Asynchronous Programming

And now asynchronous programming. Why do we need this? There is a very good reason why you should know something about it.
Consider the following case of writing a program that deals with very large files or communicates
with databases or simply performs a task that takes a long time to complete.
When you start working with a file or database, or anything similar, you will have delays because these tasks take a long time. And this will make your application not respond.
In such cases, it is very good to know how to apply simple asynchronous programming, at least to display a message on the console and not cause that our application does not respond.

AsynchronousProgramming - Microsoft Visual Studio

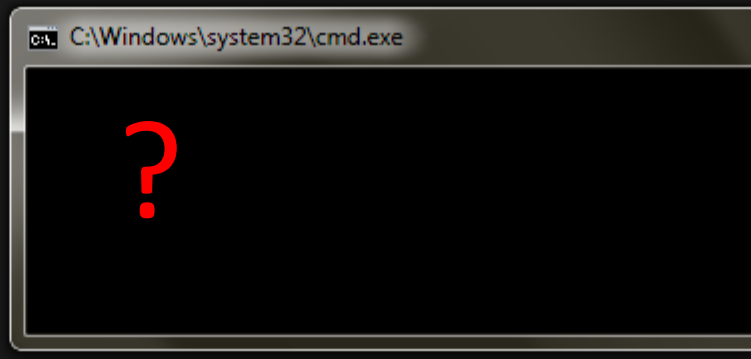
Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc

Debug | Any CPU | Rozpocznij

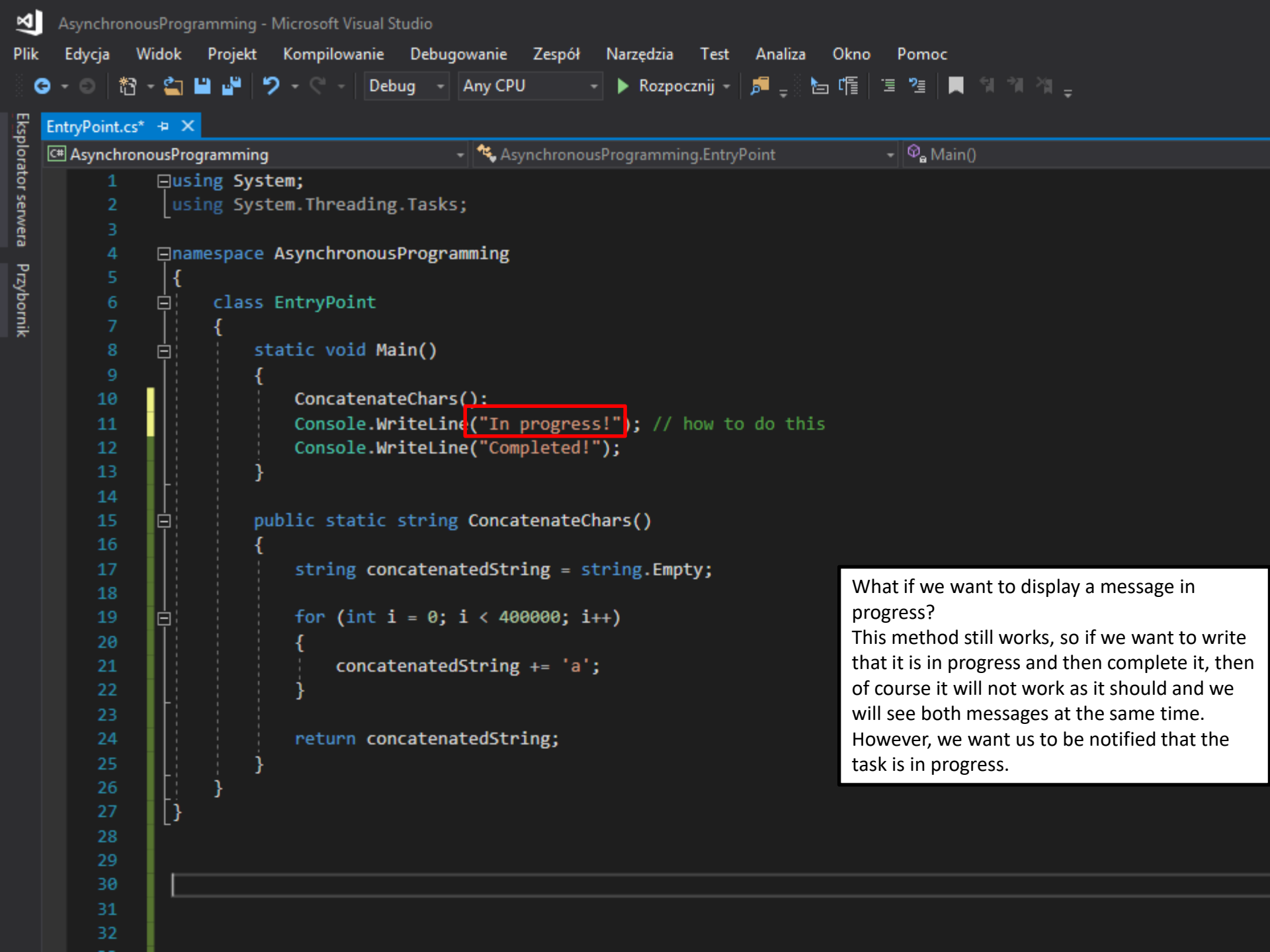
EntryPoint.cs

AsynchronousProgramming | AsynchronousProgramming.EntryPoint | Main()

```
1   using System;
2   using System.Threading.Tasks;
3
4   namespace AsynchronousProgramming
5   {
6       class EntryPoint
7       {
8           static void Main()
9           {
10              ConcatenateChars();
11              Console.WriteLine("Completed!");
12          }
13
14          public static string ConcatenateChars()
15          {
16              string concatenatedString = string.Empty;
17
18              for (int i = 0; i < 400000; i++)
19              {
20                  concatenatedString += 'a';
21              }
22
23              return concatenatedString;
24          }
25      }
26  }
27
28
29
30
31
32
```



Let's start with a very simple example.
Let's create a method that combines a very long string.
It takes so much time.
As part of this method, we will create a new linked chain.
We will create a loop to connect these subtitles hundreds of thousands of times in this way.
If we use this method now and then want to immediately print the message, the effect is that for a very long time we will not see the effect of the program.
Currently, our method does not work in the background.
It works and after receiving the method we receive a message.


AsynchronousProgramming - Microsoft Visual Studio
Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc
Debug
Any CPU
Rozpocznij
EntryPoint.cs*
AsynchronousProgramming
AsynchronousProgramming.EntryPoint
Main()
Eksplorator serwera
Przybornik
1 using System;
2 using System.Threading.Tasks;
3
4 namespace AsynchronousProgramming
5 {
6 class EntryPoint
7 {
8 static void Main()
9 {
10 ConcatenateChars();
11 Console.WriteLine("In progress!"); // how to do this
12 Console.WriteLine("Completed!");
13 }
14
15 public static string ConcatenateChars()
16 {
17 string concatenatedString = string.Empty;
18
19 for (int i = 0; i < 400000; i++)
20 {
21 concatenatedString += 'a';
22 }
23
24 return concatenatedString;
25 }
26 }
27 }
28
29
30
31
32
What if we want to display a message in progress?
This method still works, so if we want to write that it is in progress and then complete it, then of course it will not work as it should and we will see both messages at the same time.
However, we want us to be notified that the task is in progress.

