



# АСИНХРОННЕ ВИКОНАННЯ КОДУ

Лекція 11  
Об'єктно-орієнтоване програмування

# План лекції

---

- Загальні відомості про конкурентне виконання коду.
- Асинхронні операції на базі синтаксису `async-await`.
- Асинхронні потоки.





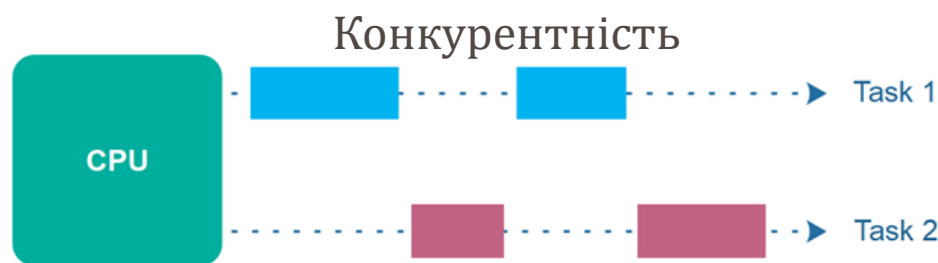
# ЗАГАЛЬНІ ВІДОМОСТІ ПРО КОНКУРЕНТНЕ ВИКОНАННЯ КОДУ

Питання 11.1. (глава 1 з книги)

# Поняття конкурентності

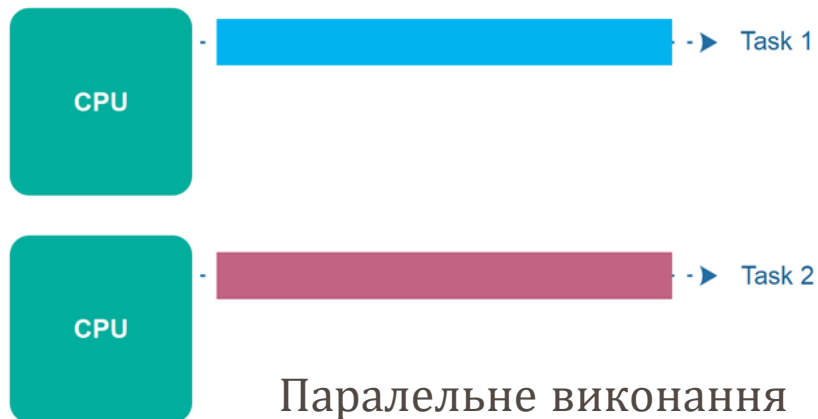
---

- **Конкурентність** – виконання відразу кількох дій одночасно.
  - Додатки використовують конкурентність, наприклад, щоб реагувати на ввід даних користувачем під час запису в БД.
  - Серверні додатки застосовують конкурентність для реакції на другий запит у ході завершення першого запиту.
- **Багатопоточність** – форма конкурентності, яка використовує кілька програмних потоків виконання.
  - Відноситься до буквального використання декількох потоків і є лише різновидом конкурентності.
  - Безпосереднє використання низькорівневих видів багатопоточності в сучасних додатках практично не має сенсу; високорівневі абстракції перевершують багатопотокові засоби старої школи як за потужністю, так і за ефективністю.
  - Багатопоточність продовжує жити в пулах потоків – корисному місці для постановки робочих операцій в чергу, яка автоматично регулюється в залежності від навантаження.

