АСИНХРОННІ ОПЕРАЦІЇ НА БАЗІ СИНТАКСИСУ ASYNC-AWAIT

Питання 11.2. (глава 2 з книги)

Рецепт 1: призупинка на заданий період часу

- ***Завдання:*** потрібно (асинхронно) призупинити виконання програми на деякий період часу.
  - Часто потрібно при модульному тестуванні, реалізації затримки для повторного використання чи програмуванні простих тайм-аутів.
- ***Вирішення:*** тип Task містить статичний метод Delay(), який повертає задачу, яка завершується після закінчення заданого часу.
  - При імітації асинхронної операції важливо перевірити синхронний успіх, асинхронний успіх та асинхронну невдачу.
  - Приклад асинхронного успіху, який повертає задачу:

```
async Task<T> DelayResult<T>(T result, TimeSpan delay)
{
    await Task.Delay(delay);
    return result;
}
```

Рецепт 1: призупинка на заданий період часу

```
async Task<string> DownloadStringWithRetries(HttpClient client, string uri)
{
    // Retry after 1 second, then after 2 seconds, then 4.
    TimeSpan nextDelay = TimeSpan.FromSeconds(1);
    for (int i = 0; i != 3; ++i) {
        try {
            return await client.GetStringAsync(uri);
        }
        catch { }

        await Task.Delay(nextDelay);
        nextDelay = nextDelay + nextDelay;
    }

    // Try one last time, allowing the error to propagate.
    return await client.GetStringAsync(uri);
}
```

- ***Експоненційна затримка*** — стратегія збільшення затримок між повторними спробами.
  - Використовуйте її для роботи з веб-службами, щоб не перевантажувати сервер повторними спробами.
  - У реальному коді рекомендують застосувати якісніше рішення (наприклад, використовуюче бібліотеку Polly NuGet);
  - Наведений код є лише прикладом використання Task.Delay().

Рецепт 1: призупинка на заданий період часу

- Task.Delay() також можна використати для організації простого тайм-аута.
  - Зазвичай для реалізації тайм-аута використовується тип CancellationTokenSource.
  - Його можна упакувати в Task.Delay() з необмеженою затримкою, щоб надати задачу, що скасовується після завершення заданого часу.
  - Використовуйте задачу з таймером у поєднанні з Task.WhenAny() для реалізації «м'якого» тайм-аута.
- Наступний приклад повертає null, якщо служба не поверне відповідь протягом 3 секунд:

```
async Task<string> DownloadStringWithTimeout(HttpClient client, string uri)
{
    using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(3));

    Task<string> downloadTask = client.GetStringAsync(uri);
    Task timeoutTask = Task.Delay(Timeout.InfiniteTimeSpan, cts.Token);

    Task completedTask = await Task.WhenAny(downloadTask, timeoutTask);
    if (completedTask == timeoutTask)
        return null;
    return await downloadTask;
}
```

Рецепт 1: призупинка на заданий період часу

- Використання Task.Delay() для реалізації «м'якого» таймаута має свої обмеження.
  - Якщо в операції відбувається тайм-аут, вона не скасовується; у попередньому прикладі задача завантаження продовжить прийом даних і завантажить всю відповідь перед тим, як її втратити.
  - Рекомендоване рішення базується на використанні *маркера скасування (cancellation token)* як тайм-аут і передачі його операції напряму (GetStringAsync в попередньому прикладі).
  - При цьому операція може виявитись неможливою для скасування; тоді Task.Delay може використовуватись іншим кодом для імітації дій, які виконуються по тайм-ауту.
- Task.Delay доречний для модульного тестування асинхронного коду або реалізації логіки повторних спроб.
  - Проте при потребі реалізації тайм-аута, кращим кандидатом буде CancellationToken.

Рецепт 2: повернення завершених задач

- ***Завдання:*** потрібно реалізувати синхронний метод з асинхронною сигнатурою.
  - Наприклад, такая ситуація може виникнути, якщо ви наслідуєте від асинхронного інтерфейсу або базового класу, проте бажаєте реалізувати його синхронно.
  - Цей прийом особливо корисний при модульному тестуванні асинхронного кода, коли потрібна проста заглушка чи імітована реалізація для асинхронного інтерфейса.

