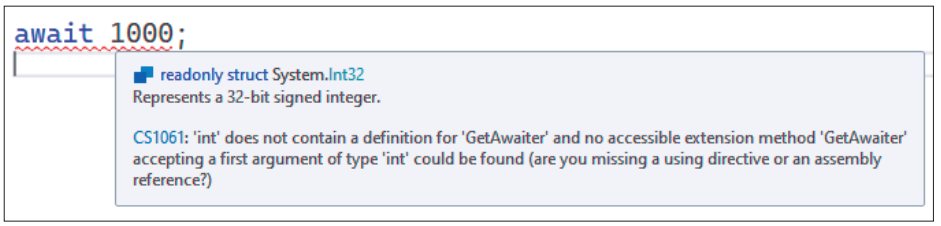
SPASS MIT AWAITABLES

await 1000;

Was soll das denn sein? Ein Syntaxfehler? Lässt der C#-Compiler nicht zu – oder doch?

Ein Test soll Klarheit bringen. Konsolenanwendung anlegen, das Statement einfügen und – siehe da –, der erste Versuch scheint schon die Annahme zu bestätigen, dass das nicht gehen kann. Die Fehlermeldung in Bild 1 kann nicht lügen. Der Hinweis allerdings, warum der Compiler das nicht mag, wirft Fragen auf. Was genau meint der Compiler mit „does not contain a definition for 'GetAwaiter'"?



Wie erwartet – das kann doch nicht funktionieren (Bild 1)

Als .NET-Adept hat man mal gelernt, dass da immer ein Ausdruck erwartet wird, der ein (bereits gestartetes) *Task*-Objekt liefert. In der Praxis wird das wohl auch meist stimmen, aber hinter den Kulissen passiert noch etwas anderes. Da gibt es noch eine verborgene, magische Namenskonvention.

Interessanterweise kann sich hinter dem *await*-Schlüsselwort jeder Ausdruckstyp verbergen, der eine öffentliche Methode namens *GetAwaiter()* besitzt. Diese muss eine Struktur vom Typ *TaskAwaiter* zurückgeben. Die Dokumentation zu *TaskAwaiter* sagt zwar „This type is intended for compiler use only" [1]; aber das soll ja nicht davon abhalten, es doch mal zu versuchen.

Dann also los: Listing 1 definiert eine Klasse namens *MagicDelay*, die einen Konstruktor mit einem *TimeSpan*-Objekt als Parameter vorsieht, dazu eine statische Factory-Methode zur einfacheren Instanzierung sowie die besagte Methode *GetAwaiter()*. Das *TaskAwaiter*-Objekt kommt von dem *Task*-Objekt, das von *Task.Delay* zurückgegeben wird.

Jetzt ist ein Aufruf wie der folgende plötzlich kein Syntaxfehler mehr, sondern funktioniert tatsächlich:

```
await MagicDelay.Milliseconds(1000);
```

Das macht zwar nichts anderes als *await Task.Delay(1000)*, es ist aber trotzdem interessant, dass es in dieser Form zulässig ist und ohne zu mucken läuft.

Na gut, aber 1000 ist ein Integer-Wert und da lässt sich ja keine zusätzliche Instanzmethode definieren. Also kommen wir so nicht weiter, oder? Es sei denn …

Was wäre denn, wenn es statt der Instanzmethode namens *GetAwaiter()* auch eine Erweiterungsmethode gäbe? Die Fehlermeldung in Bild 1 deutet so etwas ja an.

Also nächster Versuch: Die Methode *GetAwaiter()* wird aus *MagicDelay* entfernt, die Klasse mit einer ReadOnly-Eigenschaft für das *TimeSpan*-Objekt ergänzt und *GetAwaiter()* als Erweiterungsmethode in die Klasse *MagicExtensions* verlagert (Listing 2). Und siehe da, es funktioniert immer noch:

```
await MagicDelay.Milliseconds(1000);
```

Das ist doch schon mal ein guter Ansatz, da muss noch mehr gehen. Wozu der Umweg über die Klasse *MagicDelay*? Ließe