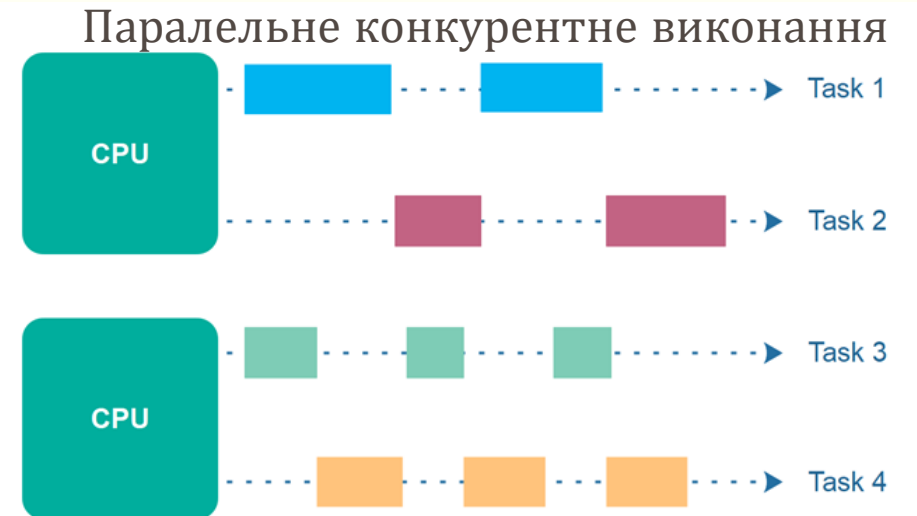


# Поняття конкурентності

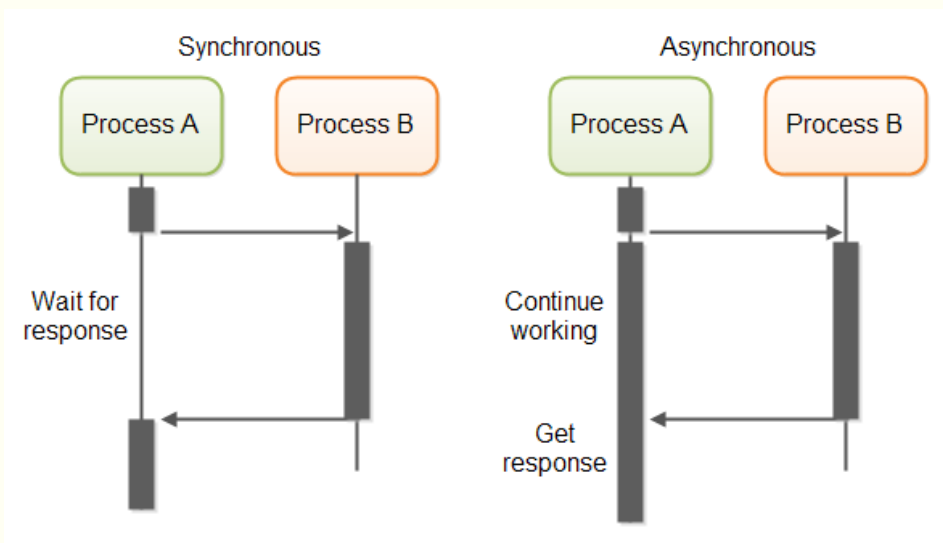
- Маючи пул потоків, стає можливо працювати з іншою формою конкурентності: **паралельною обробкою** – виконанням великого об'єму роботи за рахунок її розподілу між кількома потоками, що виконуються одночасно.
  - Паралельна обробка (або паралельне програмування) використовує багатопоточність для максимально ефективного використання багатоядерних процесорів: нерозумно доручати всю роботу одному ядру, в той час як інші простоюють.
  - Паралельна обробка розподіляє роботу між кількома потоками, кожен з яких може виконуватися незалежно на окремому ядрі.
  - Паралельна обробка є однією з різновидів багатопоточності, а багатопоточність є різновидом конкурентності.



@Марченко С.В., ЧДБК, 2021



# Поняття конкурентності



■ **Асинхронне програмування** – різновид конкурентності, що використовує обіцянки (промиси) або зворотні виклики для уникнення створення зайвих потоків.

- **Обіцянка (*future / promise*)**, або заздалегідь намічений тип – тип, який представляє деяку операцію, що завершиться в майбутньому. Приклади сучасних типів обіцянок в .NET – Task і Task<TResult>.
- Старіші асинхронні API використовують зворотні виклики або події замість обіцянок.
- Центральне місце займає ідея асинхронної операції – деякої запущеної операції, яка завершиться через деякий час.
- Хоча операція триває, вона не блокує вихідний потік; потік, який запустив операцію, вільний для виконання іншої роботи.
- Коли операція завершиться, вона сповіщує свою обіцянку або активізує зворотний виклик чи подію, щоб додаток дізнався про завершення.
- Завдяки підтримці `async` та `await` у сучасних МП асинхронне програмування стає майже таким же простим, як і синхронне (неконкурентне) програмування.