- ***Вирішення:*** можна використати Task.FromResult для створення та повернення нового об'єкта Task<T>, уже завершеного із заданим значенням:

```
interface IMyAsyncInterface
{
    Task<int> GetValueAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
    public Task<int> GetValueAsync()
    {
        return Task.FromResult(13);
    }
}
```

Рецепт 2: повернення завершених задач

- Для методів, які не мають вихідного значення, можна використати Task.CompletedTask - кешований об’єкт успішно завершеної задачі Task:

```
interface IMyAsyncInterface
{
    Task DoSomethingAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
    public Task DoSomethingAsync()
    {
        return Task.CompletedTask;
    }
}
```

  - Task.FromResult надає завершені задачі тільки для успішних результатів.

Рецепт 2: повернення завершених задач

```
Task<T> NotImplementedAsync<T>()
{
    return Task.FromException<T>(new NotImplementedException());
}

Task<int> GetValueAsync(CancellationToken cancellationToken)
{
    if (cancellationToken.IsCancellationRequested)
        return Task.FromCanceled<int>(cancellationToken);
    return Task.FromResult(13);
}
```

- Якщо потрібна задача з іншим типом результату (наприклад, задача, завершена з NotImplementedException), ви можете використати Task.FromException.
  - Аналогічно існує метод Task.FromCanceled для створення задач, уже скасованих із заданого маркера CancellationToken.

Рецепт 2: повернення завершених задач

```
static void DoSomethingSynchronously() { }

interface IMyAsyncInterface {
    Task DoSomethingAsync();
}

class MySynchronousImplementation : IMyAsyncInterface {
    public Task DoSomethingAsync() {
        try {
            DoSomethingSynchronously();
            return Task.CompletedTask;
        }
        catch (Exception ex) {
            return Task.FromException(ex);
        }
    }
}
```

- Якщо в синхронній реалізації може відбутись відмова, перехоплюйте винятки та використовуйте Task.FromException для їх повернення.
- Якщо ви реалізуєте асинхронний інтерфейс синхронним кодом, уникайте будь-яких форм блокування.
  - Уникайте блокування з наступним поверненням завершеної задачі в асинхронному методі, якщо метод може бути реалізовано асинхронно.
- Контрприклад: засоби зчитування тексту з Console у .NET BCL.
  - Console.In.ReadLineAsync() *блокує викликаючий потік, поки не буде прочитано рядок*, після чого повертає завершену задачу.
  - Така поведінка не інтуїтивна та приносить несподіванки багатьом розробникам.
  - Якщо асинхронний метод блокується, він не дозволяє викликаючому потоку запускати інші задачі, що протирічить ідеї конкурентності та може призвести до взаємоблокування.

Рецепт 2: повернення завершених задач

- Якщо ви регулярно використовуєте Task.FromResult з одним значенням, подумайте про кешування задачі.
  - Наприклад, одного разу створивши Task<int> з нульовим результатом, уникайте створення інших екземплярів, що повинні будуть знищуватись під час збирання сміття:

```
private static readonly Task<int> zeroTask = Task.FromResult(0);
Task<int> GetValueAsync()
{
    return zeroTask;
}
```

  - На логічному рівні Task.FromResult, Task.FromException і Task.FromCanceled є допоміжними методами і скороченими формами узагальненого типу TaskCompletionSource<T>.
  - TaskCompletionSource<T> є низькорівневим типом, корисним для взаємодії з іншими формами асинхронного коду.
  - Загалом слід застосовувати скорочену форму Task.FromResult і схожі формы, якщо потрібно повернути вже завершену задачу.
  - Використовуйте TaskCompletionSource<T> для повернення задачі, що завершується в деякий момент у майбутньому.

Рецепт 3: передача інформації щодо ходу виконання операції

- ***Завдання:*** потрібно відреагувати на прогрес виконання операції.
- ***Вирішення:*** використовуйте типи IProgress<T> і Progress<T>.
  - Ваш async-метод повинен отримувати аргумент IProgress<T>; тут T — тип прогресу, про який слід сповіщати:

```
async Task MyMethodAsync(IProgress<double> progress = null) {
    bool done = false;
    double percentComplete = 0;
    while (!done) {
        // ...
        progress?.Report(percentComplete);
    }
}
```

  - За діючими угодами параметр IProgress<T> може бути рівним null, якщо викликаючій стороні не потрібні сповіщення щодо прогресу; включіть відповідну перевірку в свій async-метод.
  - Метод IProgress<T>.Report зазвичай є асинхронним, тому MyMethodAsync() може продовжити виконання перед повідомленням про прогрес.
  - Краще визначити T як незмінюваний тип (або принаймні тип-значення). Якщо T є змінюваним посилальним типом, доведеться самостійно створювати окрему копію при кожному виклику IProgress<T>.Report.

