Haladó fejlesztési technikák

Szálkezelés
Thread
Task

SZÁLKEZELÉS

System.Threading.Thread

- A Thread osztály példányosításával hozható létre szál

```
using System;
using System.Threading;

namespace _01_HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t = new Thread(Koszon);
            Console.WriteLine("Szál indítása!");
            t.Start();
            Console.WriteLine("Várakozás a befejezésre...");
            t.Join();
            Console.ReadLine();
        }

        static void Koszon()
        {
            Console.WriteLine("Hello világ!");
        }
    }
}
```

System.Threading.Thread (kivonatos referencia)

Metódusok	
Start()	Szál indítása
Suspend(), Resume()	Szál felfüggesztése, illetve folytatása
Abort()	Szál leállítása
GetHashCode()	Szál azonosítójának lekérése
Sleep()	Várakozás a megadott időintervallum elteltéig
Join()	Várakozás az adott szál befejeződésére
Tulajdonságok	
CurrentCulture, CurrentUICulture	A szálhoz tartozó aktuális kultúra, illetve a szálhoz tartozó felhasználói felület kiválasztott nyelve
IsBackground	Az adott szál háttérszál vagy előtérszál*
IsThreadPoolThread	Az adott szál a ThreadPool egyik szála-e
ManagedThreadID	A szál egyedi azonosítója
Name	A szál megnevezése
Priority	A szál prioritása (fontossági szintje)
ThreadState	A szál aktuális állapota(i)

* A programok futása véget ér, ha az utolsó előtérszál is lefutott (az esetleg még futó háttérszálak ekkor automatikusan megszűnnek).

ThreadStart delegate

- **A Thread objektum konstruktora ThreadStart illetve ParameterizedThreadStart delegáltat fogad el bemeneti paraméterként**
- **ThreadStart egy visszatérési érték és bemeneti paraméter nélküli metódus lehet, azaz gyakorlatilag void()**
  - Ugyanúgy paraméterezhető lambda kifejezéssel mint az Action

```
Thread t = new Thread(Koszon);
Thread t2 = new Thread(() => {
    Thread.Sleep(1500);
    Console.WriteLine("Hello világ!");
});
t.Start();
t2.Start();
```

Szál paraméterezése

- A ParameterizedThreadStart delegált egy void(object) szignatúrájú metódus
- Szálnak bemeneti paramétert objectként kell átpasszolni a Start() metódus paramétereként

```
static void Koszon(object o)
{
    //string nev = (string)o;
    string nev = o.ToString(); //ebben az esetben eleg ez is
    Console.WriteLine("Hello " + o + "!");
}

static void Main(string[] args)
{
    Thread t = new Thread(Koszon);
    Console.WriteLine("Szál indítása!");
    t.Start("Pistike");
    Console.WriteLine("Várakozás a befejezésre...");
    t.Join();
    Console.ReadLine();
}
```

Több szál végrehajtása egyidejűleg

- **A konzolra egyszerre csak egy szál tud kiírni: amikor az egyik szál hozzáfér az erőforráshoz, a másik várakozik**
  - „versenyhelyzet”

```
Thread t = new Thread(() => {
    Thread.Sleep(1500);
    Console.WriteLine("asd");
});
t.Start();

for (int i = 0; i < 20; i++)
{
    Thread.Sleep(100);
    Console.WriteLine("Várakozás #{0}", i);
}

Console.ReadLine();
```

Több szál végrehajtása egyidejűleg

- **A konzolra egyszerre csak egy szál tud kiírni: amikor az egyik szál hozzáfér az erőforráshoz, a másik várakozik**
  - „versenyhelyzet”

Futtatáskor:

```
Várakozás #0
Várakozás #1
Várakozás #2
Várakozás #3
Várakozás #4
Várakozás #5
Várakozás #6
Várakozás #7
Várakozás #8
Várakozás #9
Várakozás #10
Várakozás #11
Várakozás #12
asd
Várakozás #13
Várakozás #14
Várakozás #15
Várakozás #16
Várakozás #17
Várakozás #18
Várakozás #19
```

Másik futtatáskor:

```
Várakozás #0
Várakozás #1
Várakozás #2
Várakozás #3
Várakozás #4
Várakozás #5
Várakozás #6
Várakozás #7
Várakozás #8
Várakozás #9
Várakozás #10
Várakozás #11
Várakozás #12
Várakozás #13
asd
Várakozás #14
Várakozás #15
Várakozás #16
Várakozás #17
Várakozás #18
Várakozás #19
```