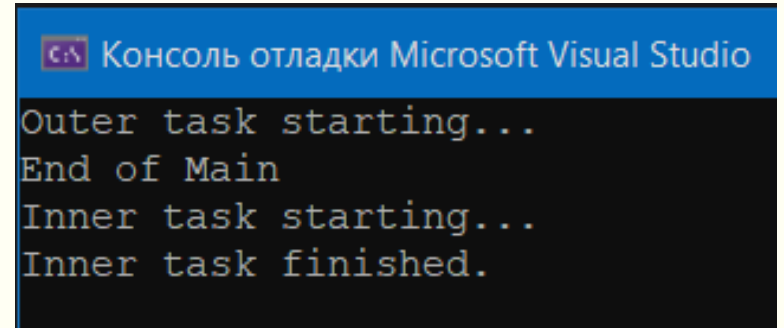
# Робота з класом Task

---

```
static void Main(string[] args)
{
    var outer = Task.Factory.StartNew(() =>           // зовнішня задача
    {
        Console.WriteLine("Outer task starting...");
        var inner = Task.Factory.StartNew(() =>      // вкладена задача
        {
            Console.WriteLine("Inner task starting...");
            Thread.Sleep(2000);
            Console.WriteLine("Inner task finished.");
        });
    });
    outer.Wait(); // очікуємо виконання зовнішньої задачі
    Console.WriteLine("End of Main");

    Console.ReadLine();
}
```

- Одна задача може запускати іншу – вкладену задачу.
  - При цьому ці задачі виконуються незалежно одна від одної.
  - Незважаючи на те, що тут ми очікуємо виконання зовнішньої задачі, вкладена задача може завершити виконання навіть після завершення методу Main:



Консоль отладки Microsoft Visual Studio

```
Outer task starting...
End of Main
Inner task starting...
Inner task finished.
```

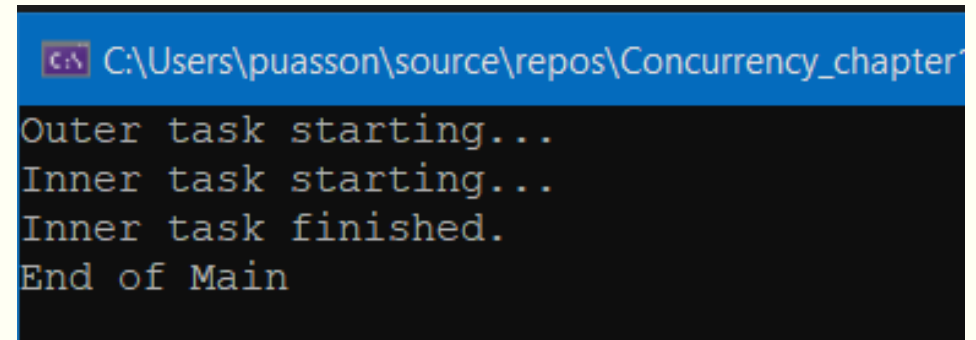
# Робота з класом Task

---

- Якщо потрібно, щоб вкладена задача вивконувалась разом із зовнішньою, необхідно використовувати значення `TaskCreationOptions.AttachedToParent`.

```
var outer = Task.Factory.StartNew(() =>           // зовнішня задача
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>        // вкладена задача
    {
        Console.WriteLine("Inner task starting...");
        Thread.Sleep(2000);
        Console.WriteLine("Inner task finished.");
    }, TaskCreationOptions.AttachedToParent);
});
outer.Wait(); // очікуємо виконання зовнішньої задачі
Console.WriteLine("End of Main");

Console.ReadLine();
```



```
C:\Users\puasson\source\repos\Concurrency_chapter
Outer task starting...
Inner task starting...
Inner task finished.
End of Main
```



# Робота з класом Task. Масив задач

---

- Можна визначити всі задачі в масиві безпосередньо через об'єкт Task:

```
Task[] tasks1 = new Task[3]
{
    new Task(() => Console.WriteLine("First Task")),
    new Task(() => Console.WriteLine("Second Task")),
    new Task(() => Console.WriteLine("Third Task"))
};

// запуск задач у масиві
foreach (var t in tasks1)
    t.Start();
```

- Або також можна використовувати методи Task.Factory.StartNew або Task.Run і відразу запускати всі задачі:

```
Task[] tasks2 = new Task[3];
int j = 1;
for (int i = 0; i < tasks2.Length; i++)
    tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));
```

- Але в будь-якому випадку знову ж можемо зіткнутися з тим, що всі завдання з масиву можуть завершитися після того, як відпрацює метод Main, в якому запускаються ці задачі.