Пример использования в вызывающем коде

```
async Task CallMyMethodAsync()
{
    var progress = new Progress<double>();
    progress.ProgressChanged += (sender, args) =>
    {
    // ...
    };
    await MyMethodAsync(progress);
}
```

- Progress<T> зберігає поточний контекст при створенні та активізує свій зворотний виклик у цьому контексті.
  - Якщо Progress<T> конструюється в UI-потоці, ви сможете оновити користувацький інтерфейс з його зворотного виклику, навіть якщо асинхронний метод викликає Report із фонового потоку.
- Якщо метод підтримує сповіщення щодо прогресу, він також повинен підтримувати скасування (тема 13).
  - IProgress<T> не обмежується одним асинхронним кодом; як прогрес, так і скасування також можуть (і повинні) використовуватись у довгостроковому синхронному коді.

Рецепт 4: очікування завершення групи задач

- ***Завдання:*** маємо кілька задач, і потрібно почекати, поки вони всі закінчаться.
- ***Вирішення:*** фреймворк надає для цього метод Task.WhenAll().
  - Метод отримує кілька задач і повертає задачу, що завершується при завершенні всіх указаних задач.

```
Task task1 = Task.Delay(TimeSpan.FromSeconds(1));
Task task2 = Task.Delay(TimeSpan.FromSeconds(2));
Task task3 = Task.Delay(TimeSpan.FromSeconds(1));

await Task.WhenAll(task1, task2, task3);
```

  - Якщо всі задачі мають однаковий тип результату і успішно завершуються, то задача Task.WhenAll повертає масив, що містить результати всіх задач:

```
Task<int> task1 = Task.FromResult(3);
Task<int> task2 = Task.FromResult(5);
Task<int> task3 = Task.FromResult(7);

int[] results = await Task.WhenAll(task1, task2, task3);

// "results" contains { 3, 5, 7 }
```

Рецепт 4: очікування завершення групи задач

```
async Task<string> DownloadAllAsync(HttpClient client,
                                    IEnumerable<string> urls)
{
    // Define the action to do for each URL.
    var downloads = urls.Select(url =>
                            client.GetStringAsync(url));
    // Note that no tasks have actually started yet
    //  because the sequence is not evaluated.

    // Start all URLs downloading simultaneously.
    Task<string>[] downloadTasks = downloads.ToArray();
    // Now the tasks have all started.

    // Asynchronously wait for all downloads to complete.
    string[] htmlPages = await Task.WhenAll(downloadTasks);

    return string.Concat(htmlPages);
}
```

- Існує перевантажена версія Task.WhenAll, яка отримує IEnumerable з задачами, проте використовувати її не рекомендується.

```
async Task ThrowNotImplementedExceptionAsync() {
    throw new NotImplementedException();
}

async Task ThrowInvalidOperationExceptionAsync() {
    throw new InvalidOperationException();
}

async Task ObserveOneExceptionAsync() {
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    try {
        await Task.WhenAll(task1, task2);
    }
    catch (Exception ex) {
        // "ex" is either NotImplementedException or InvalidOperationException
        // ...
    }
}

async Task ObserveAllExceptionsAsync() {
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    Task allTasks = Task.WhenAll(task1, task2);
    try {
        await allTasks;
    }
    catch {
        AggregateException allExceptions = allTasks.Exception;
        // ...
    }
}
```

Рецепт 4: очікування завершення групи задач

Якщо деякі задачі викидають винятки, то Task.WhenAll повідомляє про відмову своєї поверненої задачі з цим винятком.

- Якщо відразу кілька задач викидають виняток, то ці винятки поміщаються в задачу Task, яку повертає Task.WhenAll.
- Проте при очікуванні цієї задачі буде викинуто тільки один з них.
- Якщо потрібний кожний конкретний виняток, перевірте властивість Exception задачі Task, яку повертає Task.WhenAll.
- Зазвичай достатньо відреагувати тільки на першу викинуту помилку, а не на всі.

