C# async await

Asynchronous programming

Synchronous – בו זמני, מתואם לזמן מסוים.

Asynchronous - שאינו בו זמני, אינו מתואם לזמן מסוים.

מתי בדרך כלל נצטרך?

1. פעולות שתלויות בקלט או פלט מגורמים חיצוניים כמו:
   - קבלת מידע משירות אינטרנט
   - קריאה מ DB או כתיבה ל DB
   - קריאה מקובץ או כתיבה לקובץ
2. פעולות שדורשות זמן עיבוד רב

דוגמה לסדרת פעולות אסינכרוניות

דוגמא לסדרת פעולות אסינכרוניות בארוחת בוקר:

1. למזוג כוס קפה
2. לחמם מחבת ולטגן שתי ביצים
3. לחתוך סלט.
4. להכניס שתי פרוסות לחם למצנם
5. למרוח חמאה וריבה על הצנימים.
6. למזוג כוס מיץ תפוזים.

המשך דוגמא

עקרת בית מנוסה תבצע את הפעולות האלה בצורה אסינכרונית:

תחמם מחבת, תחתוך ירקות, תשפוך ביצים למחבת, תרתיח מים לקפה, וכו'.

בכל שלב בתהליך מתחילים משימה חדשה או פונים להמשך טיפול במשימה שדורשת זאת.

אם אדם אחד מכין את כל ארוחת הבוקר – אזי זו **פעולה אסינכרונית שאינה מקבילית**.

אם יותר מאדם אחד מכין את ארוחת הבוקר – אחד אחראי על החביתה, אחד על הסלט, אחד על מיץ התפוזים - אזי זו **פעולה אסינכרונית מקבילית** (parallel). במצב כזה כל אחד אחראי רק על משהו אחד ואם יש צורך להמתין, ימתין, כמו למשל להפיכת החביתה או עד שהמצנם "יקפיץ" את הפרוסות.

איך זה יראה בקוד סינכרוני?

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Egg eggs = FryEggs(2);
Console.WriteLine("eggs are ready");
Vegetables veggies = WashVegetables();
Console.WriteLine("vegetables are ready");
Toast toast = ToastBread(2);
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```

בקוד סינכרוני ההכנה תארך זמן רב, וחלק מהאוכל יתקרר עוד לפני הארוחה.

אם רוצים שהמחשב יבצע פעולות בצורה אסינכרונית, צריך לכתוב קוד אסינכרוני.

בכל שפה צריך ללמוד איך כותבים בה קוד אסינכרוני.

הפונקציות נמצאות בשקופית הבאה.

```
private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");

private static Toast ToastBread(int slices)
{
    for (int slice = 0; slice < slices; slice++)
         {
        Console.WriteLine("Putting a slice of bread in
the toaster");
 }
    Console.WriteLine("Start toasting...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

```
private static Vegetables WashVegetables()
{
    Console.WriteLine($"washing the vegetables....");
    Console.WriteLine("soaking lettuce...");
    Task.Delay(3000).Wait();
    Console.WriteLine("soaking peppers...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put vegetables on plate");
    return new Vegetables();
}


private static Egg FryEggs(int howMany)
{
    Console.WriteLine("Warming the egg pan...");
    Task.Delay(3000).Wait();
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put eggs on plate");

