Л.Р. №4: АСИНХРОННА МОДЕЛЬ ПРОГРАМУВАННЯ

План

4.1. Поняття асинхронного коду.

Код називається асинхронним, якщо він запускає якусь довготривалу операцію, але не очікує її завершення. Протилежність – це блокуючий код, який нічого не робить, поки операція не завершиться.

До числа таких довготривалих операцій можна віднести:

• мережеві запити;

• доступ до диску;

• тривалі затримки.

Основна відмінність полягає в тому, в якому потоці виконується код. В усіх популярних мовах програмування код працює в контексті якого-небудь потоку операційної системи. Якщо цей потік продовжує робити що-небудь ще, поки виконується тривала операція, то код асинхронний. Якщо потік в цей час нічого не робить, значить, він заблокований і, отже, ви написали блокуючий код.

Для асинхронного коду характерні типові труднощі: **як дізнатися, коли операція завершена**? Лише тільки після цього можна приступити до обробки її результатів. В блокуючому коді все просто – наступний оператор розміщується безпосередньо після виклику тривалої операції. Але в асинхронному світі так зробити не можна, тому що розташований в цьому місці рядок майже напевно буде виконаний раніше, ніж асинхронна операція завершиться.

Для вирішення цієї проблеми передбачається цілий ряд прийомів, що дозволяють виконати код після завершення фонової операції:

• включити потрібний код у склад самої операції, після коду, що складає її основне призначення;

• підписатися на подію, що генерується після завершення;

• передавати делегат або лямбда-вираз, який повинен бути виконаний після завершення (зворотний виклик).

Переваги асинхронного коду:

- асинхронний код розблоковує потік, з якого він був запущений (менше ресурсів, часто існує лише один потік – інтерфейс користувача);
- відкриває можливість для паралельних обчислень.

4.2. Механізм async / await.

Починаючи з версії C#5.0, з'явився механізм async/await, що суттєво спрощує асинхронне програмування, на відміну від попередніх версій.

Механізм async дає простий спосіб виразити, що має зробити програма після завершення тривалої асинхронної операції.

Приклад блокуючого методу:

```
private void DumpWebPage(string uri)
{
       WebClient webClient = new WebClient();
       string page = webClient.DownloadString(uri);
       Console.WriteLine(page);
}
```

Еквівалентний йому асинхронний метод:

```
private async void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 string page = await webClient.DownloadStringTaskAsync(uri);
 Console.WriteLine(page);
}
```

Механізм async спеціально спроектований так, щоб максимально нагадувати блокуючий код. Ми можемо розглядати тривалі або віддалені операції так, як ніби вони виконуються локально і швидко, підвищуючи продуктивність за рахунок асинхронності.

Насправді потрібно пам'ятати, що операція виконується в фоновому режимі і відбувається зворотний виклик.

Необхідно мати на увазі, що багато засобів мови в асинхронному режимі ведуть себе по-іншому, зокрема:

- винятки і блоки try-catch-finally;
- значення, що повертаються методами;
- потоки та контекст;
- продуктивність.

4.3. Клас Task<T>

Абстракція, що представляє асинхронну операцію. Універсальний варіант, Task<T>, грає роль обіцянки повернути значення (тип T), коли в майбутньому, після завершення операції, воно стане доступним.

Створення задачі. Для початку створимо об'єкт типу Task за допомогою конструктора

```
public Task(Action дія)
```

і запустимо його, викликавши метод Start ().

```
//Сконструюввати об'єкт задачі
Task tsk=new Task(DemoTask.MyTask);

//Запустити задачу на виконання
tsk.Start();
```

Точкою входу повинен служити метод, що не приймає ніяких параметрів і не повертає ніяких значень.

```
public static void MyTask()
```

```
{
      Console.WriteLine("MyTask() is started.");
      for (int count=0;count<10;count++)
      {
            Thread.Sleep(500);
            Console.WriteLine("In the method MyTask() counter = " + count);
      }
      Console.WriteLine("MyTask() is done.");
}
```

У цій програмі окреме завдання створюється на основі методу MyTask (). Після того як почне виконуватися метод Main (), завдання фактично створюється і запускається на виконання. Обидва методи MyTask () і Main () виконуються паралельно.