Рецепт 5: очікування завершення довільної задачі

- ***Завдання:*** маємо кілька задач і потрібно відреагувати на завершення довільної задачі з групи.
  - Часто задача зустрічається при виконанні кількох незалежних спроб виконання операції в структурі «першому дістається все». Наприклад, можна запитати біржеві котирування у кількох веб-служб одночасно, проте вас цікавить тільки перша відповідь.
- ***Вирішення:*** використовуйте метод Task.WhenAny.
  - Метод Task.WhenAny отримує послідовність задач і повертає задачу, яка завершується при завершенні будь-якої із задач послідовності.
  - Задача, повернена Task.WhenAny, ніколи не завершується в стані відмови чи скасування.
  - Ця «зовнішня» задача завжди завершується успішно, а її результуюче значення представляє собою першу завершену задачу Task («внутрішню»).
  - Якщо внутрішня задача завершилась з винятком, то він не розповсюджується на зовнішню задачу (повернену Task.WhenAny).
  - Зазвичай ваш код очікує внутрішню задачу за допомогою await, щоб забезпечити відстеження всіх винятків.

Рецепт 5: очікування завершення довільної задачі

```
// Returns the length of data at the first URL to respond.
async Task<int> FirstRespondingUrlAsync(HttpClient client,
    string urlA, string urlB)
{
    // Start both downloads concurrently.
    Task<byte[]> downloadTaskA = client.GetByteArrayAsync(urlA);
    Task<byte[]> downloadTaskB = client.GetByteArrayAsync(urlB);

    // Wait for either of the tasks to complete.
    Task<byte[]> completedTask =
        await Task.WhenAny(downloadTaskA, downloadTaskB);

    // Return the length of the data retrieved from that URL.
    byte[] data = await completedTask;
    return data.Length;
}
```

- Коли перша задача завершується, подумайте, чи потрібно скасувати решту задач.
  - Якщо інші задачі не скасовуються, проте до них не застосовується await, вони просто втрачаються (відпраьовують до завершення, але їх результати ігноруються.
  - Всі винятки від втрачених задач також будуть проігноровані.
  - Якщо ці задачі не будуть скасовані, вони продовжать працювати й неефективно витрачати ресурси (підключення HTTP, підключення до БД, таймеры і т. д.).
- Можна, проте не рекомендується, використовувати Task.WhenAny для реалізації тайм-аута (наприклад, при використанні Task.Delay як однієї з задач).
  - Природніше виражати тайм-аути скасуванням.
  - Додаткова перевага – дійсне скасування операцій у випадку тайм-аута.
- Інший антипаттерн Task.WhenAny — обробка задач по мірі їх завершення.
  - Вести список задач і видаляти кожну задачу зі списку при завершенні неефективно (час такої роботи $O(N^2)$, хоч існує алгоритм з часом $O(N)$).

Рецепт 6: обробка задач при завершенні

```
async Task<int> DelayAndReturnAsync(int value) {
    await Task.Delay(TimeSpan.FromSeconds(value));
    return value;
}

// Currently, this method prints "2", "3", and "1".
// The desired behavior is for this method to print
// "1", "2", and "3".
async Task ProcessTasksAsync() {
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    Task<int>[] tasks = new[] { taskA, taskB, taskC };

    // Await each task in order.
    foreach (Task<int> task in tasks) {
        var result = await task;
        Trace.WriteLine(result);
    }
}
```

- ***Завдання:*** маємо колекцію задач, які будуть використовуватись з await; потрібно організувати обробку кожної задачі після її завершення.
  - При цьому обробка кожної задачі повинна відбуватитсь відразу після завершення, без очікування інших задач.
- Приклад запускає 3 відкладені задачі, а потім очікує кожну з них.
  - Тут код очікує кожну задачу в порядку послідовності, хоч третя задача в послідовності завершується першою.
  - Код повинен здійснювати обробку (наприклад, Trace.WriteLine) при завершенні кожної задачі, не чекаючи завершення інших.

Рецепт 6: обробка задач при завершенні

```
async Task<int> DelayAndReturnAsync(int value)
{
    await Task.Delay(TimeSpan.FromSeconds(value));
    return value;
}

async Task AwaitAndProcessAsync(Task<int> task)
{
    int result = await task;
    Trace.WriteLine(result);
}

// This method now prints "1", "2", and "3".
async Task ProcessTasksAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    Task<int>[] tasks = new[] { taskA, taskB, taskC };

    IEnumerable<Task> taskQuery =
        from t in tasks select AwaitAndProcessAsync(t);
    Task[] processingTasks = taskQuery.ToArray();

    // Await all processing to complete
    await Task.WhenAll(processingTasks);
}
```

- Найпростіше рішення – рефакторинг коду та введення високорівневого async-метода, який забезпечує очікування задачі та обробку її результату.
  - Винесення обробки в окремий метод суттєво спрощує код.
  - З іншого боку, замість методу AwaitAndProcessAsync() можна використати іншу форму запису LINQ-запиту:

```
Task<int>[] tasks = new[] { taskA, taskB, taskC };

Task[] processingTasks = tasks.Select(async t =>
{
    var result = await t;
    Trace.WriteLine(result);
}).ToArray();

// Await all processing to complete
await Task.WhenAll(processingTasks);
```

