Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások

---

6. előadás

Többszálú programozás C#-ban, Windows Forms alkalmazások párhuzamosítása

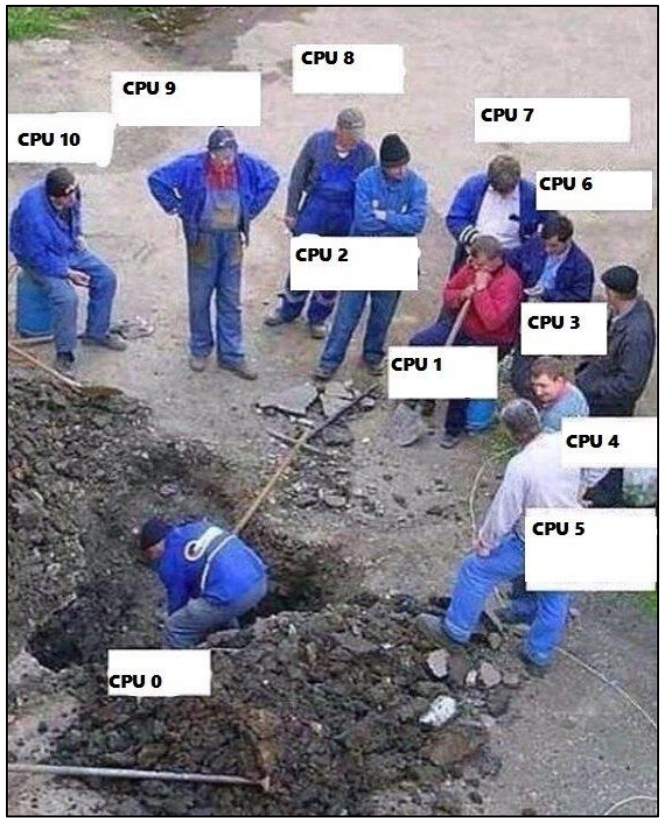
---

Cserép Máté
mcserep@inf.elte.hu
https://mcserep.web.elte.hu

Többszálú programozás C#-ban

Párhuzamos programozás

- A számítógépek több feladatot is elvégezhetnek párhuzamosan.
- A párhuzamos számítás, feldolgozás gyakran még egyszerű alkalmazások esetén is gyakran követelmény.
  - Például egy egyszerű szövegszerkesztőnek kezelnie kell a felhasználói bevitelt, függetlenül attól, hogy elfoglalt a felhasználói felület frissítésével, a szemantikai elemzéssel, stb.
- A C# nyelv és a .NET keretrendszer több eszközzel is támogatja a párhuzamos programozást.



Többszálú programozás C#-ban

Párhuzamos programozás

- Párhuzamos programozás szempontjából megkülönböztetjük a *folyamat*okat (*process*) és a *szál*akat (*thread*).
  - A folyamatok teljes végrehajtási környezettel és saját futásidejű erőforrásokkal rendelkeznek (például memória).
    - Egy C# program alapértelmezetten egyetlen folyamat.
  - Egy folyamat több szálat is tartalmazhat.
    - Ezek közös virtuális címtérrel és a rendszer erőforrásokkal rendelkeznek.
    - A szálak könnyebb súlyúak a folyamatokhoz képest.
    - Minden folyamat egy kezdeti szállal rendelkezik, amelyet gyakran elsődleges vagy fő szálnak neveznek.
- A továbbiakban a többszálú programozással foglalkozunk.

Többszálú programozás C#-ban

Szálak

- A C# programunkban új szálat többféleképpen is indíthatunk, legegyszerűbben a **`Thread`** típus példányosításával, majd a **`Start`** metódus meghívásával.
  - A szál példányosításakor paraméterként adjuk át az új szálon végrehajtandó metódust. Pl:

```
ThreadStart job = new ThreadStart(SomeMethod);
Thread thread = new Thread(job);
thread.Start();
```

  - A gyerek szálon futó feladat végrehajtása a szülő szálon bevárható a **`Thread`** objektum **`Join()`** metódusával.
  - Dönthetünk a gyerek szál terminálása mellett is (**`Abort()`**).

Többszálú programozás C#-ban

Szálak

```
class Program {
  public static void DoWork() {
     Console.WriteLine("Child thread starts");
     Console.WriteLine("Child thread goes to sleep");
     Thread.Sleep(5000); // the thread is paused for 5000 ms
     Console.WriteLine("Child thread resumes and finishes");
  }

  static void Main(string[] args) {
     ThreadStart childJob = new ThreadStart(DoWork);
     Console.WriteLine("Main thread starts");

     Thread childThread = new Thread(childJob);
     childThread.Start(); // child thread starts

     Console.WriteLine("Main thread waiting");
     childThread.Join(); // waiting for child thread to finish
     Console.WriteLine("Main thread finishes");
  }
}
```

Kód példa futtatása

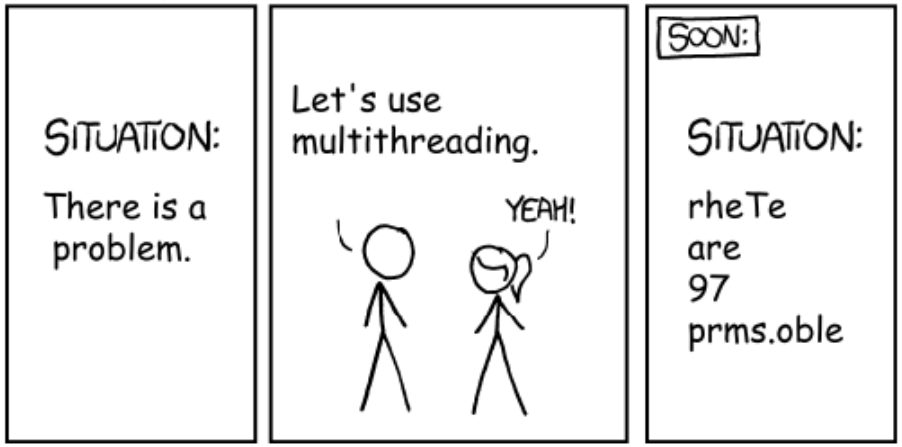
Többszálú programozás C#-ban

Szálak

- Milyen sorrendben fognak megjelenni a konzol outputon a két szál által kiírt üzenetek?