További lehetőségek többszálúság megvalósítására

- **Aszinkron metódusok**
- **Thread**
- **ThreadPool**
- **BackgroundWorker**
- **TPL / Task**
- **TPL / AsyncAwait**
- **TPL / Parallel**

Aszinkron metódusok

- **Aszinkron metódus: a feladat megkezdése után, de még mielőtt az befejeződött, elkezdődhet egy másik feladat végrehajtása**

```
//eredeti, szekvencialis megoldas:
WebClient wc = new WebClient();
Console.WriteLine("A letöltés elindult...");
wc.DownloadFile(new Uri("http://users.nik.uni-
obuda.hu/prog3/Eloadas/WHP_EA_07_Process_Thread.pptx"),
"a_prezi.pptx");
Console.WriteLine("Letöltés vége!");
```

Aszinkron metódusok

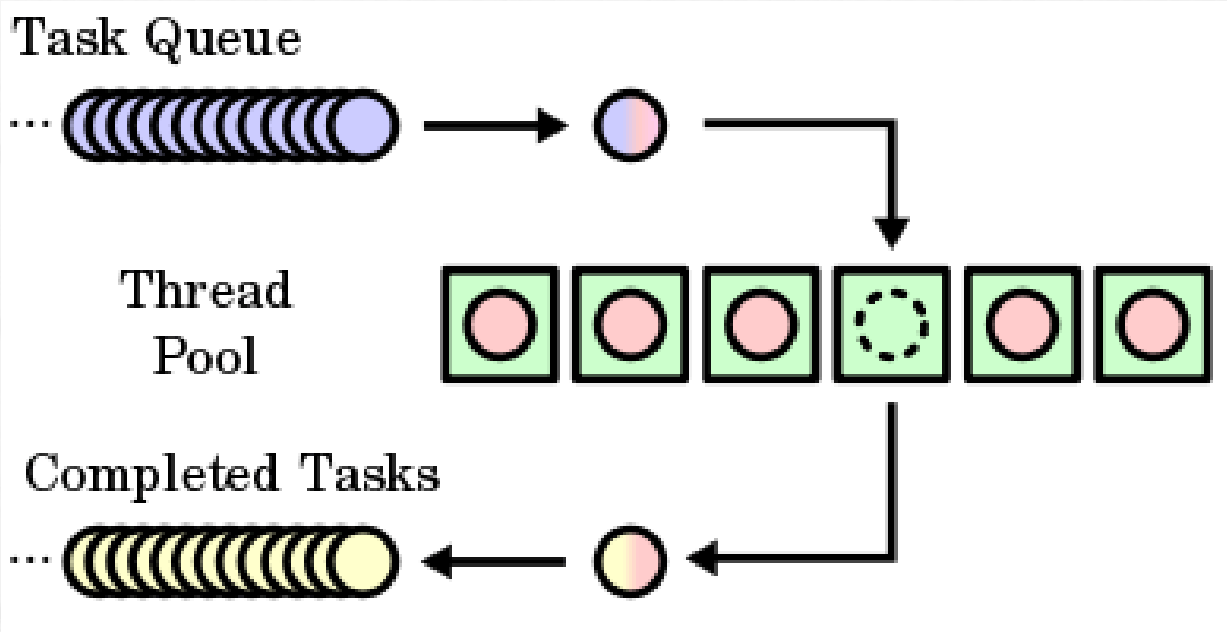
```
static void Main(string[] args)
{
    WebClient wc = new WebClient();
    wc.DownloadFileCompleted += Wc_DownloadFileCompleted;
    wc.DownloadFileAsync(new Uri("http://users.nik.uni-
obuda.hu/prog3/Eloadas/WHP_EA_07_Process_Thread.pptx"),
"a_prezi.pptx");
    Console.WriteLine("A letöltés elindult...");

    Console.ReadLine();
}

private static void Wc_DownloadFileCompleted(object sender,
System.ComponentModel.AsyncCompletedEventArgs e)
{
    Console.WriteLine("Letöltés vége!");
}
```