Рецепт 6: обробка задач при завершенні

```
async Task<int> DelayAndReturnAsync(int value)
{
    await Task.Delay(TimeSpan.FromSeconds(value));
    return value;
}

// This method now prints "1", "2", and "3".
async Task UseOrderByCompletionAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    Task<int>[] tasks = new[] { taskA, taskB, taskC };

    // Await each one as they complete.
    foreach (Task<int> task in tasks.OrderByCompletion())
    {
        int result = await task;
        Trace.WriteLine(result);
    }
}
```

- Альтернативи рефакторингу пропонують Стівен Тауб (Stephen Toub) і Джон Скіт (Jon Skeet) – методи розширення, які повертають масив задач, які завершуються по порядку.
  - Рішення Стівена Тауба - Parallel Programming with .NET,
  - Рішення Джона Скіта – в його блозі.
  - Метод розширення OrderByCompletion також доступний у бібліотеці з відкритим кодом AsyncEx (NuGet-пакет Nito.AsyncEx)

Рецепт 7: обходження контексту при продовженні

- ***Завдання:*** коли async-метод відновлює роботу після await, за умовчанням він продовжує виконання в тому ж контексті.
  - Це може створювати проблеми з швидкодією, якщо контекстом був UI-контекст, а в UI-контексті відновлює роботу велика кількість async-методів.
- ***Вирішення:*** щоб уникнути відновлення в контексті, використовуйте await для результата ConfigureAwait і передайте false у параметрі continueOnCapturedContext.

```
async Task ResumeOnContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    // This method resumes within the same context.
}

async Task ResumeWithoutContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);
    // This method discards its context when it resumes.
}
```

Скільки продовжень у UI-потоці перевищує допустимий поріг?

- Однозначної відповіді на питання немає.
  - Люциан Віщик з Microsoft (команда Universal Windows) рекомендує: близько сотні за секунду — нормально, проте близько тисячі за секунду — вже занадто багато.
- Краще обійти проблему з самого початку.
  - Для кожного написаного вами async-метода, якщо він не повинен відновлюватись у первинному контексті, використовуйте ConfigureAwait.
- Також варто враховувати контекст при написанні async-коду.
  - Зазвичай async-метод повинен або вимагати певний контекст (робота з UI-елементами або запитами / відгуками ASP.NET), або бути вільним від контекста (виконуючи фонові операції).
  - Якщо у вас є async-метод з частинами, що потребують контексту, та вільними від контексту частинами, краще розбити його на два або більше async-методи.

Рецепт 8: обробка винятків із методів async Task

- Винятки можна перехоплювати простою конструкцією try/catch, як і для синхронного коду:

```
async Task ThrowExceptionAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    throw new InvalidOperationException("Test");
}

async Task TestAsync()
{
    try
    {
        await ThrowExceptionAsync();
    }
    catch (InvalidOperationException)
    {
    }
}
```

Рецепт 8: обробка винятків із методів async Task

```
async Task ThrowExceptionAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    throw new InvalidOperationException("Test");
}

async Task TestAsync()
{
    // The exception is thrown by the method and placed on the task.
    Task task = ThrowExceptionAsync();
    try
    {
        // The exception is re-raised here, where the task is awaited.
        await task;
    }
    catch (InvalidOperationException)
    {
        // The exception is correctly caught here.
    }
}
```

- Винятки, викинуті з методів async Task, поміщаються в вихідний об’єкт Task.
  - Вони викидаються тільки при використанні await з вихідним об’єктом Task.
  - Викинутий в методі async Task виняток зберігається та включається в вихідний об’єкт Task.
  - Оскільки методи async void не мають об’єкта Task для розміщення винятку, для них використовується інша поведінка (рецепт 9).
- При використанні await з задачою Task, у якій відбулась відмова, перший виняток цієї задачі видається повторно.
  - При повторному викиданні винятків можуть виникнути питання щодо трасувань стеку. Початкове трасування стеку буде правильно збережене.
  - У більшості випадків код повинен розповсюджувати винятки з асинхронних методів, які викликаються;
  - потрібно використати await з задачою, поверненою з асинхронного методу, і цей виняток буде розповсюджуватись природним чином.

