**Einführung in Tasks, async und await**

**Synchron vs. Asynchron: Unterschied zwischen synchroner und asynchroner Programmierung.**

**Synchrone Programmierung**

In der synchronen Programmierung werden Befehle und Anweisungen **sequenziell ausgeführt**. Das bedeutet, dass eine Anweisung erst dann abgeschlossen sein muss, bevor die nächste ausgeführt wird. Dies kann zu Wartezeiten führen, insbesondere wenn eine Operation zeitintensiv ist (z. B. das Laden einer Datei, das Abrufen von Daten aus einer Datenbank oder eine HTTP-Anfrage).

**Merkmale der synchronen Programmierung:**

* Der Programmablauf ist **linear und vorhersehbar**.
* Langsame oder blockierende Operationen **halten** den gesamten Ablauf **auf**.
* Threads\* bleiben blockiert, während sie auf das Ergebnis einer Operation warten.

*\* Ein* ***Thread*** *ist die kleinste ausführbare Einheit innerhalb eines Prozesses.*

**Beispiel für eine synchrone Methode in C#:**

public static void BeispielSync()

{

Console.WriteLine("Start");

Thread.Sleep(5000); // 5 Sekunden warten

Console.WriteLine("Ende");

}

Hier wird Thread.Sleep(5000) verwendet, um eine Verzögerung von 5 Sekunden zu simulieren. Während dieser Zeit kann das Programm **keine anderen Aufgaben** ausführen.

**Asynchrone Programmierung**

In der asynchronen Programmierung können Aufgaben **parallel** ausgeführt werden, sodass eine blockierende Operation **den Hauptthread nicht anhält**. Anstatt auf eine lange laufende Operation zu warten, kann das Programm andere Aufgaben ausführen und **später** das Ergebnis der asynchronen Operation abholen.

**Merkmale der asynchronen Programmierung:**

* Der Hauptthread bleibt **nicht blockiert**, sondern kann weiterarbeiten.
* Aufgaben können parallel ausgeführt werden, was die **Effizienz** steigert.
* Asynchronität verbessert die **Reaktionsfähigkeit** von Programmen, insbesondere bei Windows-Anwendungen oder serverseitigen **Webanwendungen**.

**Beispiel für eine asynchrone Methode in C#:**

public static async Task LadeDatenASynchron()

{

Console.WriteLine("Daten werden asynchron geladen...");

await Task.Delay(3000); // Simuliert eine asynchrone Operation

Console.WriteLine("Daten wurden geladen.");

}

Hier wird await Task.Delay(3000) verwendet, um eine **nicht blockierende Verzögerung** zu simulieren. Das Programm kann in der Zwischenzeit weiterarbeiten, anstatt zu warten.

**Was ist ein „Task“?**

Ein Task in C# stellt eine **asynchrone Operation** dar, die in **einem separaten Thread** oder als nicht blockierende Hintergrundaufgabe ausgeführt werden kann.

**Eigenschaften von Tasks:**

* Tasks repräsentieren eine **laufende** oder **abgeschlossene** asynchrone Operation.
* Sie können Zustände wie "**Wartend**", "**Laufend**" oder "**Abgeschlossen**" haben.
* Sie können Ergebnisse zurückgeben (**Task<T>)** oder einfach nur eine Aufgabe ausführen (**Task**).
* Sie können mit **await** verwendet werden, um auf ihre Fertigstellung **zu warten**.

**Erstellen und Starten eines Tasks:**

public static void EinfachesTaskBeispiel()

{

Task task = Task.Run(() =>

{

Console.WriteLine("Task startet...");

Task.Delay(5000).Wait(); // simuliert 5 Sekunden Wartezeit

Console.WriteLine("Task beendet.");

});

task.Wait();

}

Hier wird **Task.Run()** verwendet, um eine Methode in einem separaten Thread auszuführen.

**Rückgabewerte mit Task<T>:**

public static Task<int> BerechneAsync()

{

Task<int> BerechneAsync()

{

return Task.Run(() =>

{

Thread.Sleep(2000); // Simuliert eine Berechnung

return 42; // Gibt ein Ergebnis zurück

});

}

return BerechneAsync();

}

Diese Methode gibt einen Task<int> zurück, der asynchron eine Zahl berechnet.

**Schlüsselwörter „async“ und „await“**

**async** markiert eine Methode als **asynchron**

Das Schlüsselwort async wird verwendet, um eine Methode als asynchron zu kennzeichnen. Eine async-Methode kann **Task** oder **Task<T>** zurückgeben und innerhalb der Methode await verwenden.

async Task MeineAsynchroneMethode()

{

Console.WriteLine("Starte asynchrone Operation...");

await Task.Delay(2000); // Simuliert eine asynchrone Verzögerung

Console.WriteLine("Asynchrone Operation abgeschlossen.");

}

**Regeln für async-Methoden:**

* Eine async-Methode **muss** entweder **void**, Task oder **Task<T>** als Rückgabetyp haben.
* Innerhalb einer async-Methode kann await verwendet werden.
* Das async-Schlüsselwort selbst macht eine Methode **nicht** **automatisch asynchron** – sie muss asynchrone Operationen enthalten.