AsynchronousProgramming - Microsoft Visual Studio

Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc

Debug | Any CPU | Rozpocznij

EntryPoint.cs

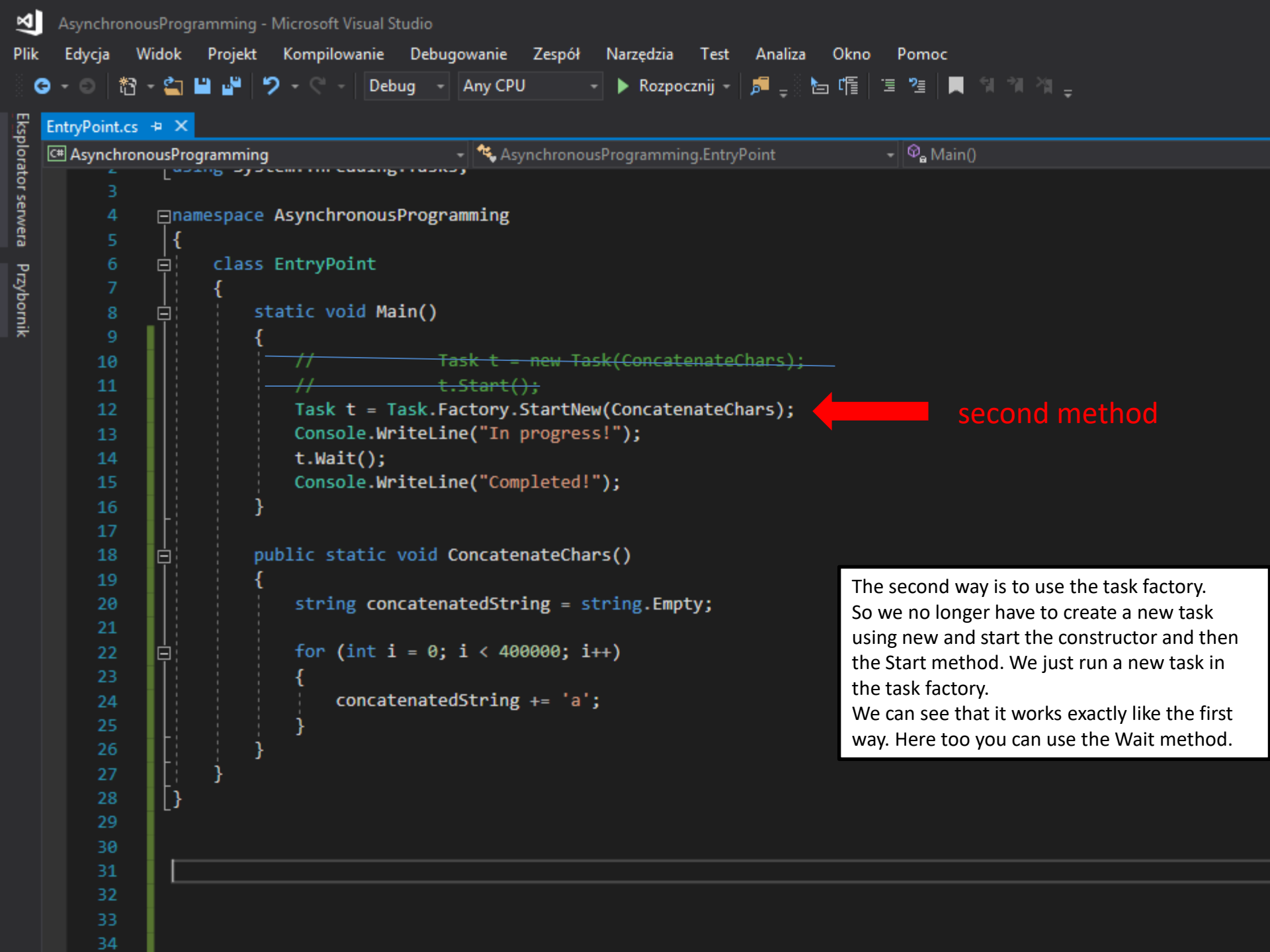
AsynchronousProgramming | AsynchronousProgramming.EntryPoint | Main()

