

## Eseményvezérelt alkalmazások: 2. gyakorlat

A munkafüzet teljesítésével egy konzolos alkalmazást fejleszthetünk, gyakorolva a szöveges fájlkezelést és a LINQ használatát C#-ban. Figyeljük meg, hogy a `DocumentStatistics` osztályban nem kezeljük a konzolt, ilyen módon könnyen újrafelhasználható lesz későbbi gyakorlaton grafikus alkalmazáshoz is (*modell*).

### 1 Dokumentum statisztikák <sup>KM</sup>

Készítsünk egy konzolos alkalmazást, amely beolvassa egy szöveges fájl tartalmát, majd a szövegre vonatkozó statisztikákat és metrikákat számol ki belőle.

A Visual Studio 2022 elindítása után válasszuk a **Create a new project** menüpontot.

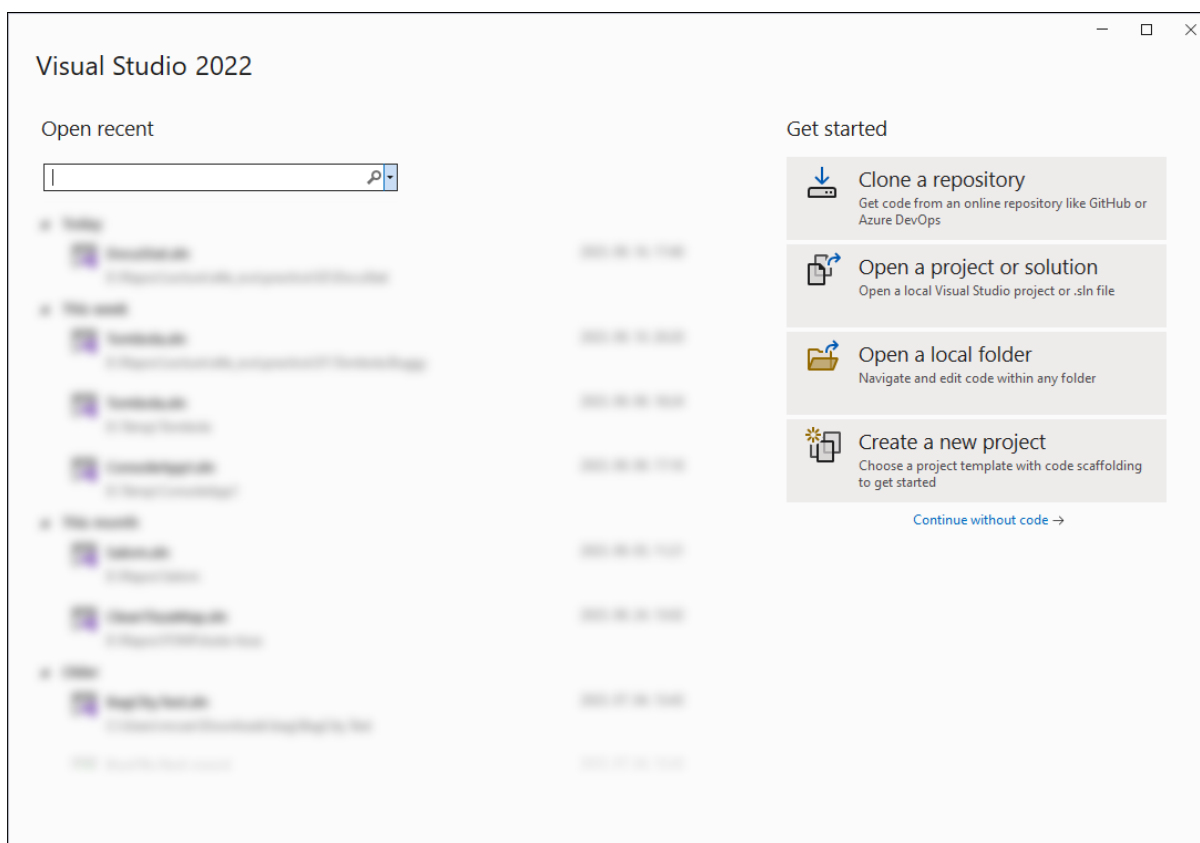


Figure 1: Új solution létrehozása

A következő ablakban válasszuk ki a **Console App** projektsablont.

