Namespaces, Assemblies & Generische Programme

1. Namespaces

1.1. Was sind Namespaces?

`Namespaces` sind logische Gruppierungen vom Code. Sie sind dafür gedacht Namensüberschneidungen zu verhindern. (Also wenn man zwei Klassen `Color` haben will, eine bei einem `Drawing` und das andere bei ner `Font` beispielsweise). Statt dem Namen der Klassen in einem Namespace ruft man `<Namespace>.<Klassenname>` auf (in unserem Beispiel Font.Color und Util.Color). Diese Namensräume sind dabei nur für den Programmierer wichtig (also in Compilezeit). Zur Laufzeit haben sie keinen Einfluss auf das Programm. Wenn man eine Klasse außerhalb eines (benannten) Namespaces anlegt, dann ist sie im implizit definierten globalen Namespace.

Man "importiert" Klassen aus anderen Namespaces mit `using`. Ich schreibe hier "importiert" in Anführungsstrichen, da man auch ohne `using` Klassen mit ihrem vollständigen Namen importieren kann. `using` erlaubt einem lediglich die Namespacenamen wegzulassen. Auch ist wichtig zu wissen, dass das Importieren eines Namespaces weder seine übergeordneten noch seine untergeordneten Namespaces importiert. `using Util.Figures` importiert also weder `Util` noch `Util.Figures.Forms`. Man kann nur `public` oder `internal` Klassen importieren, letzteres allerdings nur aus dem gleichen `Assembly`. Mehr zu Assemblies später.

1.2. Namespaces in C# vs Java

Das Äquivalent zu `Namespaces` sind in Java `Packages`. C# geht dabei bei `Namespaces` deutlich freier mit der Lokalität des Codes um. Es können mehrere `Namespaces` in einer Datei sein und `Namespaces` können auch über mehrere Dateien verteilt sein. In Java ist dagegen eine feste Struktur vorgegeben. `Packages` entsprechen Foldern und jede `Klasse` einer Datei in diesem Folder.

In Java ist das importieren von `Klassen` mit `import` möglich. Dabei kann man in Java auch alle untergeordneten `Packages` mit `import <Package>.*;` importieren. C# erlaubt dies nicht. Dafür gibt es in C# Aliases für `Namespaces`. Das heißt wir können `Namespaces` in unserer Klasse unter anderem Namen verwenden. Der Code sähe dann so aus: `using MyAlias = MyNamespace;`.

Auch im Output unterscheiden sich die beiden Sprachen. In Java werden die `Packages` im Output genau so abgebildet wie im Code. Das führt hier zu teils riesigen verzweigten und unübersichtlichen Ordnerstrukturen. In C# werden die `Namespaces` im Output nicht abgebildet, sie können also alle zusammen in einem `Assembly` liegen oder auf mehrere verteilt werden.

2. Assemblies

2.1. Was sind Assemblies?

`Assemblies` dienen ebenso wie `Namespaces` zur Gruppierung von Code. Der entscheidende Unterschied besteht darin, dass `Assemblies` zur Laufzeit existieren und `Namespaces` nur zur Compilezeit. `Assemblies` können dabei die ausführbaren Dateien (`.exe`) oder Bibliotheken (`.dll`) sein. Diese Gruppierung erlaubt dabei für eine gemeinsame Versionierung und Signierung von Codeabschnitten.

Ein `Assembly` besteht dabei aus einem einem `Manifest` (Inhaltsverzeichnis), `Metadaten` (Beschreibung), `Modules` und zu guter letzt den Dateien für den eigentlichen Code.

2.2. Modules

`Modules` existieren in C# nur im Kontext von `Assemblies`. Meist ist ein `Assembly` nur aus einem `Module` aufgebaut, doch auch mehrere sind möglich. Falls man sich für mehrere `Modules` in einem `Assembly` entscheidet, so trägt nur ein einziges von ihnen das `Manifest`, während die anderen in .netmodule Dateien kompiliert werden.