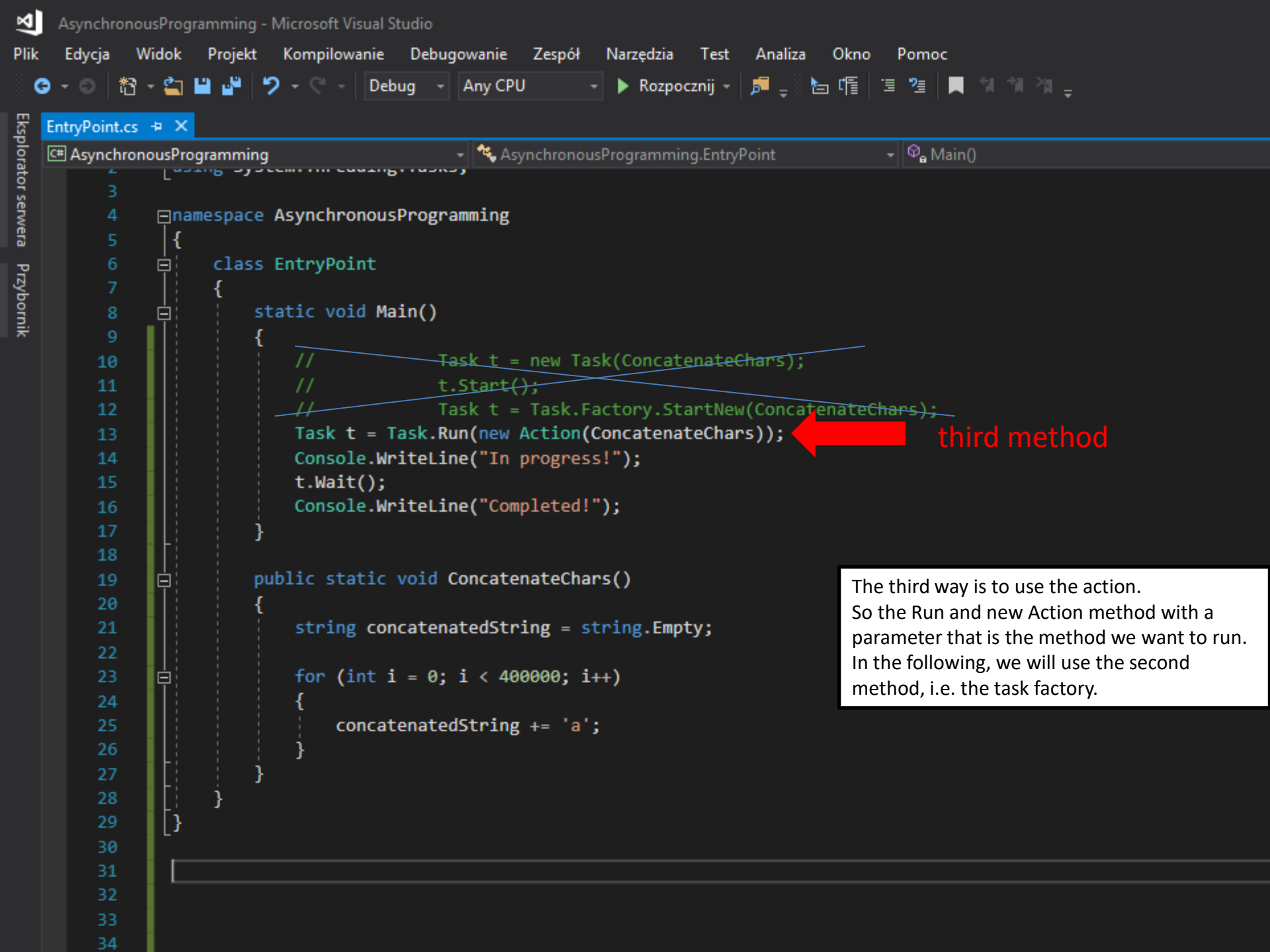
```
1   using System;
2   using System.Threading.Tasks;
3
4   namespace AsynchronousProgramming
5   {
6       class EntryPoint
7       {
8           static void Main()
9           {
10              //              ConcatenateChars();
11              Task t = new Task(ConcatenateChars);
12              t.Start();
13              Console.WriteLine("In progress!");
14              t.Wait();
15              Console.WriteLine("Completed!");
16          }
17
18          public static void ConcatenateChars()
19          {
20              string concatenatedString = string.Empty;
21
22              for (int i = 0; i < 400000; i++)
23              {
24                  concatenatedString += 'a';
25              }
26
27  //              return concatenatedString;
28          }
29      }
30  }
31
32
```

C:\Windows\system32\cmd.exe

```
In progress!
```

To do this, we must use a class called Task.
All we have to do is create a new instance of this class and pass the method as an argument to the constructor.
This constructor accepts methods that have no arguments and do not return a result.
Our method fits this signature because it meets these requirements.
For our method to start to run, you must still start the Start method belonging to the Task class.
Let's run it and see what the result is so far.
And you can see that both inscriptions appear at the same time.
To change this, we still need the Wait method. Everything that will be performed after this method will be after the task is completed in asynchronous mode. Now the subtitles appear when we intended to. Before we move on to the next lesson, I just want to show you that there are two other ways we can initiate a task in asynchronous mode.
It was the first way.


