C# Fortgeschrittenenschulung

Events und Attribute

Wiederholung: Delegates in C#

Was ist ein Delegate?

- **Definition:**
  - Ein Delegate ist ein Typ, der Referenzen auf Methoden mit einer bestimmten Signatur und einem bestimmten Rückgabewert speichern kann.
- **Vergleichbar mit:**
  - Funktionalen Zeigern in anderen Programmiersprachen.

Verwendung von Delegates

- **Hauptmerkmale:**
  - **Typsicherheit:** Delegates sorgen dafür, dass nur Methoden mit den richtigen Signaturen verwendet werden.
  - **Multicast-Unterstützung:** Delegates können mehrere Methoden aufrufen.
- **Beispiel:**

```
public delegate int MathOperation(int a, int b);

public class Calculator
{
    public int Add(int x, int y) => x + y;
    public int Subtract(int x, int y) => x - y;
}
```

Zuweisung und Aufruf eines Delegates

- **Zuweisung:**
  - Einen Delegate mit einer Methode verbinden.

```
Calculator calc = new Calculator();
MathOperation addOperation = calc.Add;
```

- **Aufruf:**

```
int result = addOperation(5, 3);  // Ergebnis: 8
```

- **Multicast:**
  - Delegates können kombiniert werden, um mehrere Methoden gleichzeitig aufzurufen.

Zuweisung mehrerer Methoden zu einem Delegate

```
Calculator calc = new Calculator();

MathOperation operations = calc.Add;
operations += calc.Subtract;
```

- **+= Operator:** Fügt Methoden zur Aufrufliste des Delegates hinzu.

Ausführung des Multicast-Delegates

```
operations(10, 5);
// Ausgabe:
// Add: 15
// Subtract: 5
```

- Methoden `Add` und `Subtract` werden der Reihe nach aufgerufen.

Entfernen von Methoden aus dem Delegate

```
operations -= calc.Subtract;

// Geänderter Aufruf
operations(10, 5);
// Ausgabe:
// Add: 15
```

- **-= Operator:** Entfernt `Subtract` aus dem Delegate.

Wichtige Überlegungen

- **Rückgabewerte:**
  - Nur der letzte Rückgabewert wird bei Multicast-Delegates beachtet (nicht relevant bei `void`-Methoden).
- **Fehlerbehandlung:**
  - Bei einer Ausnahme in einer Methode wird keine der nachfolgenden Methoden aufgerufen.

Delegates als Rückgabewerte und Parameter

- **Delegates als Parameter:**
  - Ermöglichen höhere Flexibilität und Wiederverwendbarkeit von Code.

```
public void ExecuteOperation(MathOperation operation, int a, int b)
{
    Console.WriteLine(operation(a, b));
}
```

- **Delegates als Rückgabewerte:**
  - Funktionen können Delegates zurückgeben, um flexible Verhaltensweisen zu implementieren.

Beispiel: Rückgabe eines Delegates

```
public delegate int MathOperation(int a, int b);

public class Calculator
{
    public MathOperation GetOperation(string op)
    {
        switch (op.ToLower())
        {
            case "add":
                return Add;
            case "subtract":
                return Subtract;
            default:
                throw new InvalidOperationException("Invalid operation");
        }
    }

    private int Add(int x, int y) => x + y;
    private int Subtract(int x, int y) => x - y;
}
```

Vorteile von Delegates

- **Entkopplung:**
  - Delegates ermöglichen das Entkoppeln von Methodenaufrufen von der Methode selbst.
- **Strategieänderung zur Laufzeit:**
  - Delegates können zur Laufzeit zugewiesen und geändert werden, was flexible Softwarearchitekturen ermöglicht.
- **Ereignisbehandlung:**
  - Delegates sind die Grundlage für die Ereignisbehandlung in C#.

Einführung in Events

- **Definition:**
  - Ein Event ist ein Mechanismus, mit dem eine Klasse eine Benachrichtigung auslösen kann, wenn eine bestimmte Aktion auftritt.
- **Anwendungsfälle:**
  - Benachrichtigen anderer Objekte über Änderungen.
  - Implementierungen für Benutzeroberflächen und Interaktionen.

Events in C#

Deklaration und Nutzung

```
public delegate void NotifyEventHandler(string message);

public class Publisher
{
    public event NotifyEventHandler Notify;

    public void DoSomething()
    {
        // Logik die einen Event auslöst
        Notify?.Invoke("Event ausgelöst");
    }
}
```

- Verwenden Sie den `event` -Schlüsselwort, um Ereignisse zu definieren.

Subscribing to Events

Event-Händler implementieren

```
public class Subscriber
{
    public void OnNotify(string message)
    {
        Console.WriteLine(message);
    }
}
```

- **Verbinden des Events:**

```
var publisher = new Publisher();
var subscriber = new Subscriber();

publisher.Notify += subscriber.OnNotify;
```

- **Entkoppelte Kommunikation** zwischen Publisher und Subscriber.

Einführung in Attribute

- **Definition:**
  - Attribute sind Metadaten, die zusätzliche Informationen zu den Programm-Elementen hinzufügen können.
- **Beispiele:**
  - `[Obsolete]` – Markiert veralteten Code.
  - `[Serializable]` – Macht Klassen serialisierbar.

