Πανεπιστήμιο Πελοποννήσου
Σχολή Οικονομίας, Διοίκησης και Πληροφορικής
Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Π.Μ.Σ. στην Επιστήμη και Τεχνολογία
Υπολογιστών

Μεταπτυχιακή Εργασία

Σχεδίαση εφαρμογών υψηλής απόκρισης με χρήση μοτίβων ασύγχρονου προγραμματισμού και γλώσσα προγραμματισμού C#

(Responsive applications design using Asynchronous Programming Patterns in C#)

Βλάχου Ευθυμία
ΑΜ: 2022201502003

Επιβλέποντες καθηγητές: Μασσέλος Κωνσταντίνος
Δημητρουλάκος Γρηγόριος

Τρίπολη, Φεβρουάριος 2018

Περιεχόμενα

Εκτεταμένη Περίληψη

Στις σύγχρονες εφαρμογές με γραφικό περιβάλλον εργασίες, και όχι μόνο σε αυτές, υπάρχει έντονη η ανάγκη οι εφαρμογές να αποκρίνονται στις ενέργειες του χρήστη. Αν οι εφαρμογές εκτελούν εργασίες μακράς διάρκειας, είτε αυτές είναι μεγάλης διάρκειας υπολογισμοί είτε επικοινωνία με περιφερειακές μονάδες (δίσκους, δίκτυο, κλπ), ο χρήστης έχει την αίσθηση ότι η εφαρμογή είναι μπλακαρισμένη. Αυτή η κατάσταση αποτελεί πολύ δυσάρεστη κατάσταση στο χρήστη και στην εμπειρία του με την εφαρμογή.

Η λύση σε αυτό το πρόβλημα είναι να εκτελούνται οι λειτουργίες μακράς διάρκειας ασύγχρονα. Υπάρχει ένας διαχωρισμός στην ασύγχρονη λειτουργία, που έχει να κάνει με το αν η λειτουργία είναι compute bound ή I/O bound. Πρότυπα για ασύγχρονο προγραμματισμό στη C# υπάρχουν από τη πρώτη έκδοση της γλώσσας, αλλά τα τελευταία χρόνια έχουν γίνει απλούστερα και πιο αποτελεσματικά. Επίσης απλούστερα έχουν γίνει και τα πρότυπα για αναφορά προόδου της εργασίας, καθώς και της ακύρωσης αυτής.

Extended Abstract

Modern GUI applications, but not only them, need to be highly responsive. If these applications are executing long-running operations, either they are long running computations or communication with peripheral devices (e.g. disks, network interfaces, etc), user feels the application as blocked. This situation is very unpleasant for the user and his experience with the application.

The solution to this problem is the asynchronous execution of the long running operations. We have two kinds of asynchronous operations, compute-bound asynchronous operations and I/O bound asynchronous operations. Asynchronous patterns in C# is not a new feature of the language. They exists from the first version of the language. However, they became less complex and easier to use from developers. Moreover, simpler has also become the progress reporting and cancellation patterns.

Κεφάλαιο 1

Εισαγωγή

1.1 Ασύγχρονος προγραμματισμός

Σε ένα πρόγραμμα λέμε ότι έχουμε ασύγχρονο κώδικα, όταν ο κώδικας αυτός ξεκινάει μία λειτουργία μακράς διάρκειας (long-running operation), αλλά δεν περιμένει για όσο διαρκεί αυτή η λειτουργία. Κατά αυτή την έννοια, είναι ακριβώς το αντίθετο του blocking κώδικα, ο οποίος κάθεται χωρίς να κάνει τίποτα για όσο διαρκεί η λειτουργία.

Μερικές από τις λειτουργίες μακράς διάρκειας είναι:

- Πρόσβαση στο δίκτυο
- Πρόσβαση σε δίσκους
- Καθυστερήσεις για κάποιο χρονικό διάστημα

Όλα έχουν να κάνουν με το ποιο thread τρέχει το κώδικα. Σε όλες τις γνωστές γλώσσες προγραμματισμού, ο κώδικας τρέχει σε ένα thread του λειτουργικού συστήματος. Εάν αυτό το thread συνεχίζει να κάνει άλλες ενέργειες ενώ είναι σε εξέλιξη κάποια λειτουργία μακράς διάρκειας, τότε ο κώδικας λέμε ότι είναι ασύγχρονος. Εάν το thread είναι ακόμα στον κώδικά μας, αλλά δεν κάνει καμία λειτουργία, είναι blocked, και λέμε ότι έχουμε γράψει blocked κώδικα.

Η δυσκολία με τον ασύγχρονο κώδικα είναι ότι, πολύ συχνά, θέλουμε να ξέρουμε πότε μία λειτουργία έχει τελειώσει. Όταν γράφουμε blocking κώδικα αυτό είναι εύκολο: απλώς γράφουμε άλλη μια γραμμή κώδικα αμέσως μετά από την λειτουργία μακράς διάρκειας. Στον ασύγχρονο κώδικα αυτό είναι φυσικά αδύνατο, γιατί αυτή η γραμμή κώδικα θα εκτελεστεί άμεσα και πριν ολοκληρωθεί η λειτουργία μακράς διάρκειας.

Για να λύσουμε αυτό το πρόβλημα, έχουμε επινοήσει διάφορα μοτίβα (patterns), έτσι ώστε να τρέχουν κάποιον κώδικα όταν μία background λειτουργία ολοκληρωθεί:

- Γράφοντας τον κώδικα μέσα στη background λειτουργία, αμέσως μετά από το κύριο σώμα της λειτουργίας.
- Με τη σύνδεση με κάποιο event το οποίο ενεργοποιείται μετά την ολοκλήρωση της λειτουργίας.
- Δίνοντας κάποιο delegate ή lamda expression για να εκτελεστεί μετά την ολοκλήρωση της λειτουργίας (*callback*).

Εάν η επόμενη λειτουργία πρέπει να εκτελεστεί σε ένα συγκεκριμένο thread (π.χ. σε ένα Windows Forms ή WPF UI thread), πρέπει να φροντίσουμε να δρομολογήσουμε τη λειτουργία σε αυτό το thread. Όλα αυτά είναι πολύ πολύπλοκα.

1.2 Τα οφέλη του ασύγχρονου προγραμματισμού

Ο ασύγχρονος κώδικας ελευθερώνει το thread από το οποίο ξεκίνησε να εκτελείται. Τα threads καταλαμβάνουν πόρους του συστήματος (resources), και το να χρησιμοποιούμε λιγότερα resources, είναι πάντα καλό. Συχνά, υπάρχει μόνο ένα thread που μπορεί να κάνει μία συγκεκριμένη δουλειά, όπως το UI thread, και αν δεν το

συνειδητοποιήσουμε αυτό γρήγορα, η εφαρμογή μας δε θα αποκρίνεται (unresponsive). Θα μιλήσουμε για αυτό περισσότερο στο επόμενο κεφάλαιο.

Ένα από τα πολύ συναρπαστικά οφέλη του async χαρακτηριστικού, είναι η δυνατότητα που παρέχει για να εκμεταλλευτούμε την παράλληλη επεξεργασία. Με το async μπορούμε να φτιάξουμε το πρόγραμμά μας έτσι ώστε να εκμεταλλεύεται την παράλληλη επεξεργασία, χωρίς ο κώδικας να είναι πολύπλοκος και μη διαχειρίσιμος.

1.3 Τι είναι το async

Στην έκδοση 5.0 της γλώσσας C#, η ομάδα του μεταγλωττιστή στη Microsoft πρόσθεσαν ένα νέο πολύ ισχυρό χαρακτηριστικό.

Έχει τη μορφή δύο νέων keywords:

- async
- await

Συνοδεύεται επίσης και από κάποιες προσθήκες και αλλαγές στο .NET Framework 4.5 που το ενδυναμώνουν και το κάνουν εύχρηστο.

Αυτό το χαρακτηριστικό κάνει τον *ασύγχρονο* προγραμματισμό πολύ πιο εύκολο, εξαλείφοντας την ανάγκη για πολύπλοκα patternss που ήταν απαραίτητα σε προηγούμενες εκδόσεις της C#. Έτσι μπορούμε να γράψουμε ολόκληρα προγράμματα σε ασύγχρονο στυλ.

Στη συνέχεια θα χρησιμοποιούμε τον όρο **ασύγχρονο** για να αναφερόμαστε στο γενικότερο στυλ προγραμματισμού, που έγινε ευκολότερος με την χρήση με το νέο χαρακτηριστικό **async** της C#. Ο ασύγχρονος προγραμματισμός ήταν πάντα εφικτός με την C#, αλλά απαιτούνταν πολύ δουλειά από τη μεριά του προγραμματιστή.

1.4 Τι κάνει το async χαρακτηριστικό

Το χαρακτηριστικό async είναι ένας τρόπος για να εκφράσουμε τι πρέπει να γίνει μετά την ολοκλήρωση μιας λειτουργίας μακράς διάρκειας.

Μία ασύγχρονη method μετασχηματίζεται από τον μεταγλωττιστή, για να κάνει τον ασύγχρονο κώδικα να μοιάζει πολύ με το “blocking” ισοδύναμό του κώδικα. Παρακάτω φαίνεται μία blocking method η οποία κάνει download μία ιστοσελίδα.

```
private void DumpWebPage(string uri)
{
    WebClient webClient = new WebClient();
    string page = webClient.DownloadString(uri);
    Console.WriteLine(page);
}
```

Παρακάτω φαίνεται η ισοδύναμη method χρησιμοποιώντας το async χαρακτηριστικό.

```
private async void DumpWebPageAsync(string uri)
{
    WebClient webClient = new WebClient();
    string page = await webClient.DownloadStringTaskAsync(uri);
    Console.WriteLine(page);
}
```

Οι δύο παραπάνω methods μοιάζουν πολύ. Αλλά ουσιαστικά διαφέρουν πάρα πολύ.

Η method είναι δηλωμένη σαν **async**. Αυτή η δήλωση είναι απαραίτητη για κάθε method που χρησιμοποιούν το keyword **await**. Επίσης στο όνομα της method έχουμε προσθέσει την κατάληξη **Async** για να ακολουθήσουμε το naming convention.

Το ενδιαφέρον σημείο είναι το keyword **await**. Όταν ο μεταγλωτιστής το συναντήσει κόβει τη method στα δύο. Το τι ακριβώς κάνει είναι πολύ πολύπλοκο, αλλά απλοποιημένα αυτό που κάνει είναι τα εξής:

1. Οτιδήποτε υπάρχει μετά το **await** μετακινείται σε ξεχωριστή method.
2. Χρησιμοποιούμε μία νέα έκδοση της *DownloadString* που ονομάζεται *DownloadStringTaskAsync*. Αυτή κάνει το ίδιο με την πρώτη, αλλά είναι ασύγχρονη.
3. Αυτό σημαίνει ότι η ξεχωριστή method που δημιουργήθηκε στο βήμα 1, θα εκτελεστεί μόλις τελειώσει η ασύγχρονη method στο βήμα 2.
4. Όταν το download τελειώσει, θα κληθεί η ξεχωριστή method που δημιουργήθηκε και περιέχει τον κώδικα μετά το await. **Σχηματικά** γίνεται το παρακάτω (ο παρακάτω κώδικας δεν εκτελείται και είναι απλά ενδεικτικός):

```
private void DumpWebPageAsync(string uri)
{
    WebClient webClient = new WebClient();
    webClient.DownloadStringTaskAsync(uri); // < -magic(SecondHalf);
}
private void SecondHalf(string awaitedResult)
{
    string page = awaitedResult;
    Console.WriteLine(page);
}
```

Τι συμβαίνει στο calling thread όταν τρέχει αυτόν τον κώδικα. Όταν φτάσει στην κλήση του `DownloadStringTaskAsync`, ξεκινάει το download. Αλλάό όχι σε αυτό το thread. Σε αυτό το thread, φτάνουμε στο τέλος της method και επιστρέφουμε. Το τι θα κάνει το thread είναι στο χέρι του caller thread. Εάν είναι ένα UI thread, θα επιστρέψει και θα συνεχίσει να επεξεργάζεται τις ενέργειες του χρήστη. Διαφορετικά, ίσως απελευθερώσει τα resources του. Αυτό σημαίνει ότι έχουμε γράψει ασύγχρονο κώδικα.

1.5 Το Async δεν λύνει όλα τα προβλήματα.

Το async χαρακτηριστικό έχει σκόπιμα σχεδιαστεί έτσι ώστε να μοιάζει όσο περισσότερο γίνεται με τον blocking κώδικα. Μπορούμε να δουλέψουμε με λειτουργίες μακράς διάρκειας ή με απομακρυσμένες λειτουργίες σχεδόν σαν να ήταν τοπικές και γρήγορες, διατηρώντας όμως τα οφέλη στην επίδοση από την ασύγχρονη κλήση τους.

Παρόλα αυτά, δεν είναι σχεδιασμένο για να μας αφήνει να ξεχάσουμε ότι υπάρχουν background λειτουργίες και callbacks που συμβαίνουν. Πρέπει να είμαστε προσεκτικοί με αρκετά πράγματα τα οποία συμπεριφέρονται διαφορετικά όταν χρησιμοποιούμε το async, συμπεριλαμβανομένων των παρακάτω:

- Exception και try…catch…finally blocks
- Return values των method
- Threads και context
- Performance

Χωρίς την κατανόηση του τι ακριβώς συμβαίνει, το πρόγραμμά μας θα αστοχήσει με τρόπους που θα μας εκπλήξουν και δεν θα μπορούμε να καταλάβουμε τα μηνύματα σφαλμάτων ή τον debugger για να μπορέσουμε να το φτιάξουμε.

