Informationstechnologie I - Skript zu Termin 5: Textverarbeitung

1. Grundlagen: Texte in der Programmierung

In der Informatik sind Texte fundamental eine Folge von Zeichen. Damit ein Computer diese Zeichen speichern und verarbeiten kann, müssen sie in eine numerische Form umgewandelt werden. Dieser Prozess wird als **Zeichencodierung** bezeichnet. Jede Plattform, von Ihrem Betriebssystem bis zur C#-Laufzeitumgebung, speichert Zeichen intern als Zahlen, sogenannte **Codepoints**.

Um diesen Prozess zu verstehen, müssen drei zentrale Begriffe unterschieden werden:

- **Zeichensatz (Character Set):** Ein Zeichensatz ist eine Sammlung von Zeichen, denen jeweils eine eindeutige, abstrakte Nummer zugewiesen wird. Diese Nummer nennt man **Codepoint**. Man kann sich einen Zeichensatz als ein riesiges Wörterbuch vorstellen, das jedem denkbaren Zeichen (wie 'A', '€' oder '😀') eine Nummer zuordnet. Das Unicode-Konsortium definiert den universellen Zeichensatz, der nahezu alle Schriftzeichen der Welt enthält.
- **Zeichencodierung (Character Encoding):** Eine Codierung ist das konkrete Regelwerk, das festlegt, wie die numerischen Codepoints in eine Sequenz von Bytes umgewandelt werden, damit sie in einer Datei oder im Arbeitsspeicher gespeichert werden können. Beispiele für Codierungen sind ASCII, UTF-8 und UTF-16.
- **Graphem (Grapheme):** Ein Graphem ist die kleinste semantische Einheit in einem Schriftsystem, die der Benutzer als *ein* Zeichen wahrnimmt. Ein Graphem kann aus einem oder mehreren Codepoints bestehen. Beispielsweise kann ein "ü" als einzelner Codepoint dargestellt werden oder als Kombination aus einem 'u' und einem Umlaut-Diakritikum (¨). Für den Benutzer ist beides ein einzelnes Zeichen.

2. Der historische Standard: ASCII

ASCII (American Standard Code for Information Interchange) ist einer der ältesten und grundlegendsten Zeichensätze.

- **Ursprung:** ASCII wurde ursprünglich als 7-Bit-Code entworfen, was die Darstellung von 128 (2^7) verschiedenen Zeichen ermöglicht. Diese umfassen nicht druckbare Steuerzeichen (z. B. für Zeilenumbruch), die Ziffern 0-9, die grundlegenden lateinischen Buchstaben in Groß- und Kleinschreibung sowie einige Satzzeichen.
- **Begrenzungen:** Der größte Nachteil von ASCII ist seine Beschränkung auf das englische Alphabet. Zeichen wie deutsche Umlaute (ä, ö, ü), andere europäische Schriftzeichen (z. B. é, ç) oder nicht-lateinische Schriften wie Kyrillisch oder Chinesisch sind nicht enthalten. Auch moderne Symbole wie Emojis fehlen vollständig.
- **Kompatibilität:** Trotz seiner Limitierungen ist ASCII von historischer Bedeutung und bildet die Grundlage für modernere Standards. Die ersten 128 Codepoints des Unicode-Standards (von 0 bis 127) sind identisch mit dem ASCII-Zeichensatz, was eine grundlegende Kompatibilität sicherstellt.

In C# kann man ein Zeichen direkt über seinen numerischen Wert erzeugen, was diese Zuordnung verdeutlicht:

```
// Der ASCII- und Unicode-Wert für 'A' ist 65.
// Durch das "Casting" mit (char) wird die Zahl 65
// als Zeichen interpretiert.
char buchstabe = (char)65;
Console.WriteLine(buchstabe); // Ausgabe: A
```

Heute wird reines ASCII kaum noch für allgemeine Textverarbeitung verwendet, aber es bleibt relevant für bestimmte Protokolle oder Dateiformate, die keine komplexen Zeichen benötigen.

3. Der moderne Standard: Unicode und seine Implementierung in C#

Unicode ist der globale Standard, der das Ziel verfolgt, jedem Zeichen aus jedem Schriftsystem der Welt einen eindeutigen Codepoint zuzuweisen.