Main thread starts
Child thread starts
Child thread goes to sleep
Main thread waiting
Child thread resumes and finishes
Main thread finishes

Main thread starts
Main thread waiting
Child thread starts
Child thread goes to sleep
Child thread resumes and finishes
Main thread finishes



Többszálú programozás C#-ban

Paraméterátadás az új szálnak

- Az új szálnak paramétert átadni a **`ParameterizedThreadStart`** használatával lehet. A paraméter statikus típusa **`object`** lesz, konkrét értékét a **`Start()`** metódus hívásakor adjuk meg.
  - A paraméter lehet tömb vagy egyéb gyűjtemény, így több érték is átadható.

```
public static void Main(string[] args) {
   ParameterizedThreadStart childJob =
       new ParameterizedThreadStart(DoWork);
   Console.WriteLine("Main thread starts");

   Thread childThread = new Thread(childJob);
   childThread.Start("Message from Main");

   Console.WriteLine("Main thread waiting");
   childThread.Join();
   Console.WriteLine("Main thread finishes");
}
```

Többszálú programozás C#-ban

Paraméterátadás az új szálnak

Kód példa futtatása

```
public static void DoWork(object obj) {
   Console.WriteLine("Child thread starts");

   if (obj is String)
     Console.WriteLine(obj as String);
   else
     throw new ArgumentException(
        "Parameter is not a string.", nameof(obj));

   Console.WriteLine("Child thread goes to sleep");
   Thread.Sleep(5000); // the thread is paused for 5000 ms
   Console.WriteLine("Child thread resumes and finishes");
}
```

- A gyerek szálban dobott kivételt ott kezelni is kell, a szülő szálban erre már nincsen lehetőség. Kezeletlen kivétel esetén a program terminál.

Többszálú programozás C#-ban

Szinkronizációs objektumok

- A gyerek szál nem tud egy végeredményt visszaadni a szülő szálnak.
- A szálak közötti adatkommunikáció megosztott erőforrások (jellemzően memória, közös változók) segítségével történhet.
  - Erre nem csak a gyerek szál tevékenységének végén, hanem közben, és mindkét irányban lehetőség van.
- A közös erőforrások kezelését a program *kritikus szakaszának* (*critical section*) nevezzük. Azonos erőforrásra vonatkozó kritikus szakaszok párhuzamos végrehajtása (kivéve, ha minden művelet csak olvasni próbálja) hibát, nem várt futásidejű viselkedést okozhat.
  - A szálakat a közös erőforrások használatakor szinkronizálni kell, *kölcsönös kizárás* (*mutual exclusion*) segítségével garantálva, hogy egyszerre csak egy kritikus szakasz kerül végrehajtásra.

Többszálú programozás C#-ban

Mutex

```
public Stack<T> {
    private Mutex mutex;
    private IList<T> values;

    public Stack() {
        mutex = new Mutex();
        values = new List<T>();
    }

    public void Push(T item) {
        mutex.WaitOne();
        values.Add(item);  // critical section
        mutex.ReleaseMutex();
    }
}
```

- Várakozhatunk megadott ideig vagy időpontig is: **`mutex.WaitOne(Int32)`** és **`mutex.WaitOne(TimeSpan)`**

Többszálú programozás C#-ban

Szemafor

```
public Stack<T> {
    private Semaphore sem;
    private IList<T> values;

    public Stack() {
        sem = new Semaphore();
        values = new List<T>();
    }

    public void Push(T item) {
        sem.WaitOne();
        values.Add(item);  // critical section
        sem.Release();
    }
}
```

- Megadható a kezdeti és a maximum zárolások száma:

```
Semaphore sem = new Semaphore(0, 3);
```

Többszálú programozás C#-ban

Monitorok

```
public Stack<T> {
    private IList<T> values;

    public Stack(){
        values = new List<T>();
    }

    public void Push(T item) {
        Monitor.Enter(values);
        values.Add(item);  // critical section
        Monitor.Exit(values);
    }
}
```

- Megegyező a **`lock`** utasítás használatával:

```
public void Push(T item) {
    lock(values)  {
        values.Add(item);  // critical section
    }
}
```

Többszálú programozás C#-ban

Szinkronizációs objektumok összehasonlítása

Mutex	Semaphore	Monitor
• elnevezhető • rendszer szintű hatókör • jó választás folyamatok (alkalmazások) közötti szinkronizációhoz	• elnevezhető • könnyebb súlyú • többszörös zárolás lehetősége • legfeljebb alkalmazás szintű hatókör • jó választás szálak közötti szinkronizációhoz	• név nélküli • zárolt objektummal egyező hatókör (legfeljebb alkalmazás szintű) • `lock` utasítással kényelmesen használható

Többszálú programozás C#-ban

Szálbiztos gyűjtemények

- A szálbiztos, metódusaikban kölcsönös kizárást megvalósító gyűjtemények a .NET Standard Library részét képezik (**`System.Collections.Concurrent`** névtér)
  - **`ConcurrentBag`**, **`ConcurrentDictionary`**, **`ConcurrentQueue`**, **`ConcurrentStack`**, **`BlockingCollection`** (gyártó-fogyasztó minta)
  - Műveleteik szignatúrája néhol eltér, de a szokásos interfészeket megvalósítják, pl.:

```
IDictionary<String, Object> dictionary =
   new ConcurrentDictionary<String, Object>();
```