Рецепт 9: обробка винятків із методів async void

```
sealed class MyAsyncCommand : ICommand
{
    async void ICommand.Execute(object parameter)
    {
        await Execute(parameter);
    }

    public async Task Execute(object parameter)
    {
        // ... // Asynchronous command implementation goes here.
    }

    // ... // Other members (CanExecute, etc)
    public bool CanExecute(object parameter)
    {
        CanExecuteChanged?.Invoke(null, null);
        throw new NotImplementedException();
    }
    public event EventHandler CanExecuteChanged;
}
```

- ***Завдання:*** маємо метод async void. Потрібно обробити винятки, розповсюджені з цього методу.
- ***Вирішення:*** Хорошого рішення не існує.
  - За можливості змініть метод так, щоб він повертав Task замість void.
  - Інколи це неможливо; наприклад, потрібно провести модульне тестування реалізації ICommand (яка повинна повертати void).
  - Загалом необхідно надати перевантажену версію вашого методу Execute(), яка повертає Task.
- Краще уникати розповсюдження винятків з методів async void.
  - Якщо іншого виходу немає, краще спакувати весь код у блок try з прямою обробкою винятків.

Рецепт 9: обробка винятків із методів async void

- Інший спосіб: коли метод async void розповсюджує виняток, він викидається в контексті SynchronizationContext, активному на момент початку виконання методу async void.
  - Якщо середовище виконання надає SynchronizationContext, то зазвичай воно надає механізм обробки цих високорівневих винятків на глобальному рівні.
  - Наприклад, WPF надає Application.DispatcherUnhandledException, Universal Windows – Application.UnhandledException, а ASP.NET — UseExceptionHandler.
- Також можливо обробляти винятки з методів async void, керуючи SynchronizationContext.
  - Написати власний варіант SynchronizationContext непросто, проте можна скористатись типом AsyncContext з безкоштовної NuGet-бібліотеки Nito.AsyncEx.
  - Тип AsyncContext особливо корисний для додатків, які не мають вбудованого об’єкта SynchronizationContext (наприклад, консольних додатків і служб Win32).

Рецепт 9: обробка винятків із методів async void

```
static class Program
{
    static void Main(string[] args)
    {
        try
        {
            AsyncContext.Run(() => MainAsync(args));
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine(ex);
        }
    }

    // BAD CODE!!!
    // In the real world, do not use async void unless you have to.
    static async void MainAsync(string[] args)
    {
        // ...
    }
}
```

- Тут AsyncContext використовується для запуску й обробки винятків з метода async void.
  - Одна з причин, чому слід віддавати перевагу async Task методам – їх простіше тестування.
- Якщо потрібно представити *власний* тип SynchronizationContext (наприклад, AsyncContext), не встановлюйте його в потоках, які вам не належать.
  - Як правило, цей тип не повинен розміщуватись у потоках, у яких він вже є (наприклад, UI-потоках або класичних потоках запитів ASP. NET);
  - Також не варто розміщати SynchronizationContext в потоках із пулу потоків.
  - Головний потік консольного додатку належить вам, як і всі потоки, які ви самостійно створюєте вручну.

Рецепт 10: створення ValueTask

- ***Завдання:*** потрібно створити метод, який повертатиме ValueTask<T>.
- ***Вирішення:*** ValueTask<T> використовується як вихідний тип у ситуаціях, в яких зазвичай може бути повернений синхронний результат, а асинхронна поведінка зустрічається рідше.
  - Загалом у коді додатку слід використати в якості вихідного типу Task<T>, а не ValueTask<T>.
  - Розглядати використання ValueTask<T> як вихідний тип слід лише після профілювання, яке показує можливе підвищення швидкодії.
- Проте можливі ситуації, коли потрібно реалізувати метод, що повертає ValueTask<T>.
  - Зокрема при використанні інтерфейса IAsyncDisposable, метод DisposeAsync() якого повертає ValueTask.
  - Найпростіший спосіб реалізації метода, що повертає ValueTask<T>, базується на використанні async-await:

```
public async ValueTask<int> MethodAsync()
{
    await Task.Delay(100); // asynchronous work.
    return 13;
}
```

Рецепт 10: створення ValueTask

- Часто метод, який повертає ValueTask<T>, здатний негайно повернути значення;
  - Тоді можна застосувати оптимізацію для цього сценарія з використанням конструктора ValueTask<T>, а потім передавати керування повільному асинхронному методу тільки при необхідності:

```
bool CanBehaveSynchronously;

public ValueTask<int> MethodAsync()
{
    if (CanBehaveSynchronously)
        return new ValueTask<int>(13);
    return new ValueTask<int>(SlowMethodAsync());
}

private Task<int> SlowMethodAsync() => Task.FromResult(13);
```

  - Аналогічний підхід можливий для ValueTask без параметризації.
  - Тут конструктор за умовчанням ValueTask використовується для повернення успішно завершеного об'єкта ValueTask.