Thread pooling

- **A szálak (egyenkénti) létrehozása költséges, hatékonyság szempontjából jobb ötlet lenne „újrahasznosítható” szálakat regisztrálni az operációs rendszerben, és amikor szükség lesz rá, csak felhasználni**



A kép forrása: https://www.c-sharpcorner.com/

Thread pooling

- A QueueUserWorkItem egy WaitCallback delegáltat vár, ami void(object) szignatúrájú
- Akkor kerül végrehajtásra, ha lesz szabad poolbeli Thread

```
static void Main(string[] args)
{
    for (int i = 0; i < 20; i++)
        ThreadPool.QueueUserWorkItem(Kalkulacio, i

        Console.WriteLine("Várakozunk a nagy válas

    Console.ReadLine();
}

static void Kalkulacio(object o)
{
    Thread.Sleep(1000);
    Console.WriteLine(o + ": az élet értelme 42");
}
```

```
Várakozunk a nagy válaszra ...
3: az élet értelme 42
2: az élet értelme 42
1: az élet értelme 42
0: az élet értelme 42
5: az élet értelme 42
4: az élet értelme 42
6: az élet értelme 42
7: az élet értelme 42
8: az élet értelme 42
9: az élet értelme 42
10: az élet értelme 42
12: az élet értelme 42
11: az élet értelme 42
13: az élet értelme 42
14: az élet értelme 42
16: az élet értelme 42
15: az élet értelme 42
18: az élet értelme 42
17: az élet értelme 42
19: az élet értelme 42
```

További lehetőségek többszálúság megvalósítására

- Aszinkron metódusok
- Thread
- ThreadPool
- BackgroundWorker
- **TPL / Task**
- **TPL / AsyncAwait**
- **TPL / Parallel**

Task

Thread

- Alacsony szintű eszköz szálak definiálására
- Nehezen kezelhetőek az egymásra épülések
- Költséges létrehozni, megszüntetni, váltani közöttük
- Nincs beépített lehetőség „barátságos” leállításra
- Nincs visszatérési érték definiálására lehetőség
- A metódus felparaméterezése sem túl fejlesztőbarát

Task

- **Aszinkron elvégzett „feladat”**
- **A háttérben egy `ThreadPool` egy eleme van: magasabb absztrakciós szinten dolgozunk**
- **Lehetőségek**
  - Visszatérési érték
  - Nem-blokkoló egymásra épülések
  - Kivételkezelés
  - Leállítás

Hagyományos szál vs. Task

- **Használjunk hagyományos szálakat, ha:**
  - **A szálnak a normálistól eltérő prioritással kell futnia** – a ThreadPool-on minden szál normál prioritással fut
  - **Előtérszálra van szükség** – minden ThreadPool szál háttérszál
  - **A szál extrém hosszú műveletet végez (pl. az alkalmazás teljes élettartama alatt futnia kell)**
  - **Abort()-ra lehet szükség**
- **Minden más esetben Task ajánlott**
  - Könnyen alkalmazható, robosztus eszköz
  - Széleskörű alkalmazási lehetőségek (feladatmegszakítás, kivételkezelés, ütemezés, stb)

Task indítása

```
Task task1 = new Task(new Action(PrintMessage));
Task task2 = new Task(delegate { PrintMessage(); });
Task task3 = new Task(() => PrintMessage());
Task task4 = new Task(() => { PrintMessage(); });
```

- Taskok indítása paraméterrel:
- A new Task Action-t vagy Action<object>-et vár el, a Task.Run csak Action-t!

```
new Task(LongOperation).Start();
new Task(LongOperationWithParam, 15).Start();
Task t = Task.Run(() => LongOperation());              //.NET 4.5
Task t = Task.Run(() => LongOperationWithParam(15));  //.NET 4.5
Task t = new TaskFactory().StartNew(LongOperation);
```

Task visszatérési értékkel

- Taskok indítása visszatérési értékkel rendelkező művelet esetén:

```
Task<int> task = new Task<int>(LongOperationWithReturnValue);
task.Start();                    // Func<Tresult> !
// ... más műveletek...
int result = task.Result;
Console.WriteLine(result);
```

- A Result tulajdonság megvárja a művelet végét (blokkol), ezért érdemes nem azonnal a Start után hívni (vagy lásd később)