    return new Egg();
}


private static Coffee PourCoffee()
{
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}
```

קוד סינכרוני ואסינכרוני

אחת המעלות הגדולות של קוד סינכרוני היא **הקריאות** שלו.

קל להבין אילו פעולות מתבצעות ובאיזה סדר.

כתיבת קוד אסינכרוני ב C# בעבר השפיעה מאד על הקריאות שלו.

ב C# 5.0 עם **.NET FW 4.5**. התווספו `async` ו- `await` שמקלות מאד על הכתיבה של קוד אסינכרוני ועל הקריאות שלו.

שימוש ב `async` ו `await` יכלול תמיד גם שימוש במחלקות `Task` ו `Task<T>` שנמצאות ב `System.Threading.Tasks`.

קוד אסינכרוני ו- UI

ב UI קריטי לממש קוד אסינכרוני.

ניקח כדוגמא לחיצה על כפתור במסך Win Form או בדף Web.

<u>בצורה סינכרונית</u> -

מתבצעת פונקציה כלשהי, כל המסך "יקפא" עד לסיום ריצת הפונקציה. אם מדובר בפניה לשרת וכדומה, זה יכול לארוך זמן, וחווית המשתמש נפגעת מאד אם אינו יכול לבצע שום פעולה עד לסיום הפעולה של הלחיצה הקודמת.

<u>בצורה אסינכרונית</u> –

משתמש לוחץ על כפתור, מתבצעת פונקציה, בינתיים המשתמש יכול להמשיך את הפעילות בדף, כזו שלא דרושה לה התוצאה של פעולת הלחיצה.

Don’t block, await instead

הקוד לעיל להכנת ארוחת הבוקר הוא גרוע. זהו קוד סינכרוני לביצוע פעולות אסינכרוניות.

הקוד עוצר עד לסיום ביצוע פעולה, בלי לאפשר להמשיך בפעולות אחרות בינתיים.

זה בערך כמו להכניס פרוסות למצנם ולבהות בו עד שהן קופצות, בלי יכולת לעשות כלום או אפילו לדבר עם מישהו תוך כדי.

בשלב הראשון, נכתוב קוד שיהיה מסוגל "לענות" למישהו תוך כדי, ובמילים אחרות – נגרום שה UI לא יקפא במהלך ביצוע הקוד, וניתן יהיה ללחוץ עם העכבר על המסך במקומות אחרים.

קוד מעודכן – שלב א'

ביצוע פעולות במקביל

אחרי שיפור הקוד בשלב א', קבלנו קוד שמונע "הקפאה" של UI. השלב הבא הוא לבצע את הפעולות במקביל – בדומה לאופן שבו אדם היה מבצע אותן. כמובן שזה מתאים רק למקרים שאין תלות בין פעולה לפעולה.

לצורך כך נשתמש במחלקה `System.Threading.Tasks.Task` ובעוד כמה מחלקות נלוות אליה. מחלקה זו מאפשרת להחזיק ייחוס לפעולה אותה רצינו להתחיל, ונרצה להתייחס לתוצאתה, כשתסתיים.

נבצע בקוד לעיל את השינויים הבאים:

1. ניצור משתנה מסוג `Task` – באמצעותו נחזיק "קישור" לפעולה (במילים פשוטות - פונקציה). המשתנה מסוג `Task` מסייע לנו לשמור על קשר עם הפונקציה ולהיות מסוגלים לעקוב אחרי סיום ביצוע הפונקציה, אם נרצה. חשוב להבין: ברגע שמבצעים השמה של פונקציה ל `Task` **מתבצעת קריאה לפונקציה.**

2. בנקודה בקוד שבה רוצים לוודא שביצוע הפונקציה הושלם – נבצע `await` ל `Task`, כך גם נוכל לקבל ערך מוחזר מהפונקציה, אם ישנו.
   - שורות הקוד שאחרי `await` לא יתבצעו עד שתסתיים ההרצה של הפונקציה לה מחכים.
   - אם התבצעה קריאה לפונקציה אחרת לפני ה `await`, היא **כן** יכולה להמשיך להתבצע ברקע.
   - הפונקציה הקוראת יכולה בינתיים גם היא להמשיך בביצוע הפקודות. נניח פונקציה A זימנה פונקציה א-סינכרוניתת B, וב B יש פקודת await, B תעצור אך A תוכל להמשיך בפקודות הבאות.

קוד מעודכן – שלב ב'

שילוב פעולות סינכרוניות ואסינכרוניות

- לעיתים פעולה יכולה להיות מורכבת מפעולה אסינכרונית ומפעולה סינכרונית, אחת או יותר.
- אם חלק כלשהו בפונקציה הוא אסינכרוני, אזי כל הפונקציה נחשבת לאסינכרונית.
- במקרה כזה נכון לאחד את כל הפעולות לפונקציה אחת ולוודא את סדר הפעולות הרצוי.
- הקוד בפונקציית ה Main בדוגמאות לעיל הוא דוגמא לפונקציה שמשלבת פעולות סינכרוניות ואסינכרוניות.
- כשיוצרים פונקציה ובתוכה המתנה (`await`) לסיום פעולה אסינכרונית, חובה לסמן את הפונקציה במילה `async` לפני הגדרת הסוג המוחזר.

בדוגמא של ארוחת הבוקר, הכנת הצנימים עם חמאה וריבה מורכבת מפעולה אסינכרונית אחת (הכנסת הפרסות למצנם) ושתי פעולות סינכרוניות (מריחת חמאה, מריחת ריבה).

נאחד את שלושת הפעולות לפונקציה אחת.

שילוב sync & async - קוד

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

פונקציות שימושיות של Task

- `Delay` – פונקציה סטטית, מחזירה task שמסתיימת כאשר חולף הזמן המוגדר. הזמן = פרמטר של X מילי שניות (milliseconds ובקיצור ms), שניה = 1000 ms.