Рецепт 10: створення ValueTask

```
private Func<Task> _disposeLogic;

public ValueTask DisposeAsync()
{
    if (_disposeLogic == null)
        return default;

    // Note: this simple example is not threadsafe;
    //  if multiple threads call DisposeAsync,
    //  the logic could run more than once.
    Func<Task> logic = _disposeLogic;
    _disposeLogic = null;
    return new ValueTask(logic());
}
```

- Показано реалізацію IAsyncDisposable, яка виконує свою логіку асинхронного вивільнення одноразово;
  - при майбутніх викликах метод DisposeAsync() завершується успішно та синхронно.
- Більшість методів повинні повертати Task<T>, оскільки при споживанні Task<T> виникає менше прихованих пасток, ніж при споживанні ValueTask<T>.
  - Частіше при реалізації інтерфейсів, що використовують ValueTask чи ValueTask<T>, можна просто застосовувати async та await.
- Складніші реалізації потрібні, коли ви збираєтесь використовувати ValueTask<T> самостійно.
  - Рецепт показує більш прості та розповсюджені підходи до створення екземплярів ValueTask<T> і ValueTask.
  - У більш складному випадку ви кешуєте або поміщаєте в пул реалізацію IValueTaskSource<T> і повторно використовуєте її для багатьох викликів асинхронних методів (тип ManualResetValueTaskSourceCore<TResult>).

Рецепт 11: споживання ValueTask

- Найпростіший та прямолінійний спосіб споживання ValueTask<T> або ValueTask базується на await.
  - У більшості випадків вам необхідно зробити таке:

```
ValueTask<int> MethodAsync() => new ValueTask<int>(13);

async Task ConsumingMethodAsync() {
    int value = await MethodAsync();
}
```

  - Також можна виконати await після виконання конкурентної операції, як у випадку з Task<T>:

```
ValueTask<int> MethodAsync() => new ValueTask<int>(13);

async Task ConsumingMethodAsync() {
    ValueTask<int> valueTask = MethodAsync();
    // ... // other concurrent work
    int value = await valueTask;
}
```

  - Обидва варіанти доречні, тому що ValueTask очікується лише 1 раз (обмеження ValueTask).

Рецепт 11: споживання ValueTask

- Щоб зробить щось складніше, перетворіть ValueTask<T> у Task<T> викликом AsTask:

```
ValueTask<int> MethodAsync() => new ValueTask<int>(13);

async Task ConsumingMethodAsync()
{
    Task<int> task = MethodAsync().AsTask();
    // ... // other concurrent work
    int value = await task;
    int anotherValue = await task;
}
```

- Багатократне очікування Task<T> абсолютно безпечне.
  - Доступні інші операції, наприклад, асинхронне очікування завершення кількох операцій:

```
ValueTask<int> MethodAsync() => new ValueTask<int>(13);

async Task ConsumingMethodAsync()
{
    Task<int> task1 = MethodAsync().AsTask();
    Task<int> task2 = MethodAsync().AsTask();
    int[] results = await Task.WhenAll(task1, task2);
}
```

Рецепт 11: споживання ValueTask

- Проте для кожного ValueTask<T> можна викликати AsTask() тільки один раз.
  - Найрозповсюдженіше рішення: негайно перетворити його в Task<T>, а далі ігнорувати ValueTask<T>.
  - Також не можна одночасно використовувати await і викликати AsTask для одного ValueTask<T>.
- У більшості програм слід або негайно виконати await для ValueTask<T>, або перетворити значення в Task<T>.
- Інші властивості ValueTask<T> призначені для нетривіального використання.
  - Зазвичай вони працюють не так, як інші відомі властивості; зокрема, для ValueTask<T>.Result діють жорсткіші обмеження, ніж для Task<T>.Result.
  - Код, який синхронно отримує результат від ValueTask<T>, може викликати ValueTask<T>.Result або ValueTask<T>.GetAwaiter().GetResult(), проте ці компоненти не повинні викликатись до завершення ValueTask<T>.
  - Синхронне завантаження результату з Task<T> блокує викликаючий потік до завершення задачі; ValueTask<T> таких гарантій не дає.

