Async await

Wie funktionier await?

Ein Thread ist wie eine Pipeline, die einen bestimmten Code in die CPU gibt.

Mal angenommen Thread 1 (Ui Thread) will etwas downloaden und dann auf der UI darstellen. Während des Http Requests ist die Ui gesperrt. Weil der Thread 1 darauf wartet, bis der Server geantwortet hat.

Mit await wird ein freier Thread genommen und der macht den Request. Thread 1 wird nun freigegeben. Der Benutzer kann z.B. weiter die Ui wie gewohnt nutzen.

Wenn Thread 2 fertig ist, sagt er, ich bin fertig und möchte zu dem Thread zurück kehren, der ihn erzeugt hat: Thread 1.

Thread 1 übernimmt und der Codeblock wird weiter ausgeführt.

Wenn es nun egal ist, welcher Thread den Codeblock weiter ausführen soll, kann man
[crayon-6766f185135ef823831032/]
nutzen.

.Result sollte nicht genutzt werden, da im Exceptio Stacktrace Fehlermeldungen wie „MoveNext()“ erscheinen. Diese kommen aus der kompilierten Statemachine. Besser ist es
[crayon-6766f185135f8979492399/]
zu nutzen, da dann unser Typ und unsere Exception zurück gegeben werden.

Aus einer anderen Klasse, aus einem anderen Thread in MainWindow schreiben

Möchte man aus einer anderen Klasse, aus einem anderen Thread etwas in die Mainklasse Steuerelemente schreiben, stößt man auf 2 Probleme:

1. Man kann aus Thread 2 nicht in Thread 1 schreiben

2. Man kann nicht, ohne ein Objekt angelegt zu haben nicht in die Steuerelemente schreiben.

Abhilfe schafft ein kleiner Trick.

MainWindow.cs:
[crayon-6766f18513b5a177656717/]
[crayon-6766f18513b5e796710443/]

meineAndereKlasse.cs:
[crayon-6766f18513b5f254193506/]
Kleine Erweiterung, selbes Prinzip um ein Imagecontrol zu ändern:

MainWindow.cs:
[crayon-6766f18513b61214622011/]
meineAndereKlasse.cs:
[crayon-6766f18513b63019717574/]

***Quelle**: Stackoverflow*

Backgroundworker

Wenn man aus einem Thread heraus eine Zeitaufwändige Operation durchführt, kann dies dazu führen, dass man den Eindruck bekommt, das Fenster wäre eingefroren. Der Grund ist ganz einfach, weil diese Aufgabe im ersten Thread stattfindet und solange dauert, bis es fertig ist. Jetzt stellt .Net den Threading Namespace zur Verfügung womit man für solche Aufgaben einen neuen Thread aufmachen könnte. Eben genau dies tut auch der Backgroundworker nur in einer stark vereinfachter Form.

Backgroundworker einbinden:

1. Namespace hinzufügen:
[crayon-6766f18513d41382471811/]
2. Backgroundworker deklarieren:
[crayon-6766f18513d44369701650/]
3. diesem Objekt events zuweisen:
[crayon-6766f18513d45728063825/]
4. die beiden Methoden implementieren:
[crayon-6766f18513d46435433316/]
5. worker asynchron ausführen lassen:
[crayon-6766f18513d47115977558/]

wünscht man einen Bericht über den derzeitigen Prozess, so muss man auch das event **ProgressChanged** hinzufügen, **ReportProgress(Int32)** in DoWorks ausführen lassen und in

WorkerReportsProgress das Resultat bekommen

Häufigste Problematik die man Anfangs hat ist, dass man nicht auf die Steuerelemente aus dem 1. Thread zugreifen kann. Abhilfe schafft dort
[crayon-6766f18513d49851651289/]