C# Grundlagen

Tag 6 -Grundlagen der Fehlerbehandlung und Debugging-Techniken

Agenda

- 09:00 - 09:45: Exception Handling
- 09:45 - 10:15: Best Practices und häufige Fallen
- 10:15 - 10:30: Pause
- 10:25 - 11:00: Debugging-Techniken
- 11:00 - 12:45: Übung & Abschlussprojekt
- 12:45 - 13:00: Zusammenfassung und Ausblick auf die Fortgeschrittenen-Schulung

Exception Handling in C#

Einführung

- **Definition:** Mechanismus zur Behandlung von Laufzeitfehlern.
- **Ziel:** Programmfluss bei Fehlern kontrolliert fortsetzen oder beenden.

Schlüsselkonzepte

- **Exceptions:** Objekte, die Fehlerzustände repräsentieren und Informationen über die Art des Fehlers bereitstellen.
- **Try-Catch-Blöcke:** Struktur, die potenziell fehlerverursachenden Code (Try) und Fehlerbehandlung (Catch) umgibt.
- **Finally-Block:** Optional; wird immer ausgeführt, um Bereinigungsaktionen durchzuführen.

Try-Catch-Finally Blöcke

```
try
{
    // Code der eine Exception werfen könnte
    var result = someOperation();
}
catch (SpecificException ex)
{
    // Behandlung spezifischer Exceptions
    Logger.Log(ex);
}
finally
{
    // Wird immer ausgeführt
    CleanupResources();
}
```

Exception-Hierarchie in C#

- `System.Exception` (Basisklasse)
  - `SystemException`
    - `ArgumentException`
    - `NullReferenceException`
    - `InvalidOperationException`
  - `ApplicationException`
    - Custom Exceptions

Common Exception Types

- `ArgumentException` / `ArgumentNullException`
  - Ungültige Methodenparameter
- `InvalidOperationException`
  - Ungültiger Objektzustand
- `FileNotFoundException`
  - Dateizugriffsfehler
- `FormatException`
  - Ungültige Datenformate

Weitergabe einer gefangenen Exception

```
try
{
    // Ein riskantes Stück Code
}
catch (Exception ex)
{
    // Loggen der Exception oder andere Schritte
    throw; // Ursprüngliche Exception weiterwerfen
}
```

- `throw;` wirft die ursprüngliche Exception weiter.
- Stacktrace bleibt intakt.

Unterschied: `throw` vs. `throw new`

`throw;`

- **Verwendung**: Ursprüngliche Exception weitergeben.
- **Vorteil**: Beibehaltung des ursprünglichen Stacktraces.
- **Beispiel**:

```
catch (Exception ex)
{
    // Zusätzliche Arbeit, z.B. Logging
    throw;
}
```

Unterschied: `throw` vs. `throw new`

`throw new Exception`

- **Verwendung**: Neue Exception erstellen.
- **Nachteil**: Verlust des ursprünglichen Stacktraces.
- **Beispiel**:

```
catch (Exception ex)
{
    throw new Exception("Ein Fehler ist aufgetreten", ex);
}
```

Nutzung von InnerException

- **Zweck**: Beibehaltung der ursprünglichen Ausnahmeinformationen.
- **Vorteil**: Ermöglicht das Erstellen einer neuen spezifischen Exception, während Details der ursprünglichen erhalten bleiben.

Beispiel

```
try
{
    // Problematischer Code
}
catch (Exception ex)
{
    throw new CustomException("Spezifische Nachricht", ex);
}
```

- `InnerException` enthält Details zur ursprünglichen Exception.
- Hilfreich für das Debugging und Protokollierung.