Κεφάλαιο 2

Γιατί τα προγράμματα χρειάζεται να είναι ασύγχρονα

Ο ασύγχρονος προγραμματισμός είναι απαραίτητος και χρήσιμος, αλλά η αιτία για την οποία είναι σημαντικός ποικίλει, ανάλογα με τι είδος εφαρμογή γράφουμε. Μερικά από τα οφέλη υπάρχουν σε κάθε είδος εφαρμογής, αλλά πιθανόν να είναι περισσότερο σημαντικά σε είδη εφαρμογών που πιθανόν να μη γράψουμε ποτέ.

2.1 Εφαρμογές Desktop GUI

Οι εφαρμογές desktop έχουν μία κύρια απαίτηση όσον αφορά τις επιδόσεις. Ο χρήστης πρέπει να έχει την αίσθηση της απόκρισης. Μελέτες πάνω στο χώρο του Human Computer Interaction (HCI) έχουν δείξει ότι οι χρήστες δεν προσέχουν μία αργή εφαρμογή, εφόσον η διεπαφή αποκρίνεται, και κατά προτίμηση έχει και μία κινούμενη μπάρα προόδου (animated progress indicator).

Οι χρήστες απογοητεύονται όταν η εφαρμογή «παγώνει». Τα «παγώματα» αυτά συνήθως είναι το αποτέλεσμα προγραμμάτων που δεν μπορούν να ανταποκριθούν στις ενέργειες των χρηστών κατά τη διάρκεια μιας λειτουργίας μακράς διάρκειας, είτε αυτή είναι ένας αργός υπολογισμός, είτε μία λειτουργία εισόδου/εξόδου (I/O operation), όπως για παράδειγμα μία πρόσβαση στο δίκτυο.

Όλα τα UI frameworks, συμπεριλαμβανομένων των παρακάτω, που μπορεί να χρησιμοποιήσουμε, λειτουργούν με ένα μόνο UI thread:

- WinForms
- WPF
- Silverlight

Αυτό το UI thread είναι το μόνο που μπορεί να ελέγχει τα περιεχόμενα ενός παραθύρου. Είναι επίσης το μόνο thread που ελέγχει για ενέργειες χρηστών και αποκρίνεται σε αυτές. Εάν το thread είναι απασχολημένο ή μπλοκαρισμένο για περισσότερο από μερικά millisecond, οι χρήστες θα αισθάνονται ότι η εφαρμογή είναι αργή.

Ασύγχρονος κώδικας, ακόμα και αν είναι γραμμένος χειροκίνητα, σημαίνει ότι το UI thread μπορεί να επιστρέψει στην κύρια λειτουργία του. Να ελέγχει δηλαδή τη *message queue* για ενέργειες/αιτήματα των χρηστών, και να αποκρίνεται σε αυτά. Μπορεί επίσης να εμφανίζει διάφορα οπτικά εφέ (όπως progress animations, mouse hover animations, κλπ), τα οποία είναι πολύ σημαντικά και δίνουν στο χρήστη μία καλή εντύπωση απόκρισης της εφαρμογής.

2.2 Web Application Server Code

Οι ASP .NET web servers δεν έχουν το ίδιο αυστηρό όριο του ενός thread όπως έχουν οι desktop GUI εφαρμογές. Φυσικά και πάλι υπάρχουν οφέλη από τη χρήση του ασύγχρονου προγραμματισμού. Λειτουργίες μακράς διάρκειας, κυρίως αιτήματα προς βάσεις δεδομένων, είναι πολύ κοινές σε κώδικα web εφαρμογών.

Ανάλογα με την έκδοση του IIS, θα υπάρχει ένα όριο είτε στο μέγιστο αριθμό thread που χρησιμοποιούνται για την επεξεργασία web αιτημάτων, είτε στο συνολικό αριθμό ταυτόχρονων αιτήσεων που μπορούν να εξυπηρετηθούν. Εάν στα αιτήματα ο περισσότερος χρόνος σπαταλάτε περιμένοντας για ένα μία αίτηση σε βάση δεδομένων, ίσως είναι καλή ιδέα να αυξήσουμε τον αριθμό των ταυτόχρονων αιτημάτων.

Όταν ένα thread είναι μπλοκαρισμένο, περιμένοντας για κάτι, δεν χρησιμοποιεί χρόνο από τη CPU. Παρόλα αυτά, αυτό δε σημαίνει ότι δε χρησιμοποιεί καθόλου τα resources του server. Στη πραγματικότητα, τα threads προκαλούν δύο σημαντικά overheads, ακόμα και όταν είναι μπλοκαρισμένα:

- *Memory*: Κάθε managed thread καταλαμβάνει περίπου 1MB μνήμης στα Windows. Δεν υπάρχει πρόβλημα αν έχουμε μερικές δεκάδες threads, αλλά μπορεί εύκολα να δημιουργηθεί πρόβλημα αν έχουμε μερικές εκατοντάδες threads. Εάν η μνήμη γίνεται swap στον σκληρό δίσκο, η επαναφορά των thread γίνεται αργή.
- *Scheduler overhead*: Ο scheduler του λειτουργικού συστήματος είναι υπεύθυνος για να επιλέξει ποιο thread θα εκτελεστεί, σε ποια CPU και πότε. Ακόμα και όταν τα threads είναι μπλοκαρισμένα, ο scheduler πρέπει να τα λαμβάνει υπ' όψιν, για να διαπιστώσει εάν πέρασαν σε κατάσταση μη μπλοκαρισμένη. Αυτό καθυστερεί το context switching και μπορεί να καθυστερήσει ολόκληρο το σύστημα.

Όλα τα παραπάνω επιβαρύνουν το σύστημα, αυξάνοντας τις καθυστερήσεις και μειώνοντας τη διακίνηση των δεδομένων.

Πρέπει να θυμόμαστε το εξής: Το κύριο χαρακτηριστικό του ασύγχρονου κώδικα είναι ότι το thread που ξεκίνησε μία λειτουργία μακράς διάρκειας, είναι ελεύθερο να κάνει άλλα πράγματα. Στη περίπτωση του ASP .NET κώδικα, το thread είναι από το thread pool, οπότε επιστρέφει στο thread pool κατά τη διάρκεια της λειτουργίας μακράς διάρκειας. Μπορεί πλέον να επεξεργαστεί άλλα αιτήματα, οπότε χρειάζονται λιγότερα threads για να επεξεργαστούν τον ίδιο αριθμό αιτημάτων.

2.3 Universal Windows Platform (UWP), Windows 8, Silverlight

Οι σχεδιαστές του Silverlight ήξεραν τα οφέλη του ασύγχρονου κώδικα στις UI εφαρμογές. Έτσι αποφάσισαν να ενθαρρύνουν τον κόσμο να γράφει ασύγχρονο κώδικα. Αυτό το πέτυχαν αφαιρώντας τα περισσότερα σύγχρονα APIs από το Framework. Για παράδειγμα τα Web requests υπάρχουν μόνο σαν ασύγχρονες κλήσεις.

Ο ασύγχρονος κώδικας είναι μεταδοτικός. Εάν καλούμε ένα ασύγχρονο API, ο κώδικάς μας καταλήγει ασύγχρονος. Οπότε στο Silverlight *πρέπει* να γράφουμε ασύγχρονο κώδικα-δεν υπάρχει άλλη επιλογή. Μπορούμε να έχουμε μία Wait method ή κάτι παρόμοιο για να λειτουργήσει σύγχρονα ένα ασύγχρονο API, μπλοκάροντας το thread καθώς περιμένει να κληθεί πίσω. Όμως με αυτό το τρόπο χάνουμε όλα τα πλεονεκτήματα του ασύγχρονου προγραμματισμού που προαναφέραμε.

Στις mobile εφαρμογές (Windows 10 UWP, Windows 8, Silverlight Windows Phone, κλπ) είναι πολύ σημαντικό να χρησιμοποιούμε ασύγχρονο κώδικα, καθώς τα resource είναι πολύ περιορισμένα. Οπότε η δημιουργία extra threads, μπορεί να έχει

σημαντικό αντίκτυπο στη διάρκεια της μπαταρίας της συσκευής. Σε όλα τα παραπάνω frameworks (UWP, Silverlight, κλπ), για όποια APIs διαρκούν περισσότερο από 50ms για να εκτελεστούν, διατίθενται μόνο ασύγχρονα APIs.

2.4 Παράλληλος κώδικας

Οι υπολογιστές κατασκευάζονται με όλο και περισσότερους πυρήνες, όπου όλοι λειτουργούν ανεξάρτητα από τον άλλο. Τα προγράμματα πρέπει να εκμεταλλεύονται αυτούς τους πυρήνες, αλλά η μνήμη που χρησιμοποιείται από αυτά τα προγράμματα, δεν μπορεί να γράφεται από όλους τους πυρήνες ταυτόχρονα, γιατί τα δεδομένα του ενός θα καταστρέφουν τα δεδομένα του άλλου.

Η κλασσική λύση σε αυτό το πρόβλημα είναι η χρήση locks όποτε πολλαπλοί πυρήνες θέλουν να προσπελάσουν την ίδια μνήμη. Αλλά και αυτή η λύση έχει τα προβλήματά της. Συχνά ο κώδικάς μας μπορεί να παίρνει ένα lock, μετά να κάνει μία κλήση σε μία συνάρτηση ή να ενεργοποιήσει ένα event τα οποία παίρνουν ένα άλλο lock. Αυτό είναι λάθος χρήση και άποψη για τα locks, και σημαίνει ότι τελικά περισσότερα threads καταλήγουν να περιμένουν για locks παρά να κάνουν κάτι χρήσιμο. Σε μερικές περιπτώσεις, δύο threads περιμένουν το ένα για το lock που κρατάει το άλλο, οδηγώντας σε deadlock. Τέτοια bugs είναι δύσκολο να προβλεφθούν και να φτιαχτούν.

Μία από τις πολλά υποσχόμενες λύσεις είναι το *actor* μοντέλο προγραμματισμού. Αυτή είναι μία σχεδίαση όπου κάθε κομμάτι εγγράψιμης μνήμης μπορεί να υπάρχει μόνο μέσα σε ένα *actor*. Ο μόνος τρόπος για να χρησιμοποιήσουμε αυτή τη μνήμη, είναι να στείλουμε μηνύματα στο actor, το οποίο τα επεξεργάζεται ένα τη φορά, και ίσως απαντήσει επίσης με ένα μήνυμα. Αυτό ακριβώς είναι ασύγχρονος προγραμματισμός. Η λειτουργία της ερώτησης σε έναν actor για κάτι είναι μία τυπική ασύγχρονη λειτουργία, επειδή μπορούμε να συνεχίσουμε να κάνουμε άλλα πράγματα, μέχρι να φτάσει το απαντητικό μήνυμα.

2.5 Ένα παράδειγμα

Η παρακάτω method κατεβάζει ένα favicon από ένα website και το προσθέτει σε ένα WPF WrapPanel :

```
private void AddAFavicon(string domain)
{
   WebClient webClient = new WebClient();
   byte[] bytes = webClient.DownloadData("http://" + domain + "/favicon.ico");
   Image imageControl = MakeImageControl(bytes);
   m_WrapPanel.Children.Add(imageControl);
}
```

Παρατηρούμε ότι η παραπάνω υλοποίηση είναι εντελώς σύγχρονη. Το thread μπλοκάρεται ενώ κατεβαίνει το icon. Επίσης ο χρήστης θα παρατηρήσει ότι το παράθυρο δεν αποκρίνεται για μερικά δευτερόλεπτα αφότου πατήσουμε το κουμπί για

download. Όπως γνωρίζουμε, αυτό συμβαίνει επειδή το UI thread είναι μπλοκαρισμένο καθώς κατεβαίνουν τα εικονίδια, συνεπώς δεν μπορεί να επιστρέψει για να επεξεργαστεί ενέργειες του χρήστη.

Κεφάλαιο 3

Γράφοντας Ασύγχρονο Κώδικα Χειροκίνητα

Σε αυτό το κεφάλαιο θα μιλήσουμε για τη συγγραφή ασύγχρονου κώδικα χωρίς τη βοήθεια της C# 5.0 και του async. Πρόκειται για τεχνικές τις οποίες δεν θα χρειαστεί ποτέ να χρησιμοποιήσουμε, αλλά είναι σημαντικές για να μας βοηθήσουν να καταλάβουμε τι συμβαίνει στο παρασκήνιο. Για αυτό το λόγο αυτό, θα τις εξετάσουμε πολύ συνοπτικά, επισημαίνοντας μόνο τα σημεία που είναι χρήσιμα για την κατανόηση.

3.1 Μερικά ασύγχρονα patterns της .NET

Όπως αναφέραμε τα UWP και Silverlight παρέχουν μόνο ασύγχρονες εκδόσεις των APIs όπως π.χ. την πρόσβαση στο web. Παρακάτω ακολουθεί ένα παράδειγμα, για το πως θα μπορούσαμε να κατεβάσουμε μία web page και να την εμφανίσουμε.

```
private void DumpWebPage(Uri uri)
{
    WebClient webClient = new WebClient();
    webClient.DownloadStringCompleted += OnDownloadStringCompleted;
    webClient.DownloadStringAsync(uri);
}

private void OnDownloadStringCompleted(object sender, DownloadStringCompletedEventArgs eventArgs)
{
    m_TextBlock.Text = eventArgs.Result;
}
```

Αυτός ο τύπος API ονομάζεται *Event-based Asynchronous Pattern* (EAP). Η λογική του είναι ότι αντί για μία σύγχρονη method που θα κατεβάσει την ιστοσελίδα, η οποία θα μπλοκάρει το thread μέχρι να τελειώσει, χρησιμοποιείται μία method και ένα event. Η method μοιάζει με την αντίστοιχη σύγχρονη έκδοσή της, με τη διαφορά ότι επιστρέφει void. Το event περιέχει ένα ειδικά ορισμένο EventArgs τύπο, ο οποίος περιέχει την τιμή που επιστρέφεται.