Thread esetén közös változóba gyűjtenénk a „kimenetet”

```
static void Main(string[] args)
{
    int result = 0;

    Thread t1 = new Thread(() => {
        Thread.Sleep(1500);
        result = 42;
    });
    t1.Start();
    Console.WriteLine("Feldolgozás elindult!");

    t1.Join();
    Console.WriteLine("Feldolgozás vége!");
    Console.WriteLine($"Eredmény: {result}");

    Console.ReadLine();
}
```

```
Feldolgozás elindult!
Feldolgozás vége!
Eredmény: 42
```

Task esetén adott a Result tulajdonságon keresztül

```
static void Main(string[] args)
{
    Task<int> t1 = new Task<int>(() => {
        Thread.Sleep(1500);
        return 42;
    });
    t1.Start();
    Console.WriteLine("Feldolgozás elindult!");

    t1.Wait();
    Console.WriteLine("Feldolgozás vége!");
    Console.WriteLine($"Eredmény: {t1.Result}"

    Console.ReadLine();
}
```

```
Feldolgozás elindult!
Feldolgozás vége!
Eredmény: 42
```

Task esetén a Result lekérdezése blokkol

```
static void Main(string[] args)
{
    Task<int> t1 = new Task<int>(() => {
        Thread.Sleep(1500);
        return 42;
    });
    t1.Start();
    Console.WriteLine("Feldolgozás elindult!");

    Console.WriteLine($"Feldolgozás vége!\nEredmény: {t1.Result}");

    Console.ReadLine();
}
```

```
Feldolgozás elindult!
Feldolgozás vége!
Eredmény: 42
```

Várakozás Taskra

- **Bármilyen esetben, amikor a Task műveletére várni kell (pl. eredményt képez, tömböt feltölt, beállítást végez, fájlt ment…)**

```
task.Wait();                    //várakozás, míg kész (blokkol)

Task.WaitAll(taskArray); //várakozás, míg mind kész (blokkol)
Task.WaitAny(taskArray); //várakozás, míg legalább az egyik kész (blokkol)
//vagy
Task.WaitAll(task1, task2, task3); //mint fent
Task.WaitAny(task1, task2, task3); //mint fent
```

  - Hátrány: ezek a hívások mind blokkolnak

Több Task együttes bevárása

```
List<Task> ts = new List<Task>()
{
    new Task(() => { Thread.Sleep(800); Console.WriteLine("Kész
vagyok!"); }),
    new Task(() => { Thread.Sleep(1200); Console.WriteLine("Kész!");
}),
    new Task(() => { Thread.Sleep(1000);
Console.WriteLine("Elkészült!"); }),
    new Task(() => { Thread.Sleep(1100); Console.WriteLine("Ready!");
}),
};

foreach (Task t in ts)
    t.Start();

Task.WaitAll(ts.ToArray());

Console.WriteLine("Minden feladat végrehaj
```

```
Kész vagyok!
Elkészült!
Ready!
Kész!
Minden feladat végrehajtva!
```

Continuation-ök

- **Az eredeti Task lefutása után egy új Task indul majd a megadott művelettel (a t az előző, befejeződött Taskra való referencia)**

```
Task<int> continuedTask = new Task<int>(LongOperationWithReturnValue);
//nem blokkol, mint a Wait:
continuedTask.ContinueWith(t => Console.WriteLine("The result was: " +
    t.Result));
```

  - Feltételes indítás:

```
//csak ha hiba nélkül futott az eredeti Task:
continuedTask.ContinueWith(t => Console.WriteLine("The result was: " +
    t.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);

//csak ha cancelezték az eredeti Taskot:
continuedTask.ContinueWith(t => Console.WriteLine("The task was canceled!"),
    TaskContinuationOptions.OnlyOnCanceled);

//csak ha hibára futott az eredeti Task:
continuedTask.ContinueWith(t => Console.WriteLine("The task threw " +
    "an exception: " + t.Exception.InnerException),
    TaskContinuationOptions.OnlyOnFaulted);
```

Task folytatása

```
Task t = new Task(() => { Thread.Sleep(1000);
Console.WriteLine("Kész!"); });
t.ContinueWith(x => { Console.WriteLine($"Task #{x.Id} véget ért!");
});
t.Start();

Console.WriteLine("Task elindítva!");
```

```
Task elindítva!
Kész!
Task #2 véget ért!
```