**Ne** a *Console App (.NET Framework)* sablont válasszuk, ugyanis az .NET 8 helyett a korábbi .NET Frameworkre épülő projektet hozna létre.

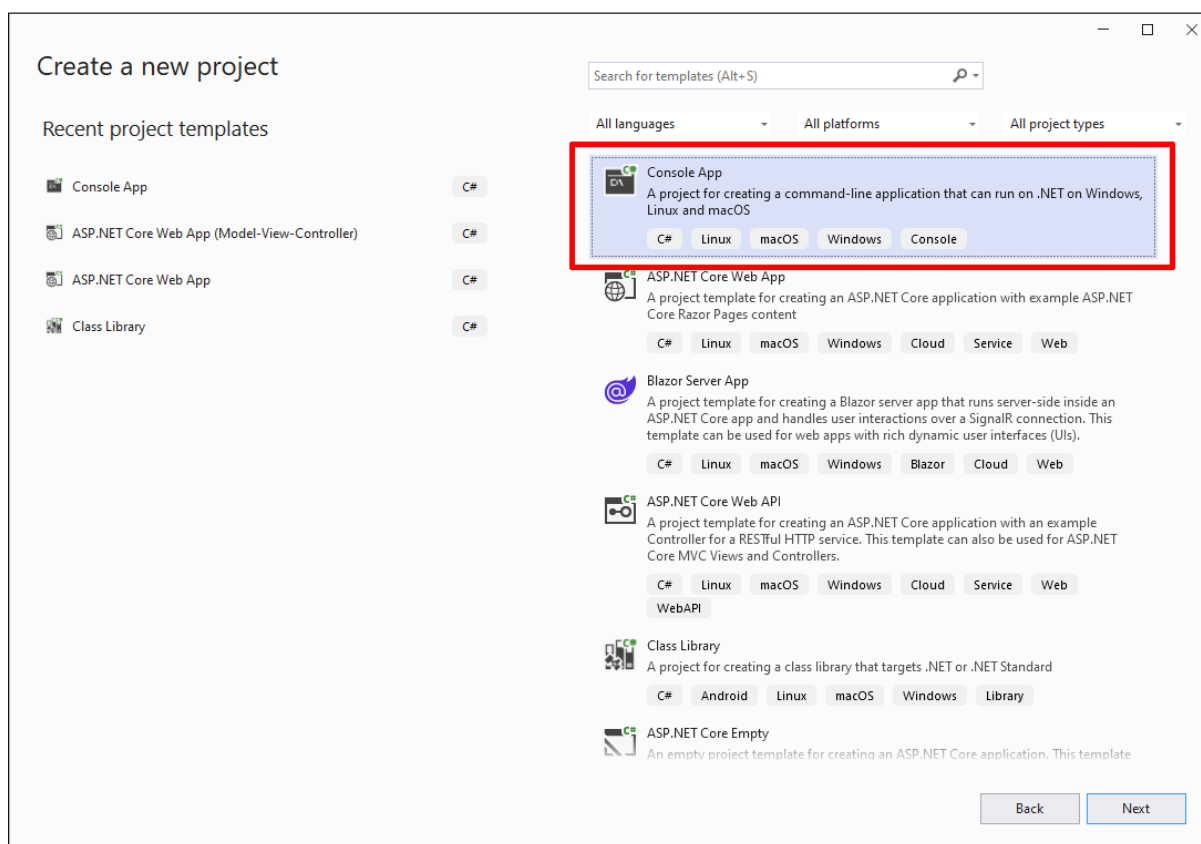
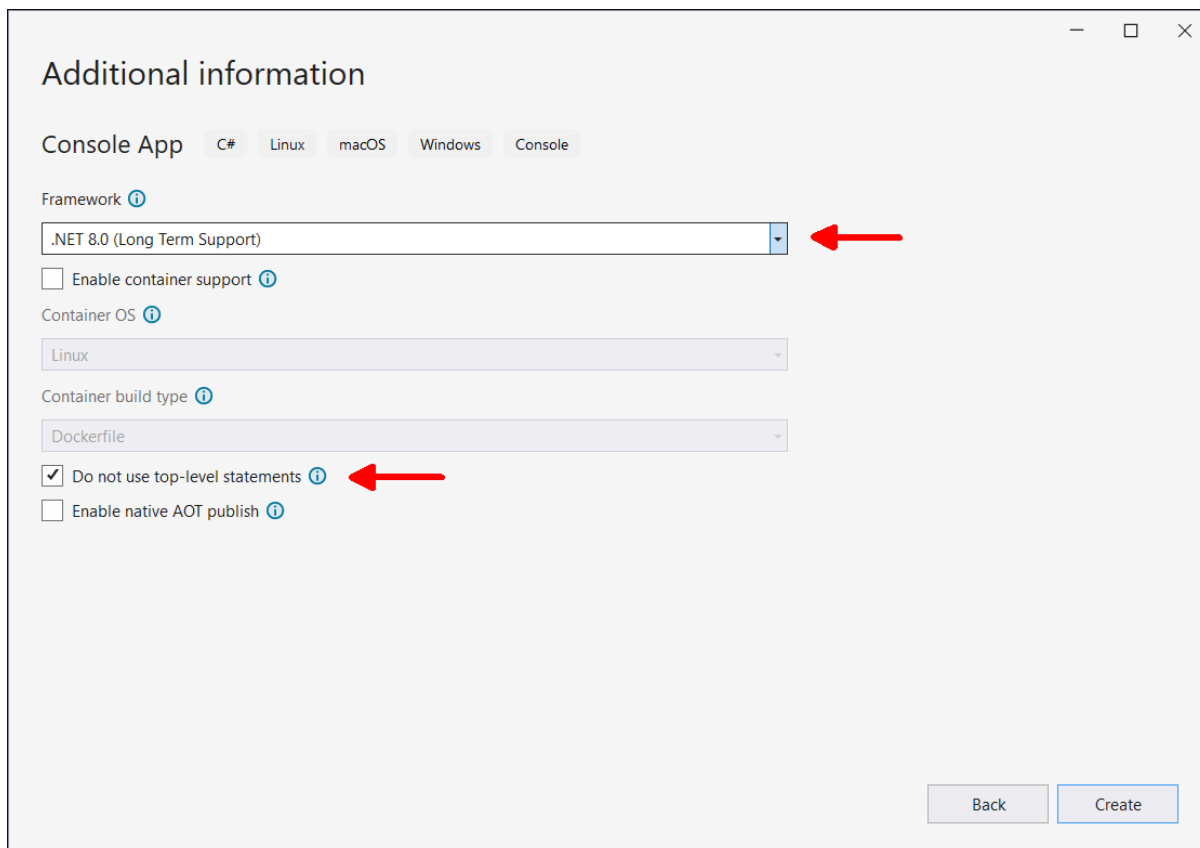