Συνδεόμαστε με το event αμέσως πριν καλέσουμε τη method. Η method επιστρέφει αμέσως (ακριβώς επειδή αυτό σημαίνει ασύγχρονος κώδικας). Αργότερα, κάποια στιγμή στο μέλλον, το event θα ενεργοποιηθεί και εμείς μπορούμε να συνεργαστούμε με αυτό.

Αυτό το μοτίβο (pattern) είναι προφανώς πολύ πολύπλοκο για να το χρησιμοποιήσουμε, όχι μόνο γιατί πρέπει να σπάσουμε σε δύο methods αυτό που θα έπρεπε κανονικά να είναι μία απλή σειρά από συνεχόμενες εντολές. Πάνω από αυτό, το γεγονός ότι συνδεόμαστε με ένα event, προσθέτει πολυπλοκότητα. Αν συνεχίσουμε να χρησιμοποιούμε το ίδιο στιγμιότυπο της WebClient για ένα άλλο αίτημα, ίσως να μην περιμένουμε ότι μπορεί να είμαστε ακόμα συνδεδεμένοι με το original event και ότι ο handler θα τρέξει ξανά.

Ένα άλλο ασύγχρονο pattern που υπάρχει στη .NET είναι σχετικό με το `IAsyncResult` interface. Ένα παράδειγμα είναι η method `BeginGetHostAddresses`, η οποία κάνει DNS lookup μιας IP διεύθυνσης για ένα ένα hostname. Αυτό το design απαιτεί δύο methods, μία που θα λέγεται `Begin`*`Method`* η οποία ξεκινάει τη λειτουργία, και μία που θα λέγεται `End`*`Method`* την οποία χρησιμοπούμε στο callback για πάρουμε το αποτέλεσμα.

```
private void LookupHostName()
```

```
{
    object unrelatedObject = "hello";
    Dns.BeginGetHostAddresses("oreilly.com", OnHostNameResolved, unrelatedObject);
}

private void OnHostNameResolved(IAsyncResult ar)
{
    object unrelatedObject = ar.AsyncState;
    IPAddress[] addresses = Dns.EndGetHostAddresses(ar);
    // Do something with addresses
    ...
}
```

Αυτό το design pattern τουλάχιστον δεν τα προβλήματα των event handlers του προηγούμενου design.

Και τα δύο αυτά όμως patterns απαιτούν το σπάσιμο της διαδικασίας σε δύο methods. Το `IAsyncResult` pattern μας επιτρέπει να περάσομε κάτι από τη πρώτη method στην δεύτερη, όπως κάναμε στο παράδειγμα με το string "`hello`". Αλλά ο τρόπος που το κάνει είναι αρκετά πολύπλοκος, απαιτώντας να περάσουμε κάτι ακόμα και αν δεν το θέλουμε, και αναγκάζοντάς μας να το κάνουμε cast από `object`. Το EAP επίσης υποστηρίζει το πέρασμα ενός object με ένα παρόμοια πολύπλοκο τρόπο.

Το πέρασμα context μεταξύ των method είναι ένα γενικότερο πρόβλημα στα ασύγχρονα patterns. Το lambda expression είναι μία λύση και μπορούμε να το χρησιμοποιήσουμε σε όλες αυτές τις περιπτώσεις.

3.2 Ένα απλούστερο ασύγχρονο pattern

Ο απλούστερος κώδικας με ασύγχρονη συμπεριφορά, χωρίς τη χρήση του async, περιλαμβάνει το πέρασμα μιας callback σαν παράμετρο στη method:

```
void GetHostAddress(string hostName, Action<IPAddress> callback)
```

Είναι ευκολότερο στη χρήση από τα άλλα patterns.

```
private void LookupHostName()
{
    GetHostAddress("oreilly.com", OnHostNameResolved);
}
private void OnHostNameResolved(IPAddress address)
{
    // Do something with address
    ...
}
```

Όπως αναφέραμε προηγουμένως, αντί να χρησιμοποιήσουμε δύο methods μπορούμε να χρησιμοποιήσουμε μία ανώνυμη method ή ένα lambda expression για callback. Αυτό έχει το σημαντικό όφελος, ότι μπορούμε να προσπελάσουμε μεταβλητές από το πρώτο μέρος της method.

```
private void LookupHostName()
```

```
{
    int aUsefulVariable = 3;
    GetHostAddress("oreilly.com", address =>
    {
        // Do something with address and aUsefulVariable
        ...
    });
}
```

Το lambda είναι λίγο δύσκολο στην ανάγνωση, και συχνά εάν χρησιμοποιούμε πολλαπλά ασύγχρονα APIs, θα χρειαστούμε πολλά lambdas το ένα μέσα στο άλλο. Ο κώδικας γίνεται έτσι δύσκολος στην ανάγνωση και δύσκολα μπορούμε να δουλέψουμε με έναν τέτοιο κώδικα.

Το μειονέκτημα αυτής της απλής προσέγγισης είναι ότι τυχών exceptions δεν γίνονται πλέον throw στον caller. Στα patterns που χρησιμοποιούνται στη .NET, η κλήση στην `End*MethodName*` ή η λήψη αποτελέσματος με την `Result` property θα κάνει rethrow την exception, έτσι ώστε να μπορεί να το χειριστεί η αρχική method. Αντί αυτού, μπορεί να καταλήξουν σε λάθος μέρος, ή να μη χειριστούν από κανέναν πουθενά.

3.3 Εισαγωγή στα Task

Η Task Parallel Library (TPL) παρουσιάστηκε στην έκδοση 4.0 του .NET Framework. Η πιο σημαντική class είναι η `Task`, η οποία αναπαριστά μία λειτουργία σε εξέλιξη. Η generic έκδοση `Task<T>`, δρα σαν μία υπόσχεση μιας τιμής (τύπου `T`) η οποία θα είναι διαθέσιμη αργότερα, όταν η λειτουργία τελειώσει.

Το async χαρακτηριστικό της C# 5.0 χρησιμοποιεί την `Task` εκτεταμένα, όπως θα δούμε αργότερα. Παρόλα αυτά, ακόμα και χωρίς το async, μπορούμε να χρησιμοποιήσουμε την `Task` και κυρίως την `Task<T>` για να γράψουμε ασύγχρονα προγράμματα. Για να το κάνουμε αυτό, ξεκινάμε μία λειτουργία η οποία θα επιστρέψει ένα `Task<T>`. Μετά χρησιμοποιούμε την `ContinueWith` method για να κάνουμε register το callback.

```
private void LookupHostName()
{
    Task<IPAddress[]> ipAddressesPromise = Dns.GetHostAddressesAsync("oreilly.com");
    ipAddressesPromise.ContinueWith(_ =>
        {
            IPAddress[] ipAddresses = ipAddressesPromise.Result;
            // Do something with address
            ...
        });
}
```

Το πλεονέκτημα της `Task` είναι ότι απαιτείται μόνο μία method στη `Dns` κλήση, κάνοντας το API καθαρότερο. Όλο το logic που έχει σχέση με την ασύγχρονη συμπεριφορά της κλήσης, μπορεί να είναι μέσα στην `Task` class, έτσι ώστε να μη εμφανίζονται ξανά και ξανά σε κάθε ασύγχρονη method. Αυτό το logic μπορεί να κάνει πολύ σημαντικά πράγματα, όπως να ασχολείται με exceptions και

`SynchronizationContexts`. Αυτά, όπως θα δούμε παρακάτω, είναι χρήσιμα στο να τρέξουμε ένα callback σε ένα συγκεκριμένο thread (π.χ. στο GUI thread).

3.4 Το πρόβλημα της Manual Asynchrony

Όπως είδαμε, υπάρχουν διάφοροι τρόποι για να υλοποιήσουμε ασύγχρονα προγράμματα. Μερικοί είναι κομψότεροι από άλλους, αλλά είδαμε πως όλοι μοιράζονται ένα ελάττωμα. Η διαδικασία που θέλουμε να γράψουμε πρέπει να χωριστεί σε δύο methods: την πραγματική method και το callback. Με τη χρήση ανώνυμης method ή lambda για το callback, μετριάζεται κάπως αυτό το πρόβλημα, αλλά ο κώδικας γίνεται αρκετά «τραχύς» και δύσκολος για να τον ακολουθήσει κανείς.

Υπάρχει και άλλο ένα πρόβλημα. Μιλήσαμε για methods που κάνουν μία ασύγχρονη κλήση, αλλά τι συμβαίνει αν θέλουμε να κάνουμε περισσότερες από μία; Ακόμη χειρότερα, τι συμβαίνει αν θέλουμε να καλέσουμε ασύγχρονες methods μέσα σε ένα loop; Η μόνη μας επιλογή είναι μία αναδρομική method, η οποία είναι πολύ δυσκολότερο να διαβαστεί από ένα απλό loop.

```
private void LookupHostNames(string[] hostNames)
{
   LookUpHostNamesHelper(hostNames, 0);
}

private static void LookUpHostNamesHelper(string[] hostNames, int i)
{
   Task<IPAddress[]> ipAddressesPromise = Dns.GetHostAddressesAsync(hostNames[i]);
   ipAddressesPromise.ContinueWith(_ =>
       {
           IPAddress[] ipAddresses = ipAddressesPromise.Result;

           //  Do something with address
           ...

           if (i + 1 < hostNames.Length)
           {
               LookUpHostNamesHelper(hostNames, i + 1);
           }
       });
}
```

Ένα επιπλέον πρόβλημα που προκαλείται από τον manual ασύγχρονο προγραμματισμό σε οποιοδήποτε από τα αναφερθέντα στυλ, είναι η κατανάλωση του ασύγχρονου κώδικα που γράψαμε. Εάν γράψουμε ασύγχρονο κώδικα, και μετά θέλουμε να τον χρησιμοποιήσουμε κάπου αλλού στο πρόγραμμά μας, θα πρέπει να παρέχουμε ένα ασύγχρονο API για χρήση. Εάν η κατανάλωση ενός τέτοιου είδους ασύγχρονου API φαίνεται μία φορά δύσκολη και πολύπλοκη, η παροχή ενός τέτοιου είναι δύο φορές περισσότερο. Και επειδή ο ασύγχρονος κώδικας είναι μεταδοτικός, όχι μόνο έχουμε έχουμε να κάνουμε με ασύγχρονα APIs, αλλά το ίδιο πρέπει να κάνει και ο caller των APIs, και ο caller των caller, μέχρι που τελικά όλο το πρόγραμμα καταλήγει πολύπλοκο και πραγματικά πολύ μπερδεμένο.

Κεφάλαιο 4

Γράφοντας Async Methods

Στο σημείο αυτό που πλέον ξέρουμε πόσο καλός και χρήσιμος είναι ο ασύγχρονος προγραμματισμός, αλλά και πόσο δύσκολος είναι στο γράψιμο, είναι ώρα να δούμε το async χαρακτηριστικό της C# 5.0. Όπως είδαμε νωρίτερα, μία method χαρακτηρισμένη ως `async` επιτρέπεται να περιέχει το `await` keyword.

```
private async void DumpWebPageAsync(string uri)
{
    WebClient webClient = new WebClient();
    string page = await webClient.DownloadStringTaskAsync(uri);
    Console.WriteLine(page);
}
```

Το `await` σε αυτό το παράδειγμα μετασχηματίζει τη method, έτσι ώστε να σταματάει προσωρινά κατά τη διάρκεια του download, και αργότερα να επανέρχεται και να συνεχίζει όταν το download τελειώσει. Αυτός ο μετασχηματισμός κάνει τη method ασύγχρονη.

4.1 Μετατροπή του παραδείγματος DownloadIcon σε Async

Θα μετατρέψουμε το προηγούμενο παράδειγμα με τα icons έτσι ώστε να κάνει χρήση του async.

Η σημαντική method είναι η `AddFavicon`, η οποία κατεβάζει το icon και μετά το προσθέτει στο UI. Θέλουμε να κάνουμε αυτή τη method ασύγχρονη, έτσι ώστε το UI thread να είναι ελεύθερο να απαντά στις ενέργειες των χρηστών κατά τη διάρκεια του download. Το πρώτο βήμα είναι να προσθέσουμε το `async` keyword στη method.

Μετά πρέπει να περιμένουμε να τελειώσει το download χρησιμοποιώντας το `await` keyword. Ως προς το συντακτικό της γλώσσας C#, το `await` λειτουργεί ως unary operator, όπως το ! not operator, ή το `(type)` cast operator. Τοποθετείται στα αριστερά μιας expression και σημαίνει *περίμενε για αυτή την expression asynchronously*.

Τέλος η κλήση της `DownloadData` πρέπει να αλλάξει στην ασύγχρονη έκδοσή της `DownloadDataTaskAsync`.

