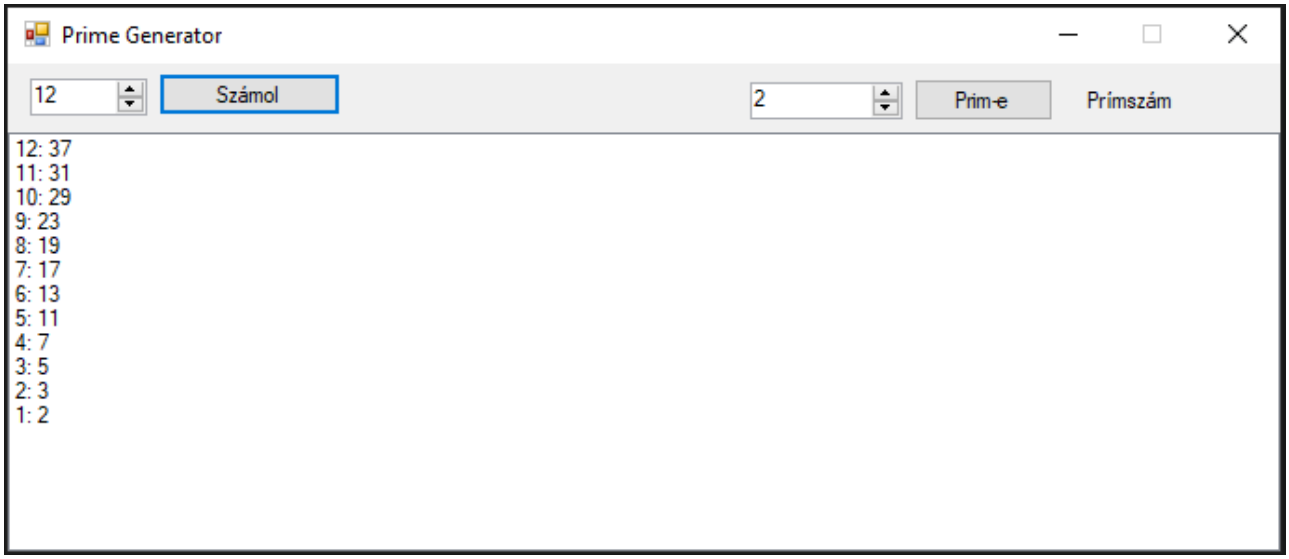
Prímszám generátor

Egy prímszám generáló WinForms alkalmazást fogunk elkészíteni kétrétegű (modell-nézet) architektúrában.

Az alkalmazás képes előállítani az első n darab prímszámot, valamint meg tudja mondani egy számról, hogy prím-e. A már kigenerált prímeket optimalizáció céljából eltároljuk, hogy egy későbbi hívásnál gyorsabban adhassuk vissza az értékeket. Nagy számok esetén az optimalizáció mellett is lassú lehet a számolás, így szükséges az alkalmazást párhuzamosítani, hogy a felület reszponzív maradjon. Adjunk lehetőséget a számolás leállítására is!

A felületen legyen látható egy *NumericUpDown*, ahol megadhatjuk, hogy az első hány prímet generáljuk ki. Mellette legyen egy számoló gomb, amire kattintva az alul lévő *ListBox*-ban megjelenítjük az előállított prímeket. Emellett legyen a felületen egy másik *NumericUpDown* is, ahol megadhatunk egy számot, és lekérdezhetjük róla, hogy prím-e. Ezt az információt egy labelen fogjuk megjeleníteni.



Első n prím generálása

Írjuk ki első funkcióként az első n prímet, a gombra kattintva. A prímelőállítás logikája kerüljön egy modell rétegbe tartozó osztályba (`PrimeGenerator`). A metódus neve legyen `GeneratePrimes`. Addig vegyük a következő számot, és teszteljük, hogy prím-e, amíg elő nem állítottunk n darabot. A szám legyen `Int64` típusú, hogy nagyobb számokkal is dolgozhassunk. Ahhoz, hogy megállapítsuk egy számról, prím-e, keressünk osztót 2 és a szám négyzetgyöke között! Lassítsuk egy kicsit az algoritmust, hogy méghosszabbá tegyük a számítást.

```
Thread.Sleep(10); // Szándékos lassítás
```

Egy prím megtalálásáról a modell váltson ki eseményt (`NewGenResult`), amire a nézet feliratkozik (`GotNewGenResult`). Az eseményargumentumban átadjuk azt, hogy hányadik prímet találtuk meg, valamint magát a számot. Ha kiváltódik az esemény, a nézetben a *ListBox*-ra kiírjuk az új prímet.

Ezt megvalósítva látszik, hogy a felület a számolás alatt nem kattintható, ezen szeretnénk a párhuzamosítással változtatni.