Figure 2: Új projekt létrehozása

A Next gombra való kattintás után megadhatjuk a projekt és a solution nevét. A .NET megoldásokban (*Solution*) gondolkodik, egy Solution több projektet is tartalmazhat. Alapértelmezetten a solution neve megegyezik az első projekt nevével, de ezt lehet módosítani.

Tovább lépve kiválaszthatjuk a cél keretrendszert: válasszuk a hosszú távú támogatással (LTS) rendelkező **.NET 8** verziót.

Pipáljuk be a **Do not use top-level statements** opciót, hogy a `Program.cs` fájlba a belépési ponthoz tartozó `Main` eljárást és a tartalmazó osztályt is legenerálja a Visual Studio.



Additional information

Console App C# Linux macOS Windows Console

Framework ⓘ  
[.NET 8.0 (Long Term Support)]

☐ Enable container support ⓘ

Container OS ⓘ  
[Linux]

Container build type ⓘ  
[Dockerfile]

☒ Do not use top-level statements ⓘ

☐ Enable native AOT publish ⓘ

Back Create

Figure 3: Cél keretrendszer kiválasztása

Az induló projekt a *Solution Explorer*-ben a projekt nevén jobb klikk, majd a **Set as StartUp Project** menüpontot választva adható meg. A sablon mindössze egy **Program** nevű osztályt generál, amelyben a statikus **Main** függvény található.