Többszálú programozás C#-ban

Atomi típusok

- Atomi adattípusok (olvasás és írás): **`bool`**, **`char`**, **`byte`**, **`sbyte`**, **`short`**, **`ushort`**, **`uint`**, **`int`**, **`float`** és referencia szerinti típusok.
- Nem atomi adattípusok: **`long`**, **`ulong`**, **`double`**, **`decimal`**, stb. Nincs garancia az atomi olvasásra, írásra, módosításra.
- Komplexebb műveletek (pl. hozzáadás, ami egy olvasás és egy írás) már az atomi adattípusokra sem atomi.
  - Ezen elemi műveletek atomi módon az **`Interlocked`** osztály használatával érhetőek el:

```
int x = 41;
Interlocked.Increment(ref x);   // increment x

SomeType y = new SomeType();
SomeType z = new SomeType();
// ...
Interlocked.Exchange(ref y, z); // replace y with z
```

Többszálú programozás C#-ban

Alacsony absztrakciós szintű szálkezelés problémái

- Problémák a **`Thread`** típussal:
  - nincs lehetőség erősen típusos paraméterátadásra (megosztott memória használható)
  - nincs lehetőség az eredmény visszaadására (megosztott memória használható)
  - nincs lehetőség a kivételek továbbítására a gyerek szálból a fő szál felé

Többszálú programozás C#-ban

Taszk-alapú aszinkron programozás

- A .NET Framework 4.0-s verziója (2010) óta elérhető a párhuzamos és késeltetett végrehajtás magasabb absztrakciós szintű koncepciója, a *taszk*ok (**`Task`**)
  - a taszk egy *lambda*-kifejezésként (**`Func`**, **`Action`**) megadott eljárást hajt végre aszinkron módon, jellemzően egy külön szálon
    - az új szál a .NET-ben elérhető szálkészletből (*thread pool*) kerül kivételre, amely a szálak újrafelhasználását biztosítja
  - a művelet a példány **`Start()`** metódusával hajtható végre, de szinkron művelet is futtatható aszinkron módon a **`Task.Run(…)`** művelete segítségével, amely egy **`Task`**-ot ad vissza
  - a művelet eredménye a **`Task`** objektum **`Result`** tulajdonságával kérhető le (amely megvárja a taszk befejezését)
    - a taszk a **`Wait()`** metódussal és változataival is megvárható

Többszálú programozás C#-ban

Taszk-alapú aszinkron programozás

- Pl.:

```
private Int32 Compute(){ /* ... */ }
  // ez eredményt előállító számítás

private void RunCompute() {
  Task<Int32> myTask =
     new Task<Int32>(() => Compute());
     // taszk létrehozása a végrehajtandó művelettel
  myTask.Start(); // taszk elindítása
  // további műveletek ...

  Int32 result = myTask.Result;
  // eredmény megvárása

  // további műveletek ...
}
```

Többszálú programozás C#-ban

Taszk-alapú aszinkron programozás

- Alternatív módon:

```
private Int32 Compute(){ /* ... */ }
  // ez eredményt előállító számítás

private void RunCompute() {
  Int32 result = Task.Run(() => Compute()).Result;
      // taszk végrehajtása és az eredmény megvárása

  // további műveletek ...
}
```

Többszálú programozás C#-ban

Taszk-alapú aszinkron programozás

```
class Program {
  public static int Add(int a, int b) {
     Console.WriteLine("Child thread starts");
     int result = a + b;
     Console.WriteLine("Child thread goes to sleep");
     Thread.Sleep(5000); // the thread is paused for 5000 ms
     Console.WriteLine("Child thread resumes and finishes");
     return result;
  }

  public static void Main(string[] args) {
     int x = 30, y = 12;
     Task<int> task = new Task<int>(() => Add(x, y));
     Console.WriteLine("Main thread starts");
     task.Start();

     Console.WriteLine("Main thread waiting");
     int sum = task.Result; // várakozás az eredményre
     Console.WriteLine("Main thread finishes, sum = {0}", sum);
  }
}
```

Kód példa futtatása

Többszálú programozás C#-ban

Aszinkron műveletek

- Az aszinkron műveletek eredménye bevárható egy másik aszinkron műveletben
  - aszinkron műveletet az **`async`** kulcsszóval hozhatunk létre
  - aszinkron műveletet bevárni az **`await`** utasítással tudunk
  - pl.:

```
private async void ReadStreamAsync(Stream str)
{
    StreamReader reader = new StreamReader(str);
    String line = await reader.ReadLineAsync();
       // aszinkron módon olvasunk, és megvárjuk
       // a művelet lefutását
    …
}
```

Többszálú programozás C#-ban

Aszinkron műveletek

- Az aszinkron műveletek az első **`await`** utasításig szinkron módon hajtódnak végre
  - Az **`await`** utasítás elérésekor az adott **`Task`** egy másik szálon kerül kiértékelésre.
  - Közben a tartalmazó metódus felfüggesztésre kerül, és a vezérlés a hívó eljáráshoz kerül vissza.
  - A **`Task`** befejezésekor **`await`** utasítást tartalmazó metódus további része a második szálon kerül végrehajtásra.
  - Kód példa: https://ideone.com/TSpZMd

Többszálú programozás C#-ban

Aszinkron műveletek

- A háttérben futtatandó tevékenységek jelentős része (pl. fájlkezelés, hálózatkezelés) aszinkron műveletként is elérhetőek (.NET Framework 4.5 és C# 5.0 óta):
  - ez a műveletek nevében konvencionálisan az **`Async`** szuffixszel jelzett, amelyet saját metódusainknál is érdemes követni
    pl.:
    ```
    StreamReader reader = ...;
    reader.ReadLineAsync(); // aszinkron olvasás
    ```
  - az aszinkronitást csak a megvalósításban kell jelölnünk, interfészben nem, csupán a taszk visszatérési értéket kell megadnunk