Custom Exception-Klassen

Benutzerdefinierte Ausnahmen bieten die Möglichkeit, spezifische Fehlerzustände innerhalb einer Anwendung darzustellen und zu handhaben.

```
public class ProductException : Exception
{
    public string ProductCode { get; }

    public ProductException(string message, string productCode)
        : base(message)
    {
        ProductCode = productCode;
    }

    public ProductException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

Exception Properties und Methods

- `Message` : Fehlerbeschreibung
- `StackTrace` : Aufrufreihenfolge
- `InnerException` : Ursprüngliche Exception
- `Source` : Name der Exception-Quelle
- `HResult` : Numerischer Fehlercode

Exception Filtering mit when

```
try
{
    ProcessOrder(order);
}
catch (OrderException ex) when (ex.OrderId > 1000)
{
    // Behandlung für bestimmte Order-IDs
}
catch (OrderException ex) when (IsRetryable(ex))
{
    // Behandlung wiederholbarer Fehler
}
```

Vorteile des Exception Handlings

- **Robustheit:** Programme können besser mit unerwarteten Situationen umgehen.
- **Wartbarkeit:** Klar strukturiertes Fehlerbehandlungssystem erleichtert das Verständnis und die Wartung von Code.
- **Fehlerverfolgung:** Hilfreiche Fehlermeldungen ermöglichen eine effektive Debugging- und Fehlerverfolgung.

Best Practices

- **Spezifische Ausnahmen fangen:** Beginnen Sie mit spezifischeren Ausnahmebehandlungen und arbeiten Sie zur Allgemeinheit hin.
- **Vermeiden Sie leere Catch-Blöcke:** Stellen Sie sicher, dass alle gefangenen Ausnahmen sinnvoll behandelt werden.
- **Finally für Bereinigung verwenden:** Nutzen Sie den Finally-Block für das Schließen von Dateien, Freigeben von Ressourcen etc.
- **Custom Exceptions:** Nur erstellen, wenn zusätzliche Informationen bereitgestellt werden müssen, die in bestehenden Ausnahmearten nicht verfügbar sind.

Performance-Aspekte

- Try-Catch Blöcke haben keinen Performance-Impact
- Exception-Handling ist nur beim Werfen kostspielig
- Exceptions nicht für Programmfluss-Kontrolle
- Validierung vor Exception-Throwing

```
// Besser
if (string.IsNullOrEmpty(name)) return false;

// Schlechter
try
{
    ProcessName(name);
}
catch (ArgumentException)
{
    return false;
}
```

Logging in C#

Logging hilft dabei, Informationen über den Programmablauf zu erfassen und unterstützt bei der Fehlerdiagnose und -überwachung.

Vorteile von Logging

- **Monitoring:** Ermöglicht Echtzeitüberwachung von Anwendungen.
- **Fehlerdiagnose:** Hilft bei der schnellen Identifizierung und Behebung von Problemen.
- **Performance-Analyse:** Unterstützt das Verständnis von Leistungsengpässen und der allgemeinen Anwendungseffizienz.

Logging-Frameworks

- **.NET built-in Logging (Microsoft.Extensions.Logging):**
  - Flexibles und erweiterbares Logging-Framework, integriert in ASP.NET Core.
  - Unterstützt verschiedene Log-Level (Information, Warning, Error, etc.).
- **NLog:**
  - Leistungsstarkes, flexibles und leicht konfigurierbares Logging-Framework.
  - Unterstützt Logging in Dateien, Datenbanken, Netzwerke, etc.
- **log4net:**
  - Teil der Apache Logging Services.
  - Bietet eine umfassende und flexible Logging-Infrastruktur.

Logging mit Microsoft.Extensions.Logging

```
dotnet add package Microsoft.Extensions.Logging
```

```
using Microsoft.Extensions.Logging;

var loggerFactory = LoggerFactory.Create(builder =>
{
    builder.AddConsole(); // Lädt das Console-Logging
});

ILogger logger = loggerFactory.CreateLogger<Program>();

logger.LogInformation("An application event occurred.");
logger.LogWarning("A potential issue detected.");
logger.LogError("An error occurred.");
```

Logging Best Practices