A fájlbeolvasáshoz és a számítások elvégzéséhez hozzunk létre egy új fájlt **DocumentStatistics.cs** néven. Ebben automatikusan létrejön egy osztály ugyanezen a néven.

- Készítsünk egy privát `_filePath` adattagot `string` típussal, amely a fájl elérési útját fogja tárolni.
- Adjunk az osztályhoz egy `FileContent` nevű tulajdonságot (*property*) is a fájl tartalmának, amely legyen publikus, a típusa legyen `String`, és rendelkezzen egy publikus `get` és egy privát `set` *accessorral* (“elérő”)!

## 2 Fájl tartalmának beolvasása <sup>KM</sup>

### 2.1 Elérési útvonal bekérése

A beolvasás két lépésből áll. Első lépésben bekérünk a felhasználótól egy érvényes abszolút útvonalat egy szöveges (txt kiterjesztésű) fájlhoz. Ezt végezzük a **Main** függvényben, amely addig olvas be a konzolról, amíg egy létező szöveges fájl nem kapunk. A beolvasást a `Console.ReadLine()` függvénnyel végezzük. A kiterjesztést a `System.IO.Path.GetExtension()` metódussal, a létezés ellenőrzését a `System.IO.File.Exists()` metódussal végezzük.

Az elérési útvonalat a **DocumentStatistics** osztály konstruktora vegye át paramétereként és mentse el a `_filePath` privát adattagba.

## 2.2 Fájl beolvasása

A második lépés a fájl tartalmának beolvasása. Ehhez készítsük el a `Load` nevű függvényt a `DocumentStatistics` osztályban. A metódusban a `System.IO.File.ReadAllText(_filePath)` meghívása stringként adja vissza a fájl tartalmát, ezzel beállíthatjuk a `FileContent` értékét. A `ReadAllText()` `System.IO.IOException` kivételt okozhat, ezt azonban majd csak a hívó metódusok fogják elkapni.

Hívjuk meg a `Main`-ben a `Load` függvényt, és egy `try-catch` blokkban kezeljük a kivételt! A `catch` ágban írjuk ki a kivétel üzenetét (`Message`) a `Console.WriteLine()` paranccsal, majd lépünk ki egy nullától különböző értékkel, így jelezve, hogy valami nem megfelelően zajlott a program futása közben. Például mínusz egyes hibakód esetén: `return -1;`. Ehhez a `Main` függvény visszatérési típusát változtassuk `int`-re és a helyes működést reprezentáló nullás értékkel térjünk vissza a függvény végén.

## 3 Szavak előfordulásának megszámlálása *EM*

A szövegen végzett műveletek elvégzéséhez definiáljunk egy `ComputeDistinctWords` nevű metódust a `DocumentStatistics` osztályban! A szavak előfordulásainak számát egy `string` - `int` párokat tartalmazó konténerben fogjuk tárolni (`Dictionary<string, int>`). Ezt a konténert is tulajdonságként vegyük fel, hasonlóan a `FileContent`-hez. Adjuk neki a `DistinctWordCount` nevet, és inicializáljuk az osztály konstruktorában.

A `ComputeDistinctWords`-ben a szöveget tördeljük szavakra a `Split()` függvény hívásával! Ha ezt paraméter nélkül hívjuk meg, automatikusan a *whitespace*-eknél fogja szétvágni a szöveget. A `Split()` eredménye egy `string` tömb (`words`). A tömbben ekkor még lehetnek üres stringek, ezeket egy `lambdakifejezéssel` távolítsuk el a tömbből:

```
words = words.Where(s => s.Length > 0).ToArray();
```

A `words` tartalmát tehát felülírjuk a nem üres stringekkel. A `Where` hívás egy `IEnumerable`-t ad vissza, amit tömbbé kell konvertálnunk az értékadáshoz (`ToArray()`).