Többszálú programozás C#-ban

Aszinkron tevékenységek megvalósítása

- pl.:

```
interface IAsyncInterface {
    Task ProcessAsync();
    Task<Int32> ComputeAsync();
    // aszinkron műveletek
       (visszatérési értékből látszik)
}
…
async Task SomeMethod(IAsyncInterface asInst) {
    Int32 result =
       await asInst.ComputeAsync();
       // eredmény bevárása
}
…
```

Többszálú programozás C#-ban

Aszinkron tevékenységek megvalósítása

```
class AsyncImplementation : IAsyncInterface
{
   private void Process(); // szinkron művelet

   public async Task ProcessAsync()
   {
      await Task.Run(() => Process());
      // a tevékenység aszinkron végrehajtása
   }
   public async Task<Int32> ComputeAsync()
   {
      await Task.Run(() => { … return value; });
   }
}
```

Többszálú programozás C#-ban

Aszinkron tevékenységek megvalósítása

```
class Program {
  public static int Add(int a, int b) {
     /* ... */
  }

  public static async Task<int> AddAsync(int a, int b) {
    return await Task.Run(() => Add(a, b));
  }

  public static void Main(string[] args) {
     int x = 30;
     int y = 12;

     Console.WriteLine("Main thread starts");
     Task<int> task = AddAsync(x, y);

     Console.WriteLine("Main thread waiting");
     int sum = task.Result;
     Console.WriteLine("Main thread finishes, sum = {0}", sum);
  }
}
```

Kód példa futtatása

Többszálú programozás C#-ban

Kivételkezelés taszkokkal

- A taszkokon belül keletkező kezeletlen kivételek visszapropagálásra kerülnek a hívó szál felé.

```
Console.WriteLine("Main thread starts");
Task<int> task = DoWorkAsync(42);

Console.WriteLine("Main thread waiting");
try {
  int result = task.Result;
  // eredmény megvárása
}
catch (Exception ex) {
  // kivételek kezelése ...
}
Console.WriteLine("Main thread finishes");
```

Többszálú programozás C#-ban

Kivételkezelés taszkokkal

- Bizonyos esetekben több kivétel is keletkezhet (például amikor több taszkra várakozunk), ezért a kivételeket minden esetben egy **`AggregateException`** példányként kaphatóak el.

```
Console.WriteLine("Main thread starts");
Task<int> taskA = DoWorkAsync(42);
Task<int> taskB = DoWorkAsync(100);

Console.WriteLine("Main thread waiting");
try {
  Task.WaitAll(new Task[] { taskA, taskB });
  // taskA.Result és taskB.Result elérhető ezen a ponton
}
catch (AggregateException ae) {
  foreach (var e in ae.InnerExceptions) {
    // kivételek kezelése ...
  }
}
Console.WriteLine("Main thread finishes");
```

Windows Forms alkalmazások párhuzamosítása

Szinkron és aszinkron tevékenységek

- A tevékenységek végrehajtásának két megközelítése van:
  - *szinkron*: a tevékenység kezdeményezője megvárja annak lefutását
    - a hívó szál blokkolódik, amíg a tevékenység lefut
    - ha sokáig tart a tevékenység, akkor az a program felületén is észrevehető
  - *aszinkron*: a tevékenység kezdeményezője nem várja meg a lefutást, illetve az eredményt
    - a tevékenység (metódus) külön szálon fut
    - az eredményt később megkapjuk (pl. eseményen át)
    - a hívó szál nem blokkolódik, folytathatja a végrehajtást

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számot egy modell állítja elő (**`FibonacciGenerator`**), a generáláshoz (**`Generate`**) a klasszikus rekurzív képletet* használjuk:

$$F(n) = \begin{cases} 1 & ha\ n < 3 \\ F(n-1) + F(n-2) & ha\ n \geq 3 \end{cases}$$

- a grafikus felületen egy listában jelenítjük meg a számokat, és egy számbeállító segítségével szabályozzuk, hányadik számra vagyunk kíváncsiak

*Valós környezetben a Fibonnaci számok a Binet formula segítségével konstans algoritmikus komplexitással előállíthatóak.

Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (`FibonacciGenerator.cs`*):*

```
public Int64 Generate(Int32 number) {
    if (number < 1)
        throw new ArgumentOutOfRangeException(…);
    if (number > 100)
        throw new ArgumentOutOfRangeException(…);

    if (number < 3)
        return 1;

     return Generate(number - 1)
            + Generate(number - 2);
}
```

Windows Forms alkalmazások párhuzamosítása

Aszinkron műveletek grafikus alkalmazásokban

- A grafikus felületű alkalmazások felépítésében fontos, hogy
  - gyorsan reagáljunk a felhasználói interakcióra, a felhasználói felület mindig aktív legyen
  - amennyiben egy nagyobb műveletet hajtunk végre, azt aszinkron módon, háttérben végezzük
- Az aszinkron műveletek alapja a *taszk* (**`Task`**), az **`async`** eljárások lényegében tulajdonképpen taszkkal térnek vissza, amely tartalmazhat eredményt is (**`Task<T>`**)
  - amennyiben meg szeretnénk várni a művelet eredményét, taszkot kell megadni visszatérési értékként

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számot egy modell állítja elő (**`FibonacciGenerator`**), a generáláshoz (**`Generate`**) a klasszikus rekurzív képletet használjuk:

$$F(n) = \begin{cases} 1 & ha\ n < 3 \\ F(n-1) + F(n-2) & ha\ n \geq 3 \end{cases}$$

- lehetőséget adunk az aszinkron használatra is (**`GenerateAsync`**), lényegében egy taszkba burkoljuk a szinkron tevékenységet
- a felület így mindig aktív lesz, figyelmeztethetjük a felhasználót a tevékenységre

Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (`MainForm.cs`*):*

```
private async void ButtonGenerate_Click(…) {
    // aszinkron lesz az eseménykezelő

    _button.Text = "Generating... Please wait.";
    …
    _listBox.Items.Insert(0,
       await _generator.GenerateAsync(…));
        // megvárjuk a generálás eredményét
    …
    _button.Text = "Generate";
    …
}
```

Windows Forms alkalmazások párhuzamosítása

Aszinkron tevékenységek megszakítása

- Az aszinkron műveletek végrehajtását adott esetben azok teljes befejezése előtt meg kívánjuk szakítani:
  - a párhuzamos szál terminálása a háttér művelet inkonzisztens állapotban történő megszakításának kockázatával jár
  - A *taszk* alapú aszinkron eljárások támogatják az abortálási igény detektálását és kezelését:
    - ```
      var source = new CancellationTokenSource();
      var token = source.Token;
      var task = new Task(() => { ... }, token);
      ```
    - megszakítási igény jelzése a taszkon kívülről:
      `source.Cancel();`
    - megszakítási igény észlelése a taszkban:
      `if(token.IsCancellationRequested(){ ... }`
      ```

Windows Forms alkalmazások párhuzamosítása

Felületi vezérlők kezelése párhuzamos végrehajtás során

- A tárgyalt módon aszinkron végrehajtásra kerül, amennyiben explicit egy új *taszk*ot indítunk el, pl. egy **`Task`** objektum **`Start()`** metódusával, vagy a **`Task.Run()`**, vagy a **`Task.Factory.StartNew()`** meghívásával.

- Továbbá az aszinkron metódusok (**`async`**) szinkronban futnak, amíg el nem érik az első várakozási kifejezést (**`await`**), utána ez már nem biztosított.

- A Windows Forms asztali grafikus keretrendszerben minden vezérlőt csak az őt létrehozó szál kezelhet
  - Különben *cross-thread operation* miatti kivételt kaphatunk (**`InvalidOperationException`**).

Windows Forms alkalmazások párhuzamosítása

Felületi vezérlők kezelése párhuzamos végrehajtás során

- Ezt megoldandó minden **`Control`** objektum rendelkezik egy **`Invoke`** metódussal, ami a paraméterül kapott lambda kifejezést a felületet birtokló szálon hajtja végre.
- Ha nem fontos a háttér szál blokkolása a frissítés megvárása, használhatjuk a **`BeginInvoke`** eljárást is.
- Használhatjuk a vezérlők **`InvokeRequired`** tulajdonságát annak ellenőrzésére, hogy az adott kontextusban szükséges-e visszatérni a felületi vezérlőt birtokló szálra annak kezeléséhez.

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számokat aszinkron módon egy modell állítja elő (**`FibonacciGenerator`**) a **`Run(n)`** metódussal, amely az első **`n`** Fibonnaci számot számítja ki
- egy új Fibonacci szám előállításakor kiváltjuk a **`NewResult`** eseményt, az utolsó, azaz az **`n.`** szám előllítását követően pedig a **`Ready`** eseményt is
- a számítás, azaz a Fibonacci számok előállítása megszakítható a **`Cancel()`** metóduson keresztül
- a felületi vezérlők háttér szálakról történő frissítéskor használjuk a **`BeginInvoke`** műveletét, amely egy lambda-kifejezéssel megadott akciót (**`Action`**) tud futtatni a felület szálán

Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (`MainForm.cs`*):*

```
private void Ready(object? sender, EventArgs e) {
   // amennyiben nem a UI szálon vagyunk, rekurzív
   // módon meghívjuk a Ready() eljárást, de már a
   // UI szálon.

   if (_btnCalculate.InvokeRequired) {
      BeginInvoke(new EventHandler(Ready),
                  sender, e);
      return;
   }
   _btnCalculate.Text = "Számol";
   …
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

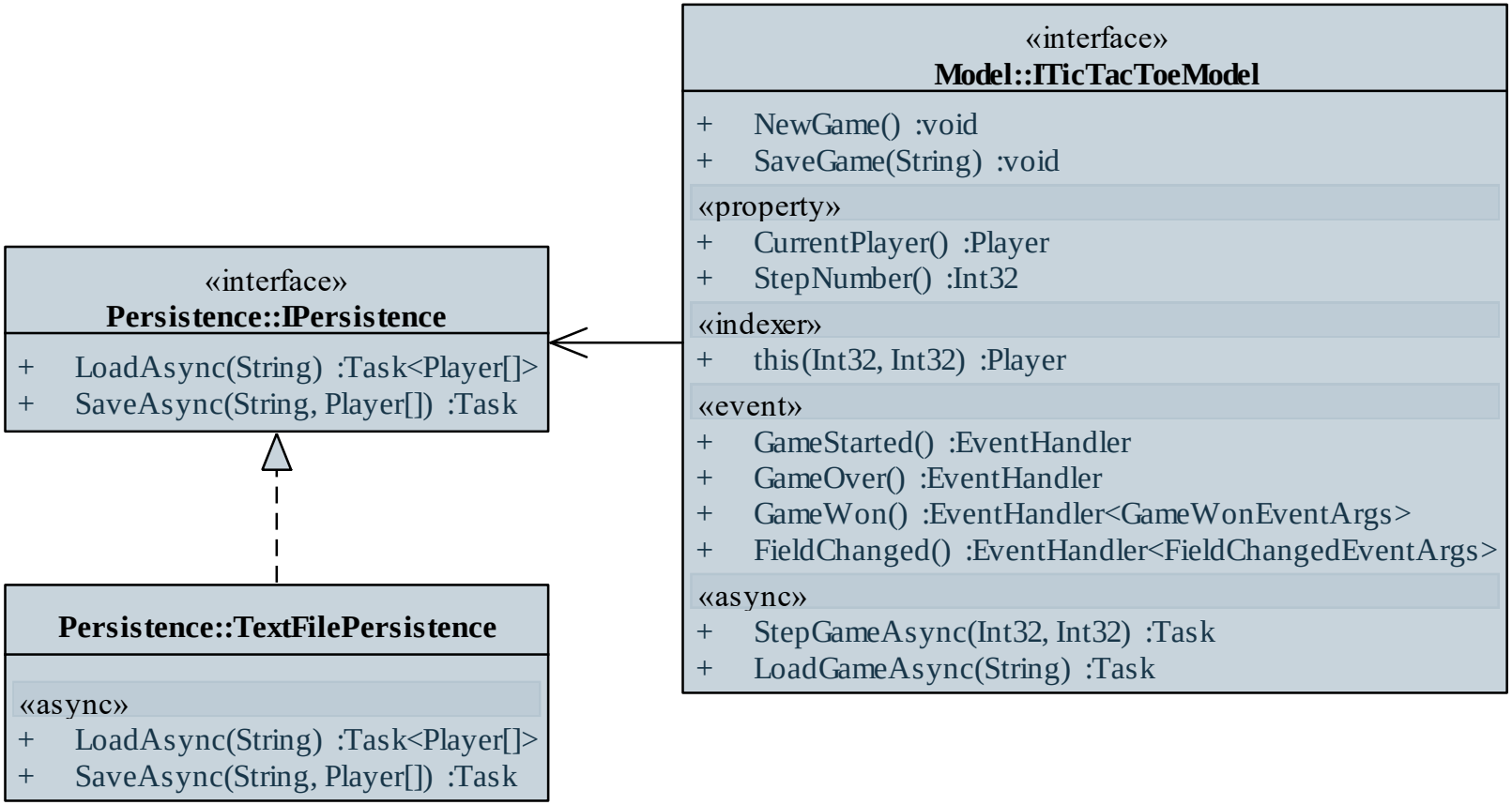
Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- hatékonysági okokból valósítsuk meg aszinkron módon a teljes fájlkezelést, így
  - az **`IPersistence`** interfész **`Load`** és **`Save`** műveletei taszkkal térnek vissza
  - az **`ITicTacToeModel`** interfésze **`LoadGame`** és **`SaveGame`** műveletei is taszkkal térnek vissza
  - minden esetben a megvalósításban aszinkron műveleteket készítünk, és aszinkron műveleteket hívunk
  - ennek megfelelően minden felhasználáskor bevárjuk az eredményt

Windows Forms alkalmazások párhuzamosítása

Példa

Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (`TextFilePersistence.cs`*):*