Párhuzamosítás

A `GeneratePrimes` metódusban lévő számolás logikáját refaktoráljuk egy külön privát függvénybe (`Generate`), és ezt futtassuk egy külön szálon. Ehhez Taskokat fogunk használni.

```
var task = new Task(() => Generate(n));
```

Ezzel önmagában még nem történik semmi, a taszkot el is kell indítani (`Start` metódus).

Arra viszont figyelnünk kell, hogy az esemény így a háttérszálból fog kiváltódni, tehát a nézetben megírt eseménykezelő a háttérszálból próbál meg változtatni a felületen. Ezt elkerüljük a következő módon.

```
if (this.generateResults.InvokeRequired)
{
     BeginInvoke(new EventHandler<GenResultEventArgs>(this.GotNewGenResult), sender, e);
     return;
}
// Ezután már szabad a generateResults ListBox-ba beszúrni...
```

Azaz, ha ezt az eseménykezelőt egy másik szálról hívták meg, mint amelyikről létrehoztuk a listboxot, akkor újra meghívjuk ezt a metódust, már a megfelelő szálon.

Ha futtatjuk a programot, ellenőrizhetjük, hogy így már nem fog lefagyni a felület, reszponzív marad.

Generálás leállítása

Számolás közben a gomb felirata legyen *Leállít,* ehhez nyilván kell tartanunk a nézetben, hogy fut-e a generálás (`isGenerationRunning`). Ha számolás közben a leállítás gombra kattintunk, akkor állítsuk le a számolást! Ehhez vezessünk be a modellben egy új metódust (`CancelPrimeGenerating`). Egy taszk leállítását egy `CancellationToken`-en keresztül kérhetjük. Egy cancellation tokent úgy hozhatunk létre, hogy példányosítjuk a forrást (`CancellationTokenSource`), majd elkérjük a neki megfelelő tokent.

```
this.genCancelSource = new CancellationTokenSource();
this.genCancelToken = this.genCancelSource.Token;
```

A taszk pedig a következőképpen hozható létre.

```
var task = new Task(() => Generate(n), genCancelToken);
```

A source-ot el szeretnénk érni a `CancelPrimeGenerating` metódusban is, így emeljük ki osztályszintű változóba. A leállítást a `Cancel` metódussal kérhetjük.

Önmagában ezzel még nem fogjuk megállítani a futó számítást, erre fel kell készítenünk a `Generate` metódust. A ciklusba szervezett logikát megállíthatjuk egy adott ponton a break utasítással. A metódusból lekérdezhető a tokenen keresztül, hogy leállították-e a számítást. Ha igen, ne folytassuk tovább a prímgenerálást.

```
if (genCancelToken.IsCancellationRequested)
    break;
```

Generálás végének jelzése

Szeretnénk a folyamat végén jelezni a nézet felé, hogy a számítás leállt. Ezt a `GenerationReady` eseményen keresztül tesszük, az eseménykezelő pedig átírja a gomb szövegét, valamint frissítjük az `isGenerationRunning` változót.

A megtalált prímeket eltárolhatjuk egy cache-be, hogy a későbbi számításokat gyorsítsuk. Erre alkalmas egy lista, amin a generálás előtt végigmegyünk, és jelezzük a nézetnek, hogy ezeket a prímeket már megtaláltuk. A generálást pedig a legutolsó talált prímtől folytatjuk.

Annak eldöntése, hogy egy szám prím-e, a fentiekkel hasonló módon kivezethető a felületre külön taszkként, külön cancellation tokennel és source-szal.

Egységtesztek létrehozása

Hozzunk létre egy új *Unit Test Project*et a solution alá! Legyen a neve *PrimeNumberGeneratorTest*. Létrejött egy tesztosztály, valamint egy tesztmetódus. Vegyük fel adattagként a tesztelendő osztály egy példányát, és vegyünk fel egy olyan metódust, ami minden tesztmetódus előtt le fog futni, és inicializálja a példányt. Amikor hivatkozni próbálunk a modellre, a Visual Studio meg fogja kérdezni, hogy szeretnénk-e egy másik projektben lévő osztályra hivatkozni. Legyen a válaszunk igen.

```
[TestInitialize]
public void Initialize()
{
     generator = new PrimeGenerator();
}
```

Teszteljük le azt, hogy néhány nem prímszámra valóban hamisat ad-e a számolás. Az eredményt eseményen keresztül kapjuk, így fel kell rá iratkoznunk a tesztmetódusban.

```
generator.CalculationResult += delegate (object sender, CalcResultEventArgs e)
{
     Assert.IsFalse(e.IsPrime);
};
```

Ezután hívjuk meg néhány nem prímszámra a generator alkalmas metódusát! Készítsünk tesztet néhány prímszámra is!