A maradék szavakon iteráljunk végig. A szavak elejéről vegyük le az összes nem betű karaktert. A karaktereket a `Char.IsLetter()` metódussal vizsgáljuk meg! Az aktuális szóból a `Remove` metódus hívásával távolíthatjuk el karaktereket. Ennek paraméterként adjuk meg az eltávolítás kezdőindexét (ez a 0. indexű elem lesz), és az eltávolítandó karakterek számát. Ne felejtjük el mindig vizsgálni, hogy van-e még karakter a szóban, így kerüljük el a túlindexelést.

```
while (words[i].Length > 0 && !Char.IsLetter(words[i][0]))
{
    words[i] = words[i].Remove(0, 1);
}
```

Ugyanezt tegyük meg fordítva, a szó utolsó karakterétől kezdve. A `^1` indexeléssel hivatkozhatunk a szó utolsó karakterére (természetesen a szó hosszából számított indexelés is jó megoldás.)

```
while (words[i].Length > 0 && !Char.IsLetter(words[i][^1]))
{
    words[i] = words[i].Remove(words[i].Length - 1, 1);
}
```

Alternatív megoldásként a nem betű karakterek eltávolítására használhatjuk a LINQ nyújtotta lehetőségeket. A `words[i].SkipWhile(...)` metódusnak paraméterül megadhatunk egy feltételt, hogy addig hagyjuk el a string karaktereit, amíg ez teljesül. A karaktereket továbbra is a `Char.IsLetter()` metódussal vizsgáljuk meg. Ezt a köztes eredményt a `Reverse()` metódussal megfordítjuk, majd a megfordított tömbön is elvégezzük a korábbi műveletet. Ezután ne felejtsük el visszafordítani az eredményt. Az eredményt a `String.Concat(...)` metódussal tudjuk visszaalakítani egyszerű stringgé.

```
string.Concat(
    words[i]
    .SkipWhile(c => !char.IsLetter(c))
    .Reverse()
    .SkipWhile(c => !char.IsLetter(c))
    .Reverse()
);
```

Ezután lehetséges, hogy újabb üres string keletkezett (pl. egy évszám esetén). Ezt vizsgáljuk meg a `String.IsNullOrEmpty()` függvénnyel, és ha igazat kapunk, lépünk tovább a következő szóra!

Az aktuális szót alakítsuk át a `ToLower()` függvénnyel, hogy csak kisbetűket tartalmazzon, így elkerüljük a kis- és nagybetűs különbségekből adódó duplikációkat.

Vizsgáljuk meg, hogy a szó benne van-e már a `DistinctWordCount` kulcsai között (`ContainsKey()`). Ha igen, növeljük a szóhoz tartozó értéket 1-gyel (használható az index operátor `[]`), egyébként vegyük fel a szót a `DistinctWordCount`-ba az `Add()` metódussal, értéke 1 legyen.

A `Main`-ben ezután hívjuk meg `ComputeDistinctWords` függvényt. A `Dictionary` egy rendezetlen típus, ezért a kiírás előtt rendeznünk kell az adatokat az előfordulások száma szerint. Rendezzük az értékek szerint `DistinctWordCount`-ot az `OrderByDescending()` metódussal, amelynek a rendezés feltételét egy lambdakifejezésként adjuk át, majd az eredmény `IEnumerable`-t mentjük el egy lokális változóba.

A rendezést LINQ lekérdezéssel, többféle szintaxissal is végezhetjük.

1) *Method Syntax:*

```
var pairs = stat.DistinctWordCount.OrderByDescending(p => p.Value);
```

2) *Query Syntax:*

```
var pairs = from pair in stat.DistinctWordCount
            orderby pair.Value descending
            select pair;
```

A két módszer ugyanazt az eredményt adja. Ezután a `pairs`-en végigiterálhatunk egy `foreach` ciklussal, hogy kiírjuk a kulcs-érték párokat.

```
foreach (var pair in pairs)
{
    Console.WriteLine($"{pair.Key}: {pair.Value}");
}
```

### 3.1 Eredmény szűrése hossz és előfordulás alapján

Tegyük paraméterezhetővé, hogy csak a felhasználó által megadott minimális előfordulás számot (`minOccurrence`) és karakterszámot (`minLength`) elérő szavakat jelenítsük meg az eredmények között.