```
await Task.Delay(3000);
```

- `Run` – מריצה פונקציה מסוימת שמתקבלת כפרמטר. אפשר גם לשלוח lambda expression
- `WhenAny` – הפונקציה מקבלת אוסף של `Task` וברגע שסיימה ביצוע של **אחת** מהן מחזירה `Task` של ה `Task` שהסתיימה. (`Task<Task>`)
- `WhenAll` – הפונקציה מקבלת אוסף של `Task` ומחזירה `Task` שמייצגת את סיום הרצת **כל** רשימת המשימות.

`WhenAny, WhenAll` מתאימות רק ל tasks שמחזירות את אותו ערך

- `WaitAny`
- `WaitAll` - ממתינה לסיום

בקצרה:

1. ניתן להתחיל ביצוע של כמה פונקציות במקביל באמצעות Task עבור פונקציה שלא מחזירה ערך, ו Task<T> עבור פונקציה שמחזירה ערך T.
2. קריאה לפונקציה כשמצד שמאל ישנו אובייקט מסוג Task כלשהו, תאפשר המשך של ביצוע הקוד תוך כדי ביצוע הפונקציה.
3. ניתן לבצע קריאה לפונקציה אסינכרונית רק מתוך פונקציה אסינכרונית (מסומנת ב async)
4. כאשר רוצים לוודא סיום ביצוע פונקציה ניתן להמתין לה באמצעות המילה await.
5. ניתן לבצע await על פונקציה ישירות או על Task של הפונקציה.
6. ניתן לבצע await רק לפונקציה אסינכרונית (מסומנת ב async), כמובן (אחרת קריאה לפונקציה פשוט מחזירה ערך...)
7. await ניתן לבצע רק בתוך פונקציה שמסומנת כ async
8. אם מפעילים כמה פונקציות, בלי שימוש ב Task, אחת אחרי השניה, עם await לכל אחת, הפונקציות יתבצעו אחת אחרי השניה ולא במקביל, אבל ה UI לפחות יהיה משוחרר.

Overview of the asynchronous model

משתמשים ב Task ו Task<T> שהם המודלים לפעולות אסינכרוניות. ברוב המקרים השימוש פשוט:

1. פעולות I\O (קלט פלט מול גורמים שונים) – מבצעים await לפונקציה שמחזירה Task או Task<T>, בתוך פונקציה שמסומנת ב async.
2. פעולות שתלויות בזמן עיבוד רב – מבצעים await לפונקציה שמופעלת ברקע באמצעות Task.Run או Task.

async function name convention

קונבנציה (convention) של פונקציות אסינכרוניות:

- אם מחזירה ערך שניתן לחכות לו (awaitable) – שם הפונקציה יסתיים ב Async
- אם לא מחזירה ערך והיא א-סינכרונית – שם הפונקציה יתחיל במילה Begin או Start

בפונקציות שנכתוב נתאים את עצמנו לקונבנציה הזו.

Task Parallel Library (TPL)

1. Task Parallel Library - מה זה?
2. Creating a Task – Implicitly & Explicitly
3. Continuation Tasks – משימות המשך
4. Attached & Detached
5. המתנה לסיום משימה – באופן סינכרוני ואסינכרוני
6. Flaten ,AggregateException – Handling Exceptions
7. Cancelling a Task
8. For, ForEach, Invoke – Parallel
9. – PLINQ
10. async with controller actions

Task Parallel Library - מה זה?

קבוצה של מחלקות ופונקציונליות שנועדו לשרת Task.

Task – פעולה א-סינכרונית.

ה Task דומה ל Thread אבל היא יותר דקלרטיבית – מגדירה מה רוצים לעשות, ולא "איך" זה יקרה.

Task Parallelism– task אחת או יותר שמתבצעות בו–זמנית. מדובר כמובן במשימות עצמאיות.

יתרונות לשימוש ב task:

1. קוד יעיל יותר, ניצול משאבי המערכת. (משתמשת ב ThreadPool ששודרג עם אלגוריתמים שונים המסייעים להתאים את מספר ה Threads הרצוי, עם איזון לשמירה על מקסימום תפוקה. בזכות השדרוג הזה, task נחשבת קלילה וניתן להשתמש בה רבות בקוד כדי לקבל מקביליות וקוד אסינכרוני יעיל.
2. דרך נוחה יותר לכתיבת קוד איסנכרוני ומקבילי בהשוואה ל thread שקדם לה.

Creating a Task - Implicitly

ניתן ליצור task בצורה מרומזת באופן הבא:

1. שימוש בפונקציה Parallel.Invoke – מקבלת מספר בלתי מוגבל של delegates, לדוגמא:

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

א - Creating a Task – Explicitly

ניתן ליצור task בצורה מפורשת באופנים הבאים:

1. יצירת מופע של System.Threading.Tasks.Task - עבור פונקציה שאינה מחזירה ערך, או של System.Threading.Tasks.Task<Tresult> - עבור פונקציה שמחזירה ערך.

כדי להפעיל את הפונקציה יש לזמן את הפונקציה Start.