Die Aufsplittung in mehrere `Modules` hat dabei mehrere Vorteile. Zum einen erlaubt sie "Lazy Loading", also das Laden von Codeabschnitten erst wenn sie benötigt werden. Zum anderen könnte man theoretisch einzelne `Modules` in verschiedenen `.NET` Sprachen schreiben. Praktisch wird letzteres wohl eher weniger Anwendung finden, aber es ist möglich!

Man kompiliert `Klassen` in einzelne `Modules` und `Assemblies` mit dem Compilerflag `/target:module` zu `.netmodule` bzw. `/target:library` zu `.dll` (oder ohne Flag als `.exe`).

Wenn man auf `Klassen` einer anderen `Assemblie` zugreifen will, hat man dabei 2 Optionen: Entweder kann man mit `/reference:A.dll` A eigenständig lassen. Oder man kann mit `/addmodule:A.netmodule` A in die eigene `Assembly` einbinden. Bei letzterem gehört A dann zum eigenen `Assembly` und kann nicht mehr separat verwendet werden.

2.3. Versionierung

Bei `Assemblies` wird grundlegend zwischen `privaten` und `public Assemblies` unterschieden. `Private Assemblies` sind dabei nur für die eigene Anwendung gedacht und werden nicht von anderen Anwendungen verwendet. Das erlaubt ihnen ohne Versionierung einfach im gleichen Ordner wie die Anwendung zu liegen. `Public Assemblies` können dagegen von anderen Anwendungen verwendet werden und müssen daher versioniert werden. Sie liegen in einem globalen `Assembly Cache`, der sich von allen Anwendungen geteilt wird.

Die Versionierung Erfolgt dabei über 4 Eigenschaften:

- Version
  - Major.Minor.Build.Revision
  - `4.2.1024.0`
- Culture
  - `en-US`

- Public Key
  - `mypubkey.snk`
- Name
  - `MyLib`

Dabei ist lediglich der Public Key verpflichtend, die anderen Eigenschaften können weggelassen werden. Wenn einem beim Laden die Versionsnummer in Teilen egal ist, kann man diese durch einen * wie folgt markieren: `4.2.*`. Standardmäßig wird bis auf die Revision verglichen, dies lässt sich jedoch auch mit einer XML-Konfigurationsdatei ändern.

2.4. Signierung

`Public Assemblies` müssen signiert werden, damit man bei späterer Verwendung ihre Authentizität garantieren kann. Signierung beschreibt dabei die Erstellung von einem privaten und einem öffentlichen Schlüssel. Aus dem privaten Schlüssel wird mit der Prüfsumme des Codes eine `Signatur` erstellt. Diese kann dann mit dem öffentlichen Schlüssel überprüft werden. Wenn eine Datei referenziert wird, so speichert sich die referenzierende Datei den `public Key`, um später die `Signatur` überprüfen zu können. Wenn diese Datei dann geladen werden soll, wird erst wieder eine Prüfsumme erstellt und mit der von dem `public Key` entschlüsselten `Signatur` verglichen. Wenn diese übereinstimmen, so ist die Datei authentisch. Sonst wird ein `Runtime Error` geworfen.

2.5. Assemblies vs Namespaces

Sowohl `Assemblies` als auch `Namespaces` dienen zur Gruppierung von Code. Da `Assemblies` aber vor allem ein Laufzeitkonzept sind und `Namespaces` ein Compilezeitkonzept, sind sie nicht direkt miteinander verknüpft. Man kann daher `Assemblies` über Teile mehrerer `Namespaces` erstrecken.