A paramétereket a felhasználótól az alkalmazás kérje be, a konzolról történő olvasáskor ügyeljünk arra, hogy a felhasználó nem csak egész számot adhat meg (`int.Parse()` vagy `int.TryParse()`). A kódredundancia csökkentése érdekében készítsünk egy `static int ReadPositive(string message)` segéd eljárást a `Program` osztályba, amely sikeresen beolvas egy pozitív egész számot a konzol inputról (újból próbálkozik, ha nem sikerült).

A `pairs` gyűjteményt szűrjük az így kapott minimális előfordulás szám és karakterszám szerint:

```
var pairs = stat.DistinctWordCount
    .Where(p => p.Value >= minOccurrence)
    .Where(p => p.Key.Length >= minLength)
    .OrderByDescending(p => p.Value);
```

### 3.2 Eredmény szűrése kizárt szavak alapján

A felhasználó adhasson meg egy kizárt szavak listáját is (`ignoredWords`). A szavakat egy sorban is beolvashatjuk, majd egy elválasztó karakter mentén (szóköz vagy vessző) a `string` típus `Split()` függvényével feldarabolhatjuk, így egy `List<string>`-et kapunk a kizárt szavakkal.

```
var pairs = stat.DistinctWordCount
    .Where(p => p.Value >= minOccurrence)
    .Where(p => p.Key.Length >= minLength)
    .Where(p => !ignoredWords.Contains(p.Key))
    .OrderByDescending(p => p.Value);
```

## 4 További metrikák <sup>OP</sup>

### 4.1 Karakterek száma

Számítsuk ki a szöveg hosszát szóközökkel és anélkül!

Adjunk a `DocumentStatistics` osztályhoz egy `CharacterCount` és egy `NonWhiteSpaceCharacterCount` tulajdonságot. Típusuk legyen `int`, publikusan csak olvashatóak legyenek.

A `Load` eljárásban számítsuk ki, hogy hány karakterből áll a beolvasott fájl, és ezek közül hány nem *whitespace* karakter.

A `Main` eljárásban ezeket a metrikákat is jelenítsük meg a fájl feldolgozását követően.

### 4.2 Mondatok száma

Számítsuk ki a szöveget alkotó mondatok számát!

Adjunk a `DocumentStatistics` osztályhoz egy `SentenceCount` tulajdonságot. Típusa legyen `int`, publikusan csak olvasható legyen. Egészítsük ki az osztályt egy `ComputeSentenceCount` privát metódussal is, amely a `SentenceCount` értékét kiszámítja. Ezt a `Load` eljárásban fogjuk meghívni.

A mondatokat úgy számlálhatjuk meg, hogy a mondatvégi írásjeleket számoljuk meg (`.`, `?`, `!`) a szövegben. Ügyeljünk arra is, hogy egymást követő több mondatvégi írásjellet (pl. `?!)` csak egy mondatnak számítsunk.

A metrika nem lesz tökéletes, hiszen például pont karakter nem csak mondatvégi írásjelként szerepelhet a szövegben, de a feladat céljára ez így megfelelő lesz. Jobb eredmény például mesterséges intelligencia alkalmazásával lenne elérhető. Ld. a [NLP \(Natural Language Processing\)](#) témakörét!

A `Main` eljárásban a mondatok számát is jelenítsük meg a fájl feldolgozását követően.

### 4.3 Tulajdonnevek száma

Számítsuk ki a szövegben található tulajdonnevek számát!

Adjunk a `DocumentStatistics` osztályhoz egy `ProperNounCount` tulajdonságot. Típusa legyen `int`, publikusan csak olvasható legyen. Egészítsük ki az osztályt egy `ComputeProperNounCount` privát metódussal is, amely a `ProperNounCount` értékét kiszámítja. Ezt is a `Load` eljárásban fogjuk meghívni.