```
Task taskA = new Task(() => Console.WriteLine("Hello taskA."));
// Start the task.
taskA.Start();
```

2. שימוש בפונקציה Task.Run שמייצרת ומפעילה task בפקודה אחת. נוח מאד ליצור משימות בצורה כזו, אם אין צורך לטפל בהן בהמשך הקוד.

```
Task taskA = Task.Run(() => Console.WriteLine("Hello from taskA."));
```

3. Task.Factory.StartNew – אופציה זו מאפשרת להגדיר הגדרות שונות ל Task (כפרמטר נוסף לפונקציה, מלבד הפעולה עצמה)

ב - Creating a Task – Explicitly

3. Task.Factory.StartNew – אופציה זו מאפשרת להגדיר הגדרות שונות ל Task (כפרמטר נוסף לפונקציה, מלבד הפעולה עצמה) , או שליחת נתונים שמשויכים ל Task, נקראים State.

```
Task<Double>[] taskArray = { Task<Double>.Factory.StartNew(() => DoComputation(1.0)),
                             Task<Double>.Factory.StartNew(() => DoComputation(100.0)),
                             Task<Double>.Factory.StartNew(() => DoComputation(1000.0)) };
var results = new Double[taskArrayWithFactory.Length];
Double sum = 0;
for (int i = 0; i < taskArray.Length; i++) {
    results[i] = taskArray[i].Result;//if the result is not ready, will wait for it
    sum += results[i];
}
Console.WriteLine("{0:N1}", sum);
```

lambda expression בלולאות

לעיתים מייצרים tasks בתוך לולאה, ומעונינים להשתמש במשתנה שמוגדר ב scope של הלולאה בתוך ה lambda expression. המשתנה אמנם יהיה נגיש בתוך הביטוי, אבל במקרים מסוימים הוא יועבר כ reference ולא כערך (גם אם הוא value type) מה שיגרו שבסופו של דבר הערך האחרון שיהיה בו יתקבל בכל ה task.

הפתרון: ליצור משתנה מקומי מחדש ולשלוח כ state ל Factory.StartNew

State מתקבל כפרמטר לפונקציה ב lambda expression.

<הדגמה בקוד>

משימות המשך

ניתן להגדיר משימה כמשימת המשך של משימה אחרת.

במקרה כזה, רק כאשר תסתיים משימת ה"אב", תתבצע המשימה הבאה.

למשימת ההמשך יישלח reference של המשימה הקודמת לה, כך היא תוכל גם לגשת למאפיין Result של המשימה הקודמת, ולהשתמש בו במידת הצורך.

<דוגמת קוד>

Attached & Detached Tasks

כאשר מייצרים משימה באמצעות Factory.startNew, אפשר להגדיר את המשימה עם האופציה AttachedToParent. ה parent ממתינה לכל המשימות ששויכו אליה (בצורה מרומזת, לא צריך לציין זאת במפורש).

Task ספציפי יכול להגדיר על עצמו DenyChildattach כדי למנוע את האפשרות לשייך אליו child task.

כך לא "יסתכן" בצורך להמתין לתת-משימות.

נשים לב: לא ניתן לשייך לפונקצית ה Main תת משימות, ולכן Attached tasks לא יפתרו את הבעיה שקיימת ב console applications , בה ה Main מסתיימת לפני שמשימות שנוצרו בה הסתיימו.

המתנה לסיום משימה

לעיתים נרצה להמתין לסיום משימה. זה יכול לקרות כאשר:

1. יש צורך בתוצאה שלה
2. פונקצית ה Main של console app עלולה להסתיים לפני שכל המשימות בה הסתיימו.