Повернення значення із задачі. Задача може повертати значення. Це дуже зручно з двох причин. По-перше, це означає, що за допомогою задачі можна обчислити деякий результат. Подібним чином підтримуються паралельні обчислення. І по-друге, процес, що викликає, виявиться заблокованим до тих пір, поки не буде отримано результат. Це означає, що для організації очікування результату не потрібно ніякої особливої синхронізації.

Для того щоб повернути результат із задачі, досить створити цю задачу, використовуючи узагальнену форму `Task <TResult>` класу `Task`.

```
public Task(Func<TResult> функція)
public Task(Func<Object, TResult> функція, Object стан)
```

де функція означає делегат, що виконується. Тип `Func` використовується саме в тих випадках, коли задача повертає результат. У першому конструкторі створюється задача без аргументів, а в другому конструкторі - задача, що приймає аргумент типу `Object`, який передається як `стан`.

У наступній програмі створюються два методи. Перший із них, `MyTask()`, не приймає параметрів, а просто повертає логічне значення `true` типу `bool`. Другий метод, `SumIt()`, приймає єдиний параметр, який перетворюється до типу `int`, і повертає суму чисел, що менше заданого параметру.

```
namespace ReturnResult
{
      class Program
      {
            //Метод без аргументів, що повертає результат
            static bool MyTask()
            {
                  return true;
            }

            //Метод, що повертає суму чисел, що менше заданого параметру
            static int SumIt(object v)
            {
                  int x=(int) v;
                  int sum=0;
                  for (; x > 0; x--)
                        sum += x;
                  return sum;
            }

            static void Main(string[] args)
```

```
        {
            Console.WriteLine("Main Thread is starting.");

            //Сконструюввати та запустити об'єкт першої задачі
            Task<bool> tsk1=Task<bool>.Factory.StartNew(MyTask);
            Console.WriteLine("The Result after running of MyTask =
"+tsk1.Result);

            //Сконструюввати та запустити об'єкт другої задачі
            Task<int> tsk2=Task<int>.Factory.StartNew(SumIt,5);
            Console.WriteLine("The Result after running of SumIt = " +
tsk2.Result);

            tsk1.Dispose();
            tsk2.Dispose();

            Console.WriteLine("Main() is done.");
            Console.ReadLine();
        }
    }
}
```

4.4. Написання асинхронних методів

Метод, відмічений ключовим словом `async`, може вмістити ключове слово `await`. Вираз `await` призводить до перетворення методу таким чином, що він призупиняється на час виконання асинхронного коду і відновлює свою роботу після його завершення. В результаті метод стає асинхронним.

1. Помічаємо метод ключовим словом `async`.

2. Далі застосовуємо ключове слово `await` для очікування виконання асинхронного коду. `await` - це унарний оператор, такий ж самий як оператор ! або оператор перетворення типу (type). Він розташовується ліворуч від виразу і означає, що потрібно дочекатися завершення асинхронного виконання цього виразу.

```
private async void button1_Click(object sender, EventArgs e)
{
     int res =await Process(100);
}
```

Розглянемо детальніше вираз `await`. Сигнатура методу `Process` має вигляд:

```
Task<int> Process(int count)
```

Тип значення, що повертається – `Task<int>`. Клас `Task` представляє асинхронну операцію та вміє викликати наш код після її завершення. Щоб скористуватися цією можливістю вручну, необхідно викликати метод `ContinueWith`, передавши йому код, що має бути виконаним після того, як довготривала операція завершиться. Саме так і поступає оператор `await`, щоб виконати решту `async`-методу.

Якщо застосувати `await` до об'єкту типу `Task<T>`, то ми отримаємо *вираз `await`*, який сам має тип `T`. Це означає, що результат оператора можна присвоїти змінній, яка використовується далі в методі, що ми і бачили у прикладі.

Проте якщо `await` застосовується до об'єкта неуніверсального класу `Task`, то виходить *оператор `await`*, значення якого нічому не можна привласнити (як і результат методу типу `void`).