Az egyszerűség kedvéért definiáljuk a tulajdonneveket úgy, mint olyan szavak, amik nagy betűvel kezdődnek, és a megelőző karakter nem mondatvégi írásjel. (A *whitespace* karaktereket figyelmen kívül hagyva.)

A `Main` eljárásban a tulajdonnevek számát is jelenítsük meg a fájl feldolgozását követően.

## 4.4 Coleman-Liau olvashatósági index

Számítsuk ki a szövegre vonatkozó **Coleman Liau olvashatósági indexet**! A metrika formulája a következő:

$$(0.0588 * L) - (0.296 * S) - 15.8$$

Ahol  $L$  a 100 szóra jutó átlagos karakterhossz,  $S$  pedig a 100 szóra jutó mondatok száma átlagosan.

Az index által mutatott érték hozzávetőleges iránymutatást ad, hogy az USA oktatási rendszere szerint hányadik osztályos tanuló részére lesz jól érthető a szöveg.

Adjunk a `DocumentStatistics` osztályhoz egy `ColemanLieuIndex` tulajdonságot. Típusa legyen `int`, publikusan csak olvasható legyen. Egészítsük ki az osztályt egy `ComputeColemanLieuIndex` privát metódussal is, amely a `ColemanLieuIndex` értékét kiszámítja. Ezt is a `Load` eljárásban fogjuk meghívni.

A szöveg teljes karakterhossza a `NonWhiteSpaceCharacterCount`, a mondatok száma a `SentenceCount` tulajdonságban már rendelkezésünkre áll, a szavak száma pedig a `DistinctWordCount` tulajdonságból összegezzük egyszerűen.

```
int totalwordCount = DistinctWordCount.Sum(w => w.Value);
```

A `Main` eljárásban a *Coleman Liau olvashatósági indexet* is jelenítsük meg a fájl feldolgozását követően.

## 4.5 Flesch Reading Ease metrika

Adjuk meg a szöveg **Flesch Reading Ease** pontszámát a következő formulával:

$$206.835 - 1.015 * \frac{\text{szószám}}{\text{mondatyszám}} - 84.6 * \frac{\text{szótagszám}}{\text{szószám}}$$

A *Flesch Reading Ease* 0 és 100 közötti pontszámot ad, a magasabb szám könnyebben olvasható szöveget jelent. Például 60 és 70 közötti pontszám javasolt 7-8. osztályos tanulóknak, míg 30 alatti pontszám felsőoktatásban tanulóknak.

Adjunk a `DocumentStatistics` osztályhoz egy `FleschReadingEase` tulajdonságot. Típusa legyen `int`, publikusan csak olvasható legyen. Egészítsük ki az osztályt egy `ComputeFleschReadingEase` privát metódussal is, amely a `FleschReadingEase` értékét kiszámítja. Ezt is a `Load` eljárásban fogjuk meghívni.

A korábban létrehozottakon kívül ehhez már csak egy olyan segédeljárás szükséges, ami a szövegben található szótagokat számlálja meg. Ehhez kellően jó megoldás, ha feltesszük, hogy minden magánhangzó egy szótagot képez, kivéve a dupla magánhangzókat. Tekintsük a `y` betűt is magánhangzónak a szótagoláshoz a kiejtési szabályok miatt.

Tovább javítható a pontosság, ha számításba vesszük, hogy a szavak végén szereplő "es", "ed", vagy "le" általában szintén nem képez új szótagot az angol nyelvben, illetve a szó végi "e" csak akkor számít szótagnak, ha az egyetlen magánhangzó a szóban.  
A nyelvtani szabályok precízebb implementálásával jobb értékeket kaphatunk, de a feladat céljára egy közelítően pontos szótagszám is megfelelő most.

Ezen logika alapján készítsünk egy `CountSyllables` metódust a `DocumentStatistics` osztályban, ami a paraméterül kapott szóra visszaadja a benne szereplő szótagok számát, majd ezt futtassuk le a szöveg összes szavára.

A `Main` eljárásban a *Flesch Reading Ease* metrikát is jelenítsük meg a fájl feldolgozását követően.