**Strings & StringBuilder**

**Strings**

* string ist ein **Referenztyp**, verhält sich aber oft wie ein Werttyp.
* string ist **immutable** (unveränderlich): Das heißt: Änderungen erzeugen **neue Objekte**.
* Strings speichern **Zeichenfolgen** und verwenden den **UTF-16-Zeichensatz**.
* Strings können direkt über string, System.String oder mit var deklariert werden:

string text1 = "Hallo";

String text2 = "Welt“;

var text3 = "Welt“;

**Wie C# Strings speichert**

C# speichert Strings in einem **internen String-Pool** (String Interning), um Speicher zu sparen.  
Das bedeutet, dass gleiche String-Werte auf dieselbe Speicheradresse verweisen:

**string a = "Hallo";**

**string b = "Hallo";**

**Console.WriteLine(object.ReferenceEquals(a, b));** // True

**ABER:** Wenn wir Strings **zur Laufzeit** erzeugen, passiert das nicht automatisch:

**string x = "Hallo";**

**string y = new string("Hallo".ToCharArray()); // Erzeugt ein neues Objekt**

**Console.WriteLine(object.ReferenceEquals(x, y));** // False

**Also:**

* Strings, **die direkt zugewiesen** werden ("Hallo"), nutzen den **String-Pool.**
* Strings, die über **new erstellt** werden, landen **nicht** im String-Pool.

**Warum sind Strings immutable?**

* **Sicherheit:** Strings werden häufig für **Passwörter, Verbindungen oder Dateioperationen** genutzt. Durch Immutabilität bleiben sie unverändert.
* **Thread-Sicherheit:** Da ein string nicht geändert werden kann, gibt es keine Thread-Synchronisationsprobleme.

**Leistungsprobleme bei string-Operationen**

Da string unveränderlich ist, führt **jede Änderung** zur Erstellung eines **neuen Objekts** im Speicher:

string text = "Hallo";

text += " Welt"; // **Neues Objekt wird erzeugt, altes bleibt im Speicher!**

text += "!";

**Problem:**

* Viele String-Änderungen → viele neue Objekte → hoher Speicherverbrauch
* Schlechte Performance, wenn Strings oft modifiziert werden

**Die Lösung ist:** **StringBuilder**

**Davor aber noch..:**

**wichtige String-Methoden**

|  |  |
| --- | --- |
| Methode | Beschreibung |
| .Length | Gibt die Länge des Strings zurück |
| .ToUpper() / .ToLower() | Wandelt den String in Groß- oder Kleinbuchstaben um |
| .Trim() | Entfernt Leerzeichen am Anfang und Ende |
| .Substring(start, len) | Gibt einen Teilstring zurück |
| .Replace("alt", "neu") | Ersetzt Zeichen oder Teilstrings |
| .Contains("text") | Prüft, ob der String eine bestimmte Zeichenfolge enthält |
| .IndexOf("text") | Gibt die Position einer Zeichenfolge zurück (-1, wenn nicht gefunden) |
| .Split(' ') | Zerlegt den String anhand eines Trennzeichens |
| .Join(";", array) | Verbindet mehrere Strings mit einem Trennzeichen |
| .StartsWith() / .EndsWith() | Prüft, ob der String mit einer bestimmten Zeichenfolge beginnt/endet |

**StringBuilder: Ein mutable String-Typ**

StringBuilder ist eine Klasse aus System.Text, die **mutable Strings** bereitstellt.  
Das bedeutet, dass Änderungen am ursprünglichen Objekt vorgenommen werden, **ohne neue Objekte zu erzeugen**.

StringBuilder sb = new StringBuilder("Hallo");

sb.Append(" Welt");

sb.Append("!");

Console.WriteLine(sb.ToString()); // Hallo Welt!

**Vorteile von StringBuilder**

* Keine neuen Objekte bei Änderungen
* Besser für **Schleifen und große Texte**
* Spart **Speicher** und verbessert **Performance**

**Wichtige Methoden von StringBuilder**

|  |  |
| --- | --- |
| Methode | Beschreibung |
| .Append("Text") | Fügt einen String ans Ende hinzu |
| .AppendLine("Text") | Fügt eine Zeile mit \n hinzu |
| .Insert(index, "Text") | Fügt einen String an einer bestimmten Position ein |
| .Remove(startIndex, length) | Entfernt Zeichen |
| .Replace("alt", "neu") | Ersetzt Zeichen oder Teilstrings |
| .ToString() | Konvertiert StringBuilder in einen string |

**Vergleich**

|  |  |  |
| --- | --- | --- |
| Merkmal | string | StringBuilder |
| Mutable? | Nein | Ja |
| Performance | Langsam bei vielen Änderungen | Schneller bei vielen Änderungen |
| Speicher | Mehr Verbrauch | Weniger Verbrauch |

**Wichtig:**

* Nutze **string**, wenn du **wenige Änderungen** vornimmst.
* Nutze **StringBuilder**, wenn du **viele Änderungen** machst.

**Wann ist StringBuilder besser?**

* Wenn du **viele Änderungen an einem String** vornimmst (z. B. += in einer Schleife).
* Wenn du mit **sehr langen Strings** arbeitest (> **1000 Zeichen**).
* Wenn du **zahlreiche Verkettungen** durchführst (z. B. in einer Schleife mit for oder while).

**Ab wann lohnt sich StringBuilder wirklich?**

Das hängt von der Anzahl der Änderungen ab:

|  |  |  |
| --- | --- | --- |
| Anzahl der Änderungen | string | StringBuilder |
| 1–5 | Schnell | Overhead |
| 5–50 | Geht noch | Effizienter |
| 50+ | Langsam (hohe Speicherbelastung) | Deutlich besser |

**Beispiel: Schleifen mit string vs. StringBuilder**

**Langsame Variante mit string (O(n²) wegen Kopien)**

Hier werden **10.000 neue Strings im Speicher erzeugt** → **sehr ineffizient!**

string result = "";

for (int i = 0; i < 10000; i++)

{

result += i + " ";

}

**Schnelle Variante mit StringBuilder (O(n))**

**Nur ein Speicherblock wird genutzt → viel schneller!**

var sb = new StringBuilder();

for (int i = 0; i < 10000; i++)

{

sb.Append(i).Append(" ");

}

string result = sb.ToString();