# Робота з класом Task. Масив задач

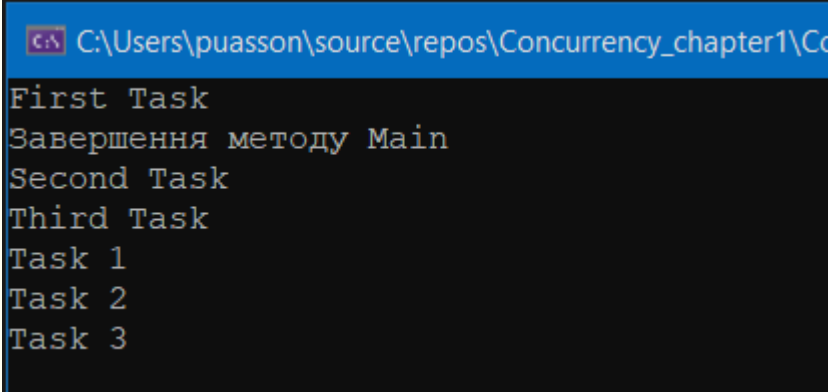
---

```
Task[] tasks1 = new Task[3]
{
    new Task(() => Console.WriteLine("First Task")),
    new Task(() => Console.WriteLine("Second Task")),
    new Task(() => Console.WriteLine("Third Task"))
};
foreach (var t in tasks1)
    t.Start();

Task[] tasks2 = new Task[3];
int j = 1;
for (int i = 0; i < tasks2.Length; i++)
    tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));

Console.WriteLine("Завершення методу Main");

Console.ReadLine();
```



```
C:\Users\puasson\source\repos\Concurrency_chapter1\Cc
First Task
Завершення методу Main
Second Task
Third Task
Task 1
Task 2
Task 3
```

# Поняття конкурентності

---

- Інша форма конкурентності – *реактивне програмування (reactive programming)*.
  - Асинхронне програмування передбачає, що додаток запускає операцію, яка завершиться в майбутньому.
  - Реактивне програмування тісно пов'язане з асинхронним програмуванням, але в його основі лежать асинхронні події замість асинхронних операцій.
  - Асинхронні події можуть не мати фактичного «початку», можуть відбуватися в будь-який час і можуть ініціюватися багаторазово.
  - Один із прикладів такого роду - введення даних користувачем.
- Реактивне програмування - декларативний стиль програмування, при якому додаток реагує на події.
  - Якщо розглядати додаток як величезний скінченний автомат, поведінка програми може бути описана як реакція на серію подій з оновленням свого стану на кожному події.
  - З сучасними фреймворками цей метод дуже корисний в реальних додатках.
  - Реактивне програмування не обов'язково конкурентне, але воно тісно пов'язане з конкурентністю.

# Вступ до асинхронного програмування

---

- Дві основних переваги:
  - Для програм з графічним інтерфейсом користувача асинхронне програмування забезпечує швидкий відгук.
  - Для програм, які працюють на стороні сервера асинхронне програмування забезпечує масштабованість. Серверний додаток може в деякій мірі масштабуватися за рахунок використання пулу потоків, але асинхронний серверний додаток зазвичай володіє на порядок кращими можливостями масштабування.
- Обидві переваги зумовлені тим, що асинхронне програмування вивільняє потоки.
  - Для GUI-програм асинхронне програмування звільняє UI-потік; це дозволяє графічному додатку зберегти високу швидкість відгуку на введення користувача.
  - Для серверних додатків асинхронне програмування звільняє потоки запитів і дозволяє серверу використовувати свої потоки для обслуговування більшої кількості запитів.