AsynchronousProgramming - Microsoft Visual Studio
Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc
Debug
Any CPU
Rozpocznij
EntryPoint.cs
AsynchronousProgramming
AsynchronousProgramming.EntryPoint
Main()
Eksplorator serwera
Przybornik
namespace AsynchronousProgramming
{
class EntryPoint
{
static void Main()
{
// Task t = new Task(ConcatenateChars);
// t.Start();
Task t = Task.Factory.StartNew(ConcatenateChars);
Console.WriteLine("In progress!");
t.Wait();
Console.WriteLine("Completed!");
}
public static void ConcatenateChars()
{
string concatenatedString = string.Empty;
for (int i = 0; i < 400000; i++)
{
concatenatedString += 'a';
}
}
}
}
second method
The second way is to use the task factory.
So we no longer have to create a new task using new and start the constructor and then the Start method. We just run a new task in the task factory.
We can see that it works exactly like the first way. Here too you can use the Wait method.


AsynchronousProgramming - Microsoft Visual Studio
Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc
Debug
Any CPU
Rozpocznij
EntryPoint.cs
AsynchronousProgramming
AsynchronousProgramming.EntryPoint
Main()
Eksplorator serwera
Przybornik
namespace AsynchronousProgramming
{
class EntryPoint
{
static void Main()
{
// Task t = new Task(ConcatenateChars);
// t.Start();
// Task t = Task.Factory.StartNew(ConcatenateChars);
Task t = Task.Run(new Action(ConcatenateChars));
Console.WriteLine("In progress!");
t.Wait();
Console.WriteLine("Completed!");
}
public static void ConcatenateChars()
{
string concatenatedString = string.Empty;
for (int i = 0; i < 400000; i++)
{
concatenatedString += 'a';
}
}
}
}
third method
The third way is to use the action.
So the Run and new Action method with a parameter that is the method we want to run.
In the following, we will use the second method, i.e. the task factory.

Using Task on methods with arguments

Okay, you've been through asynchronous method calls, but that's not a very useful way. We don't get any results and we can't use any arguments.
Let's say we want to change the amount of concatenation - we will do it by creating input arguments for the method.
So let's do it.

AsynchronousProgramming - Microsoft Visual Studio

Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc

Debug | Any CPU | Rozpocznij

EntryPoint.cs

AsynchronousProgramming | AsynchronousProgramming.EntryPoint | Main()

```
3
4   namespace AsynchronousProgramming
5   {
6       class EntryPoint
7       {
8           static void Main()
9           {
10              Task t = Task.Factory.StartNew(() =>
11              {
12                  ConcatenateChars('a', 400000);
13              });
14              Console.WriteLine("In progress!");
15              t.Wait();
16              Console.WriteLine("Completed!");
17          }
18
19          public static void ConcatenateChars(char charToConcatenate, int count)
20          {
21              string concatenatedString = string.Empty;
22
23              for (int i = 0; i < count; i++)
24              {
25                  concatenatedString += charToConcatenate;
26              }
27          }
28      }
29  }
30
31
32
33
34
```

C:\Windows\system32\cmd.exe

```
In progress!
```

If we create a method with two parameters. The first parameter indicates the character we will be adding, and the second parameter will determine how many times this operation is to be performed. We will then receive an error. Basically, this means that our method no longer corresponds to the action signature that is required for use. To get around this problem, we can use the Lambda expression and call this method an anonymous method. So we'll create a normal method that will be able to call another method. It looks something like this.
We use empty parentheses because we don't need to have any input arguments to satisfy the Action delegate.

AsynchronousProgramming - Microsoft Visual Studio

Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc

Debug Any CPU Rozpocznij

Eksplorator serwera Przybornik

EntryPoint.cs

AsynchronousProgramming AsynchronousProgramming.EntryPoint Main()

```
3
4   namespace AsynchronousProgramming
5   {
6       class EntryPoint
7       {
8           static void Main()
9           {
10              Task t = Task.Factory.StartNew(() => ConcatenateChars('a', 400000));
11              Console.WriteLine("In progress!");
12              t.Wait();
13              Console.WriteLine("Completed!");
14          }
15
16          public static void ConcatenateChars(char charToConcatenate, int count)
17          {
18              string concatenatedString = string.Empty;
19
20              for (int i = 0; i < count; i++)
21              {
22                  concatenatedString += charToConcatenate;
23              }
24          }
25      }
26  }
27
28
29
30
31
32
33
34
```

Since we only have one line of code here, we simply created a method that allows us to call our second method without restrictions.
We don't have to use curly braces, so we can just do it in one line. And it will continue to work.