- **Universelle Codepoints:** Unicode definiert eine riesige Tabelle von Codepoints. Die Notation für einen Unicode-Codepoint ist `U+` gefolgt von einer hexadezimalen Zahl, z. B. `U+0041` für 'A' oder `U+20AC` für das Euro-Symbol '€'.
- **Encodings:** Um diese Codepoints effizient zu speichern, wurden verschiedene Codierungen entwickelt:
  - **UTF-8:** Die heute im Web dominante Codierung. Sie ist variabel in der Byte-Länge: ASCII-Zeichen benötigen nur 1 Byte, während andere Zeichen 2, 3 oder 4 Bytes belegen. Dies macht sie speichereffizient für Texte, die hauptsächlich aus lateinischen Buchstaben bestehen.
  - **UTF-16:** Eine Codierung, die meist 2 Bytes (16 Bit) pro Zeichen verwendet. Sie ist der interne Standard in C# und der .NET-Plattform sowie in Betriebssystemen wie Windows.
  - **UTF-32:** Eine Codierung mit fester Länge, bei der jedes Zeichen 4 Bytes (32 Bit) belegt. Sie ist einfach zu verarbeiten, aber speicherintensiv.

Unicode in C#

C# abstrahiert die Komplexität der Codierung weitgehend, aber es ist entscheidend zu verstehen, wie es intern funktioniert:

- Der Datentyp `char` in C# repräsentiert eine **UTF-16 Codeunit**, also eine 16-Bit-Zahl (2 Bytes).
- Der Datentyp `string` ist eine unveränderliche Sequenz von `char`-Werten, also eine Folge von UTF-16 Codeunits.

Die meisten gängigen Zeichen (wie alle lateinischen, griechischen, kyrillischen Zeichen) passen in eine einzelne 16-Bit-Codeunit. Dieser Bereich wird als **Basic Multilingual Plane (BMP)** bezeichnet.

Für Zeichen mit Codepoints oberhalb der BMP (z. B. viele Emojis oder historische Schriftzeichen) reicht eine 16-Bit-Codeunit nicht aus. UTF-16 löst dieses Problem durch sogenannte **Surrogate Pairs**: Ein solches Zeichen wird durch *zwei* aufeinanderfolgende `char`-Werte repräsentiert.

```
// Das Euro-Symbol hat den Codepoint U+20AC, der in 16 Bit passt.
char euro = '\u20AC'; // '€'

// Das "grinsende Gesicht"-Emoji hat den Codepoint U+1F600.
// Dieser Wert ist größer als FFFF (hex) und passt nicht in 16 Bit.
string emoji = "😀";

Console.WriteLine(euro);      // Ausgabe: €
Console.WriteLine(emoji);     // Ausgabe: 😀

// Die Length-Eigenschaft zählt die UTF-16 Codeunits (char-Werte).
Console.WriteLine(euro.ToString().Length); // Ausgabe: 1
Console.WriteLine(emoji.Length);           // Ausgabe: 2
```

Das Beispiel zeigt, dass `emoji.Length` den Wert `2` liefert, da das Emoji als Surrogate Pair aus zwei `char`-Werten gespeichert wird. Dies ist ein entscheidender Punkt: **`string.Length` gibt nicht die Anzahl der sichtbaren Zeichen (Grapheme) zurück, sondern die Anzahl der UTF-16-Codeunits.** Für die korrekte Verarbeitung von Graphemen bietet C# die Klasse `System.Globalization.StringInfo`.

Codepoint-Notation in C#

Die `U+XXXX`-Notation ist eine plattformunabhängige Art, ein Unicode-Zeichen zu identifizieren. In C#-Code werden Unicode-Codepoints mit Escape-Sequenzen dargestellt:

- `\uXXXX`: Für Codepoints innerhalb der BMP (vier hexadezimale Ziffern).
- `\UXXXXXXXX`: Für Codepoints außerhalb der BMP (acht hexadezimale Ziffern), die Surrogate Pairs erfordern.

```
// '\u' für einen 4-stelligen Hex-Wert (BMP)
char euro = '\u20AC';         // U+20AC => '€'

// '\U' für einen 8-stelligen Hex-Wert. C# erzeugt daraus
// automatisch das korrekte Surrogate-Pair.
string grin = "\U0001F600";   // U+1F600 => 😀
```

4. Fundamentale Datentypen: `char` und `string`

Der Datentyp `char`

Der Datentyp `char` ist die kleinste Baueinheit für Text in C# und repräsentiert ein einzelnes Unicode-Zeichen (genauer: eine UTF-16-Codeunit). Literale dieses Typs werden in **einfachen Anführungszeichen** (`'`) deklariert.

```
char buchstabe = 'A';
char ziffer = '9';
char euroSymbol = '€';
```

Ein `char` kann Buchstaben, Ziffern, Satzzeichen oder Sonderzeichen enthalten.