# Вступ до асинхронного програмування

---

- У сучасних асинхронних додатках .NET використовуються 2 ключових слова: `async` і `await`.
  - Ключове слово `async` додається в оголошення методу та має подвійне призначення: дозволяє використовувати ключове слово `await` всередині цього методу та наказує компілятору згенерувати для цього методу скінченний автомат за аналогією з роботою `yield return`.
  - Метод з ключовим словом `async` може повернути `Task<TResult>`, якщо він повертає значення; `Task` – якщо він не повертає значення; або будь-який інший «подібний» тип - такий, як `ValueTask`.
  - Крім того, `async`-метод може повернути `IAsyncEnumerable<T>` або `IAsyncEnumerator<T>`, якщо він повертає кілька значень у переліченні.
  - «Подібні» типи представляють обіцянки; вони можуть повідомляти викликаючий код про завершення `async`-методу.
- Уникайте `async void`! Можливо створити `async`-метод, який повертає `void`, але це слід робити тільки при написанні `async`-обробника подій.
  - Звичайний `async`-метод без значення, що повертається повинен повертати `Task`, а не `void`.

# Приклад асинхронного коду

---

```
async Task DoSomethingAsync()
{
    int value = 8;

    await Task.Delay(TimeSpan.FromSeconds(1));

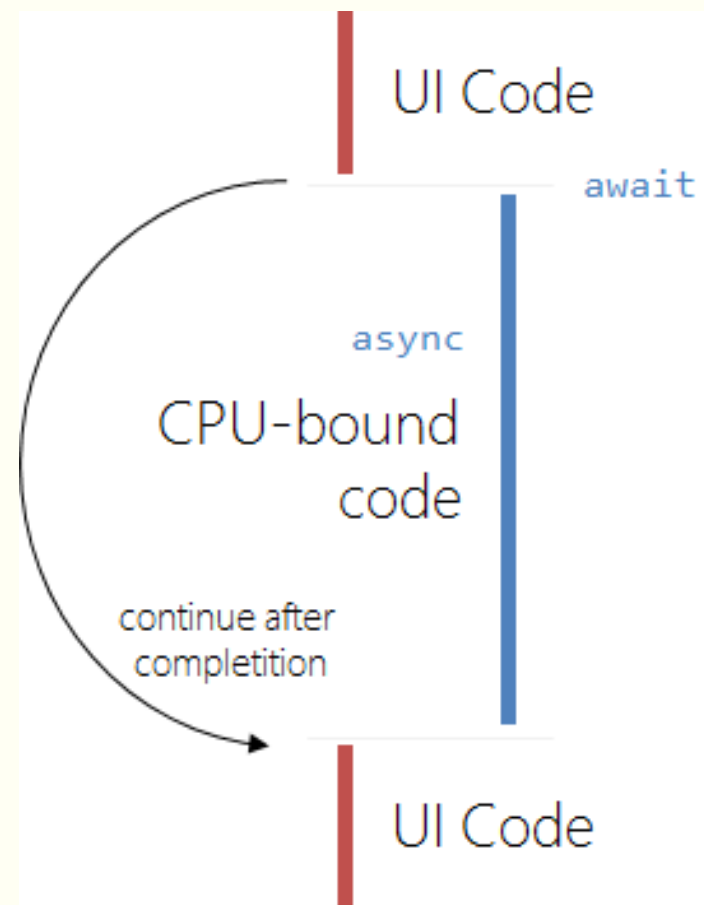
    value *= 2;

    await Task.Delay(TimeSpan.FromSeconds(1));

    return value;
}
```

- `async`-метод починає виконуватися синхронно, як і будь-який інший метод.
  - У середині `async`-методу команда `await` виконує асинхронне очікування за своїм аргументом.
  - Спочатку вона перевіряє, чи завершилася операція: якщо так, то метод залишиться активним (синхронно).
  - Інакше `await` призупиняє `async`-метод і повертає незавершену задачу.
  - Коли операція завершиться пізніше, метод `async` продовжує виконання.
- `async`-метод може розглядатися як такий, що складається з декількох синхронних частин, розділених командами `await`.
  - Перша синхронна частина виконується в потоці, який викликав метод, але де виконуються інші синхронні частини? Відповідь на це питання не проста.