Μία async method δεν είναι αυτομάτως ασύγχρονη. Οι async methods απλώς κάνουν ευκολότερη τη κατανάλωση άλλων ασύγχρονων method. Ξεκινούν να τρέχουν σύγχρονα, μέχρι να καλέσουν μία ασύγχρονη method και να την περιμένουν. Όταν το κάνουν αυτό έχουν απαραιτήτως γίνει ασύγχρονες. Μερικές φορές μία async method δεν περιμένει τίποτα, οπότε στη περίπτωση αυτή τρέχει σύγχρονα.

```
private async void AddAFavicon(string domain)
{
   WebClient webClient = new WebClient();
   byte[] bytes = await webClient.DownloadDataTaskAsync("http://"+domain+"/favicon.ico");
   Image imageControl = MakeImageControl(bytes);
   m_WrapPanel.Children.Add(imageControl);
}
```

Ας συγκρίνουμε αυτή τη method με τις άλλες δύο εκδόσεις που είδαμε νωρίτερα. Μοιάζει περισσότερο με τη σύγχρονη έκδοση. Δεν υπάρχει extra method, μόνο λίγος

extra κώδικας στην ίδια δομή. Παρόλα αυτά, συμπεριφέρεται πολύ περισσότερο σαν την ασύγχρονη έκδοση που γράψαμε manually νωρίτερα.

4.2 Task και await

Ας δούμε λίγο πως λειτουργεί το `await`. Παρακάτω φαίνεται το signature της `WebClient.DownloadStringTaskAsync` method:

```
Task<string> DownloadStringTaskAsync(string address)
```

Ο τύπος που επιστρέφεται είναι `Task<string>`. Όπως έχουμε ήδη πει ένα `Task` αναπαριστά μία λειτουργία σε εξέλιξη, και η υποκλάση `Task<T>` αναπαριστά μία λειτουργία η οποία θα έχει ένα αποτέλεσμα τύπου *T* κάποια στιγμή στο μέλλον. Μπορούμε να φανταστούμε την `Task<T>` ως μία υπόσχεση για ένα *T* όταν η λειτουργία μακράς διάρκειας ολοκληρωθεί.

Οι `Task` και `Task<T>` μπορούν και οι δύο να αναπαραστήσουν ασύγχρονες λειτουργίες, και επίσης και οι δύο έχουν τη δυνατότητα να καλέσουν πίσω τον κώδικά μας όταν η λειτουργία τελειώσει. Για να χρησιμοποιήσουμε αυτή τη δυνατότητα manually, χρησιμοποιούμε την `ContinueWith` method που διαθέτουν και οι δύο, έτσι ώστε να περάσουμε ένα delegate προς εκτέλεση όταν τελειώσει η λειτουργία μακράς διάρκειας. Το `await` χρησιμοποιεί την ίδια δυνατότητα για να εκτελέσει το υπόλοιπο του ασύγχρονου κώδικα με τον ίδιο τρόπο.

Εάν τοποθετήσουμε ένα `await` σε ένα `Task<T>`, τότε γίνεται μία *await expression*, και όλη η expression έχει τύπο *T*. Αυτό σημαίνει ότι μπορούμε να εκχωρήσουμε σε μία μεταβλητή το αποτέλεσμα της αναμενόμενης expression, και να το χρησιμοποιήσουμε στο υπόλοιπο της method, όπως είδαμε στα παραδείγματα. Παρόλα αυτά, όταν περιμένουμε με await μία non-generic `Task`, έχουμε μία *await statement* η οποία δε μπορεί να εκχωρηθεί πουθενά, ακριβώς όπως μία κλήση σε μία `void` method. Αυτό ισχύει και βγάζει νόημα, γιατί το `Task` δεν υπόσχεται μία τιμή ως αποτέλεσμα, αλλά απλά αναπαριστά τη λειτουργία ως έχει.

```
await smtpClient.SendMailAsync(mailMessage);
```

Μπορούμε να σπάσουμε την await expression, έτσι ώστε να μπορούμε να προσπελάσουμε την Task απευθείας, ή να κάνουμε κάτι άλλο πριν την περιμένουμε με await.

```
Task<string> myTask = webClient.DownloadStringTaskAsync(uri);
// Do something here
string page = await myTask;
```

Είναι σημαντικό να καταλάβουμε πλήρως τις συνέπειες που έχει το παραπάνω. Η method `DownloadStringTaskAsync` εκτελείται πρώτη. Αρχίζει να εκτελείται σύγχρονα, στο τρέχον thread, και εφόσον έχει αρχίσει το download, επιστρέφει ένα `Task<string>` επίσης στο ίδιο thread. Μόνο αργότερα όταν περιμένουμε με `await` το `Task<string>` κάνει κάτι ειδικό ο μεταγλωτιστής. Το ίδιο ισχύει εάν γράψουμε την `await` στην ίδια γραμμή με την κλήση της ασύγχρονης.

Η λειτουγία μακράς διάρκειας ξεκινάει εφόσον έγινε κλήση στην `DownloadStringTaskAsync`, πράγμα το οποίο μας δίνει έναν πολύ απλό τρόπο να εκτελέσουμε πολλαπλές ασύγχρονες λειτουργίες ταυτόχρονα. Μπορούμε απλά να

ξεκινήσουμε πολλές ασύγχρονες λειτουργίες, να κρατήσουμε τα Tasks, και μετά να περιμένουμε όλα αυτά με await.

```
Task<string> firstTask = webClient1.DownloadStringTaskAsync("http://oreilly.com");
Task<string> secondTask = webClient2.DownloadStringTaskAsync("http://simpletalk.com");
string firstPage = await firstTask;
string secondPage = await secondTask;
```

Τα παραπάνω είναι ένας επικίνδυνος τρόπος για να περιμένουμε πολλαπλά Tasks, εάν τύχει να δημιουργήσουν exceptions. Εάν και οι δύο λειτουργίες δημιουργήσουν exception, το πρώτο `await` θα προωθήσει το exception του, που σημαίνει ότι η `secondTask` δεν θα γίνει ποτέ await. Η exception της δεν θα παρατηρηθεί, και ανάλογα με την έκδοση της .NET και των ρυθμίσεων, ίσως χαθεί ή ακόμα και να ξαναδημιουργηθεί (το exception) σε ένα μη αναμενόμενο thread, καταλήγοντας να τερματιστεί η εφαρμογή. Θα δούμε αργότερα καλύτερους τρόπους για να κάνουμε το παραπάνω.

4.3 Τύποι επιστροφής ασύγχρονων μεθόδων

Υπάρχουν τρεις τύποι, τους οποίους μία `async` method μπορεί να επιστρέφει:

- `void`
- `Task`
- `Task<T>` for some type *T*

Κανένας άλλος τύπος δεν επιτρέπεται, καθώς γενικά οι async methods δεν έχουν τελειώσει τη στιγμή που επιστρέφουν. Τυπικά, μία async method θα περιμένει μία λειτουργία μακράς διάρκειας, το οποίο σημαίνει ότι η method επιστρέφει γρήγορα, αλλά θα συνεχίσει στο μέλλον. Αυτό σημαίνει ότι κανένα χρήσιμο αποτέλεσμα δεν είναι διαθέσιμο όταν επιστρέφει η method. Το αποτέλεσμα θα είναι διαθέσιμο αργότερα.

Στο σημείο αυτό, να ξεκαθαρίσουμε τη διαφορά ανάμεσα στον *τύπο επιστροφής (return type)* της method - για παράδειγμα, `Task<string>` - και στον *τύπο αποτελέσματος (result type)* τον οποίο ο προγραμματιστής περιμένει να επιστρέψει στην καλούσα method, ο οποίος σε αυτή τη περίπτωση είναι `string`. Σε απλές non-async methods, ο τύπος επιστροφής και ο τύπος αποτελέσματος, είναι πάντα το ίδιο πράγμα, αλλά στις ασύγχρονες methods η διαφορά είναι σημαντική.

Είναι εμφανές ότι το `void` είναι μία λογική επιλογή ως τύπος επιστροφής σε μία ασύγχρονη κατάσταση. Μία `async void` method είναι μία ασύγχρονη λειτουργία “fire and forget”. Ο caller δε μπορεί ποτέ να περιμένει για αποτέλεσμα, και δεν μπορεί να ξέρει πότε ολοκληρώθηκε μία λειτουργία ή εάν ήταν επιτυχής. Πρέπει να χρησιμοποιούμε `void` μόνο όταν ξέρουμε ότι κανένας caller δεν χρειάζεται να ξέρει πότε μία λειτουργία ολοκληρώθηκε ή εάν ήταν επιτυχής. Στη πράξη αυτό σημαίνει ότι η `void` χρησιμοποιείται πολύ σπάνια. Η πιο συχνή χρήση των `async void` methods είναι στο όριο μεταξύ async κώδικα και άλλου κώδικα. Για παράδειγμα ένας UI event handler πρέπει να επιστρέφει `void`.

Οι async methods που επιστρέφουν `Task` επιτρέπουν στον caller να περιμένει την λειτουργία να τελειώσει, και προωθούν τα exceptions που τυχών συνέβησαν κατά τη διάρκεια της ασύγχρονης λειτουργίας. Όταν κανένα αποτέλεσμα δεν απαιτείται, μία `async Task` method είναι καλύτερη από μία `async void` method, επειδή επιτρέπει στον caller να χρησιμοποιήσει το `await` για να την περιμένει, κάνοντας ταυτόχρονα και το exception handling ευκολότερο.

Τέλος, οι async methods που επιστρέφουν `Task<T>`, για παράδειγμα `Task<string>`, χρησιμοποιούνται όταν η ασύγχρονη λειτουργία έχει αποτέλεσμα.

4.4 Async, Method Signatures και Interfaces

Το `async` keyword εμφανίζεται στη δήλωση μιας method, ακριβώς όπως τα `static` και `public` keywords. Πέρα από αυτό, το `async` δεν είναι μέρος του signature της method, με την έννοια του overriding, της υλοποίησης interfaces, ή της κλήσης.

Η μόνη επίδραση που έχει το async keyword είναι στη μεταγλώττιση της method στην οποία εφαρμόζεται, αντίθετα με τα άλλα keywords τα οποία αλλάζουν τον τρόπο που η method αλληλοεπιδρά με τον έξω κόσμο. Για το λόγο αυτό, οι κανόνες γύρω από το method override και της υλοποίησης interface αγνοούν τελείως το `async` keyword.

```
class BaseClass
{
    public virtual async Task<int> MyMethod()
    {
      ...
    }
}

class SubClass : BaseClass
{
    // This overrides AlexsMethod above
    public override Task<int> MyMethod()
    {
      ...
    }
}
```

Τα interfaces δε μπορούν να χρησιμοποιήσουν το `async` στη δήλωση μίας method, γιατί απλά δεν χρειάζεται. Εάν ένα interface απαιτεί μια method να επιστρέφει `Task`, η υλοποίηση της method ίσως επιλέξει να χρησιμοποιήσει `async`, αλλά το εάν θα το χρησιμοποιήσει τελικά ή όχι είναι θέμα της υπολοποίησης της method. Το interface δε χρειάζεται να ορίσει εάν θα χρησιμοποιηθεί το `async` ή όχι.

4.5 Το return στις Async Methods

Το return statement έχει διαφορετική συμπεριφορά σε μία async method. Στις μη-async methods, όπως ξέρουμε, η χρήση του return statement εξαρτάται από τον τύπο επιστροφής της method:

`void` *methods*

Το return statement είναι προαιρετικό, και αν υπάρχει πρέπει να είναι απλώς return.

Methods that return a type `T`

Το return statement πρέπει να έχει μία expression τύπου *`T`* (π.χ. `return 5+x;`) και πρέπει να υπάρχει στο τέλος της method σε όλα τα code paths.

Σε μία method μαρκαρισμένη σαν `async`, οι κανόνες είναι σχεδόν ίδιοι:

`void` *methods και methods που επιστρέφουν* `Task`

Το return statement είναι προαιρετικό, και αν υπάρχει πρέπει να είναι απλώς return.

Methods that return `Task<T>`

Το return statement πρέπει να έχει μία expression τύπου *`T`* και πρέπει να υπάρχει στο τέλος της method σε όλα τα code paths.

Στις async methods, ο τύπος επιστροφής μίας method είναι διαφορετικός από τον τύπο της expression που υπάρχει στο `return` statement. Ο μετασχηματισμός που κάνει ο μεταγλωτιστής, «περιτυλίγει» τη τιμή που θέλουμε να επιστρέψουμε μέσα σε ένα `Task<`*`T`*`>`, πριν το δώσει στον caller. Φυσικά στη πραγματικότητα το `Task<`*`T`*`>` δημιουργείται αμέσως, και απλώς θα «γεμίσει» με την τιμή αποτελέσματος αργότερα, όταν η εργασία μακράς διάρκειας θα έχει τελειώσει.

4.6 Οι Async Methods είναι «μεταδοτικές»

Όπως είδαμε μέχρι τώρα, ο καλύτερος τρόπος για να «καταναλώσουμε» ένα `Task` που επιστρέφεται από ένα ασύγχρονο API, είναι να το περιμένουμε με `await` μέσα σε μία async method. Όταν το κάνουμε αυτό, η method θα επιστρέψει επίσης `Task`. Για να εκμεταλλευτούμε τα οφέλη του ασύγχρονου στυλ προγραμματισμού, ο κώδικας που καλεί τη method μας δεν πρέπει να μπλοκάρεται περιμένοντας το `Task` να ολοκληρωθεί, επομένως πιθανόν να περιμένει με `await` την method μας.