Вираз `await` можна розділити на дві частини:

```
Task<int> myTask = Process(100);
int res =await myTask;
```

Важливо чітко розуміти, що при цьому відбувається. У першому рядку викликається метод `Process(100)`. Він починає виконуватися синхронно в поточному потоці і, приступивши до обчислення, повертає об'єкт `Task<int>` - все ще в поточному потоці. І лише коли ми виконуємо `await` для цього об'єкта, компілятор робить щось незвичайне. Це відноситься і до випадку, коли оператор `await` безпосередньо передує викликові асинхронного методу.

Довготривала операція починається тоді, коли йде звернення до методу `Process`, і це дозволяє дуже просто організувати одночасне виконання декількох асинхронних операцій.

```
Task<int> myTask1 = Process(100);
Task<int> myTask2 = Process(200);

int res1 =await myTask1;
int res2 =await myTask2;
```

Достатньо просто запустити їх по черзі, зберегти всі об'єкти `Task`, а потім чекати їх завершення за допомогою `await`.

4.5. Тип значення, що повертається асинхронним методом.

Метод, позначений ключовим словом `async`, може повертати значення трьох типів:

- `void`
- `Task`
- `Task<T>`, де `T` - певний тип.

Ніякі інші типи не допускаються, тому що в загальному випадку виконання асинхронного методу не завершується в момент повернення управління. Як правило, асинхронний метод чекає завершення тривалої операції, тобто управління він повертає відразу, але результат в цей момент ще не отримано і, отже, недоступний у програмі, що викликає.

У звичайних, не асинхронних, методах тип значення збігається з типом результату, тоді як в асинхронних методах вони різні - і ***це дуже суттєво***.

Метод, описаний як `async void`, можна розглядати як операцію виду «запустив і забув». Програма, що викликає, не чекає результату і не може дізнатися, коли і як операція завершується. Використовувати тип `void` слід, коли ви впевнені, що програмі, яка викликає, байдуже, коли завершиться операція і чи завершиться вона успішно.

`Async`-методи, які повертають тип `Task`, дозволяють програмі, що викликає, чекати завершення асинхронної операції та поширюють винятки, що мали місце при її виконанні. Якщо значення результату несуттєве, то метод типу `async Task` краще методу типу `async void`, тому що програма, що викликає, отримує можливість дізнатися про завершення операції, що спрощує впорядковування задач і обробку винятків.

Нарешті, `async`-методи, які повертають тип `Task<T>`, наприклад `Task<int>`, використовуються, коли асинхронна операція повертає якийсь результат.

Специфікатор async не вважається частиною сигнатури методу.

Оператор await може зустрічатися всередині блоку try, але не всередині блоків catch або finally.

4.6. Відміна асинхронних операцій.

Часто важливо мати можливість скасування паралельної операції після її запуску, наприклад, у відповідь на запит користувача. Реалізувати це найпростіше за допомогою прапорця скасування, що пов'язаний не з типом Task, а з типом CancellationToken.

Для цього потрібно, щоб такий метод, що підтримує скасування, повинен мати перевантажений варіант, в якому за звичайними параметрами слідує параметр типу CancellationToken:

```
Task<int> Process(int count, CancellationToken cancellTocken)
```

Клас CancellationTokenSource використовується для створення об'єктів CancellationToken і керування ними:

```
CancellationTokenSource cts = new CancellationTokenSource();
button2.Click += (x,y) => cts.Cancel();

label1.Text =(await Process(100, onChangeProgress,cts.Token)).ToString();
```

Виклик методу Cancel об'єкта CancellationTokenSource переводить його у стан «скасований». В асинхронному методі виконується перевірка того, чи має бути скасована операція. Якщо в методі є цикл, то таку перевірку можна проводити на кожній ітерації:

```
Task<int> Process(int count, IProgress<int> ChangeProgressBar,
CancellationToken cancellTocken)
{
      return Task.Run(() =>
      {
            int i;
            for (i = 1; i <= count; i++)
            {
                  if (cancellTocken.IsCancellationRequested)
                        return i;

                  Thread.Sleep(100);
            }
            return i;
      });
}
```

Зручною особливістю підходу до скасування, заснованого на маркерах CancellationToken, є той факт, що той самий маркер можна поширити на стільки частин асинхронної операції, на скільки необхідно, - досить просто передати його всім частинам. Неважливо, працюють вони паралельно або послідовно, йде мова про повільні обчислення або віддалені операції, - один маркер скасовує все.