Több Task együttes folytatása

```
List<Task> ts = new List<Task>() {
    new Task(() => { Thread.Sleep(800); Console.WriteLine("Kész
vagyok!"); }),
    new Task(() => { Thread.Sleep(1200); Console.WriteLine("Kész!");
}),
    new Task(() => { Thread.Sleep(1000);
Console.WriteLine("Elkészült!"); }),
    new Task(() => { Thread.Sleep(1100); Console.WriteLine("Ready!");
}),
};

Task.WhenAll(ts.ToArray()).ContinueWith(x
    Console.WriteLine("Minden feladat végr
});

foreach (Task t in ts)
    t.Start();

Console.ReadLine();
```

```
Kész vagyok!
Elkészült!
Ready!
Kész!
Minden feladat végrehajtva!
```

Eredménnyel rendelkező Taskok együttes folytatása

```
List<Task<int>> ts = new List<Task<int>>() {
    new Task<int>(() => { Thread.Sleep(800); return 800; }),
    new Task<int>(() => { Thread.Sleep(600); return 600; }),
    new Task<int>(() => { Thread.Sleep(1200); return 1200; }),
    new Task<int>(() => { Thread.Sleep(1100); return 1100; }),
};

Task.WhenAll(ts.ToArray()).ContinueWith(x => {
    //x.Result: int[]
    Console.WriteLine("Vége! Az eredmények összege: " +
x.Result.Sum());
});

foreach (var t in ts)
    t.Start();

Console.ReadLine();
```

```
Vége! Az eredmények összege: 3700
```

Hibá

```
Task elindult
Baj történt, az alábbi üzenettel: One or more errors occurred.
```

- Ha e
  Wai
  Exce

```
Task t = new Task(() => {
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});

t.Start();
Console.WriteLine("Task elindult");
try { t.Wait(); }
catch (Exception e)
{
    Console.WriteLine("Baj történt, az alábbi üzenettel: " +
e.Message);
}
```

Hibakezelés a Taskban

- Ha egy Taskban hiba történik, az Exception lenyelődik, és a Wait() vagy Result hívásakor dobódik el egy gyűjteményes Exception (AggregateException) formájában

```
Task t = new Task(() => {
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});
t.Start();
Console.WriteLine("Task elindult");
try { t.Wait(); }
catch (AggregateException e)
{
    foreach (var ie in e.InnerExceptions)
        Console.WriteLine("Baj történt, az alábbi üzenettel: " +
ie.Message);
}
```

Hiba

```
Task elindult
Baj történt, az alábbi üzenettel: houston, baj van!
```

- Ha e
  Wai
  Exce

```
Task t = new Task(() => {
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});
t.Start();
Console.WriteLine("Task elindult");
try { t.Wait(); }
catch (AggregateException e)
{
    Console.WriteLine("Baj történt, az alábbi üzenettel: " +
string.Join("\n", e.InnerExceptions.Select(x => x.Message)));
}
```

AggregateException

```
Task t = new Task(() => {
    new Task(() => { throw new Exception("asd"); },
TaskCreationOptions.AttachedToParent).Start();
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});

t.Start();
Console.WriteLine("Task elindult");
```

AggregateException

```
try { t.Wait(); }
catch (AggregateException e) {
    e.Handle(ie =>
    {
        if (ie.Message.Contains("houston"))
        {
            Console.WriteLine("ezt ismerjük, kezeljük");
            return true;
        }
        Console.WriteLine("Baj történt, az alábbi üzenettel: " +
ie.Message);
    });
}
```

```
Task elindult
ezt ismerjük, kezeljük
Baj történt, az alábbi üzenettel: One or more errors occurred.

Unhandled Exception: System.AggregateException: One or more errors
occurred. ---> System.AggregateException: One or more errors occurred.
---> System.Exception: asd
   at xx_Eloadashoz.Program.<>c.<Main>b__0_1() in ...
Press any key to continue . . .
```

Hibakezelés Continuationnel

```
Task t = new Task(() => {
    new Task(() => { throw new Exception("asd"); },
TaskCreationOptions.AttachedToParent).Start();
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});

t.ContinueWith(x => { Console.WriteLine("Hiba volt: " +
x.Exception.Message); }, TaskContinuationOptions.OnlyOnFaulted);

t.Start();
Console.WriteLine("Task elindult");
```

```
Task elindult
Hiba volt: One or more errors occurred.
```