```
public async Task<Player[]> LoadAsync(String path)
    …
    Byte[] fileData =
     await Task.Run(() => File.ReadAllBytes(path));
       // fájl bináris tartalmának aszinkron
       // beolvasása
    …
    return fileData.Select(fileByte =>
       (Player)fileByte).ToArray();
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

.NET Core 3.0-tól (illetve .NET Standard 2.1-től) aszinkron segédeljárás is elérhető a fájl tartalmának beolvasására:

```
public async Task<Player[]> LoadAsync(String path)
    …
    Byte[] fileData =
     await File.ReadAllBytesAsync(path);
       // fájl bináris tartalmának aszinkron
       // beolvasása
    …
    return fileData.Select(fileByte =>
       (Player)fileByte).ToArray();
}
```

Windows Forms alkalmazások párhuzamosítása

Párhuzamosítás időzítővel

- Az időzítés egy másik lehetséges formája az aszinkron tevékenység végrehajtásnak, amely a grafikus felülettől függetlenül is használható a **`System.Timers.Timer`** időzítővel
  - kezelhető az intervallum (**`Interval`**), indítás és leállítás (**`Start`**, **`Stop`**), valamint az időzített esemény kiváltása (**`Elapsed`**)
  - a **`System.Windows.Forms.Timer`** vezérlővel ellentétben párhuzamosan fut a háttérben, és nagyobb pontosságot garantál
  - hátránya, hogy amennyiben grafikus felületű alkalmazással használjuk, szinkronizálást kell végeznünk a felülettel
  - hasonlóan, mint a *taszk*oknál láttuk, ez feloldható a vezérlő **`BeginInvoke`** műveletével, amely egy lambda-kifejezéssel megadott akciót (**`Action`**) tud futtatni a felület szálán

