C# Fortgeschrittenenschulung

Asynchrone und Parallele Entwicklung & LINQ

Asynchrone Entwicklung in C#

Warum asynchron?

- **Reaktionsfähigkeit erhöhen:**
  - Benutzeroberflächenblockaden vermeiden.
- **Effiziente Ressourcennutzung:**
  - Warten auf E/A-Operationen minimieren.
- **Konzepte:**
  - `async` und `await` für asynchrone Methoden
  - Verwendung von `Task` zur Rückgabe von Ergebnissen

Grundlagen von async und await

```
public async Task<string> FetchDataAsync()
{
    await Task.Delay(1000); // Simuliert einen Netzwerkaufruf
    return "Data fetched";
}

public async Task CallAsyncMethod()
{
    string data = await FetchDataAsync();
    Console.WriteLine(data);
}
```

- `async`: Markiert eine Methode als asynchron.
- `await`: Wartet auf das Ende einer asynchronen Operation.

Wann `await` verwenden und wann nicht

Wann `await` verwenden?

- **Lange laufende I/O-Operationen:**
  - Netzwerkzugriffe, Dateisystemoperationen, Datenbankabfragen.
  - Beispiel: `await httpClient.GetStringAsync(url);`
- **Vermeidung von UI-Blockaden:**
  - Sicherstellen, dass UI-Anwendungen reaktionsfähig bleiben.
  - Beispiel: Daten während der GUI-Verarbeitung abrufen.
- **Parallele Aufrufe:**
  - Mehrere unabhängige Operationen, die gleichzeitig laufen können.
  - Beispiel: `await Task.WhenAll(task1, task2);`

Wann `await` vermeiden?

- **Kurze rechenintensive Aufgaben:**
  - CPU-Arithmetik oder kleine datenverarbeitende Methoden.
  - Beispiel: Algorithmen innerhalb einer tight-loop.
- **Wenn Synchronität erforderlich ist:**
  - Reihenfolge von Berechnungen, die von früheren Ergebnissen abhängen.
  - Beispiel: Aufbau einer Datenstruktur.
- **Verarbeitung von In-Memory-Daten:**
  - Operationen, die sofortige Ergebnisse über kleine Datensätze liefern.
  - Beispiel: LINQ über kleine Listen.

Wichtige Überlegungen

- **Vermeidung von Overhead:**
  - Asynchrone Methoden haben einen gewissen Overhead; also nur verwenden, wenn nötig.
- **Fehlerbehandlung:**
  - Ausnahmen genau managen und loggen.

Verwalten mehrerer asynchroner Ausführungen

Warum mehrere asynchrone Aufgaben?

- **Parallelität maximieren:**
  - Mehrere unabhängige asynchrone Operationen effizient gleichzeitig durchführen.
- **Wartezeit minimieren:**
  - Gesamtwartezeit reduzieren, indem auf alle Tasks gleichzeitig gewartet wird.

Verwendung von Task.WhenAll

- **Zweck:**
  - Warte, bis alle bereitgestellten Tasks abgeschlossen sind.

```
Task task1 = DoWorkAsync();
Task task2 = DoOtherWorkAsync();

await Task.WhenAll(task1, task2);
```

- **Vorteile:**
  - Setzt den Code nach Abschluss aller Tasks fort.
  - Unterstützt die asynchrone Weiterverarbeitung nach Beendigung.

Verwendung von Task.WaitAll

- **Zweck:**
  - Blockiert den aufrufenden Thread, bis alle bereitgestellten Tasks beendet sind.

```
Task task1 = Task.Run(() => DoWork());
Task task2 = Task.Run(() => DoOtherWork());

Task.WaitAll(task1, task2);
```

- **Verwendungskontext:**
  - Verwendet in nicht-asynchronen Methoden, um synchron auf die Beendigung aller Tasks zu warten.