2.6. Decompilierung

Man kann analog zu Java Bytecode auch C# Code dekompilieren. Man kann mit Programmen wie `mono` exe's in Assemblyartigen CIL-Code umwandeln. Umsetzung Linux:

- `sudo apt-get install mono-complete` //Mono installieren
- `mcs MyProgram.cs` //Compiliert zu MyProgram.exe
- `monodis --output=<outputfile> MyProgram.exe` //Dekompiliert zu outputfile

Der Output sieht dann wie folgt aus:

```
.namespace HelloWorld
{
  .class private auto ansi beforefieldinit Hello
    extends [mscorlib]System.Object
  {

    // method line 1
    .method public hidebysig specialname rtspecialname
```

```
            instance default void '.ctor' ()  cil managed
    {
        // Method begins at RVA 0x2050
    // Code size 7 (0x7)
    .maxstack 8
    IL_0000:  ldarg.0
    IL_0001:  call instance void object::'.ctor'()
    IL_0006:  ret
    } // end of method Hello::.ctor

    // method line 2
    .method private static hidebysig
           default void Main (string[] args)  cil managed
    {
        // Method begins at RVA 0x2058
    .entrypoint
    // Code size 11 (0xb)
    .maxstack 8
    IL_0000:  ldstr "Hello World!"
    IL_0005:  call void class [mscorlib]System.Console::WriteLine(string)
    IL_000a:  ret
    } // end of method Hello::Main

  } // end of class HelloWorld.Hello
}
```

Man kann hier wunderbar `Namespaces` und `Klassen` erkennen. Desweiteren sieht man die Funktion des Programs: Lade "Hello World!" in den Stack und gib ihn mit `WriteLine` aus. Hier abgebildet ist also ein Hello World Program.

3. Generische Programmierung

3.1. Naiver Ansatz

Wenn wir eine Klasse schreiben wollen, die mit vielen Datentypen umgeht, so könnte man ja einfach ausnutzen, dass alle Klassen von `object` erben. Wenn wir nun aber einfach `object` verwenden, so verlieren wir die Typsicherheit. Aber das ist ja kein Problem? Ich weiß schließlich, was ich da rein packe!
Stellen Sie sich folgenden Fall vor:

```
Buffer intBuffer = new Buffer();
intBuffer.Put(3);
intBuffer.Put(4); // kein Problem

Console.WriteLine("Enter Age: ");
intBuffer.Put(Console.ReadLine());
int x = (int) intBuffer.Get(); // Laufzeitfehler
```

Was ist passiert? Wir haben einen Buffer geschrieben, der nur `int`s aufnehmen soll. Aber beim Einlesen von der Konsole haben wir einen `string` eingelesen. Da wir aber `object` verwenden, wird der `string` einfach in den Buffer gepackt. Der Compiler kann uns hier nicht mehr helfen, da er nicht weiß, dass wir nur `int's erwarten. Beim Auslesen erwarten wir nun einen Integer, erhalten aber einen String. Daher wird hier ein Laufzeitfehler geworfen. Deswegen ist das verlassen auf Benennung von Objekten nicht ausreichend.

3.2. Generische Typen

```
class Buffer<Elem>{
    public void Put(Elem obj){...}
    public Elem Get(){...}
}
```

Mit diesem Code können wir nun einen Buffer für jeden Typen erstellen. Beim Initialisieren der `Klasse` wird der Typ festgelegt. Von nun an können nur noch Objekte dieses Typs in den Buffer gepackt werden.

```
Buffer<int> buf = new Buffer<int>();
buf.Put(3);
int x = buf.Get(); // keine Konvertierung nötig
```

Das hat den weiteren Vorteil, dass wir beim Auslesen keine Konvertierung mehr vornehmen müssen. Konvertierungen sind immer teuer, da sie eine neue Instanz erstellen müssen. Wir reduzieren also nicht nur Fehler, sondern erhöhen auch Performance!