Windows Forms alkalmazások párhuzamosítása

Párhuzamosítás időzítővel

- Pl.:

```
Timers.Timer myTimer = new Timer(); // időzítő
myTimer.Elapsed +=
    new ElapsedEventHandler(Timer_Elapsed);
    // időzített esemény
…
void Timer_Elapsed(…){
    // itt nem használhatjuk a felületet
    BeginInvoke(new Action(() => {
        // itt már igen
        myLabel.Text = e.SignalTime.ToString();
            // kiírjuk az eltelt időt a felületre
    }));
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

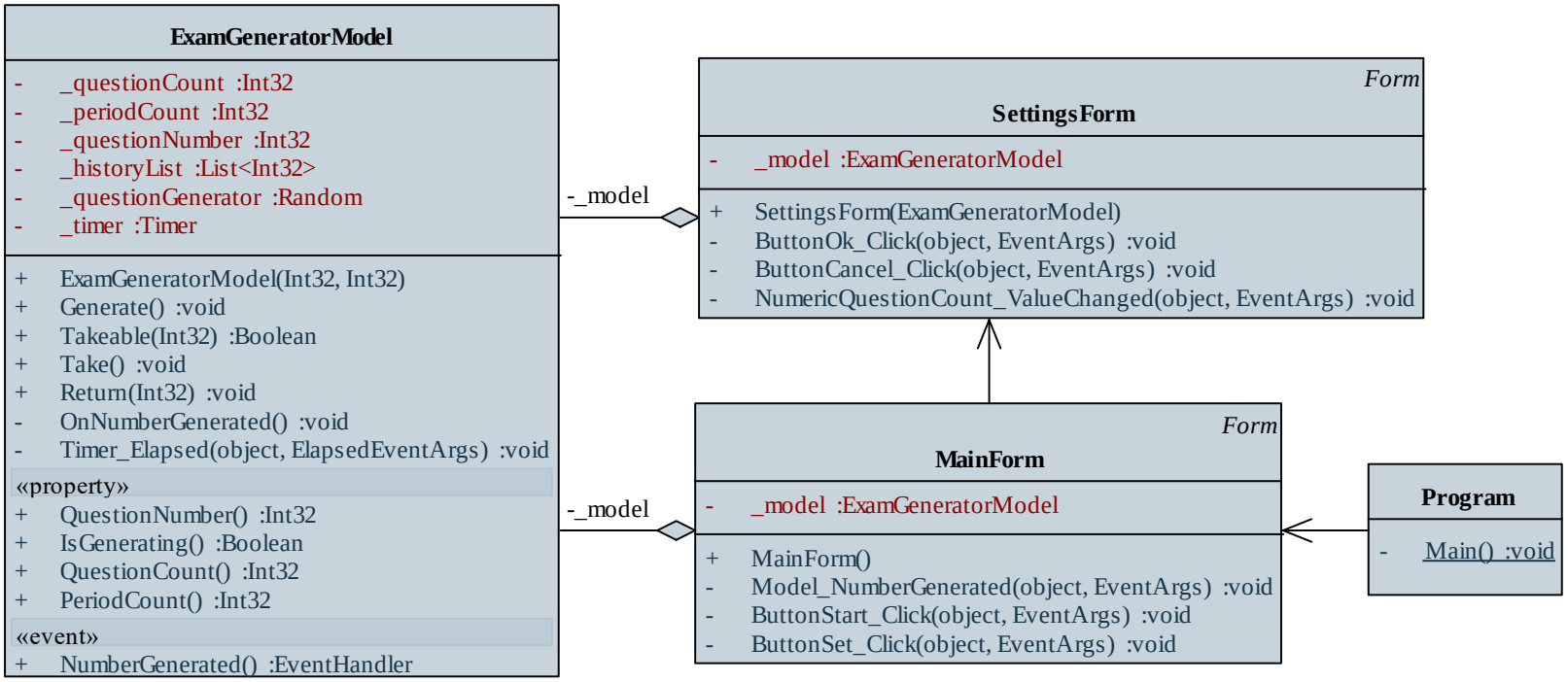
Feladat: Készítsünk egy vizsgatétel generáló alkalmazást kétrétegű architektúrában.

- a modell (**`ExamGeneratorModel`**) végzi a tételek generálását (**`Generate`**), amihez időzítőt használ, továbbá eseménnyel (**`NumberGenerated`**) jelzi, ha generált egy új számot
- emellett lehetőség van a tétel elfogadására (**`Take`**), illetve a korábban húzott tételek visszahelyezésére (**`Return`**)
- a generálás legyen megszakítható, a modell megfelelően valósítsa meg az **`IDisposable`** interfészt
- mindkét nézet kapcsolatban áll a modellel, a főablak az esemény hatására frissíti a megjelenítést (ügyelve a szinkronizációra)

Windows Forms alkalmazások párhuzamosítása

Példa

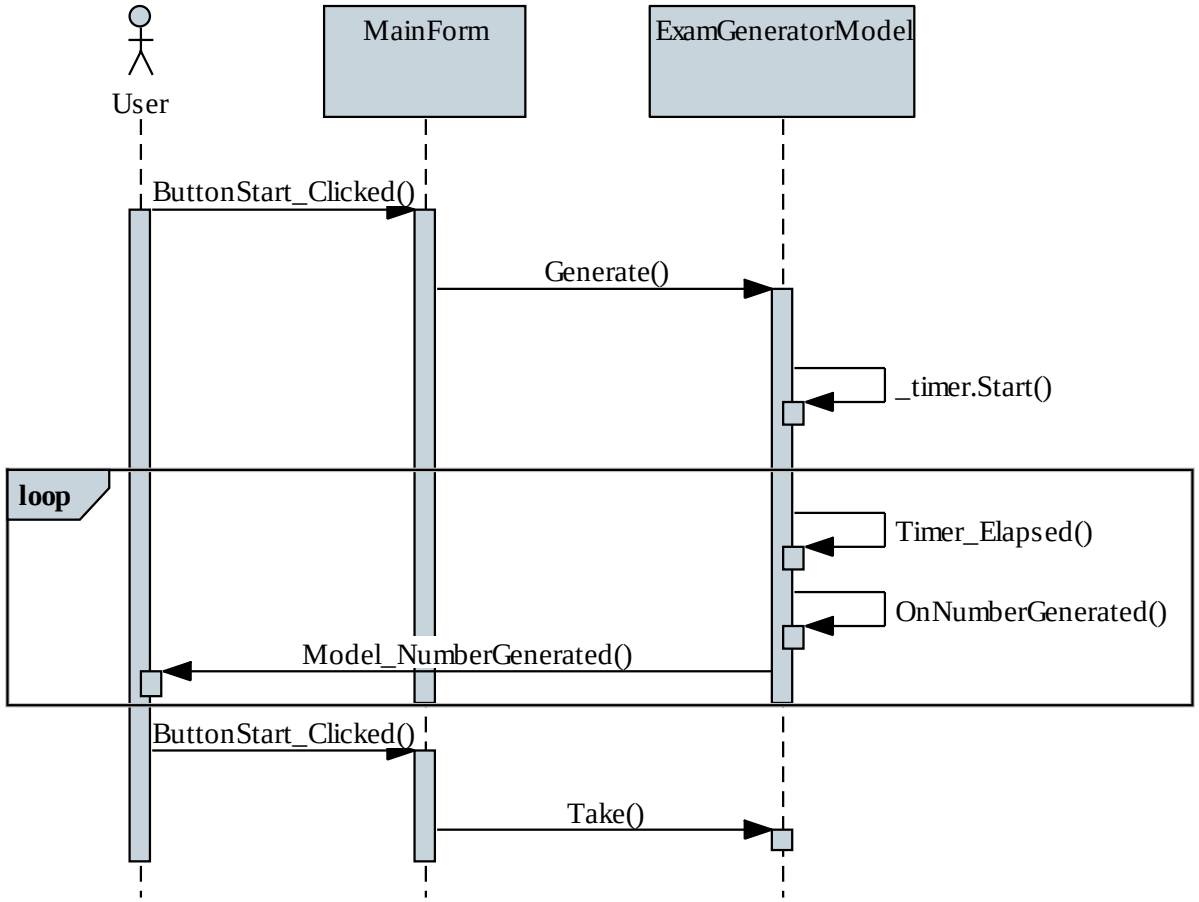
Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (`MainForm.cs`*):*