**await wartet auf die Fertigstellung eines Tasks, ohne den aktuellen Thread zu blockieren**

Das Schlüsselwort **await** wird verwendet, um auf die Fertigstellung eines Tasks zu warten. Es **pausiert** die aktuelle Methode**, gibt aber den Thread frei,** sodass andere Operationen weiterlaufen können.

**Beispiel:**

public static async Task EinfachesTaskBeispielASync()

{

// Einfaches async/await-Beispiel

Console.WriteLine("Start");

await WarteAsync();

Console.WriteLine("Ende");

static async Task WarteAsync()

{

Console.WriteLine("Warten beginnt");

await Task.Delay(3000); // Warten ohne Blockieren des Threads

Console.WriteLine("Warten beendet");

}

}

Hier sorgt await Task.Delay(3000) dafür, dass die Methode für 3 Sekunden pausiert, aber der Thread nicht blockiert wird.

**Vorteile von await:**

* Ermöglicht nicht-blockierende Wartezeiten.
* Verhindert, dass die Anwendung während lang andauernder Operationen einfriert.
* Erhöht die Performance und Benutzerfreundlichkeit.

|  |  |
| --- | --- |
| Konzept | Erklärung |
| Synchrone Programmierung | Befehle werden **nacheinander** ausgeführt, blockierende Operationen können den Ablauf verzögern. |
| Asynchrone Programmierung | Lässt Operationen im Hintergrund laufen, ohne den Hauptthread zu blockieren. |
| Task | Repräsentiert eine asynchrone Operation, kann einen Wert zurückgeben (Task<T>). |
| async | Markiert eine Methode als **asynchron**, ermöglicht die **Nutzung von await.** |
| await | Wartet auf die Fertigstellung eines Tasks, ohne den Thread zu blockieren. |

**Übung**

Erstellen Sie eine Methode, die zwei asynchrone Methoden aufruft, bei der eine 5 Sekunden und die andere 2 Sekunden wartet.

**Gemeinsame Übung**

Das "Frühstücks-Beispiel" ist ein bekanntes und anschauliches Anwendungsbeispiel, um synchrone und asynchrone Programmierung zu erklären. Es verdeutlicht, wie man mehrere Aufgaben gleichzeitig ausführen kann, anstatt sie nacheinander abzuarbeiten.

**Das Grundprinzip des Frühstücks-Beispiels**

Stellen Sie sich vor, Sie bereiten ein Frühstück vor. Eine synchrone Vorgehensweise würde so aussehen:

1. Kaffee kochen (dauert 5 Minuten)
2. Eierspeise machen (dauert 7 Minuten)
3. Toast toasten (dauert 3 Minuten)
4. Frühstück servieren

Insgesamt würde das Frühstück **15 Minuten** dauern (5 + 7 + 3 Minuten), weil alles nacheinander erledigt wird.

In der **asynchronen Version** bereiten Sie die Aufgaben so vor, dass alle gleichzeitig starten können:

1. Starten Sie den Kaffeekocher.
2. Während der Kaffee läuft, beginnen Sie, die Eierspeise zu machen
3. Starten Sie den Toaster.
4. Warten Sie, bis alle Aufgaben (Kaffee, Eierspeise, Toast) abgeschlossen sind.

Durch diese Vorgehensweise ist die Zeit, die für die Zubereitung des Frühstücks benötigt wird, nur so lang wie die längste einzelne Aufgabe (in diesem Fall 7 Minuten für die Eierspeise) — statt 15 Minuten.

**Erklärung des Codes**

**Paralleles Starten der Aufgaben**

Task kaffeeTask = KaffeeKochenAsync();

Task eierTask = EierspeiseAsync();

Task toastTask = ToastToastenAsync();

Alle drei Aufgaben werden gleichzeitig gestartet, da sie nicht blockieren.

**Task.WhenAll()**

await Task.WhenAll(kaffeeTask, eierTask, toastTask);

Das Programm wartet, bis **alle Aufgaben abgeschlossen sind**, bevor es mit der nächsten Zeile fortfährt.

**Asynchrone Methoden**

static async Task KaffeeKochenAsync()

{

Console.WriteLine("Kaffee kochen gestartet...");

await Task.Delay(5000); // simuliert 5 Sekunden Wartezeit

Console.WriteLine("Kaffee ist fertig!");

1. }

Jede Methode verwendet **await Task.Delay(),** um die **Zubereitungszeit zu simulieren.** Das Blockieren des Threads wird vermieden, und andere Aufgaben können währenddessen weiterlaufen.

**Nützliche Methoden und Klassen**

|  |  |
| --- | --- |
| Methode/Schlüsselwort | Beschreibung |
| Task.Delay(int ms) | Simuliert eine **asynchrone** Verzögerung. |
| HttpClient.GetStringAsync(url) | Sendet eine asynchrone Web-Anfrage. |
| Task.WhenAll(tasks) | Wartet auf mehrere Tasks. |
| Task.WhenAny(tasks) | Wartet **auf den ersten** abgeschlossenen Task. |