Using Task on methods with return type

We learned how to call a method asynchronously and how to use a method that has specific call arguments.
The next logical step for us is to recover some data from this method.

AsynchronousProgramming - Microsoft Visual Studio

Plik Edycja Widok Projekt Kompilowanie Debugowanie Zespół Narzędzia Test Analiza Okno Pomoc

Debug | Any CPU | Rozpocznij

EntryPoint.cs

AsynchronousProgramming | AsynchronousProgramming.EntryPoint | Main()

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace AsynchronousProgramming
5  {
6      class EntryPoint
7      {
8          static void Main()
9          {
10             Task<string> t = Task.Factory.StartNew(() => ConcatenateChars('a', 400000));
11             Console.WriteLine("In progress!");
12             t.Wait();
13             Console.WriteLine("Completed!");
14             Console.WriteLine("The length of the result is " + t.Result.Length);
15         }
16
17         public static string ConcatenateChars(char charToConcatenate, int count)
18         {
19             string concatenatedString = string.Empty;
20
21             for (int i = 0; i < count; i++)
22             {
23                 concatenatedString += charToConcatenate;
24             }
25             return concatenatedString;
26         }
27     }
28 }
29
30
31
32
```

C:\Windows\system32\cmd.exe

```
In progress!
Completed!
The length of the result is 400000
Aby kontynuować, naciśnij dowolny klawisz . . .
```

If it's a method that returns something. Let's say - a string of characters that we want to use later in our code.
We need to make our method return a string.
It also requires our task to have a generic type string, because a task that does not contain types is a task that is used for methods that are void.
If we run the code now, it will still work, but we won't have access to the output of the method.
So we have the result, but we can't use it yet.
So when you have a method that returns a result, all you have to do is do the task and read the Result property from the Task class object. Of course, we can read it only after completing the method, i.e. after the Wait method.

The async and await keywords

Up to this point you have all the components that will make your methods asynchronous. You know how to call methods, pass arguments and get results.
In this part of the lecture you will learn keywords that will simply make your life easier in creating asynchronous code.
Let's do it and see where it takes us.

Let's use the async keyword here in the declaration of our method, we'll see that an error has occurred - asynchronous methods must be void. Let's create a new task that will be a platform for asynchronous invocation of our method. Now we need to complete the anonymous method. You can see here that we just use the same method. We're just going to transfer this code to our task factory. So remember that we need to keep the keyword return because we want to return a string. Because we want to catch our result only after completing the method - we will use the keyword await. Basically, we've created a sort of bubble.

```
4   namespace AsynchronousProgramming
5   {
6       class EntryPoint
7       {
8           static void Main()
9           {
10              int count = 200000;
11              char charToConcatenate = '1';
12              Task<string> t = ConcatenateCharsAsync(charToConcatenate, count);
13              Console.WriteLine("In progress");
14              Console.WriteLine("The length of the result is " + t.Result.Length);
15              string text = ConcatenateChars(charToConcatenate, count);
16              Console.WriteLine("Synchronously - The length: " + text.Length);
17          }
18
19          public static string ConcatenateChars(char charToConcatenate, int count)
20          {
21              string concatenatedString = string.Empty;
22              for (int i = 0; i < count; i++)
23              {
24                  concatenatedString += charToConcatenate;
25              }
26              return concatenatedString;
27          }
28
29          public async static Task<string> ConcatenateCharsAsync(char charToConcatenate, int count)
30          {
31              return await Task<string>.Factory.StartNew(() =>
32              {
33                  return ConcatenateChars(charToConcatenate, count);
34              });
35          }
```

```
C:\Windows\system32\cmd.exe
In progress
The length of the result is 200000
Synchronously - The length: 200000
Aby kontynuować, naciśnij dowolny klawisz . . .
```

asynchronously

synchronously

We can call our method asynchronously and synchronously. As you can see during the asynchronous form, we can perform other tasks. However, when we are going to read the Result property, the task is waiting for the calculations to complete. As you can see the results obtained are the same.