Unterschiede zwischen Task.WhenAll und Task.WaitAll

- **Task.WhenAll:**
  - Asynchrone Verarbeitung
  - Nicht blockierend; weiterverarbeitender Code kann noch andere asynchrone Aufrufe verwenden.
- **Task.WaitAll:**
  - Synchrone Verarbeitung
  - Blockiert den aktuellen Thread; nicht empfehlenswert für UI-Threads.

Beispielanwendung

- **Fehlerbehandlung:**
  - Bei `Task.WhenAll`: Sammle alle Ausnahmen und arbeite weiterhin asynchron.
  - Bei `Task.WaitAll`: Behandle Ausnahmen beim in den Funktionen.

```
try
{
    await Task.WhenAll(task1, task2);
}
catch (Exception ex)
{
    Console.WriteLine($"Exception handled: {ex.Message}");
}
```

Einsatzempfehlungen

- **Verwenden Sie `Task.WhenAll` für:**
  - Asynchrone Methoden, effiziente Wartezeit.
  - Arbeiten in Umgebungen, die nicht UI-blockierend sind.
- **Verwenden Sie `Task.WaitAll` für:**
  - Ausschließlich in nicht-asynchronen Kontexten/Methode, wo Blockierung akzeptabel ist.

Parallele Entwicklung in C#

Warum parallel?

- **Beschleunigung durch Parallelisierung:**
  - Mehrere Prozessoren/Kerne verwenden, um Aufgaben gleichzeitig zu verarbeiten.

Bedeutung von Parallelität

- **Definition:**
  - Gleichzeitige Ausführung von Aufgaben, um die Rechenressourcen effizienter zu nutzen.
- **Vorteile:**
  - Erhöht die Anwendungsleistung durch parallele Daten- und Aufgabenverarbeitung.

Grundlagen der Parallel-Klasse

- **Namespace:**
  - `System.Threading.Tasks`
- **Primäre Methoden:**
  - `Parallel.For`
  - `Parallel.ForEach`

Verwendung von Parallel.For

```
Parallel.For(0, 10, i =>
{
    Console.WriteLine($"Index {i} processed by Thread {Thread.CurrentThread.ManagedThreadId}");
});
```

- **Vorteile:**
  - Ermöglicht die iterative, parallele Ausführung von Schleifen, um umfangreiche Aufgaben zu beschleunigen.

Verwendung von Parallel.ForEach

```
Parallel.ForEach(myCollection, item =>
{
    Console.WriteLine($"Processing {item}");
});
```

- **Anwendung:**
  - Nützlich bei der parallelen Verarbeitung von Sammlungen und Listen.
- **Leistungssteigerung:**
  - Bessere Ressourcennutzung bei vielfältigen Aufgaben.

Teilkreismuster und parallele Verarbeitung

- **Teilkreismuster:**
  - Aufteilung von Aufgaben in kleinere Pakete, die parallel bearbeitet werden.
- **Vorteile:**
  - Bessere Lastverteilung und Nutzung von Multikernsystemen.

Einführung in LINQ

Was ist LINQ?

- **Language Integrated Query:**
  - Bietet eine einheitliche Methode zur Arbeit mit Datenquellen.
- **Vorteile:**
  - Starke Typprüfung
  - Lesbarkeit und Wartbarkeit

LINQ-Syntaxarten

- Query-Expression-Syntax
- Methoden-basierte Syntax

Beispiele für LINQ-Abfragen

Query-Expression-Syntax

```
var results =
    from num in numbers
    where num > 10
    select num;
```

Methoden-basierte Syntax

```
var results = numbers.Where(num => num > 10);
```

- **Filterung:** `Where`
- **Projektionen:** `Select`
- **Aggregationen:** `Sum`, `Average`

Filtern: `Where`

Query-Syntax

```
var query = from num in numbers
            where num > 5
            select num;
```

Methoden-Syntax

```
var query = numbers.Where(num => num > 5);
```

- **Verwendung:**
  - Auswahl von Elementen basierend auf externen Kriterien.