Παρακάτω είναι ένα παράδειγμα μιας βοηθητικής method που επιστρέφει τον αριθμό των χαρακτήρων σε μία ιστοσελίδα:

```
private async Task<int> GetPageSizeAsync(string url)
{
    WebClient webClient = new WebClient();
    string page = await webClient.DownloadStringTaskAsync(url);
    return page.Length;
}
```

Παρακάτω είναι μία method η οποία χρησιμοποιεί την παραπάνω βοηθητική method για να βρεί την μεγαλύτερη ιστοσελίδα σε ένα πίνακα με URLs. Η method αυτή επίσης επιστρέφει το αποτέλεσμά της ασύγχρονα:

```
private async Task<string> FindLargestWebPage(string[] urls)
{
    string largest = null;
    int largestSize = 0;
    foreach (string url in urls)
    {
        int size = await GetPageSizeAsync(url);
        if (size > largestSize)
        {
            size = largestSize;
            largest = url;
        }
    }
    return largest;
}
```

Με αυτό το τρόπο καταλήγουμε να γράψουμε αλυσιδωτές async methods, όπου κάθε μία περιμένει την άλλη. Ο ασύγχρονος προγραμματισμός είναι ένα μεταδοτικό μοντέλο προγραμματισμού, που διαπερνάει όλο το κώδικά μας. Επειδή όμως είναι εύκολο να γράψουμε async methods, αυτό δε αποτελεί πρόβλημα.

4.7 Async Anonymous Delegates and Lambdas

Όπως οι συνηθισμένες methods μπορούν να είναι ασύγχρονες, έτσι και οι δύο τύποι anonymous methods μπορούν να είναι ασύγχρονες. Η σύνταξη μοιάζει πολύ με τις συνηθισμένες κλασικές methods. Παρακάτω φαίνεται πως μπορούμε να φτιάξουμε ένα ασύγχρονο anonymous delegate και ένα ασύγχρονο lambda αντίστοιχα:

```
Func<Task<int>> getNumberAsync = async delegate { return 3; };

Func<Task<string>> getWordAsync = async () => "hello";
```

Για τις παραπάνω, ισχύουν οι ίδιοι κανόνες με τις συνηθισμένες ασύγχρονες methods.

Κεφάλαιο 5

Task-Based Ασύγχρονο Μοτίβο

(Task-Based Asynchronous Pattern)

Το Task-Based ασύγχρονο μοτίβο (Task-Based Asynchronous Pattern TAP) είναι ένα σύνολο απαιτήσεων από τη Microsoft για να γράφουμε ασύγχρονα APIs χρησιμοποιώντας το `Task`. Αυτό το μοτίβο δημιουργεί APIs τα οποία μπορούν καταναλωθούν χρησιμοποιώντας το `await`.

5.1 Προδιαγραφές του TAP

Όπως ξέρουμε οι κανόνες για σωστή σχεδίαση μιας σύγχρονης method είναι:

- Πρέπει να έχει λίγες παραμέτρους, ή ίσως και καμία. Οι `ref` και `out` παράμετροι πρέπει να αποφεύγονται αν είναι δυνατόν.
- Πρέπει να έχει έναν τύπο επιστροφής, αν χρειάζεται, ο οποίος πραγματικά να εκφράζει το αποτέλεσμα του κώδικα μέσα στη method, σε αντίθεση με C++ κώδικα όπου συνηθίζεται να επιστέφεται μία τιμή για ένδειξη επιτυχίας ή αποτυχίας.
- Πρέπει να έχει ένα όνομα που εξηγεί τη συμπεριφορά της method.
- Κοινές και αναμενόμενες αποτυχίες πρέπει να είναι μέρος του τύπου επιστροφής, ενώ οι μη αναμενόμενες αποτυχίες πρέπει να δημιουργούν exceptions.

Παρακάτω φαίνεται μία καλά σχεδιασμένη σύγχρονη method, η οποία είναι μέρος της `Dns` class.

```
public static IPHostEntry GetHostEntry(string hostNameOrAddress)
```

Το TAP παρέχει το ίδιο επίπεδο οδηγιών για τη σχεδίαση μίας ασύγχρονης method:

- Πρέπει να έχει τον ίδιο αριθμό παραμέτρων που έχει και η ισοδύναμη σύγχρονη. Παράμετροι `ref` και `out` δεν πρέπει ποτέ να χρησιμοποιούνται.
- Πρέπει να επιστρέφει `Task` ή `Task<T>` ανάλογα με το αν η σύγχρονη method θα είχε ή όχι τύπο επιστροφής. Το task θα πρέπει να ολοκληρωθεί κάποια στιγμή στο μέλλον, παρέχοντας την τυχόν τιμή επιστροφής στη method.
- Θα πρέπει να ονομάζεται *`Name`*`Async` όπου *`Name`* είναι το όνομα της πιθανής ισοδύναμης σύγχρονης.
- Ένα exception που τυχόν θα συμβεί από λάθος στη χρήση της συνάρτησης, θα πρέπει να στέλνεται κατευθείαν από τη method. Κάθε άλλο exception θα πρέπει να τοποθετείται στο `Task`.

Παρακάτων φαίνεται μία καλά σχεδιασμένη TAP method:

```
public static Task<IPHostEntry> GetHostEntryAsync(string hostNameOrAddress)
```

Τα παραπάνω πιθανόν να φαίνονται εντελώς προφανή, αλλά όπως είδαμε πιο πριν το παραπάνω είναι το τρίτο επίσημο ασύγχρονο μοτίβο (pattern) που υπάρχει στο .NET

framework. Τα προηγούμενα ήταν αρκετά πολύπλοκα όπως έχουμε δει και σίγουρα δυσκόλευαν πολλούς προγραμματιστές.

Η κεντρική ιδέα στο TAP είναι η ασύγχρονη method να επιστρέφει ένα `Task`, το οποίο ενθυλακώνει την υπόσχεση για την ολοκλήρωση στο μέλλον μιας λειτουργίας μακράς διάρκειας. Χωρίς την ιδέα αυτή, τα προηγούμενα ασύγχρονα μοτίβα χρειάζονταν είτε να προσθέσουμε επιπλέον παραμέτρους στις method, είτε να προσθέσουμε επιπλέον methods ή events για να υποστηρίζεται ο callback μηχανισμός. Το Task μπορεί να περιέχει οποιαδήποτε υποδομή είναι απαραίτητη για να υποστηρίζεται ο callback μηχανισμός, χωρίς να μολύνουμε τον κώδικά μας με αυτές τις λεπτομέρειες.

Ένα επιπλέον όφελος είναι ότι, επειδή ο μηχανισμός ενός ασύγχρονου callback είναι τώρα στο Task, δε χρειάζεται να τον ξαναγράφουμε κάθε φορά που γίνεται μία ασύγχρονη κλήση. Σε συνέχεια, αυτό σημαίνει ότι είναι ανεκτό ο μηχανισμός να είναι πιο πολύπλοκος και αποτελεσματικός, και να μπορεί να κάνει πράγματα όπως επαναφορά του context, συμπεριλαμβανόμενου του synchronization context, καθώς γίνεται το callback. Επίσης παρέχει ένα κοινό API για να δουλεύουμε με ασύγχρονες λειτουργίες, κάνοντας χαρακτηριστικά του compiler όπως το async υλοποιήσιμα, τα οποία δε θα μπορούσαν να υλοποιηθούν με άλλα μοτίβα.

5.2 Χρησιμοποιώντας το Task για Compute-Intensive λειτουργίες

Μερικές φορές, μία λειτουργία μακράς διάρκειας δεν κάνει κάποια αιτήματα εξυπηρέτησης από το δίκτυο ή κάποια πρόσβαση στο δίσκο. Απλώς παίρνει πολύ χρόνο επειδή είναι ένας δύσκολος υπολογισμός που χρειάζεται πολύ χρόνο από τον επεξεργαστή για να ολοκληρωθεί. Φυσικά δεν περιμένουμε να μπορούμε να κάνουμε κάτι τέτοιο χωρίς να δεσμεύσουμε το thread, όπως θα κάναμε με μία πρόσβαση στο δίκτυο. Όμως σε προγράμματα με user interface, θέλουμε να αποφύγουμε το «πάγωμα» του UI thread. Για να το πετύχουμε αυτό θα πρέπει να αφήνουμε το UI thread να επεξεργάζεται άλλα events, και να χρησιμοποιήσουμε ένα διαφορετικό thread για τον υπολογισμό μακράς διάρκειας.

Το `Task` παρέχει έναν εύκολο τρόπο για να το κάνουμε αυτό:

```
Task t = Task.Run(() => MyLongComputation(a, b));
```

Η `Task.Run` χρησιμοποιεί ένα thread από το `ThreadPool` για την εκτέλεση του delegate που της δίνουμε. Στη παραπάνω περίπτωση χρησιμοποιήσαμε ένα lambda expression για να είναι πιο εύκολο να περάσουμε τοπικές μεταβλητές στη συνάρτηση υπολογισμού. Το `Task` που προκύπτει αρχίζει να εκτελείται αμέσως, και μπορούμε να το περιμένουμε με `await` όπως κάθε άλλο `Task`:

```
await Task.Run(() => MyLongComputation(a, b));
```

Αυτός είναι ένας πολύ απλός τρόπος για να εκτελέσουμε κάτι σε ένα background thread.

Εάν χρειαζόμαστε περισσότερο έλεγχο πάνω στο ποιο thread τρέχει τον υπολογισμό, η `Task` έχει ένα static property που λέγεται `Factory` τύπου `TaskFactory`. Αυτός ο τύπος έχει μία method που λέγεται `StartNew` με διάφορα overloads για τον έλεγχο της εκτέλεσης του υπολογισμού:

```
Task t = Task.Factory.StartNew(() => MyLongComputation(a, b),
                               cancellationToken,
                               TaskCreationOptions.LongRunning,
                               taskScheduler);
```

Εάν γράφουμε μία βιβλιοθήκη που περιέχει πολλές compute-intensive methods, ίσως μπούμε στον πειρασμό να παρέχουμε ασύγχρονες εκδόσεις των method μας, οι οποίες θα καλούν την Task.Run για να ξεκινήσουν την εργασία σε ένα background thread. Αυτό δεν είναι καλή ιδέα, γιατί ο caller του API μας ξέρει περισσότερα για τις απαιτήσεις σε thread της εφαρμογής του, από ότι εμείς. Για παράδειγμα, στις web εφαρμογές, δεν υπάρχει όφελος από τη χρήση του thread pool. Η Task.Run είναι πολύ εύκολη στη χρήση, οπότε ας αφήσουμε τους callers να τη χρησιμοποιήσουν αν χρειάζεται.

5.3 Puppet Task

Το TAP είναι πραγματικά εύκολο να το «καταναλώσουμε», οπότε είναι φυσικό να θέλουμε να το παρέχουμε σε κάθε API που φτιάχνουμε. Ξέρουμε ήδη πως να το κάνουμε αυτό όταν «καταναλώνουμε» άλλα TAP APIs, χρησιμοποιώντας μία async method. Αλλά τι συμβαίνει όταν μία λειτουργία μακράς διάρκειας δεν είναι ήδη διαθέσιμη ως TAP API; Ίσως είναι ένα API που χρησιμοποιεί άλλο ασύγχρονο μοτίβο. Ίσως πάλι δεν «καταναλώνουμε» ένα API, αλλά κάνουμε κάτι ασύγχρονο εντελώς manually.

Το εργαλείο που χρειαζόμαστε εδώ είναι το `TaskCompletionSource<T>`. Με αυτό το τρόπο μπορούμε να φτιάξουμε μία `Task` που θα μπορούμε να την ελέγχουμε άμεσα εμείς. Θα είναι δηλαδή ένα *puppet* Task. Έτσι μπορούμε να κάνουμε την `Task` να τερματίζει όποια στιγμή θέλουμε, και μπορούμε να την κάνουμε να αποτύχει στέλνοντάς της ένα exception όποια στιγμή θέλουμε.

Ας δούμε ένα παράδειγμα. Ας υποθέσουμε ότι θέλουμε να ενθυλακώσουμε ένα prompt που θα εμφανίζεται στον χρήστη, με την παρακάτω method:

```
Task<bool> GetUserPermission()
```

Το prompt ας πούμε ότι είναι ένα custom dialog box που έχουμε γράψει εμείς, το οποίο ζητάει τη συγκατάθεση του χρήστη για κάτι. Επειδή αυτό θα χρειάζεται να γίνεται σε πολλά σημεία της εφαρμογής, είναι σημαντικό να είναι μία εύκολη στην κλήση method. Θα μπορούσε ιδανικά να είναι μία ασύγχρονη method, επειδή θέλουμε να απελευθερώσουμε το UI thread για να εμφανιστεί το dialog box. Αλλά δε μοιάζει καθόλου με τις παραδοσιακές ασύγχρονες method που κάνουν κλήσεις στο δίκτυο ή που εκτελούν λειτουργίες μακράς διάρκειας. Σε αυτή τη περίπτωση περιμένουμε τον χρήστη. Ας δούμε το σώμα αυτής της method:

```
private Task<bool> GetUserPermission()
{
  // Make a TaskCompletionSource so we can return a puppet Task
  TaskCompletionSource<bool> tcs = new TaskCompletionSource<bool>();
  // Create the dialog ready
  PermissionDialog dialog = new PermissionDialog();
  // When the user is finished with the dialog, complete the Task using SetResult
  dialog.Closed += delegate { tcs.SetResult(dialog.PermissionGranted); };
  // Show the dialog
  dialog.Show();
```

```
    // Return the puppet Task, which isn't completed yet
    return tcs.Task;
}
```

Η `TaskCompletionSource<bool>` δημιουργεί το `Task`, και μας το παρέχει σαν property. Μπορούμε αργότερα να χρησιμοποιήσουμε τη `SetResult` method της `TaskCompletionSource` για να αναγκάσουμε το `Task` να ολοκληρωθεί.