Der Datentyp `string`

Der Datentyp `string` repräsentiert eine Sequenz von `char`-Werten. String-Literale werden in **doppelten Anführungszeichen** (`"`) deklariert. Er ist der zentrale Datentyp für die Arbeit mit Texten.

```
string gruss = "Hallo Welt";
string einzelnesZeichenAlsString = "A"; // Nicht zu verwechseln mit char 'A'
string leererString = ""; // Ein String kann auch null Zeichen enthalten.
```

5. Das Konzept der Unveränderlichkeit (Immutability)

Ein `string` in C# ist **unveränderlich (immutable)**. Das bedeutet, dass ein einmal erstelltes String-Objekt in seinem Zustand nicht mehr modifiziert werden kann. Jede Operation, die einen String scheinbar ändert (wie das Umwandeln in Großbuchstaben oder das Anhängen von Text), erzeugt im Hintergrund ein **neues** String-Objekt mit dem geänderten Inhalt. Das ursprüngliche Objekt bleibt unberührt.

```
string text = "Hallo";

// Die Methode ToUpper() erzeugt den neuen String "HALLO".
// Da das Ergebnis keiner Variable zugewiesen wird, geht es sofort verloren.
text.ToUpper();
Console.WriteLine(text); // Gibt weiterhin "Hallo" aus.

// Hier wird das neue Objekt in einer neuen Variable gespeichert.
string neuerText = text.ToUpper();
Console.WriteLine(neuerText); // Gibt "HALLO" aus.
```

Warum ist das so? Die Unveränderlichkeit bietet mehrere Vorteile:

- **Vorhersagbarkeit und Sicherheit:** Wenn Sie einen String an eine Methode übergeben, können Sie sicher sein, dass diese Methode den ursprünglichen String nicht verändern kann.
- **Effizienz:** Mehrere Variablen können auf dasselbe String-Objekt im Speicher verweisen, ohne dass die Gefahr besteht, dass eine Variable den Wert für eine andere "mitändert".

Wegen der Immutability gilt als Faustregel: **Das Ergebnis einer String-Methode muss fast immer einer neuen oder derselben Variable zugewiesen werden.**

6. Eigenschaften und Zugriff

Die `Length`-Eigenschaft und der Index-Zugriff `[]`

Die `Length`-Eigenschaft eines Strings gibt die Gesamtzahl der `char`-Werte (UTF-16-Codeunits) an. Der Zugriff auf einzelne Zeichen erfolgt über einen **nullbasierten Index** mit eckigen Klammern `[]`. Das bedeutet, das erste Zeichen hat den Index 0, das zweite den Index 1 und so weiter.

```
string text = "C#-Code"; // text.Length ist 7
// Index:   0123456

// Zugriff auf das erste Zeichen
char erstesZeichen = text;              // Ergibt 'C'

// Zugriff auf das letzte Zeichen
char letztesZeichen = text[text.Length - 1]; // Ergibt 'e'
```

Achtung: Der Versuch, auf einen Index zuzugreifen, der außerhalb des gültigen Bereichs liegt (z. B. kleiner als 0 oder größer/gleich `Length`), führt zu einer `IndexOutOfRangeException` und beendet das Programm mit einem Fehler. Das Muster `text.Length - 1` für den Zugriff auf das letzte Element ist ein universelles und wichtiges Konzept in der Programmierung.

7. Iteration über Strings

Es gibt zwei primäre Methoden, um die Zeichen eines Strings zu durchlaufen:

Iteration mit `foreach`

Die `foreach`-Schleife ist der einfachste und lesbarste Weg, jedes Zeichen eines Strings zu verarbeiten, wenn die Position (der Index) des Zeichens nicht von Bedeutung ist.

```
string text = "Hallo";

// Die Schleife durchläuft jedes Zeichen in 'text'.
// In jeder Iteration enthält die Variable 'buchstabe'
// das nächste Zeichen.
foreach (char buchstabe in text)
{
    Console.WriteLine(buchstabe);
}
```

Diese Schleife ist weniger fehleranfällig als eine manuelle Index-Verwaltung und sollte bevorzugt werden, wenn nur der Wert der Zeichen benötigt wird.

Iteration mit `for`

Eine `for`-Schleife bietet die volle Kontrolle über den Iterationsprozess, da sie direkt mit dem Index arbeitet. Sie ist notwendig, wenn die Position eines Zeichens für die Programmlogik relevant ist.

```
string text = "C#-Code";

// Die Schleife läuft, solange der Index i kleiner als die Länge des Strings ist.
for (int i = 0; i < text.Length; i++)
{
    char aktuellesZeichen = text[i];
    Console.WriteLine($"Index {i}: {aktuellesZeichen}");
}
```

Diese Form der Iteration gibt die volle Kontrolle über Start, Ende und Schrittweite und ist unverzichtbar, wenn der Index des Zeichens für die Logik benötigt wird, z. B. um Zeichen von vorne und hinten zu vergleichen.

8. String-Formatierung

Das Zusammenfügen von Strings und Variablen ist eine alltägliche Aufgabe.