```
public MainForm(){
    _model = new ExamGeneratorModel(10, 0);
    _model.NumberGenerated +=
        new EventHandler(Model_NumberGenerated);
            // modell eseménye
}
private void Model_NumberGenerated(object sender,
                    EventArgs e){
  BeginInvoke(new Action(() => {
      _textNumber.Text =
          _model.QuestionNumber.ToString();
  })); // szinkronizált végrehajtás
}
```

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- A .NET alkalmazások rendelkezhetnek egy szinkronizációs kontextussal (**`SynchronizationContext.Current`**), amelynek megadásával implicit szinkronizáció követelhető meg a várakozási kifejezések (**`await`**) után a hívó szállal.
  - Konzolos alkalmazások esetén nincsen (alapértelemzett) szinkronizációs kontextus objektum, az **`await`** utáni utasítások nem garantált, hogy a hívó szálon kerülnek végrehajtásra.
  - Windows Forms alkalmazások rendelkeznek egy alapértelmezett szinkronizációs kontextussal.
    - a **`SynchronizationContext.Current`** értéke egy **`WindowsFormsSynchronizationContext`** típusú objektum.

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- A **`WindowsFormsSynchronizationContext`** implementálja, hogy **`await`** utasítás után a UI szálra visszatérünk a **`BeginInvoke`** használatával.
- Ilyen módon Windows Forms alkalmazásokban nem szükséges az **`await`** utasítások után az explicit **`Invoke`** / **`BeginInvoke`** használata, mert azt tapasztaljuk, hogy már egyébként is a UI szálon vagyunk.
- Egy másik szálról kiváltott esemény kezelésére ez nem vonatkozik (pl. a **`System.Timers.Timer`** időzítő **`Elapsed`** eseményére), azok továbbra is az adott szálon kerülnek kiváltásra, és ha kezelésükhöz a felületi vezérlők kezelése szükséges, manuálisan szinkronizálnunk kell a szálakat.

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- Windows Forms alkalmazásokban az implicit szinkronizáció a UI szálra await utasításokat követően kikapcsolható a **`ConfigureAwait(false)`** meghívásával a taszkra:

```
await MethodAsync().ConfigureAwait(false);
```

  - Javítható a program teljesítménye, ha egyébként gyakori és szükségtelen szinkronizációra lenne szükség.

- Ha a bevárni kívánt taszk időközben elkészült, akkor szinkronizáció nélkül is az eredeti szálon maradhatunk:

```
Task task = …
…
await task.ConfigureAwait(false);
// ha a task elkészült, akkor az eredeti szálon
// folytatódik a végrehajtás, ha még nem,
// akkor egy másikon
```

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- Finomabb vezérlés igényekor a taszkok a **`TaskScheduler`** típussal ütemezhetőek, hogy mely szálon kerüljenek végrehajtásra
  - A **`TaskScheduler.Default`** az alapértelmezett, *thread pool* alapú ütemező, amely egy új *thread pool*-ból elérhető szálon ütemezi a végrehajtást.
  - A statikus **`FromCurrentSynchronizationContext`** metódussal egyszerűen kérhető az aktuális szinkronizációs kontextushoz egy **`TaskScheduler`** objektum.
  - A **`TaskScheduler.Current`** egy taszkon belül az aktuális, egyébként az alapértelemezett ütemezőt adja meg.
  - Egy **`TaskScheduler`** típusú paraméter átadható a taszkok példányosításakor.

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- Jellemzően nincsen szükség szinkronizációra, de a grafikus felület vezérlőinek elérésekor igen.
- Taszkok végrehajtása egymás után láncolható a **`ContinueWith`** metódus használatával.

```
TaskScheduler scheduler =
TaskScheduler.FromCurrentSynchronizationContext();
  // taszk szinkronizációs objektum

Task.Run(() => DoBackgroundWork())
   .ContinueWith(() => { label.Text = "Ready." },
    scheduler);
   // a DoBackgroundWork() futtatása aszinkron módon
   // háttérszálon történik;
   // majd a UI (szöveges címke) frissítése szinkron,
   // szálbiztos módon történik
```

Windows Forms alkalmazások párhuzamosítása

Taszkok szinkronizálása

- Egy lehetséges haladó felhasználás a **`TaskScheduler`**-ek használatára a keretendszerben implementált **`ConcurrentExclusiveSchedulerPair`**.

- Ennek a *scheduler*-nek két ütemezője van, a **`ConcurrentScheduler`** és a **`ExclusiveScheduler`**. Az előbbivel ütemezett taszkok futhatnak párhuzamosan, míg az utóbbival ütemezett taszkok nem.

```
var cesp = new ConcurrentExclusiveSchedulerPair();
Task task = Task.Factory.StartNew(() =>  {
     // olyan tevékenység, amelynek elvégzésekor
     // garantáltan ne fusson más taszk a
     // cesp objektummal ütemezve
     },
     … , cesp.ExclusiveScheduler);
```