```
try
{
    ProcessOrder(order);
}
catch (Exception ex)
{
    _logger.LogError(
        ex,
        "Fehler bei Order {OrderId}: {Message}",
        order.Id,
        ex.Message
    );
    throw; // Re-throw wenn nötig
}
```

Performance messen in C#

Warum Performance messen?

- **Optimierungspotential**: Identifizieren von Flaschenhälsen.
- **Ressourceneffizienz**: Bessere Nutzung von CPU, Speicher und E/A.
- **Skalierbarkeit**: Sicherstellen, dass Anwendungen mit zunehmender Last umgehen können.
- **Benutzerzufriedenheit**: Verbesserung der Reaktionsfähigkeit.

Einfache Zeitmessung mit Stopwatch

```
using System.Diagnostics;

var stopwatch = Stopwatch.StartNew();

// Zu messender Code

stopwatch.Stop();
Console.WriteLine($"Code executed in {stopwatch.ElapsedMilliseconds} ms");
```

- **Vorteil**: Einfach und direkt.
- **Nachteil**: Manuelle Integration erforderlich, geringe Detailtiefe.

Benchmarking mit BenchmarkDotNet

- **Vorteil**: Präzise und reproduzierbare Ergebnisse.
- **Zweck**: Mikromessungen und Vergleiche von Codepfaden.

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

public class MyBenchmark
{
    [Benchmark]
    public void MyMethod()
    {
        // Zu benchmarkender Code
    }
}

class Program
{
    static void Main(string[] args)
    {
        var summary = BenchmarkRunner.Run<MyBenchmark>();
    }
}
```

Debugging-Techniken

Visual Studio Debugger Features

- Breakpoints (F9)
- Step Over (F10)
- Step Into (F11)
- Step Out (Shift + F11)
- Continue (F5)

Watch Windows und Quick Watch

- Variablen überwachen
- Ausdrücke auswerten
- Object Members inspizieren
- Array-Visualisierung
- Custom Visualizer

Debug vs. Release Builds

Debug Build

- Optimierungen deaktiviert
- Debugging-Symbole
- DEBUG konstante definiert

Release Build

- Optimierungen aktiviert
- Keine Debugging-Symbole
- RELEASE konstante definiert

Using Statement und IDisposable

Das `using`-Statement sorgt dafür, dass Ressourcen wie Streams, Dateien oder Datenbankverbindungen automatisch freigegeben werden, sobald sie nicht mehr benötigt werden.

Vorteile

- **Ressourcensicherheit:** Stellt sicher, dass Ressourcen auch bei Ausnahmen korrekt freigegeben werden, wodurch Speicherlecks verhindert werden.
- **Verbesserte Lesbarkeit:** Der Code wird klarer und leichter lesbar, da es explizit zeigt, wann Ressourcen verwendet und freigegeben werden.
- **Kürzerer Code:** Reduziert Boilerplate-Code, da die Notwendigkeit für explizite Aufrufe der `Dispose`-Methode entfällt.

Using Statement und IDisposable

```
// Traditionell
using (var reader = new StreamReader("file.txt"))
{
    var content = reader.ReadToEnd();
}

// C# 8.0+ Syntax
using var writer = new StreamWriter("log.txt");
writer.WriteLine("Log entry");
// Automatische Dispose am Ende des Blocks
```

Übung 1: Datei-Verarbeitungssystem

Ziele

- Implementierung robuster Fehlerbehandlung
- Verständnis von Exception-Hierarchien
- Praktische Anwendung von Logging
- Verwendung von Using-Statements

Fokus

- Try-Catch-Finally Blöcke
- Custom Exceptions
- Ressourcen-Management
- Fehler-Logging

Abschlussprojekt: Mediathek

- Entwickeln Sie eine C#-Konsolenanwendung für eine Mediathek mit Büchern, Zeitungen, CDs und DVDs.
- Nutzen Sie Vererbung und Polymorphismus, um Medientypen zu verwalten.
- Implementieren Sie Funktionen zum Hinzufügen, Anzeigen und Suchen von Medien.