Модульне тестування async-методів

```
class TestMethodAttribute : Attribute { }

class Sut {
    public Task<bool> MyMethodAsync() => Task.FromResult(true);
    public async void MyVoidMethodAsync() { }
}

class ch07r01A
{
    [TestMethod]
    public async Task MyMethodAsync_ReturnsFalse()
    {
        var objectUnderTest = new Sut(); // ...;
        bool result = await objectUnderTest.MyMethodAsync();
        Assert.IsFalse(result);
    }
}
```

- ***Завдання:*** маємо async-метод, для якого потрібно провести модульне тестування.
- ***Вирішення:*** більшість сучаних фреймворків модульного тестування (MSTest, NUnit, xUnit та ін.) підтримують методи модульного тестування async Task.
  - Наведено приклад async-модульного теста в MSTest.
  - Фреймворк поміщає, що метод має вихідний тип Task, і очікує завершення задачі перед тим, як зробити помітку щодо проходження чи відмови тесту.

Модульне тестування async-методів

- Якщо ваш фреймворк модульного тестування не підтримує модульні тести async Task, то йому додведеться допомогти з очікуванням тестованої асинхронної операції.
  - Один з варіантов — використати GetAwaiter().GetResult() для синхронного блокування по задачі; якщо після цього застосувати GetAwaiter().GetResult() замість Wait(), це дозволить уникнути обгортання AggregateException, коли в задачі виникне виняток.
  - Інший спосіб – тип AsyncContext з NuGet-пакета Nito.AsyncEx:

```
[TestMethod]
public void MyMethodAsync_ReturnsFalse()
{
    AsyncContext.Run(async () =>
    {
        var objectUnderTest = new Sut(); // ...;
        bool result = await objectUnderTest.MyMethodAsync();
        Assert.IsFalse(result);
    });
}
```

  - AsyncContext.Run() очікує завершення всіх асинхронних методів.
  - Імітація (mocking) асинхронних залежностей здається дещо незграбною: завжди бажано перевірити, як ваші методи реагують на синхронний успіх (імітація з Task.FromResult), синхронні помилки (імітація з Task.FromException) та асинхронний успіх (імітація з Task.Yield і вихідним значенням).

Модульне тестування async-методів

```
interface IMyInterface
{
    Task<int> SomethingAsync();
}

class SynchronousSuccess : IMyInterface
{
    public Task<int> SomethingAsync()
    {
        return Task.FromResult(13);
    }
}

class SynchronousError : IMyInterface
{
    public Task<int> SomethingAsync()
    {
        return Task.FromException<int>(new InvalidOperationException());
    }
}

class AsynchronousSuccess : IMyInterface
{
    public async Task<int> SomethingAsync()
    {
        await Task.Yield(); // force asynchronous behavior
        return 13;
    }
}
```

- Task.FromResult і Task.FromException розглядались у рецепті 2.
  - Task.Yield може використовуватись для примусового застосування асинхронної поведінки та задіюється переважно при модульному тестуванні.
- При тестуванні асинхронного коду взаємоблокування та стани гонитви можуть проявлятись частіше, ніж при тестуванні синхронного коду.
  - Може бути корисним призначення тайм-аута на рівні тестів; у Visual Studio можна додати в рішення файл тестових налаштувань, у якому задавати тайм-аути для окремих тестів.
  - Значення за умовчанням досить велике, оптимальним може бути двохсекундний тайм-аут на рівні тестів.

Модульне тестування методів async void

- ***Завдання:*** маємо метод async void, для якого необхідно написати модульні тести.
- ***Вирішення:*** Такої ситуації слід уникати! Якщо метод async void можна перетворити в метод async Task – зробіть це!
  - Якщо ваш метод зобов'язаний бути методом asyn void (наприклад, для відповідності сигнатурі методу інтерфейса), розгляньте можливість написання 2 методів: метода async Task, який містить усю логіку, та обгортки async void, яка просто викликає метод async Task та очікує результат.
  - Метод async void задовольняє вимогам архітектури, а метод async Task (з усією логікою) придатний для тестування.
- Якщо змінити метод неможливо, модульне тестування метода async void все ж можливе.
  - Використовуйте клас AsyncContext з бібліотеки Nito.AsyncEx:

```
// Not a recommended solution
[TestMethod]
public void MyMethodAsync_DoesNotThrow()
{
    AsyncContext.Run(() => {
        var objectUnderTest = new Sut(); // ...;
        objectUnderTest.MyVoidMethodAsync();
    });
}
```

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: Асинхронні потоки