VAWI OBJC WS 23/24 Block B

Hausarbeit

Aufgabe 2

von: Schubert, Patrick

1 Neuerungen in C#10.0 oder C#11.0

Mit Einführung einer neuen Version, wie z. B. C#10.0 oder C#11.0 wird die Sprache C# stetig verbessert. Dem Folgend sollen einige dieser Verbesserungen näher untersucht werden.

1.1 Verwendung einer globalen using Direktive

Mit einer sog. globalen „using“ Anweisung - auch Direktive genannt - können ab C#10.0 Namensräume (sog. Namespaces) definiert werden, ohne diesen voll qualifiziert anzugeben [1]. Voll qualifiziert meint hier, dass durch das Hinzufügen des Schlüsselworts global nicht mehr jede einzelne Quelldatei im Projekt damit beziffert werden muss (s. Abb. 1). Einmal definiert, werden alle Typen aus einem Namensraum für ein Projekt importiert [2]. Im Gegensatz dazu, würden die Typen nur für eine Datei gelten. Dabei könnte die Syntax z. B. folgendermaßen aussehen: global using <voll-qualifizierter-namensraum> [2].

Hauptvorteile der zuvor erläuterten globalen using – Anweisungen sind u. A. die Lesbarkeit des Codes zu verbessern, indem redundante using – Anweisungen in mehreren Dateien vermieden werden. Außerdem erspart dies Zeit für Softwareentwickler und vermindert Fehler im Code, wenn beispielsweise ein using vergessen würde. Folglich lässt sich der Code leichter verwalten. Nachteilig ist beispielsweise, dass nicht direkt auf einen inkludierten Namespace (Namensraum) geschlossen werden kann, wenn nicht direkt in der jeweiligen Datei gearbeitet wird.

1.2 Verbesserungen bei Lambdaausdrücken

Allgemein wird durch einen Lambdaausdruck eine sog. anonyme Funktion erstellt. Dabei bezeichnet das Wort anonym eine Funktion, die keinen Namen besitzt. Beispielsweise definiert (Eingabeparameter) => Ausdruck einen solchen Lambdaausdruck. Für einen einzigen bzw. für mehrere Eingabeparameter müssen deren Datentypen nicht explizit angegeben werden. Sie werden vom Compiler selbst abgeleitet. Schließlich können mit Einführung von C#10 natürliche Datentypen für solche Lambdaausdrücke festgelegt werden, obwohl grundsätzlich für diese, kein eigenes Typsystem existiert. Vor der Einführung von C#10 konnte ein sog. Delegat wie z. B. Func<…> - ein Delegat mit Rückgabewert - oder Action<…> - ein Delegat ohne einen Rückgabewert –

definiert und dadurch ein Typ erzwungen werden [3]. Das folgende Beispiel zeigt einen Lambdaausdruck mit der einen Eingabeparameter x entgegennimmt und den Rückgabewert x im Quadrat einer Variablen eines Func-Delegaten zuweist:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Abb. 1: Func<…> - Delegat (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression, abgerufen am: 01.04.2024)

Durch die Version C#10.0 kann der Compiler einen natürlichen Datentyp selbst bestimmen, wie im Folgenden Beispiel gezeigt:

```
var parse = (string s) => int.Parse(s);
```

Abb. 2: Natürlichen Datentypen bestimmen (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression, abgerufen am: 01.04.2024)

Parse wird dabei automatisch durch den Compiler als Func<string, int> abgeleitet. Wenn kein geeigneter Datentyp bestimmt werden kann, wird dieser synthetisiert [3]. D. h. er wird einem weniger expliziten Datentyp, wie z. B. System.Object oder System.Delegate zugewiesen:

```
object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Abb. 3: Synthetisierte Datentypen (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression, abgerufen am: 01.04.2024)

Sollte der Compiler dennoch keinen Datentyp selbstständig ableiten können, weil beispielsweise kein Datentyp zu einem Eingabeparameter angegeben wird, muss der Delegat deklariert bzw. zugewiesen werden [3]. Dies soll das nachfolgende Beispiel verdeutlichen:

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda

Func<string, int> parse = s => int.Parse(s);
```