Projektion: `Select`

Query-Syntax

```
var query = from num in numbers
            select num * num;
```

Methoden-Syntax

```
var query = numbers.Select(num => num * num);
```

- **Verwendung:**
  - Neue Formate oder Datenformen aus bestehenden Elementen erstellen.

Sortieren: `OrderBy` und `OrderByDescending`

Query-Syntax

```
var query = from num in numbers
            orderby num
            select num;
```

Methoden-Syntax

```
var query = numbers.OrderBy(num => num);
```

- **Verwendung:**
  - Sortieren von Elementen in auf- oder absteigender Reihenfolge.

Gruppierung: `GroupBy`

Query-Syntax

```
var query = from s in students
            group s by s.Grade;
```

Methoden-Syntax

```
var query = students.GroupBy(s => s.Grade);
```

- **Verwendung:**
  - Zusammenfassen von Elementen, die gemeinsame Attribute teilen.

Aggregation: `Sum`, `Average`

Query-Syntax

(nicht direkt unterstützt)

Methoden-Syntax

```
var total = numbers.Sum();
var average = numbers.Average();
```

- **Verwendung:**
  - Berechnung von Summen, Durchschnitten und anderen Aggregaten.

Mengenoperationen: `Distinct`

Query-Syntax

(nicht direkt unterstützt)

Methoden-Syntax

```
var distinctNumbers = numbers.Distinct();
```

- **Verwendung:**
  - Entfernt doppelte Einträge aus den Ergebnissen.

Verknüpfung: `Join`

Query-Syntax

```
var query = from p in people
            join c in cars on p.Id equals c.OwnerId
            select new { p.Name, c.Model };
```

Methoden-Syntax

```
var query = people.Join(cars, p => p.Id, c => c.OwnerId,
            (p, c) => new { p.Name, c.Model });
```

- **Verwendung:**
  - Verknüpfen von Datensätzen aus zwei Sammlungen mit einem gemeinsamen Schlüssel.

Kombinieren: `Concat`

Query-Syntax

(nicht unterstützt)

Methoden-Syntax

```
var combined = list1.Concat(list2);
```

- **Verwendung:**
  - Zusammenführen von zwei oder mehr Sequenzen.

LINQ und PLINQ

- **PLINQ:**
  - Parallele Ausführung von LINQ-Abfragen
  - Nutzung der Systemressourcen effizienter

```
var parallelQuery = numbers.AsParallel().Where(n => n > 10);
```

Wichtige Überlegungen bei PLINQ

Threadsicherheit

- **Gemeinsame Ressourcen:**
  - Synchronisation sicherstellen oder vermeiden.
- **Seiteneffekte vermeiden:**
  - Operations in PLINQ sollten keine gemeinsam genutzten Daten ändern.

Leistung

- **Grenzen der Parallelität:**
  - Bei kleinen Datensätzen kann der Overhead die Vorteile aufheben.
- **Partitionierung:**
  - Standardmäßig automatisch, überlegende Anpassung mit `WithDegreeOfParallelism`.

Fehlerbehandlung und Reihenfolge

Exception Handling

- **AggregateException:**
  - PLINQ versteckt Ausnahmen; Behandlung implementieren.

Ordnung und Reihenfolge

- **Optionen zur Reihenfolge:**
  - `AsOrdered()` und `AsUnordered()` zur Kontrolle der Verarbeitung.

Parallelisierbarkeit

- **Eignung:**
  - Nicht alle Abfragen sind zur Parallelisierung geeignet, vor allem, wenn Reihenfolgen benötigt werden.

Praktische Übungen

Anstehende Übungen

- **Asynchrone Methoden implementieren:**
  - Webanfragen asynchron verarbeiten.
- **Parallelisierung nutzen:**
  - Datensätze mit Parallel.ForEach verarbeiten.
- **Komplexe LINQ-Abfragen erstellen:**
  - Arbeiten mit Datenstrukturen und LINQ.