Path\File.txt"; // Verbatim string literal |
| --- |

* **Repeating Characters:** Using the string constructor.

| Console.Write(new string('\*', 10)); // \*\*\*\*\*\*\*\*\*\* |
| --- |

* **From char array:**

| char[] ca = "Hello".ToCharArray(); string s = new string(ca); // s = "Hello" |
| --- |

**Null vs. Empty Strings:**

* An **empty string** has a length of zero.

| string empty = ""; Console.WriteLine(empty.Length == 0); // True Console.WriteLine(empty == string.Empty); // True (string.Empty is an empty string constant) |
| --- |

* A **null string** means the string variable refers to no object.

| string nullString = null; Console.WriteLine(nullString == null); // True Console.WriteLine(nullString == ""); // False Console.WriteLine(nullString.Length); // NullReferenceException if nullString is null |
| --- |

* string.IsNullOrEmpty(string value) is a convenient static method to check if a string is either null or empty without risking a NullReferenceException.

**Accessing Characters:**

* **Indexer:** string implements an indexer to access individual characters (zero-indexed).

| string str = "abcde"; char letter = str[1]; // letter == 'b' |
| --- |

* **foreach:** string also implements IEnumerable<char>, allowing you to iterate over its characters.

| foreach (char c in "123") Console.Write(c + ","); // 1,2,3, |
| --- |

**Searching within Strings:**

* StartsWith(), EndsWith(), Contains(): Return true or false. These methods have overloads that accept a StringComparison enum to control case and culture sensitivity.
* IndexOf(): Returns the first zero-based index of a character or substring, or -1 if not found. Overloads allow specifying a startPosition and StringComparison.
* LastIndexOf(): Similar to IndexOf() but searches backward.
* IndexOfAny(): Returns the first matching position of *any* character from a specified set.

**Manipulating Strings (Returning New Strings):**

* Substring(): Extracts a portion of the string.
* Insert(), Remove(): Inserts or removes characters at a specified position.
* PadLeft(), PadRight(): Pads the string to a given length with a specified character (or space).
* TrimStart(), TrimEnd(), Trim(): Removes specified characters (defaulting to whitespace) from the start, end, or both.
* Replace(): Replaces all non-overlapping occurrences of a character or substring.
* ToUpper(), ToLower(), ToUpperInvariant(), ToLowerInvariant(): Return case-converted new strings (with the same culture sensitivity considerations as char methods).

**Splitting and Joining Strings:**

* Split(): Divides a string into a string[] based on delimiters (defaulting to whitespace). Overloads allow specifying char[] or string[] delimiters and StringSplitOptions (e.g., RemoveEmptyEntries).
* string.Join(): (Static method) Concatenates a string array into a single string, with a specified delimiter between elements.
* string.Concat(): (Static method) Concatenates multiple strings or a string array without any separator. This is internally what the + operator for strings translates to.

**String.Format() and Interpolated Strings:**

string.Format() is a powerful static method for building strings by embedding variables into a composite format string.

| string composite = "It's {0} degrees in {1} on this {2} morning"; string s = string.Format(composite, 35, "Perth", DateTime.Now.DayOfWeek); // s == "It's 35 degrees in Perth on this Thursday morning" |
| --- |

* {0}, {1} etc., are **format items** corresponding to argument positions.
* You can specify a **minimum width** ({0,-20} for left-aligned, {1,15} for right-aligned).
* You can specify a **format string** ({1,15:C} for currency).

**Interpolated String Literals (C# 6+):**

A more modern and readable alternative to string.Format(), prefixed with $. Expressions are embedded directly in braces {}.

| string s = $"It's hot this {DateTime.Now.DayOfWeek} morning"; // You can still use alignment and format strings: Console.WriteLine($"Name={ "Mary",-20} Credit Limit={500,15:C}"); |
| --- |

### **Comparing Strings**

When comparing strings, .NET distinguishes between **equality comparison** (are two strings semantically the same?) and **order comparison** (which string comes first alphabetically?).

Ordinal vs. Culture Comparison:

This is a critical distinction in string comparison:

* **Ordinal Comparison:** Interprets characters simply as their numeric Unicode values. It's fast and deterministic (always the same result regardless of locale).
  + Example: 'A' (65) comes before 'a' (97). So, "Atom" < "atom".
* **Culture-Sensitive Comparison:** Interprets characters with reference to a particular alphabet and language rules. It's necessary for correct alphabetical sorting and text processing in different languages.
  + **Current Culture:** Based on the user's operating system settings.
  + **Invariant Culture:** A culture that is the same on every computer (based on English rules), useful for consistent behavior across different environments.
  + Example: Invariant culture treats 'a' and 'A' as adjacent for sorting purposes, so "atom" might come before "Atom" in a sorted list.