Επειδή ακολουθήσαμε το TAP, ο caller μπορεί απλώς να περιμένει με await τη συγκατάθεση του χρήστη. Η κλήση είναι πολύ απλή:

```
if (await GetUserPermission())
  { ....
```

Δεν υπάρχει non-generic έκδοση της `TaskCompletionSource<T>`. Παρόλα αυτά, επειδή η `Task<T>` είναι μία υποκλάση της `Task`, μπορούμε να χρησιμοποιήσουμε την `Task<T>` οπουδήποτε θέλουμε την `Task`. Στη περίπτωση αυτή, μπορούμε να χρησιμοποιήσουμε την `TaskCompletionSource<object>`, και να καλέσουμε την `SetResult(null)` για να ολοκληρώσουμε την Task.

5.4 Αλληλεπίδραση με παλιά ασύγχρονα μοτίβα

Η ομάδα της .NET στη Microsoft δημιούργησε TAP εκδόσεις για όλα τα σημαντικά ασύγχρονα APIs στο .NET framework. Είναι όμως ενδιαφέρον να δούμε πως μπορούμε να φτιάξουμε μία TAP method από non-TAP ασύγχρονο κώδικα. Θα χρησιμοποιήσουμε και εδώ την `TaskCompletionSource<T>`.

Ας πάρουμε το DNS lookup παράδειγμα που χρησιμοποιήσαμε νωρίτερα. Στην .NET 4.0, η ασύγχρονη έκδοση της DNS lookup method χρησιμοποιούσε το `IAsyncResult` ασύγχρονο μοτίβο. Αυτό σημαίνει ότι αποτελούνταν από μία `Begin` method και από μία `End` method:

```
IAsyncResult BeginGetHostEntry(string hostNameOrAddress,
                               AsyncCallback requestCallback,
                               object stateObject)
IPHostEntry EndGetHostEntry(IAsyncResult asyncResult)
```

Η τυπική διαδικασία είναι να κάνουμε consume αυτό το API χρησιμοποιώντας ένα lambda σαν callback και να καλέσουμε την End method μέσα από το lambda. Αυτό ακριβώς θα κάνουμε εδώ, αλλά αντί να κάνουμε οτιδήποτε μέσα στο callback, θα χρησιμοποιήσουμε απλώς την `TaskCompletionSource<T>` για να ολοκληρώσουμε την `Task`.

```
public static Task<IPHostEntry> GetHostEntryAsync(string hostNameOrAddress)
{
   TaskCompletionSource<IPHostEntry> tcs = new TaskCompletionSource<IPHostEntry>();
   Dns.BeginGetHostEntry(hostNameOrAddress, asyncResult =>
   {
      try
      {
         IPHostEntry result = Dns.EndGetHostEntry(asyncResult);
         tcs.SetResult(result);
      }
      catch (Exception e)
      {
         tcs.SetException(e);
      }
```

```
    }, null);
    return tcs.Task;
}
```

Αυτός ο κώδικας γίνεται πολύ πιο πολύπλοκος στην περίπτωση που συμβεί exception. Εάν το DNS resolve αποτύχει, θα συμβεί ένα exception όταν καλέσουμε την `EndGetHostEntry`. Όταν συμβεί ένα exception, πρέπει να το βάλουμε στην `TaskCompletionSource<T>` έτσι ώστε ο caller να μπορεί να πάρει την exception σύμφωνα με το TAP στυλ προγραμματισμού.

Στη πραγματικότητα υπάρχουν τόσα πολλά τέτοια ασύγχρονα APIs τα οποία ακολουθούν αυτό το μοτίβο, που η .NET framework ομάδα της Microsoft έφτιαξε μία utility method για να τα μετατρέπουμε σε TAP:

```
Task t = Task<IPHostEntry>.Factory.FromAsync<string>(Dns.BeginGetHostEntry,
                                                     Dns.EndGetHostEntry,
                                                     hostNameOrAddress,
                                                     null);
```

Παίρνει τις `Begin` και `End` methods σαν delegates, και χρησιμοποιεί έναν μηχανισμό παρόμοιο με αυτό που κάναμε παραπάνω.

5.5 Up-Front εργασία

Ξέρουμε ήδη ότι όταν καλούμε μία TAP ασύγχρονη method, η method τρέχει στο τρέχον thread όπως όλες οι άλλες methods. Η διαφορά είναι ότι μία TAP method πιθανόν να μην έχει τελείωσει τις εργασίες της πριν επιστρέψει. Θα επιστρέψει ένα `Task` γρήγορα, και αυτό το Task θα ολοκληρωθεί μόλις οι εργασίες ολοκληρωθούν.

Κάποιος κώδικας στη method θα τρέξει σύγχρονα στο τρέχον thread. Στη περίπτωση μιας async method αυτό θα είναι μέχρι τον operand του πρώτου await, όπως είδαμε νωρίτερα.

Το TAP επιβάλει ότι οι σύγχρονες λειτουργίες που γίνονται από μία TAP method, πρέπει να είναι οι ελάχιστες δυνατόν. Μπορούμε για παράδειγμα να ελέγξουμε αν κάποια arguments είναι έγκυρα, αλλά δεν μπορούμε να κάνουμε λειτουργίες και υπολογισμούς μακράς διάρκειας. Οι υβριδικές methods, οι οποίες κάνουν κάποιους υπολογισμούς ακολουθούμενες από μία πρόσβαση στο δίκτυο ή κάτι παρόμοιο, είναι καλές, αλλά θα πρέπει να χρησιμοποιούμε την `Task.Run` για να μετακινήσουμε τον υπολογισμό σε ένα background thread. Ας σκεφτούμε μία ρουτίνα που κάνει upload μία εικόνα σε ένα website, αλλά πρέπει να την κάνει resize πρώτα για εξοικονομήσει bandwidth:

```
Image resized = await Task.Run(() => ResizeImage(originalImage));
await UploadImage(resized);
```

Όταν βλέπουμε μία method η οποία φαίνεται ότι ακολουθεί το TAP, περιμένουμε να επιστρέψει γρήγορα. Ο καθένας που παίρνει τον κώδικά μας και τον χρησιμοποιεί σε μία UI εφαρμογή, θα εκπλαγεί αν εμείς εσωτερικά κάνουμε ένα αργό resize στην εικόνα σύγχρονα.

Κεφάλαιο 6

Ποιο thread τρέχει τον κώδικά μου;

Όπως είπαμε νωρίτερα, όλα γύρω από τον ασύγχρονο προγραμματισμό έχουν να κάνουν με τα threads. Στην C# αυτό σημαίνει ότι πρέπει να καταλάβουμε ποιο .NET thread τρέχει τον κώδικά μας σε ποια σημεία του προγράμματος, και τι συμβαίνει στα threads κατά τη διάρκεια των λειτουργιών μακράς διάρκειας.

6.1 Πριν το πρώτο await

Σε κάθε async method που γράφουμε, κάποιος κώδικας θα είναι πριν το πρώτο `await` keyword. Αυτός ο κώδικας τρέχει στο calling thread. Τίποτα το ενδιαφέρον δε συμβαίνει πριν το πρώτο `await`.

Το παραπάνω είναι μία από τις πιο συνηθισμένες παρανοήσεις σχετικά με το async. Το async ποτέ δεν στέλνει τη method μας για εκτέλεση σε ένα background thread. Ο μόνος τρόπος για να το κάνουμε αυτό είναι να χρησιμοποιήσουμε κάτι σαν την `Task.Run`, η οποία υπάρχει αποκλειστικά για αυτό το σκοπό.

Στην περίπτωση μιας UI εφαρμογής, αυτό σημαίνει ότι ο κώδικας πριν από το πρώτο `await` τρέχει στο UI thread. Αντίστοιχα, σε μία ASP .NET web εφαρμογή, τρέχει σε ένα ASP .NET worker thread.

Είναι πολύ συνηθισμένο στη γραμμή με το πρώτο `await`, η expression που κάνουμε await να είναι μία άλλη async method. Επειδή η expression εκτελείται πρίν από το πρώτο await, θα εκτελεστεί στο calling thread. Αυτό σημαίνει ότι το calling thread θα συνεχίσει να εκτελεί κώδικα βαθιά μέσα στην εφαρμογή μας, μέχρι κάποια method να επιστρέψει πραγματικά ένα `Task`. Η method που θα το κάνει αυτό μπορεί να είναι μία framework method, ή μπορεί να είναι μία method που χρησιμοποιεί την `TaskCompletionSource` για να δημιουργήσει μία puppet `Task`. Εκείνη η method είναι η πηγή της ασύγχρονης διαδικασίας στο πρόγραμμά μας-όλες οι async methods απλώς αναπαράγουν την ασύγχρονη διαδικασία.

Ο κώδικας που τρέχει πριν φτάσουμε στο πρώτο πραγματικά ασύγχρονο σημείο μπορεί να είναι αρκετά χρονοβόρος. Σε μία UI εφαρμογή όπου αυτός ο κώδικας τρέχει στο UI thread, το UI δεν θα αποκρίνεται. Οπότε το ότι χρησιμοποιούμε async δεν εγγυάται ότι το UI θα αποκρίνεται. Επομένως θα πρέπει να προσέχουμε όλος αυτός ο κώδικας να είναι σύντομος.

6.2 Κατά τη διάρκεια της ασύγχρονης λειτουγίας

Ποιο thread εκτελεί πραγματικά την ασύγχρονη λειτουρία;

Αυτή είναι μία ερώτηση trick. Για τυπικές λειτουργίες όπως πρόσβαση στο δίκτυο, δεν υπάρχουν threads που να μπλοκάρονται περιμένοντας τη λειτουργία να ολοκληρωθεί. Φυσικά αν χρησιμοποιούμε το async για να περιμένουμε για έναν υπολογισμό χρησιμοποιώντας την `Task.Run`, υπάρχει ένα thread από το thread pool το οποίο κάνει τον υπολογισμό.

Υπάρχει ένα thread που περιμένει για αιτήσεις από το δίκτυο να ολοκληρωθούν, το οποίο όμως είναι κοινό για όλες τις αιτήσεις. Αυτό στα Windows ονομάζεται *IO*

Completion Port thread. Όταν η αίτηση από το δίκτυο ολοκληρωθεί, ένας interrupt handler στο λειτουργικό σύστημα προσθέτει ένα job σε μία ουρά για το IO Completion Port. Για να εκτελεστούν 1000 αιτήσεις στο δίκτυο, οι αιτήσεις ξεκινάνε όλες, και καθώς έρχονται οι απαντήσεις, επεξεργάζονται όλες από το ένα IO Completion Port.

Στη πραγματικότητα, τα IO Completion Port threads είναι περισσότερα από ένα, ανάλογα με το πόσα cores έχει η CPU. Παρόλα αυτά και πάλι, ο αριθμός των thread είναι ό ίδιος είτε υπάρχουν 10 είτε 1000 αιτήσεις στο δίκτυο.

6.3 Synchronization Context

Το `SynchronizationContext` είναι μία class του .NET framework, η οποία έχει τη δυνατότητα να εκτελεί κώδικα σε έναν συγκεκριμένο τύπο thread. Υπάρχουν διάφορα `SynchronizationContexts` που χρησιμοποιούνται από τη .NET. Τα πιο σημαντικά από αυτά είναι τα UI thread contexts που χρησιμοποιούνται από τις εφαρμογές WinForms και WPF.

Ένα instance της `SynchronizationContext` class από μόνο του, δεν κάνει κάτι πολύ χρήσιμο, οπότε όλα τα πραγματικά instances αυτής ουσιαστικά είναι subclasses. Περιέχουν static members που μας επιτρέπουν να διαβάσουμε και να ελέγξουμε το *τρέχον* `SynchronizationContext`. Το τρέχον SynchronizationContext είναι μία property του τρέχοντως thread. Η ιδέα είναι ότι σε οποιοσδήποτε σημείο εκτέλεσης σε ένα special thread, θα πρέπει να μπορούμε να «συλλάβουμε» το τρέχων SynchronizationContext και να το αποθηκεύσουμε. Αργότερα μπορούμε να το χρησιμοποιήσουμε για να τρέξουμε κώδικα πίσω στο special thread που ξεκινήσαμε. Όλα αυτά θα πρέπει να μπορούν να γίνουν χωρίς να χρειάζεται να ξέρουμε ακριβώς σε ποιο thread ξεκινήσαμε. Εφόσον μπορούμε να χρησιμοποιήσουμε το `SynchonizationContext`, μπορούμε να γυρίσουμε πίσω σε αυτό.

Η σημαντική method στη `SynchronizationContext` είναι η `Post`, η οποία μπορεί να κάνει ένα delegate να τρέξει στο σωστό context.

6.4 await και SynchronizationContext

Ξέρουμε ότι ο κώδικας πριν το πρώτο await τρέχει στο calling thread, αλλά τι συμβαίνει όταν η method επιστρέψει μετά το await;

Στη πραγματικότητα, τις περισσότερες φορές, ο κώδικας τρέχει επίσης στο calling thread παρόλο που το calling thread πιθανόν έχει κάνει άλλα πράγματα στο μεταξύ. Αυτό κάνει τα πράγματα πολύ εύκολα για τον προγραμματιστή.