3.3. Constraints

```
class OrderedBuffer <Elem, Prio>
where Prio : IComparable<Prio>
{
    public void Put(Elem obj, Prio prio){...}
    public Elem Get(){...}
}
```

Manchmal wollen wir nicht nur Daten eines Types abspeichern, sondern auch Funktionen auf ihm ausführen. Neben den Funktionen aus Object können wir aber Standardmäßig nichts auf generischen Typen ausführen. Es könnten schließlich auch Klassen ohne diese Funktionen eingegeben werden. Hier kommen Constraints ins Spiel. Constraints erlauben uns, minimale Anforderung an die Typen zu stellen. In unserem Beispiel müssen Prio's verglichen werden können. Dafür implementieren sie `IComparable<Prio>`. Jetzt können wir die Funktion `CompareTo` auf Prio's ausführen.

```
class Stack<T,E> where E: Exception, new() {
```

```
    public void Push(T obj){
        if (!spaceLeft()) throw new E();
        ...
    }
}
```

Wir können in Constaints auch Konstruktor Definitionen erfordern. Hier erfordern wir, dass E's Konstruktor ohne Argumente aufgerufen werden kann, damit wir selbst ein neues Objekt der Klasse E erstellen können.

3.4. Vererbung

```
class Buffer<Elem> : List<Elem> {
    public void Put(Elem obj){...}
    public Elem Get(){...}
}

class intBuffer<Prio> : List<int> {
    ...
}
```

Generische Klassen können auch von anderen Klassen erben. Dabei können sie generische Typen vererben oder diese spezifizieren. Es kann aber keine generische Typen erhalten bleiben, wenn die Klasse selbst nicht generisch ist. Bei der Instanziierung der Klasse muss schließlich auch der generische Typ feststehen.

Man kann auch Methoden einer generischen Klasse überschreiben. Diese haben dann aber nicht mehr den generischen Typ, sondern den Typ der Klasse.

Das coole an generischen Klassen in C# ist, dass man auch zur Runtime auf die Informationen zugreifen kann. Ganz im Gegensatz zu einem Java, wo zur Runtime alle generischen Klassen objects sind. Bei C# sind dagegen auch Typvergleiche möglich, da es in der Runtime erst eine Schablone für die generischen Typen erstellt. Diese Schablone wird dann für jeden Werttyp (int, char, ...) konkretisiert. Referenztypen (string, object, ...) müssen sich dagegen eine Konkretisierung teilen. Diese Konkretisierungen werden erst mit dem ersten Auftreten des Typs erstellt.

3.5. Generic Methods

```
T Max<T> (T[] a) where T: IComparable{
    T max = a[0];
    for (int i = 1; i < a.Length; i++){
        if (a[i].CompareTo(max) > 0) max = a[i];
    }
    return max;
}
```

Generische Klassen bringen uns nichts, wenn man nicht auch generische Methoden schreiben könnte. Diese werden sehr ähnlich zu Klassen aufgebaut. Man kann wie bei Klassen auch hier Constraints festlegen.

```
int[] numbers = {3, 4, 5};
int max = Max<int>(numbers);
max = Max(numbers); // Type inference
```

Man kann diese Methoden dann nutzen in dem man auch hier den Typ bei der Nutzung festlegt. Meistens ist dies aber gar nicht nötig. Wenn eines der Inputargumente vom Typ ist, so kann der Compiler diesen Typen inferieren. Sollte dies nicht der Fall sein, muss man ihn jedoch beim Aufruf angeben.

3.6. Generic Delegates

```
public Sequence<T> Select (Filter<T> matches){
    Sequence<T> result = new Sequence<T>();
    for(Node p = head; p != null; p = p.next){
        if (matches(p.data)) result.Add(p.data);
    }
    return result;
}

Sequence<int> numbers = new Sequence<int>();
numbers.Add(3); numbers.Add(-5); numbers.Add(7);

Sequence<int> positive = numbers.Select(delegate(int x) {return x > 0;});
```

Wenn man generische Methoden hat, sind auch generische Delegates erwartet. Diese sind am weitesten Verbreitet als Filtermethoden um Listen zu filtern. Hier wird ein Delegate erwartet, dass einen Wert vom Typ T entgegen nimmt und einen Boolean zurück gibt. Im Beispiel wird eine Liste von ints gefiltert, sodass nur positive ints zurück gegeben werden. Wir mussten hier wieder wegen Typinferenz den Typen nicht explizit angeben.