Abb. 4: Zuweisung eines Delegaten aufgrund des fehlenden Datentyps beim Eingabeparameter (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression, abgerufen am: 01.04.2024)

Wie bei den Eingabeparametern gilt das zuvor gesagte auch für den sog. Rückgabetyp, was im folgenden Beispiel gezeigt werden soll:

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type

var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Abb. 5: Zuweisung eines Delegaten aufgrund des fehlenden Datentyps beim Rückgabewert (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression, abgerufen am: 01.04.2024)

Schließlich kann der Rückgabetyp nicht eindeutig abgeleitet werden, da es ein Integer oder ein String sein könnte. Je nachdem, welchen Wahrheitswert b besitzt. Folglich wird er mit System.Object bzw. object deklariert.

Ein großer Vorteil durch die Verbesserung der Lambdaausdrücke ist, dass Entwickler heute nicht mehr die Datentypen von Eingabeparametern und Rückgabewerten in Lambdaausdrücken explizit angeben müssen, indem sie Delegat-Typen wie Func<…> oder Action<…> verwenden müssen. Dies macht den Code schlanker, lesbarer und senkt Entwicklungszeiten. Außerdem ist der Code weniger Fehleranfällig, da synthetisierte Datentypen – beispielsweise System.Object oder System.Delegate - immer noch dafür sorgen, dass etwaiger Code kompiliert und ausgeführt wird. Andererseits könnte eine automatische Typableitung dazu führen, dass Entwickler weniger Kontrolle über die tatsächlichen Datentypen haben, obwohl es sich bei C# um eine statisch typisierte Sprache handelt. Ältere Compiler sind möglicherweise nicht in der Lage die Syntax korrekt zu verarbeiten. Schlussendlich ist nicht klar, wie der Compiler die Datentypen intern ableitet.

1.3 Unformatierte Zeichenfolgeliterale

Mit Einführung von C#11.0 wurde ein neues Format für Zeichenfolgenliterale eingeführt. Ein unformatiertes Zeichenfolgenliteral beginnt mit mindestens drei doppelten geraden Anführungszeichen oben (""") und endet mit der gleichen Anzahl doppelter Anführungszeichen (s. Abb. 6). Dabei kann in den Zeichenfolgenliteralen beliebiger Text enthalten sein, einschließlich Leerzeichen, Zeilenvorschubzeichen, eingebettete Anführungszeichen und andere Sonderzeichen, ohne dass Escapesequenzen erforderlich sind. Zeilenvorschübe nach dem öffnenden und vor den schließenden Anführungszeichen gehören nicht zum endgültigen Inhalt. Abbildung 6 soll das zuvor Gesagte verdeutlichen [4]:

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

Abb. 6: Unformatierte Zeichenfolgenliterale (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-11?source=recommendations#raw-string-literals, abgerufen am: 01.04.2024)

Außerdem können die zuvor genannten Zeichenfolgenliterale mit einer Zeichenfolgeninterpolation kombiniert werden, um beispielsweise Klammern in den Ausgabetext einzuschließen oder zuvor definierte Variablen einzubetten. Mehrere $-Zeichen geben dabei an, wie viele aufeinander folgende Klammern die Interpolation starten und beenden, wie folgende Abbildung zeigt [4]:

```
var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

Abb. 7: Zeichenfolgeninterpolation (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-11?source=recommendations#raw-string-literals, abgerufen am: 01.04.2024)

Nachfolgend soll der Hinweis ergehen, dass beispielsweise bis C#10.0 folgendes zulässig war:

```
var x = $"""
    Hello
{1 + 1}
    World
    """;
```

Abb. 8: Verstoß gegen Konvention in C#11.0 (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-11?source=recommendations#raw-string-literals, abgerufen am: 01.04.2024)

Dies verstieß gegen die Regel, dass der Zeileninhalt (einschließlich der Stelle, an der eine Interpolation beginnt) mit demselben Leerzeichen wie die letzte """; Zeile beginnen muss. Mit Einführung von C#11.0 ist es nun erforderlich, dass das Obige wie folgt geschrieben wird:

```
var x = $"""
    Hello
    {1 + 1}
    World
    """;
```

Abb. 9: Korrekte Schreibweise Zeichenfolgenliterale in C#11.0 (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-11?source=recommendations#raw-string-literals, abgerufen am: 01.04.2024)

Durch die Verbesserungen von Zeichenfolgenliteralen können schließlich Zeichenfolgen über mehrere Zeilen hinweg ohne Escape-Sequenzen dargestellt werden. Dies macht den Code lesbarer und erleichtert die Wartung. Insbesondere bei langen Zeichenfolgen oder solchen mit vielen Sonderzeichen ist dies sehr hilfreich. Das Risiko möglicher Fehler wird damit reduziert. Außerdem wird dadurch der Code insgesamt kompakter, da weniger manuelle Formatierungen erforderlich sind. Sonderzeichen können einfacher integriert werden.

1.4 Neues Listenmuster

Eine weitere Neuerung in C#11 sind sog. Listenmuster, die einen Musterabgleich von einem Array (sog. Feld) oder einer Liste - allgemein handelt es sich dabei um Sequenzen - erleichtern. Dabei können beliebige Elemente wie z. B. Konstanten, Typen, Eigenschaften oder relationale Muster mit einer Eingabesequenz verglichen werden [5]. Dies soll im folgenden Beispiel durch Einträge in einem Array verdeutlicht werden:

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]);  // True
Console.WriteLine(numbers is [1, 2, 4]);  // False
Console.WriteLine(numbers is [1, 2, 3, 4]);  // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]);  // True
```

Abb. 10: Anwendung eines Listenmusters (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/patterns#list-patterns, abgerufen am: 01.04.2024)

Die integer Zahlen 1, 2 und 3 im Array numbers entsprechen dabei den Mustern, die mit dem Schlüsselwort „is“ und der gezeigten Logik abgefragt werden.

Weiterhin kann mit dem sog. Ausschussmuster „_“ (Unterstrich) oder dem sog. var – Muster alias „var variablenname“ - var steht dabei für Variable - gearbeitet werden [5]. Durch einen Unterstrich „_“ wird ein Zeichen nicht berücksichtigt bzw. ausgeschlossen. Im folgenden Beispiel soll die gleiche Eingabesequenz wie in Abb. 10 verwendet werden:

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is {first}.");
}
// Output:
// The first element of a three-item list is 1.
```

Abb. 11: Anwendung des Ausschussmusters und var – Musters (Quelle: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/patterns#list-patterns, abgerufen am: 01.04.2024)

Schließlich wird deutlich, dass das erste Element (die Zahl 1) des Arrays mit einer definierten Variablen z. B. „first“ abgefragt werden kann. Die Zahlen 2 und 3, also die Elemente eins und zwei des Arrays werden dabei durch Anwendung das Ausschussmusters ignoriert.

Durch Einführung der sog. Listenmuster können Eingabesequenzen und damit die Inhalte von Listen oder Arrays leichter abgefragt werden. Hierzu müssen keine Schleifen mehr verwendet werden. Dies steigern die Performance und Lesbarkeit des Codes. Durch eine breite Palette von Mustern könne Entwickler flexibel Datenstrukturen effizient analysieren und auf unterschiedliche Muster unterschiedlich reagieren. Dadurch wird der Code robuster. Zudem ist der Code insgesamt kompakter, da z. B. mit nur einer einzigen Zeile Code eine gesamte Datenstruktur überprüft werden kann. Folglich können if – Anweisungen oder Switch – Case Anweisungen für die Überprüfung von Sequenzen entfallen.

Literaturverzeichnis

[1] Microsoft. "What's New in C# 10.0". URL: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-10#global-using-directives [zuletzt abgerufen am: 01.04.2024].

[2] Microsoft. "Using-Direktive (C#-Programmierhandbuch)". URL: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/using-directive [zuletzt abgerufen am: 01.04.2024].

[3] Microsoft. "Lambdaausdrücke (C#-Programmierhandbuch)". URL: https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression [zuletzt abgerufen am: 01.04.2024].

[4] Microsoft. "What's New in C# 11.0". URL: https://learn.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-11?source=recommendations#raw-string-literals [zuletzt abgerufen am: 01.04.2024].

[5] Microsoft. "What's New in C# 11.0".https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/operators/patterns#list-patterns [zuletzt abgerufen am: 01.04.2024].