4.7. Інформування про хід виконання асинхронної операції

Крім збереження можливостей відгуку інтерфейсу і надання користувачеві можливості скасувати операцію, часто буває корисно повідомляти, скільки часу залишилося до кінця повільної операції. Для цього в шаблоні використовується спеціальний типи.

Потрібно передати асинхронному методу об'єкт, який реалізує інтерфейс `IProgress<T>`, що буде використовуватися для інформування про хід роботи. Параметр типу `IProgress<T>` розміщується в кінці списку параметрів, після параметру типу `CancellationToken`.

```
Task<int> Process(int count, IProgress<int> ChangeProgressBar,
                  CancellationToken cancellTocken)
```

Щоб скористатися таким методом, ми повинні створити клас, який реалізує інтерфейс `IProgress<T>`. Потрібно або створити об'єкт, передавши його конструктору лямбда-вираз, або підписатися на подію для отримання інформації про хід виконання, яку можна буде відобразити в інтерфейсі.

```
IProgress<int> onChangeProgress = new Progress<int>((i) =>
{
     label1.Text = i.ToString();
     progressBar1.Value = i;
});
```

Якщо потрібно повідомляти про хід виконання з самого асинхронного методу, то треба лише викликати метод `Report` інтерфейсу `IProgress<T>`:

```
Task<int> Process(int count, IProgress<int> ChangeProgressBar,
CancellationToken cancellTocken)
{
     return Task.Run(() =>
     {
          int i;
          for (i = 1; i <= count; i++)
          {
               if (cancellTocken.IsCancellationRequested)
                    return i;

               ChangeProgressBar.Report(i);
               Thread.Sleep(100);
          }
          return i;
     });
}
```

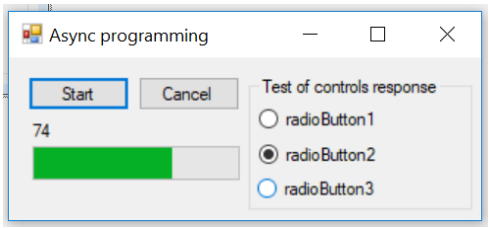
Важливо правильно вибрати параметр типу `T`. Якщо потрібно всього лише показати відсоток, то можна взяти тип `int`.

```
namespace WindowsFormsApp4
{
     public partial class Form1 : Form
     {
          public Form1()
          {
               InitializeComponent();
          }

          private async void button1_Click(object sender, EventArgs e)
```

```
        {
            IProgress<int> onChangeProgress = new Progress<int>((i) =>
            {
                label1.Text = i.ToString();
                progressBar1.Value = i;
            });

            CancellationTokenSource cts = new CancellationTokenSource();
            button2.Click += delegate { cts.Cancel(); };

            label1.Text =(await Process(100,
onChangeProgress,cts.Token)).ToString();
        }

        Task<int> Process(int count, IProgress<int> ChangeProgressBar,
CancellationToken cancellTocken)
        {
            return Task.Run(() =>
            {
                int i;
                for (i = 1; i <= count; i++)
                {
                    //label1.Text = i.ToString();
                    //progressBar1.Value = i;
                    if (cancellTocken.IsCancellationRequested)
                        return i;

                    ChangeProgressBar.Report(i);
                    Thread.Sleep(100);
                }
                return i;
            });
        }
    }
}
```

4.8. Завдання для виконання

1. Розробити програму за шаблоном Windows Form Application, у якій запускається на виконання асинхронна задача (кнопка Start) виконання довготривалої операції. Виконання асинхронної задачі організувати таким чином, щоб вона виконувалася в окремому потоці й елементи керування форми реагували в цей час на дії користувача.



2. Удосконалити розроблену програму таким чином, щоб асинхронна операція повертала результат обчислень та відображала його на елементі керування форми.

3. Доповнити функціонал розробленої програми можливістю відображати хід виконання операції (відсоток у текстовій та графічній формах) за допомогою відповідних елементів керування форми.

4. Розширити функціонал розробленої програми можливістю дочасно перервати виконання довготривалої операції (кнопка Cancel).