Task leállítása

- **A Task-ban végzett műveletben kell kezelni a leállítás lehetőségét**
- **Szinkronizált CancellationTokenen keresztül történik a Task értesítése, így megoldva a közös változó használatának problémáját**
- **Használható ellenőrzési módok:**
  - **cancellationToken.IsCancellationRequested**: bool tulajdonság. Ha igaz, ki kell lépni a függvényből
  - **cancellationToken.ThrowIfCancellationRequested():** Exception keletkezik, ha leállítás volt kérelmezve (ezzel kilép a függvényből)
  - Előny: az Exception egyértelműen mutatja, hogy a leállás nem a művelet vége miatt történt

Task leállítása

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        if (cts.Token.IsCancellationRequested)
            return;
    }
}, cts.Token);

t.Start(
Console.
Thread.S
Console.
cts.Canc
```

```
Task elindult!
0       1       2       3       4       5       6       7       8
9       10      11      12      13      14
15      16      17      18      19      20      21      22      23
24      25      26      27      28      29
30      31      32      33      34      35      36      37      38
39      40      41      42      43      44
45      46      47      48      49      50      51      52      53
Leállítás kezdeményezése...
54
```

Task leállítása - kivétel

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        cts.Token.ThrowIfCancellationRequested();
    }
}, cts.Token);

t.Start();
Console.WriteLine("Task elindult!");
Thread.Sleep(1200);
Console.WriteLine("Leállítás kezdeményezése...");
cts.Cancel();
```

Task leállítása

```
try { t.Wait(); }
catch (AggregateException e) {
    e.Handle(ie => {
        if (ie is OperationCanceledException)
            Console.WriteLine("Leállítva!");
        return ie is OperationCanceledException;
    });
}
```

```
Task elindult!
0       1       2       3       4       5       6       7       8       9
10      11      12      13      14
15      16      17      18      19      20      21      22      23      24
25      26      27      28      29
30      31      32      33      34      35      36      37      38      39
40      41      42      43      44
45      46      47      48      49      50      51      52      53 Leállítás
kezdeményezése...
54      Leállítva!
```

Task leállítása - continuation

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        cts.Token.ThrowIfCancellationRequested();
    }
}, cts.Token);

t.ContinueWith(x => { Console.WriteLine("Leállítva!"); },
TaskContinuationOptions.OnlyOnCanceled);

t.Start();
Console.WriteLine("Task elindult!");
Thread.Sleep(600);
Console.WriteLine("Leállítás kezdeményezése...");
cts.Cancel();
```

GUI-elem kezelése

- **Windows-os grafikusfelület-elemekhez általában nem lehet hozzányúlni, csak a létrehozó szálról (GUI szál)**
  - Még közvetve sem
  - WPF, Windows Forms is!
  - Van kevés kivétel (bizonyos függvények, adatkötésnél a PropertyChanged)
- **Általános megoldás: Invoke**
  - Függvény végrehajtatása a GUI szállal

```
Dispatcher.Invoke(() =>
{
    label.Content = ...;
    listBox.Items.Add(...);
});
```

GUI-elem kezelése

- **Rövidebb módszer, ha a Task eredményét kiíró műveletet is külön Taskként indítjuk, megadott taskütemező (TaskScheduler) segítségével**
- **Beépített taskütemezők:**
  - Thread Pool Task Scheduler: a ThreadPool-on indítja a taskokat (alapértelmezett)
  - Synchronization Context Task Scheduler: a felhasználói felület szálján indítja a taskokat – ezzel kell indítani, ha GUI-elemet akarunk kezelni

    Referencia „megszerzése”: a GUI szálján

    `TaskScheduler.FromCurrentSynchronizationContext()`

```
Task<int> taskWithReturnValue =
    Task.Run(() => { Thread.Sleep(1000); return 6; });

taskWithReturnValue.ContinueWith(
    t => textBox1.Text = "Result: " + t.Result,
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnRanToCompletion,
    TaskScheduler.FromCurrentSynchronizationContext());
```

Források

- Szabó-Resch Miklós Zsolt és Cseri Orsolya Eszter Haladó Programozás előadásfóliái
- Kertész Gábor Párhuzamos és Elosztott Rendszerek Programozása előadásfóliái
- MSDN