Listing 1: Die Methode GetAwaiter() macht die Klasse „awaitable"

```
public class MagicDelay
{
  private readonly TimeSpan timeSpan;

  private MagicDelay(TimeSpan timeSpan)
  {
    this.timeSpan = timeSpan;
  }

  public static MagicDelay Milliseconds(
    int milliseconds)
  {
    return new MagicDelay(
      TimeSpan.FromMilliseconds(milliseconds));
  }

  // Magic function...
  public TaskAwaiter GetAwaiter()
  {
    return Task.Delay(timeSpan).GetAwaiter();
  }
}
```

sich das nicht direkt mit *TimeSpan* machen? Mit einer Erweiterungsmethode wie hier?

```
public static class MagicExtensions
{
  public static TaskAwaiter GetAwaiter(
    this TimeSpan timeSpan)
  {
    return Task.Delay(timeSpan).GetAwaiter();
  }
}
```

Und dann ein Aufruf in dieser Form:

```
await TimeSpan.FromMilliseconds(2000);
```

Wow – das geht auch. Dann müsste es doch auch möglich sein, eine solche Extension-Methode für den Integer-Typ vorzusehen. Mal sehen, ob das mit dem folgenden Vierzeiler vielleicht schon funktioniert. Ein Integer-Wert als Zeitangabe in Millisekunden:

```
public static class MagicExtensions
{
  public static TaskAwaiter GetAwaiter(
    this int milliseconds)
  {
    return Task.Delay(TimeSpan.FromMilliseconds(
      milliseconds)).GetAwaiter();
  }
}
```

Implementieren, ausprobieren:

```
await 1000;
```

Und nun übersetzen. Der Compiler meckert nicht – Syntax korrekt – starten – läuft! Wer hätte das gedacht? Es geht also doch.

Braucht man so was?

Okay, braucht man nun nicht wirklich. Aber interessant ist es schon, dass man mit einfachen Tricks eine solche Syntax zulässig machen kann. „Bei await 1000 weiß man doch nicht, welche Einheit sich dahinter verbergen soll", werden Kritiker jetzt sagen. Sind das Sekunden, Millisekunden oder was? Andererseits muss man zugeben, dass dies bei *Task.Delay(1000)* auch nicht besser gelöst wurde.

Ob es praktische Anwendungen für diesen Ansatz gibt oder ob es nur eine Lösung für ein nicht existierendes Problem ist – wir wissen es nicht. Vielleicht haben Sie als Leser kreative Ideen, was sich mit *GetAwaiter()*-Erweiterungsmethoden sinnvoll umsetzen ließe? Lassen Sie es uns wissen.

Den Quellcode zu den Beispielen finden Sie bei GitHub [2]. Inspiriert wurde dieser Artikel übrigens von einem YouTube-Video von Nick Chapsas mit dem lustigen Titel „How to await ANYTHING in C#" [3]. Also, wenn das nicht Anregung genug ist, es selbst mal auszuprobieren. Daher hier noch eine Übungsaufgabe: Bringen Sie doch mal Folgendes zum Laufen:

```
await "nächste Ausgabe der dotnetpro";
```

Viel Erfolg! ■

[1] TaskAwaiter Struct, www.dotnetpro.de/SL2306Await1
[2] Jürgen Fuchs, Quellcode zu den Beispielen bei GitHub, www.dotnetpro.de/SL2306Await2
[3] Nick Chapsas, How to await ANYTHING in C#, www.dotnetpro.de/SL2306Await3

Listing 2: GetAwaiter() als Erweiterungsmethode

```
public class MagicDelay
{
  private readonly TimeSpan timeSpan;
  public TimeSpan TimeSpan => timeSpan;

  private MagicDelay(TimeSpan timeSpan)
  {
    this.timeSpan = timeSpan;
  }

  public static MagicDelay Milliseconds(
    int milliseconds)
  {
    return new MagicDelay(TimeSpan.FromMilliseconds(
      milliseconds));
  }
}

public static class MagicExtensions
{
  public static TaskAwaiter GetAwaiter(
    this MagicDelay magicDelay)
  {
    return Task.Delay(magicDelay.TimeSpan).
      GetAwaiter();
  }
}
```

Joachim Fuchs
Dr. Joachim Fuchs ist begeistert von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte sind XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode	A2306Await