Η C# χρησιμοποιεί το `SynchronizationContext` για να το κάνει αυτό. Όπως είδαμε νωρίτερα, όταν περιμένουμε με await ένα `Task`, το τρέχων `SynchronizationContext` αποθηκεύεται. Αργότερα, μόλις έρθει η ώρα να επιστρέψει και να συνεχίσει η method, η υποδομή του `await` keyword χρησιμοποιεί την method `Post` για να επαναφέρει τη method για εκτέλεση στο αποθηκευμένο `SynchronizationContext`.

Όμως, η method μπορεί να επανέλθει σε διαφορετικό thread από αυτό που ξεκίνησε αν:

- Το SynchronizationContext είχε πολλαπλά thread, όπως το thread pool.
- Δεν υπήρχε SynchronizationContext όταν η εκτέλεση έφτασε στο `await`, όπως συμβαίνει για παράδειγμα σε μία console application.
- Έχουμε ρυθμίσει το `Task` να μη χρησιμοποιήσει το SynchronizationContext για τη συνέχεια της εκτέλεσης.

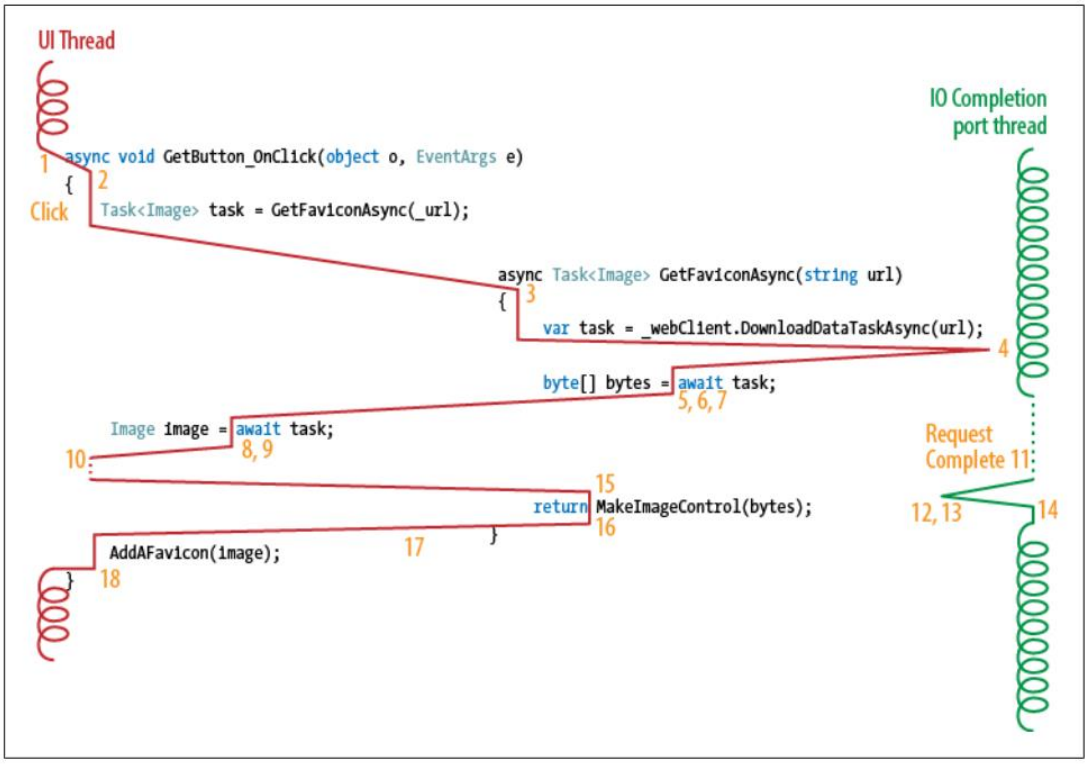
Ευτυχώς σε UI εφαρμογές όπου είναι σημαντικό η εκτέλεση να συνεχίσει στο ίδιο thread δεν ισχύουν τα παραπάνω, οπότε μπορούμε ελεύθερα και με ασφάλεια να εκτελέσουμε κώδικα στο UI thread μετά το `await`.

6.5 Ο κύκλος ζωής μιας Async λειτουργίας

Ας δούμε μία έκδοση του παραδείγματος favicon, το οποίο μας δείχνει ακριβώς ποιο thread τρέχει ποιον κώδικα. Έχουμε δύο async methods:

```
async void GetButton_OnClick(...)
async Task<Image> GetFaviconAsync(...)
```

Ο event handler `GetButton_OnClick` καλεί την `GetFaviconAsync`, η οποία με τη σειρά της καλεί την `WebClient.DownloadData TaskAsync`. Παρακάτω φαίνεται ένα διάγραμμα της σειράς των γεγονότων που συμβαίνουν κατά την εκτέλεση της method.



Εικόνα 1 Ο κύκλος ζωής μιας async λειτουργίας

1. Ο χρήστης κάνει click στο button, οπότε ο event handler `GetButton_OnClick` γίνεται queue για εκτέλεση από το UI thread.
2. Το UI thread εκτελεί το πρώτο μισό της `GetButton_OnClick`, συμπεριλαμβανομένης της κλήσης της `GetFaviconAsync`.
3. Το UI thread συνεχίζει μέσα στην `GetFaviconAsync` και εκτελεί το πρώτο μισό της, συμπεριλαμβανομένης της κλήσης της `WebClient.DownloadDataTaskAsync`.
4. Το UI thread συνεχίζει μέσα στην `WebClient.DownloadData TaskAsync`, η οποία ξεκινάει το download και επιστρέφει ένα `Task`.
5. Το UI thread αφήνει την `WebClient.DownloadDataTask Async`, και φτάνει στο `await` της `GetFaviconAsync.`
6. Το τρέχων `SynchronizationContext` γίνεται capture – είναι το UI thread.
7. Η `GetFaviconAsync` σταματάει από το await, και το Task από την `WebClient.DownloadDataTaskAsync` ενημερώνεται ότι πρέπει να επαναφέρει την εκτέλεση της `GetFaviconAsync` όταν τελειώσει το download (στο captured `SynchronizationContext`).
8. Το UI thread φεύγει από την `GetFaviconAsync`, η οποία επιστρέφει ένα `Task`, και φτάνει στο `await` στην `GetButton_OnClick`.
9. Ομοίως, η `GetButton_OnClick` σταματάει από την `await`.
10. Το UI thread φεύγει από την `GetButton_OnClick`, και είναι ελεύθερο να επεξεργαστεί άλλες ενέργειες του χρήστη.

    Στο σημείο αυτό, περιμένουμε το icon να κατέβει. Αυτό μπορεί να πάρει μερικά δευτερόλεπτα. Παρατηρούμε ότι το UI thread είναι ελεύθερο να επεξεργαστεί άλλες ενέργειες του χρήστη, και το IO Completion Port δεν έχει εμπλακεί ακόμα. Ο συνολικός αριθμός των μπλοκαρισμένων thread κατά τη διάρκεια αυτής της λειτουργίας είναι 0.

11. Το download τελειώνει, οπότε το IO Completion Port κάνει queue το γεγονός.
12. Το IO Completion Port thread ορίζει το Task που επεστράφει από την `DownloadDataTaskAsync` ως ολοκληρωμένο.
13. Το IO Completion Port thread τρέχει κώδικα μέσα στην `Task` για να χειριστεί την ολοκλήρωση, ο οποίος καλεί την `Post` στο captured `SynchronizationContext` (το UI thread) για να συνεχίσει.
14. Το IO Completion Port thread είναι ελεύθερο να εργαστεί πάνω σε άλλα IO αιτήματα.
15. Το UI thread βρίσκει την `Post`ed εντολή και επαναφέρει την εκτέλεση της `GetFaviconAsync`, εκτελώντας το δεύτερο μισό της, μέχρι το τέλος.

16. Καθώς το UI thread φεύγει από την `GetFaviconAsync`, θέτει το `Task` που επεστράφει από την `GetFaviconAsync` ως ολοκληρωμένο.
17. Επειδή αυτή τη φορά, το τρέχων `SynchronizationContext` είναι το ίδιο με αυτό που είχε γίνει capture, δεν χρειάζεται να εκτελεστεί η `Post` και το UI thread συνεχίζει σύγχρονα.

    Αυτή η λογική είναι λίγο αναξιόπιστη σε WPF εφαρμογές, επειδή το WPF συχνά δημιουργεί καινούργια `SynchronizationContext` objects. Παρόλο που είναι ισοδύναμα, δημιουργείται η εντύπωση στο TLP ότι χρειάζεται να εκτελεστεί η Post ξανά.

18. Το UI thread επαναφέρει την `GetButton_OnClick`, εκτελώντας το δεύτερο μισό της, μέχρι το τέλος.

Παρατηρούμε ότι *κάθε* γραμμή του κώδικά μας εκτελέστηκε από το UI thread. Το IO Completion Port thread έτρεξε μόνο για όσο χρειάστηκε για να κάνει `Post` μία εντολή προς το UI thread, το οποίο τελικά έτρεξε το δεύτερο μισό και των δύο method.

6.6 Επιλέγοντας μη χρήση του SynchronizationContext

Κάθε υλοποίηση του `SynchronizationContext` πραγματοποιεί το `Post` με διαφορετικό τρόπο. Μερικές από αυτές τις υλοποιήσεις είναι σχετικά expensive. Για να αποφύγουμε αυτό το κόστος, η .NET δεν θα χρησιμοποιήσει την `Post` όταν το captured `SynchronizationContext` είναι το ίδιο με το τρέχων τη στιγμή της ολοκλήρωσης του `Task`.

Όταν το `SynchronizationContext` είναι διαφορετικό, είναι απαραίτητη η (expensive) `Post`. Σε κώδικα που είναι performance critical, ή σε library κώδικα, όπου δεν μας ενδιαφέρει ποιο thread χρησιμοποιούμε, μπορούμε να επιλέξουμε να μην επιβαρυνθούμε με αυτό το κόστος στο performance. Αυτό γίνεται καλώντας την `ConfigureAwait` στο `Task` πριν το κάνουμε await. Εάν το κάνουμε αυτό, δεν θα γίνει `Post` back στο original `SynchronizationContext` κατά την επιστροφή.

```
byte[] bytes = await client.DownloadDataTaskAsync(url).ConfigureAwait(false);
```

Παρόλα αυτά, η `ConfigureAwait` δεν κάνει πάντα αυτό που περιμένουμε ότι θα κάνει. Είναι σχεδιασμένη ως μία ειδοποίηση προς τη .NET ότι δε μας πειράζει σε ποιο thread επανέρχεται η method μας, παρά αυστηρά ως μία εντολή. Το τι κάνει εξαρτάται από το ποιο thread ολοκλήρωσε το `Task` που περιμένουμε. Εάν αυτό το thread δεν είναι σημαντικό, ίσως από το thread pool, θα πρέπει να συνεχίσει να εκτελεί τον κώδικά μας. Αλλά εάν είναι ένα σημαντικό thread κάποιου είδους, η .NET θα προτιμήσει να το ελευθερώσει για να κάνει άλλα πράγματα, και η method μας θα επανέλθει στο thread pool αντί για το αρχικό (σημαντικό) thread. Η .NET χρησιμοποιεί το τρέχων `SynchronizationContext` του thread για να κρίνει εάν είναι σημαντικό.

6.7 Αλληλοεπιδρώντας με σύγχρονο κώδικα

Είναι πιθανό να μας τύχει να δουλεύουμε σε μία ήδη υπάρχουσα εφαρμογή, και παρόλο που ο καινούργιος κώδικας που γράφουμε μπορεί να είναι ασύγχρονος χρησιμοποιώντας το TAP, ίσως χρειαστεί να επικοινωνήσουμε με παλιό σύγχρονο κώδικα. Όταν το κάνουμε αυτό, συνήθως χάνουμε τα οφέλη του ασύγχρονου προγραμματισμού.

Το να κάνουμε consume σύγχρονο κώδικα από ασύγχρονο κώδικα είναι εύκολο. Εάν έχουμε ένα blocking API, μπορούμε απλά να το τρέξουμε στο thread pool και να το περιμένουμε, με χρήση της `Task.Run`. Χρησιμοποιούμε ένα thread, αλλά αυτό είναι αναπόφευκτο.

```
var result = await Task.Run(() => MyOldMethod());
```

Το να κάνουμε consume ασύγχρονο κώδικα από σύγχρονο κώδικα, ή να υλοποιήσουμε ένα σύγχρονο API, επίσης φαίνεται εύκολο, αλλά μπορεί να έχει κρυμμένα προβλήματα. Η `Task` έχει μία property που λέγεται `Result`, η οποία μπλοκάρει το thread περιμένοντας το Task να ολοκληρωθεί. Μπορούμε να το χρησιμοποιήσουμε σε παρόμοια μέρη με την `await`, αλλά χωρίς να χρειάζεται η method μας να είναι μαρκαρισμένη ως `async` ή να επιστρέφει `Task`. Και πάλι όμως σπαταλάμε ένα thread. Αυτή τη φορά το calling thread μπλοκάρεται.

```
var result = MyMethodAsync().Result;
```

Η παραπάνω τεχνική αποτυγχάνει όποτε χρησιμοποιείται από ένα `SynchronizationContext` με ένα μόνο thread, όπως το UI thread. Ας σκεφτούμε τι ζητάμε από το UI thread να κάνει. Είναι μπλοκαρισμένο, περιμένοντας για το `Task` από τη `MyMethodAsync` να ολοκληρωθεί. Το πιθανότερο είναι η `MyMethodAsync` να καλεί μία άλλη TAP method, και να την περιμένει. Όταν η λειτουργία ολοκληρωθεί, χρησιμοποιείται το captured `SynchronizationContext` (το UI thread) για να κάνει `Post` την εντολή επαναφοράς της `MyMethodAsync`. Αλλά το UI thread δεν θα πάρει ποτέ αυτό το μήνυμα, γιατί είναι ακόμα μπλοκαρισμένο. Επομένως έχουμε γράψει ένα deadlock.