3.7. Null

Wieso ein eigenes Kapitel für Null-Werte? `null` ist `null`, oder nicht? Das stimmt so leider nur für Objekte. Wenn man einen Werttyp hat, so kann dieser nicht `null` sein. Ein integer hat zum Beispiel als `null` den Wert 0.

Statt also einen generischen Typen auf `null` zu setzen gibt es stattdessen die Methode `default(T)` in C#. Diese gibt den für den jeweiligen Datentyp richtigen Default Wert zurück. Das ist auch wichtig bei Vergleichen zu bedenken, da ein Vergleich ala `if (x != null)` bei Werttypen immer true zurückgeben würde.

3.8. Ko- und Kontravarianz

```
List<String> stringList = new List<String>();
List<Object> objectList;
objectList = stringList; //Compilefehler!
```

Gucken Sie sich das Beispiel an. Wieso können wir `stringList` nicht auf objectList legen? Man kann doch `Strings` auch als `? ` betrachten. Das Problem ist, dass wir in objectList auch andere `Objekte` speichern können.

```
objectList.Add(new Integer(3));
String s = stringList.Get(0); // ClassCastException!
```

Wenn wir jetzt auf `stringList` zugreifen, so kann es sein, dass wir einen `Integer` zurück bekommen.

Dieses Problem wird als **Kovarianz** und **Kontravarianz** bezeichnet.

3.8.1. Kovarianz

Kovarianz bedeutet, dass wenn 2 Typen kompatibel sind, auch ihre abgeleiteten generischen Typen kompatibel sind. Diese Kompatibilität wird durch das `out` Keyword angegeben.

```
interface Sequence<out T> {
    T Get();
}
```

Dies geht, da man out nur für Rückgabetypen verwenden kann. Das heißt wir haben nicht das Problem von oben, dass ein neuer Wert in die Liste eingefügt werden kann. Das heißt, dass man hier `Sequence<String>` als `Sequence<Object>` betrachten kann. Hier sei angemerkt, dass dies nur bei Referenzwerten funktioniert.

3.8.2. Kontravarianz

Kontravarianz bedeutet, dass wenn 2 Typen kompatibel sind, ihre abgeleiteten generischen Typen umgekehrt kompatibel sind. Diese Kompatibilität wird durch das `in` Keyword angegeben.

```
interface IComparer<in T> {
    int Compare(T x, T y);
}

IComparer<Object> objectComparer = ...;
IComparer<String> stringComparer = objectComparer; //valid
```

Bei in Dürfen die Typen nur als Input verwendet werden. Deswegen kann man hier den `objectComparer` auf den `stringComparer` legen. Dann werden die Strings als Objecte betrachtet. Ob

das sinnig ist sei dem Programmierer selbst überlassen. Aber es ist möglich und typsicher.

3.9. Generische Programmierung in C# vs Java

Allgemein ist C# in der generischen Programmierung viel mächtiger als Java. Das liegt vor allem wohl daran, dass es in Java erst nachträglich eingefügt wurde; in C# war es von Anfang an dabei. Das führt dazu, dass Generische Typen keine Umsetzung im Java Bytecode haben. Das verhindert in Java die Reflection von generischen Typen, wie es in C# möglich ist. Und selbst Arrays generischer Typen können nur unhandlich und nicht Typsicher von Java generiert werden.

```
new T[10]; // Compilefehler!
(T[]) new Object[10]; // Unhandlich
```

4. Abschluss

Dies war eine kleine Einführung in C# Assemblies, Namespaces und Generics. Es gibt noch viel zu entdecken, ich würde dafür die ausführliche Dokumentation von Microsoft empfehlen.

4.1. Quellen

- Kompaktkurs C# 7 dpunkt.Verlag
- C# Dokumentation https://docs.microsoft.com/en-us/dotnet/csharp/