**String Equality Comparison:**

* The == operator and the default string.Equals() (without StringComparison argument) perform **ordinal, case-sensitive comparison**. This is chosen for its efficiency and determinism.
* For **culture-aware or case-insensitive equality**, use overloads of string.Equals() that accept a StringComparison enum.

| public enum StringComparison {  CurrentCulture, CurrentCultureIgnoreCase,  InvariantCulture, InvariantCultureIgnoreCase,  Ordinal, OrdinalIgnoreCase } // Example: Console.WriteLine(string.Equals("foo", "FOO", StringComparison.OrdinalIgnoreCase)); // True |
| --- |

**String Order Comparison:**

* string.CompareTo() (instance method): Performs **culture-sensitive, case-sensitive order comparison**. It's used by default in sorted collections (IComparable).
* string.Compare() (static method): Allows for fine-grained control over order comparison, accepting StringComparison enum or CultureInfo objects for specific culture rules. Returns a positive number (first comes after), negative number (first comes before), or zero (same order position).
* string.CompareOrdinal() (static method): Performs ordinal order comparison.

### **StringBuilder: Mutable Strings for Efficiency**

Since string objects are immutable, repeated string manipulations (like concatenating many strings in a loop) create numerous intermediate string objects, which can be inefficient and put pressure on the garbage collector.

The System.Text.StringBuilder class provides a **mutable (editable)** string. With a StringBuilder, you can Append(), Insert(), Remove(), and Replace() substrings directly without creating new objects for each operation.

| using System.Text;  StringBuilder sb = new StringBuilder(); // Optional initial string and capacity for (int i = 0; i < 50; i++) {  sb.Append(i).Append(","); // Efficiently adds to the same StringBuilder object } Console.WriteLine(sb.ToString()); // Convert to immutable string for final result |
| --- |

* AppendLine(): Appends a string and a newline sequence.
* AppendFormat(): Accepts a composite format string, similar to String.Format().
* Length property and writable indexer are available.
* To clear a StringBuilder, set Length = 0 or create a new instance. Setting Length = 0 does **not** free the allocated memory; for that, a new StringBuilder instance is required.

### **Text Encodings and Unicode**

* **Character Set:** An allocation of characters to unique numeric codes (code points).
  + **Unicode:** The dominant character set today, encompassing virtually all spoken world languages and symbols (around one million code points).
  + **ASCII:** The first 128 characters of Unicode, representing basic English alphabet, numbers, and symbols. Older and less comprehensive but efficient (1 byte per character).
* **Text Encoding:** A mapping from a character's numeric code point to a binary representation (bytes). Encodings are crucial when reading/writing text to files, streams, or networks.

**Categories of Text Encodings in .NET:**

1. **Legacy Encodings:** Map Unicode characters to other, older character sets (e.g., EBCDIC, 8-bit code pages like Windows-1252). ASCII encoding also falls here.
2. **Standard Unicode Encoding Schemes:**
   * **UTF-8:** Most common for text files and internet. Uses 1 to 4 bytes per character. ASCII characters use 1 byte, making it ASCII-compatible. This is the **default encoding for file and stream I/O in .NET**.
   * **UTF-16:** Uses 1 or 2 16-bit words per character. This is the **internal representation for char and string in .NET**.
   * **UTF-32:** Least space-efficient; uses 4 bytes per character. Rarely used but allows easy random access.

**Obtaining an Encoding Object (System.Text.Encoding):**

* Static properties for common encodings: Encoding.UTF8, Encoding.Unicode (which is UTF-16), Encoding.UTF32, Encoding.ASCII.
* Encoding.GetEncoding(string characterSetName): For other encodings (e.g., "GB18030"). In .NET 5+ and .NET Core, you might need to register CodePagesEncodingProvider.Instance first.

**Using Encoding for File I/O:**

You use an Encoding object to specify how text is written to or read from a file or stream.

| System.IO.File.WriteAllText("data.txt", "Testing...", Encoding.Unicode); // Writes in UTF-16 |
| --- |

If the encoding is omitted, UTF-8 is the default.

**Converting to/from Byte Arrays:**

Encoding objects have GetBytes() (string to byte[]) and GetString() (byte[] to string) methods:

| byte[] utf8Bytes = Encoding.UTF8.GetBytes("0123456789"); // Length: 10 bytes byte[] utf16Bytes = Encoding.Unicode.GetBytes("0123456789"); // Length: 20 bytes string original1 = Encoding.UTF8.GetString(utf8Bytes); // "0123456789" |
| --- |