Async

כשיש פונקציה זמן הריצה שלה ארוך מן הרגיל לדוגמה קריאה מקובץ או מסד נתונים, פניה ל web api שמחזיר המון נתונים או העלאת והורדת קבצים וכן כל פעולה אחרת הלוקחת זמן רב.

כאשר הפונקציה מופעלת בצורה סינכרונית, הקוד לאחר הפעלת הפונקציה ימתין לסיום הריצה של הפונקציה. ובמידה ויש לנו UI אז הוא יקפא ללא תגובה כל עוד הריצה הנוכחית לא תסתיים.

דוגמה לקוד סינכרוני - פעולה אחרי פעולה.

```
LongProcess();
ShortProcess();

static void LongProcess()

{
Console.WriteLine("LongProcess Started");

//some code that takes long execution time

System.Threading.Thread.Sleep(4000); // hold execution for 4 seconds

Console.WriteLine("LongProcess Completed");
}

static void ShortProcess()

{
Console.WriteLine("ShortProcess Started");

//do something here

Console.WriteLine("ShortProcess Completed");

}
```

Asynchronous programming

בתכנות אסינכרוני הקוד רץ באופן נפרד. כך שלא צריך להמתין לסיומו.

מתוך ההרצה הראשית ניתן להריץ מספר פעולות במקביל.

האפליקציה העיקרית תמשיך למשפט הבא. מבלי שמשהו יעצור ויחסום אותה.

ב net. מממשים את זה ע"י המילים async, await ו task.

כך ניהיה יותר קל לכתוב קוד אסינכרוני שרץ ברקע.

```
static async void LongProcess(){
   Console.WriteLine("LongProcess Started");
   await Task.Delay(4000); // hold execution for 4 seconds
   Console.WriteLine("LongProcess Completed");
}
```

```
static void ShortProcess() {
   Console.WriteLine("ShortProcess Started");
   //do something here
   Console.WriteLine("ShortProcess Completed");
}
```

Async

המילה הזו מסמנת פונקציה כאסינכרונית. פונקציה זו תרוץ בנפרד מהקוד הראשי שהוא ימשיך לרוץ כרגיל. פעולה שרצה ברקע של הקוד שהפעיל אותה.

פונקציה המסומנת בasync לרוב תחזיר Task .

כל פונקציה שמפעילה פונקציה כזו תצטרך להיות ג"כ async . לכן פונקציית הmain לעולם תיהיה אסינכרונית.

Await

המילה הזו מורה שיש לחכות לסיום הפעולה המסומנת בawait. הקוד הראשי יעצור עד שיחזור ערך מהפונקציה.

ניתן להשתמש בawait רק עבור פעולת async

פונקציה שרוצים להפעיל אותה עם await לא יכולה להחזיר void

מקובל להשתמש בawait רק לפני שצריך את הערך המוחזר

שימי לב – כל מה שלא תלוי בפעולה האסינכרונית יוכל לרוץ בינתיים . הקוד שכן תלוי בפעולה או בערך המוחזר שלה – צריך להמתין וזאת ע"י המילה await.

TASK

פונקציה אסניכרונית מחזירה task. הtask מיצג את הפעולה האסינכרונית. במידה ונרצה להחזיר ערך מהפונקציה נשתמש בTask<TResult>. במקרה כזה הערך המוחזר יהיה בresult של האוביקט מסוג task

לדוגמה

```
Task<int> task = ReadFile(filePath);
int length = await task;
```

או בקיצור:

```
Var length = Await ReadFile(filePath);
```

פעולה המשלבת פעולות סינכרוניות ואסינכרוניות נכניס לתוך פונקציה אסינכרונית אחת שתכיל את הפעולות לפי הסדר הנכון.

עפ"י הקונבנציה פונקציה אסינכרונית המחזירה ערך תקרא בסיומת Async ושאינה מחזירה ערך תתחיל בbegin או בstart

פונקציות שימושיות של המחלקה TASK

Delay - פונקציה סטטית, מחזירה TASK ומסתימת לאחר סיום פרק הזמן הנשלח לפונקציה במילי שניות.

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

WhenAny - מקבלת אוסף של Task וממתינה עד לביצוע של אחת מהן. אם האחרות לא תבוטלנה – הן תמשכנה לרוץ אבל אין אחריות על כך (למשל סיום Main ב console app) ואף אחד גם לא יתעניין בערך המוחזר מהן, אם ישנו.

WhenAll – מקבלת כנ"ל וממתינה לסיום כל המשימות.

WhenAny, WhenAll מתאימות רק ל tasks שמחזירות את אותו ערך

(WaitAny/ WaitAll) עוצר ממש הכל עד לסיום.

Run - מריצה פונקציה מסוימת שמתקבלת כפרמטר אפשר גם לשלוח lambda expression. שימושי עבור פעולות של זמן עיבוד רב.

Task Parallel Library

זוהי ספריה עם יכולות שונות סביב ה TASK. פרוש המילה **Parallel** היא מקבילי

כשאנו משתמשים בtask אנחנו מגדירים מה לעשות ופחות מתעסקים באיך זה יקרה וזה נחשב לקוד יותר נוח ויעיל בהשוואה לthreads

יצירה של task יכולה להיות בצורה מרומזת או מפורשת:

Creating a Task – Implicitly

ע"י הפונקציה invoke() במחלקה Parallel שמקבלת מספר לא מוגבל של delegates

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

Creating a Task – Explicitly

1. יצירת מופע ע"י new Task ואח"כ הפעלה שלו באמצעות הפונקציה start()

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

2. שימוש בפונקציה Task.Run שמייצרת פונקציה ומפעילה אותה גם. נוח למשימות שאין להן עוד טיפול בקוד

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

3. שימוש בפונקציה Task.Factory.StartNew שמקבלת הגדרות נוספות ושליחת פרמטרים לTask:

```
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);
```

הערה: באם מיצרים tasks ע"י ()=> בתוך לולאה ומעבירים משתנים יש ליצור משתנה מקומי חדש כדי שיהיה לנו את הערך עצמו ולא reference .

אופציות של משימות אב ובן

.*AttachedToParent* מגדירים על הבן ואז האבא מחכה לסיום של הבנים. שימי לב שמשמעותי רק כשמחכים למשימת האב, בלי זה ההמתנה תהיה רלוונטית רק לאב ולא לבנים.

DenyChildAttach משימה שמוגדרת עם אופציה זו , גם עם יש לה בנים מקושרים היא לא מחכה לסיומם

לסיכום בקוד אסינכרוני , מכניסים לעבודה פעולה בזמן שמחכים למשהו שני

לעומת זאת בTP (task parallelism) יש כמה משימות שמבוצעות במקביל, שימושי במשימות שלוקחות זמן עיבוד רב שמתבצעות ברקע.