# Вступ до асинхронного програмування



- При виконанні `await` для задачі (найрозповсюдженіший сценарій) у момент, коли `await` вирішує призупинити метод, зберігається контекст.
  - Це поточний об'єкт `SynchronizationContext`, якщо тільки він не дорівнює `null` (тоді контекстом є поточний об'єкт `TaskScheduler`).
  - Метод відновлює виконання в цьому збереженому контексті.
  - Зазвичай контекстом є UI-контекст (для UI-поточку) або контекст пулу потоків (у більшості інших ситуацій).
  - У ASP.NET Classic контекстом також може бути контекст запиту ASP.NET.
  - У ASP.NET Core використовується контекст пулу потоків замість спеціального контексту запиту.
- Таким чином, у наведеному коді всі синхронні частини намагаються відновити продовження в вихідному контексті.
  - Якщо викликати метод `DoSomethingAsync` з UI-поточку, кожна з його синхронних частин буде виконуватися в цьому UI-поточці, але якщо викликати його з потоку з пулу потоків, то кожна з синхронних частин буде виконуватися в будь-якому потоці з пулу потоків.
  - Щоб обійти цю поведінку за умовчанням, можна виконати `await` по результату методу розширення `ConfigureAwait` з передачею `false` в параметрі `continueOnCapturedContext`.

# Вступ до асинхронного програмування

---

```
async Task DoSomethingAsync()
{
    int value = 13;

    // Asynchronously wait 1 second.
    await Task.Delay(TimeSpan.FromSeconds(1))
        .ConfigureAwait(false);

    value *= 2;

    // Asynchronously wait 1 second.
    await Task.Delay(TimeSpan.FromSeconds(1))
        .ConfigureAwait(false);

    Trace.WriteLine(value);
}
```

- Код починає виконання в викликаючому потоці, а після припинення `await` він відновлює виконання в потоці з пулу потоків.
  - Хороша практика: викликати `ConfigureAwait` в базових «бібліотечних» методах і відновлювати контекст тільки тоді, коли буде потрібно - в ваших зовнішніх методах «користувачького інтерфейсу».
- Ключове слово `await` також може працювати з будь-яким об'єктом, який допускає очікування (`awaitable`), побудованим за певною схемою.
  - Наприклад, бібліотека `Base Class Library` включає тип `ValueTask<T>`, який зменшує витрати пам'яті, якщо результат в основному є синхронним, зокрема, при зчитуванні з кешу.
  - Тип `ValueTask<T>` не перетворюється в `Task<T>` напряму, а будується за схемою, що допускає очікування, тому може використовуватись з `await`.
  - У більшості випадків `await` отримує `Task` або `Task<TResult>`.



## Існує 2 основних способи створення екземплярів Task

---

- Деякі задачі представляють реальний код, який повинен виконуватись процесором – *обчислювальні задачі* – створюються викликом `Task.Run` (або `TaskFactory.StartNew`, якщо мають виконуватись за певним розкладом).
- Інші задачі представляють сповіщення (базуються на подіях) та створюються за допомогою `TaskCompletionSource<TResult>` (або однієї із скорочених форм).
  - Більшість задач вводу/виводу використовує `TaskCompletionSource<TResult>`.

# Обробка помилок з async-await виглядає логічно

---

```
abstract class ch01_03
{
    public abstract Task PossibleExceptionAsync();
    public abstract void LogException(Exception ex);

    async Task TrySomethingAsync()
    {
        try
        {
            await PossibleExceptionAsync();
        }
        catch (NotSupportedException ex)
        {
            LogException(ex);
            throw;
        }
    }
}
```

- У коді PossibleExceptionAsync може викинути виняток NotSupportedException, проте TrySomethingAsync може природно перехопити його.
  - Трасування стеку перехопленого винятку зберігається без штучного пакування в TargetInvocationException чи AggregateException.

# Вступ до асинхронного програмування

---

```
abstract class ch01_04
{
    public abstract Task PossibleExceptionAsync();
    public abstract void LogException(Exception ex);

    async Task TrySomethingAsync()
    {
        // The exception will end up on the Task, not thrown directly.
        Task task = PossibleExceptionAsync();

        try
        {
            // The Task's exception will be raised here, at the await.
            await task;
        }
        catch (NotSupportedException ex)
        {
            LogException(ex);
            throw;
        }
    }
}
```

- Коли async-метод викидає виняток, він поміщується в вихідний об'єкт Task, і задача Task завершується.
  - При виконанні await для цього Task-об'єкта оператор await отримує цей виняток і (заново) викидає його так, що початкове трасування стека зберігається.
  - Тут код буде працювати очікувано, якщо PossibleExceptionAsync є async-методом.