Γενικά πρέπει να αποφεύγεται η συγγραφή blocking κώδικα. Αν βρεθούμε στη παραπάνω κατάσταση, μπορούμε με χρήση της `ConfigureAwait`, να προσπεράσουμε το deadlock πρόβλημα, μεταφέροντας την εκτέλεση του δεύτερου μισού της method μας στο thread pool.

Κεφάλαιο 7

Async Basics Cookbook

7.1 Pausing for a period of time

Πρόβλημα

Θέλουμε να περιμένουμε ασύγχρονα για ένα χρονικό διάστημα.

Λύση

Η `Task` class έχει μία static method την `Delay` η οποία επιστρέφει ένα Task το οποίο ολοκληρώνεται μετά το προκαθορισμένο χρονικό διάστημα. Το παρακάτω παράδειγμα ορίζει ένα task το οποίο ολοκληρώνεται ασύγχρονα:

```
static async Task<T> DelayResult<T>(T result, int millisecondsDelay)
{
    await Task.Delay(millisecondsDelay);
    return result;
}
```

Σχόλια

Η `Task.Delay` είναι καλή επιλογή για unit testing ασύγχρονου κώδικα ή για την υλοποίηση retry αλγορίθμων. Παρόλα αυτά, αν θέλουμε να υλοποιήσουμε ένα timeout, καλύτερη επιλογή είναι η χρήση του `CancellationToken`.

7.2 Reporting Progress

Πρόβλημα

Θέλουμε να έχουμε ενημέρωση προόδου καθώς εκτελείται μία ασύγχρονη λειτουργία.

Λύση

```
public async void StartProcessingButton_Click(object sender, EventArgs e)
{
    // The Progress<T> constructor captures our UI context,
    // so the lambda will be run on the UI thread.
    Progress<int> progress = new Progress<int>(percent =>
    {
        textBox1.Text = percent + "%";
    });

    // DoProcessing is run on the thread pool.
    await Task.Run(() => DoProcessing(progress));
    textBox1.Text = "Done!";
}

public void DoProcessing(IProgress<int> progress)
{
    for (int i = 0; i < 100; i = i + 1)
    {
```

```
        Thread.Sleep(100); // CPU-bound work
        if (progress != null)
            progress.Report(i);
    }
}
```

Σχόλια

Η `IProgress<T>` παράμετρος μπορεί να είναι `null` αν ο χρήστης δεν θέλει αναφορά προόδου. Επομένως, θα πρέπει να κάνουμε τον σχετικό έλεγχο για `null` μέσα στην async method μας.

Η `Progress<T>` θα κάνει capture το τρέχων SynchronizationContext όταν γίνει contruct και θα καλέσει το callback της σε αυτό το context. Αυτό σημαίνει πως εάν κατασκευάσουμε την `Progress<T>` στο UI thread, τότε μπορούμε να ενημερώσουμε το UI από αυτό το callback, ακόμα και αν η ασύγχρονη method καλεί την `Report` από ένα background thread.

7.3 Περιμένοντας ένα σύνολο από Tasks να ολοκληρωθούν

Πρόβλημα

Έχουμε πολλές Tasks και θέλουμε να περιμένουμε να ολοκληρωθούν όλες.

Λύση

Το .NET framework παρέχει την `Task.WhenAll` method για αυτό το σκόπο. Η method παίρνει διάφορες tasks και επιστρέφει ένα task το οποίο θα ολοκληρωθεί όταν όλες οι άλλες ολοκληρωθούν.

```
Task task1 = Task.Delay(100);
Task task2 = Task.Delay(300);
Task task3 = Task.Delay(200);

await Task.WhenAll(task1, task2, task3);
```

Όταν τα tasks είναι τύπου `Task<T>`, τότε το αποτέλεσμα της `Task.WhenAll` είναι ένας πίνακας από `T` results.

Σχόλια

Εάν ένα από τα tasks δημιουργήσει ένα exception, τότε η `Task.WhenAll` θα ενημερώσει το return task της με αυτό το exception, οπότε η return task θα δημιουργήσει το exception. Εάν δημιουργηθούν exceptions από πολλές tasks, τότε όλα αυτά τα exceptions θα δημιουργηθούν από τη returned task. Παρόλα αυτά, όταν το task γίνεται await, μόνο ένα από αυτά τα exceptions θα δημιουργηθεί.

7.4 Περιμένοντας να ολοκληρωθεί μία Task από ένα σύνολο

Πρόβλημα

Έχουμε πολλές Tasks και θέλουμε να ανταποκριθούμε σε μία από αυτές η οποία θα είναι η πρώτη που θα ολοκληρωθεί.

Λύση

Το .NET framework παρέχει την `Task.WhenAny` method για αυτό το σκόπο. Η method παίρνει διάφορες tasks και επιστρέφει ένα task το οποίο θα ολοκληρωθεί όταν θα ολοκληρωθεί ένα Task από το σύνολο. Το επιστρεφόμενο Task είναι το Task που ολοκληρώθηκε. Το παραπάνω είναι πολύπλοκο στη περιγραφή, αλλά με ένα παράδειγμα φαίνεται πόσο απλό είναι.

```
// Returns the length of data at the first URL to respond.
private static async Task<int> FirstRespondingUrlAsync(string urlA, string urlB)
{
    var httpClient = new HttpClient();
    // Start both downloads concurrently.
    Task<byte[]> downloadTaskA = httpClient.GetByteArrayAsync(urlA);
    Task<byte[]> downloadTaskB = httpClient.GetByteArrayAsync(urlB);
    // Wait for either of the tasks to complete.
    Task<byte[]> completedTask =
    await Task.WhenAny(downloadTaskA, downloadTaskB);
    // Return the length of the data retrieved from that URL.
    byte[] data = await completedTask;
    return data.Length;
}
```

Σχόλια

Το Task που επιστρέφεται από την `Task.WhenAny` δεν ολοκληρώνεται ποτέ σε κατάσταση fault ή canceled. Πάντα επιστρέφει το πρώτο Task που ολοκληρώθηκε. Εάν αυτό το Task ολοκληρώθηκε με ένα exception, τότε το exception δεν προωθείται στο Task που επιστρέφεται από την `Task.WhenAny`. Για αυτό το λόγο, πρέπει να κάνουμε await το Task αφού ολοκληρωθεί.

Όταν ολοκληρωθεί το πρώτο task, πρέπει να λάβουμε υπ' όψιν αν θα ακυρώσουμε τα υπόλοιπα tasks. Εάν τα υπόλοιπα tasks δεν ακυρωθούν αλλά δε γίνουν ούτε await, τότε εγκαταλείπονται. Τα εγκαταλελειμμένα tasks θα έρθουν σε κατάσταση run to completion, και τα αποτελέσματά τους θα αγνοηθούν. Τυχόν exceptions επίσης αγνοηθούν.

7.5 Αποφεύγοντας το Context για τα Continuations

Πρόβλημα

Όταν μία async method συνεχίζει την εκτέλεσή της μετά από ένα await, εξ ορισμού θα συνεχίσει να εκτελείται στο ίδιο context. Αυτό μπορεί να προκαλέσει performance

προβλήματα εάν αυτό το context είναι το UI context και κάνουν resume σε αυτό ένας μεγάλος αριθμός από async methods.

Λύση

Για να αποφύγουμε να γίνει resume σε ένα context, κάνουμε await το αποτέλεσμα της ConfigureAwait και της δίνουμε false σαν παράμετρο.

```
async Task ResumeOnContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    // This method resumes within the same context.
}
async Task ResumeWithoutContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);
    // This method discards its context when it resumes.
}
```

Σχόλια

Αν έχουμε πολλά continuations τα οποία πρέπει να τρέξουν στο UI thread, μπορεί να προκληθεί performance πρόβλημα. Αυτός ο τύπος performance προβλήματος είναι δύσκολο να διαγνωσθεί, καθώς δεν είναι μια μόνο method που επιβραδύνει το σύστημα. Όσο πιο πολύπλοκη γίνεται η εφαρμογή τόσο μεγαλώνει το πρόβλημα.

Η ερώτηση είναι, *πόσα continuations* στο UI thread είναι *πολλά*; Η Microsoft σαν οδηγό ορίζει το εξής: εκατό ή περισσότερα ανά δευτερόλεπτο είναι OK, αλλά χίλια ή περισσότερα είναι πολλά.

Είναι προτιμότερο να αποφύγουμε αυτό το πρόβλημα από την αρχή. Για κάθε async method που γράφουμε, αν δεν χρειάζεται να κάνει resume στο original context, να χρησιμοποιούμε το `ConfigureAwait`. Δεν υπάρχει κάποιο μειονέκτημα σε αυτό.

7.6 Αποστολή αιτημάτων ακύρωσης (cancellation requests)

Πρόβλημα

Έχουμε κώδικα που μπορεί να ακυρωθεί η εκτέλεσή του (δέχεται δηλαδή ένα `CancellationToken`) και πρέπει να τον ακυρώσουμε.

Λύση

Ο τύπος `CancellationTokenSource` είναι το source για ένα `CancellationToken`. Το `CancellationToken` απλώς επιτρέπει στον κώδικα να ανταποκριθεί σε αιτήματα ακύρωσης. Το `CancellationTokenSource` επιτρέπει στον κώδικα να ζητήσει ακύρωση.

Το `Token` property της `CancellationTokenSource` επιστρέφει ένα `CancellationToken`, και η `Cancel` method δημιουργεί το πραγματικό αίτημα για ακύρωση.

Στο παρακάτω παράδειγμα όταν πατηθεί το Start button θα δημιουργηθεί ένα `CancellationTokenSource` και θα ξεκινήσει μία ασύγχρονη λειτουργία. Όταν πατηθεί το Cancel button θα εκτελεστεί η Cancel method και θα ακυρωθεί η ασύγχρονη λειτουργία.

```
private CancellationTokenSource _cts;

private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    StartButton.IsEnabled = false;
    CancelButton.IsEnabled = true;
    _cts = new CancellationTokenSource();
    var token = _cts.Token;
    await Task.Delay(TimeSpan.FromSeconds(5), token);
    MessageBox.Show("Delay completed successfully.");
    StartButton.IsEnabled = true;
    CancelButton.IsEnabled = false;
}
private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    _cts.Cancel();
}
```

Σχόλια

Ο παραπάνω μηχανισμός ακύρωσης δεν χρησιμοποιείται μόνο σε GUI εφαρμογές όπως στο παραπάνω παράδειγμα, αλλά και σε άλλους τύπους εφαρμογών όπως ASP .NET web εφαρμογές κλπ.

Κεφάλαιο 8

Μελλοντικές κατευθύνσεις

8.1 Συμπεράσματα

Με τον ασύγχρονο προγραμματισμό μπορούμε να δημιουργήσουμε εφαρμογές υψηλής απόκρισης. Αυτό οδηγεί σε πιο φιλικές προς τον χρήστη εφαρμογές, βελτιώνοντας δραστικά την εμπειρία του χρήστη (user experience). Όλες οι σύγχρονες γλώσσες προγραμματισμού σε συνεργασία με τα σύγχρονα λειτουργικά συστήματα, παρέχουν τεχνικές και μοτίβα ασύγχρονου προγραμματισμού.

Στο .NET οικοσύστημα της Microsoft και στη γλώσσα C#, υπάρχουν σημαντικά εργαλεία και λειτουργίες που μας επιτρέπουν να γράψουμε ασύγχρονο κώδικα ευκολότερα και γρηγορότερα από ότι το παρελθόν. Συγκεκριμένα με το async/await μοτίβο και την Task Parallel Library (TPL), η συγγραφή ασύγχρονου και παράλληλου κώδικα είναι πιο εύκολη από ποτέ.

8.2 Μελλοντικές κατευθύνσεις

Ο ασύγχρονος προγραμματισμός είναι ένα αντικείμενο που βρίσκεται συνεχώς σε εξέλιξη. Άλλωστε όπως ήδη περιγράφηκε, από την πρώτη κιόλας έκδοση της γλώσσας C# υπήρχαν ασύγχρονα μοτίβα. Με τα χρόνια και καθώς η γλώσσα C# και η .NET εξελίσσονταν, εξελίχθηκαν και τα ασύγχρονα πρότυπα. Η συνεχής μελέτη του ασύγχρονου προγραμματισμού είναι σίγουρο ότι θα οδηγήσει σε περεταίρω βελτιώση και απλοποίηση των σχετικών μοτίβων.

Αναφορές

[1] CLR via C#, 4th edition, Jeffrey Richter, Microsoft Press 2012

[2] Windows via C++, 5th edition , Jeffrey Richter-Christophe Nasarre, Microsoft Press 2008

[3] Concurrency in C# Cookbook, Stephen Cleary, O' Reilly Media 2014

[4] Async in C# 5.0, Alex Davies, O' Reilly Media 2012

[5] https://blog.stephencleary.com/2013/11/there-is-no-thread.html, Stephen Cleary, 2013