3. עלולים להצטרך לטפל בשגיאות שהתרחשו במשימות.

המתנה לסיום משימה מתאפשרת במספר דרכים.

המתנה – Wait ,WaitAny, WaitAll

WaitAll - פונקציה זו עוצרת את כל ה thread מלהמשיך עד לסיום המשימות שנשלחו לפונקציה זו.

WaitAny – כנ"ל, אבל עד לסיום אחת מהמשימות שנשלחו לפונקציה.

שתי הנ"ל הן פונקציות סטטיות.

Wait – המתנה לסיום המשימה עליה מופעלת הפונקציה.

נשים לב: גישה ל Result של משימה תבצע למעשה המתנה עד לסיומה (תעצור את ה Thread)

המתנה א-סינכרונית –WhenAll

פונקציה אסינכרונית שמקבלת רשימת משימות שעליה "לעקוב" אחר סיומן. מחזירה task.

סטטוס ה task יכול להיות כמה אפשרויות – מוגדרות ב TaskStatus enum:

1. הצלחה – אם כל המשימות הסתיימו בהצלחה, RanToCompletion
2. כשלון – מספיק שאחת הסתיימה בכשלון כדי שה Task הכולל יוחזר בסטטוס כשלון - Faulted
3. ביטול – מספיק שביטלו אחת המשימות כדי שה task הכולל יוחזר בסטטוס ביטול – Cancelled

WhenAll עבור tasks עם ערך מוחזר:

אם הסתיימה ב RanToCompletion יוחזר מערך שלכל התוצאות. לכן חייבת לקבל tasks שמחזירות אותו סוג.

WhenAny

תמתין לסיום אחת מהמשימות שנשלחו אליה כפרמטר.

אם האחרות לא תבוטלנה – הן תמשכנה לרוץ אבל אין אחריות על כך (למשל סיום Main ב console app) וא אחד גם לא יתעניין בערך המוחזר מהן, אם ישנו.

Redundant operations: Consider an algorithm or operation that can be performed in many ways. You can use the WhenAny method to select the operation that finishes first and then cancel the remaining operations.

Interleaved operations: You can start multiple operations that must finish and use the WhenAny method to process results as each operation finishes. After one operation finishes, you can start one or more tasks.

Throttled operations: You can use the WhenAny method to extend the previous scenario by limiting the number of concurrent operations.

Expired operations: You can use the WhenAny method to select between one or more tasks and a task that finishes after a specific time, such as a task that's returned by the Delay method. The Delay method is described in the following section.

טיפול בשגיאות

שגיאות שמתרחשות בתוך task יתגלגלו לפונקציה שיצרה את ה tasks.

מכיוון שיכולות להתרחש כמה שגיאות – נניח הפעלנו 3 tasks, אזי יכולה להתרחש שגיאה בשתיים מהמשימות (או כמובן בשלושתן). במצב כזה נוצר AggregateException – שכשמו, אוסף את כל ה exceptions שמתרחשים בתהליך ביצוע task.

נכתוב בלוק try…catch ובתוכו את ההמתנה לסיום המשימה (עם אחת מפונקציות ה Wait) או גישה ל Result.

אם יש קינון של משימות (משימה 1 יצרה משימה 2 שיצרה משימה 3), עדיין יתרחש AggregateException אבל פרטי השגיאה יופיעו בתוכו כ InnerException.

https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library