Konkatenation mit dem `+`-Operator

Der `+`-Operator kann verwendet werden, um zwei oder mehr Strings zu verketten (aneinanderzufügen).

```
string vorname = "Max";
string nachname = "Mustermann";

// Mehrere Strings werden zu einem neuen String zusammengefügt.
string vollerName = vorname + " " + nachname; // Ergebnis: "Max Mustermann"

Console.WriteLine("Benutzer: " + vollerName);
```

Diese Methode ist einfach und intuitiv für wenige Verknüpfungen. Bei vielen Operationen, insbesondere in Schleifen, kann sie jedoch ineffizient sein, da bei jeder Konkatenation ein neues String-Objekt erzeugt wird.

String-Interpolation mit `$`

String-Interpolation ist der moderne, lesbare und empfohlene Ansatz in C#. Ein String-Literal wird mit einem `$`-Zeichen eingeleitet. Variablen oder Ausdrücke können dann direkt in geschweiften Klammern `{}` in den String eingebettet werden.

```
string name = "Anna";
int alter = 28;

// Die Werte von 'name' und 'alter' werden direkt in den String eingesetzt.
string gruss = $"Name: {name}, Alter: {alter}.";

Console.WriteLine(gruss); // Ausgabe: Name: Anna, Alter: 28.
```

Interpolation erhöht die Lesbarkeit des Codes erheblich, da die finale Struktur des Strings direkt im Code sichtbar ist. Sie sollte als Standardmethode zur String-Formatierung verwendet werden.

9. Wichtige String-Methoden

Die `string`-Klasse in C# bietet eine Vielzahl nützlicher Methoden zur Bearbeitung und Analyse von Texten. Hier sind einige der wichtigsten:

`ToUpper()` / `ToLower()`

Diese Methoden geben einen **neuen** String zurück, in dem alle Buchstaben in Groß- bzw. Kleinbuchstaben umgewandelt wurden. Das Original-Objekt bleibt unverändert.

```
string original = "MischText";
string gross = original.ToUpper(); // Ergibt "MISCHTEXT"
string klein = original.ToLower(); // Ergibt "mischtext"
```

Diese Methoden sind besonders nützlich für Vergleiche, bei denen die Groß- und Kleinschreibung ignoriert werden soll.

`Trim()`

Die `Trim()`-Methode gibt einen neuen String zurück, bei dem alle führenden und nachgestellten Whitespace-Zeichen (Leerzeichen, Tabulatoren, Zeilenumbrüche) entfernt wurden.

```
string userInput = "   Wichtige Eingabe   ";
string bereinigt = userInput.Trim(); // Ergibt "Wichtige Eingabe"
```

Dies ist ein Standardverfahren zur Bereinigung von Benutzereingaben, um Fehler bei Vergleichen oder der Weiterverarbeitung zu vermeiden, die durch unsichtbare Zeichen entstehen können.

`IndexOf()`

Diese Methode sucht nach dem ersten Vorkommen eines Zeichens oder einer Zeichenkette innerhalb eines Strings und gibt dessen nullbasierten Startindex zurück. Wenn der gesuchte Text nicht gefunden wird, gibt die Methode `-1` zurück.

```
string text = "Hallo Welt, schöne Welt!";
int position1 = text.IndexOf("Welt");  // Ergibt 6 (der Startindex von "Welt")
int position2 = text.IndexOf("Erde");  // Ergibt -1 (da "Erde" nicht enthalten ist)
```

`IndexOf` ist fundamental, um zu prüfen, ob ein Text einen anderen enthält, und um dessen Position zu bestimmen.

`Substring()`

`Substring()` extrahiert einen Teilstring. Es gibt zwei Überladungen:

1. `Substring(int startIndex)`: Extrahiert den Teil vom `startIndex` bis zum Ende des Strings.
2. `Substring(int startIndex, int length)`: Extrahiert einen Teil mit einer bestimmten `length` ab dem `startIndex`.

```
string text = "Programmieren";

// 1. Ab Startindex 8 bis zum Ende
string teil1 = text.Substring(8); // Ergibt "ieren"

// 2. Ab Startindex 0 mit einer Länge von 4 Zeichen
string teil2 = text.Substring(0, 4); // Ergibt "Prog"
```

Diese Methode wird oft in Kombination mit `IndexOf` verwendet, um dynamisch Textteile basierend auf Trennzeichen oder Schlüsselwörtern zu extrahieren.

`Replace()`

Diese Methode gibt einen neuen String zurück, in dem alle Vorkommen eines bestimmten Zeichens oder einer Zeichenkette durch eine andere ersetzt wurden.

```
string satz = "Ein guter Tag ist ein schöner Tag.";
string neuerSatz = satz.Replace("Tag", "Moment");
// Ergibt: "Ein guter Moment ist ein schöner Moment."
```

Auch hier ist es wichtig, sich an die Unveränderlichkeit zu erinnern: `Replace` modifiziert nicht den ursprünglichen String, sondern gibt eine neue, modifizierte Kopie zurück. Das Ergebnis muss also einer Variablen zugewiesen werden.