Benutzerdefinierte Attribute

Eigenes Attribut erstellen

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class InformationAttribute : Attribute
{
    public string Description { get; }

    public InformationAttribute(string description)
    {
        Description = description;
    }
}
```

- Attribute werden von `System.Attribute` abgeleitet.

Typische Einsatzgebiete von Attributen

- **Dokumentation:**
  - `[Obsolete]` kennzeichnet veraltete Methoden und Klassen.
  - `[Description]` fügt informativen Text hinzu, z.B. in benutzerdefinierten Attributen und APIs.
- **Serialisierung:**
  - `[Serializable]` markiert Klassen, die für die Serialisierung vorgesehen sind.
  - `[DataContract]` und `[DataMember]` werden in WCF verwendet, um Datenverträge zu definieren.

Weitere Einsatzgebiete

- **Codekompilation:**
  - `[Conditional]` ermöglicht bedingte Methodenaufrufe, z.B. bei Debugging.
- **Sicherheit:**
  - `[PrincipalPermission]` überprüft Benutzerrechte in sicherheitskritischen Anwendungen.
- **Testing:**
  - `[TestMethod]` in Testframeworks markiert Methoden als Testfälle.

Benutzerdefinierte Attribute

- **Spezifische Anwendungsfälle:**
  - Erstellen Sie benutzerdefinierte Attribute für domänenspezifische Aufgaben wie Protokollierung oder Validierung.
- **Beispiel:**
  - Ein Attribut zur Validierung von Eingabewerten, das vor Methoden verwendet wird, um sicherzustellen, dass Eingaben bestimmten Kriterien entsprechen.

Reflection in C#

Was ist Reflection?

- **Definition:**
  - Mechanismus zur Laufzeitinspektion und Manipulation der eigenen Struktur und Metadaten einer Anwendung.
- **Einsatzgebiete:**
  - Dynamisches Erzeugen von Typen
  - Zugriff auf Metadaten
  - Laufzeitstrukturanalyse

Grundlagen der Reflection

- **Namespace:**
  - `System.Reflection`
- **Zentrale Klassen:**
  - `Assembly` : Repräsentiert eine geladene .NET-Anwendung
  - `Type` : Repräsentiert Informationen über einen Typ
  - `MethodInfo` , `PropertyInfo` , `FieldInfo` : Repräsentieren Methoden, Eigenschaften und Felder

Typanalyse mit Reflection

```
Type type = typeof(SomeClass);

Console.WriteLine("Methods:");
foreach (var method in type.GetMethods())
{
    Console.WriteLine($"{method.Name}");
}

Console.WriteLine("Properties:");
foreach (var prop in type.GetProperties())
{
    Console.WriteLine($"{prop.Name}");
}
```

- **Zugriff auf Typ-Konstruktionselemente:** Methoden, Eigenschaften und Felder zur Laufzeit auflisten.

Instanziierung mit Reflection

```
Type type = typeof(SomeClass);
object instance = Activator.CreateInstance(type);

Console.WriteLine($"Instance created: {instance}");
```

- **Dynamische Objektinstanziierung:** Erzeugt Instanzen eines Typs bei Laufzeit.

Methoden mit Reflection aufrufen

```
MethodInfo method = type.GetMethod("DoSomething");
method.Invoke(instance, new object[] { "Hello, World!" });
```

- **Methodenaufrufe:** Dynamischer Aufruf von Methoden einer Instanz.

Eigenschaften mit Reflection ändern

```
PropertyInfo property = type.GetProperty("SomeProperty");
property.SetValue(instance, newValue);

var value = property.GetValue(instance);
Console.WriteLine($"Property Value: {value}");
```

- **Zugriff auf Eigenschaften:** Setzen und Abrufen von Eigenschaftswerten zur Laufzeit.

Vorteile und Herausforderungen

- **Vorteile:**
  - Erhöhte Flexibilität und dynamische Code-Prüfung.
  - Unterstützung bei Frameworks, die Variabilität brauchen, z.B. ORMs.
- **Herausforderungen:**
  - Komplexität und potenzieller Leistungseinbruch.
  - Erhöhtes Sicherheitsrisiko, da Typinformationen zur Laufzeit modifizierbar sind.

Attribute mit Reflection nutzen

Auslesen von Attributen

```
var attrs = typeof(MyClass).GetCustomAttributes(false);

foreach(var attr in attrs)
{
    if(attr is InformationAttribute info)
    {
        Console.WriteLine(info.Description);
    }
}
```

- **Reflection** ermöglicht das Auslesen von Attributen zur Laufzeit.

Übungszeit

Anstehende Übungen zu Events und Attributen

- **Übung 1: Implementierung und Nutzung von Events**
  - Erstellen und Verwalten von Events innerhalb einer Klasse-Struktur.
- **Übung 2: Erstellung und Verarbeitung von Attributen**
  - Entwickeln eines benutzerdefinierten Attributs und Auswertung mithilfe von Reflection.

Ziel der Übungen

- **Vertiefung des Verständnisses:**
  - Praktische Anwendung der Konzepte von Events und Attributen.
- **Erhöhung der Flexibilität:**
  - Lernen, wie man durch Events entkoppelte Anwendungen entwickelt.
- **Metadaten verstehen:**
  - Nutzen von Attributen zur Verbesserung von Code-Kommentar und -Ressourcenauswertung.