# Вступ до асинхронного програмування

---

```
class ch01_05
{
    async Task WaitAsync()
    {
        // This await will capture the current context ...
        await Task.Delay(TimeSpan.FromSeconds(1));
        // ... and will attempt to resume the method here
        // in that context.
    }

    void Deadlock()
    {
        // Start the delay.
        Task task = WaitAsync();

        // Synchronously block, waiting for the async method
        // to complete.
        task.Wait();
    }
}
```

- Якщо викликається async-метод, слід виконати await для задачі, яку цей метод повертає.
  - Намгайтесь не викликати Task.Wait(), Task<TResult>.Result або GetAwaiter().GetResult(), оскільки це призводить до взаємоблокування (deadlock).
- Тут код створює дедлок при виклику з UI-контексту або контексту ASP.NET Classic, оскільки обидва допускають виконання тільки одного потоку.
  - Deadlock() викличе WaitAsync(), що запускає затримку, а потім (синхронно) очікує завершення цього метода з блокуванням контекстного потоку.
  - Коли затримка завершиться, await намагається відновити WaitAsync() у збереженому контексті, проте не зможе, оскільки в контексті вже є заблокований потік (контекст не дорукає більше 1 потоку в будь-який момент часу).
- Запобігти взаємоблокуванню можна 2 способами:
  - застосувати ConfigureAwait(false) в WaitAsync (змусить await ігнорувати його контекст)
  - застосувати await з викликом WaitAsync (перетворює Deadlock() в async-метод).

# Вступ до асинхронного програмування

---

- Асинхронні потоки беруть основу `async` та `await` і розширяють її для роботи з багатьма значеннями.
  - Асинхронні потоки будуються на основі концепції асинхронних перелічуваних об'єктів, які схожі на звичайні перелічувані об'єкти (`enumerables`), за винятком того, що дозволяють виконати асинхронну роботу при отриманні наступного елемента послідовності.
- Асинхронні потоки особливо корисні тоді, коли маємо послідовність даних, які надходять поодиноці або блоками.
  - Наприклад, якщо додаток обробляє відповідь API, у якому використовується розбиття на сторінки з параметрами `limit` та `offset`, асинхронні потоки можуть стати ідеальною абстракцією.

# Вступ до паралельного програмування

---

- Паралельне програмування слід використовувати, коли серйозний обсяг обчислювальної роботи може бути розділений на незалежні блоки.
  - Паралельне програмування тимчасово підвищує завантаження процесора для поліпшення пропускної спроможності системи; це корисно в клієнтських системах, в яких процесор часто простоює, але в серверних системах зазвичай недоречно.
  - У більшості серверів присутні деякі вбудовані засоби паралелізму; наприклад, ASP.NET обробляє кілька запитів паралельно.
  - Написання паралельного коду на сервері може приносити користь в деяких ситуаціях (якщо вам відомо, що кількість одночасно обслуговуваних користувачів завжди буде низькою), але, як правило, паралельне програмування на сервері буде конфліктувати з вбудованими паралельними засобами і не принесе ніякої реальної користі.
- Існує 2 форми паралельного програмування: паралелізм даних і паралелізм задач.
  - Паралелізм даних виникає тоді, коли є набір переважно незалежних фрагментів даних, які очікують обробки.
  - Під паралелізмом задач розуміється ситуація, в якій є деякий пул роботи, де кожен фрагмент роботи в основному не залежить від інших.
  - Паралелізм задач може бути динамічним: якщо один фрагмент роботи породжує кілька додаткових фрагментів роботи, вони можуть бути додані в пул роботи.

# Вступ до паралельного програмування

---

- Відомі різні підходи до реалізації паралелізму даних.
  - Метод `Parallel.ForEach()` є аналогом циклу `foreach` і повинен використовуватися там, де це можливо.
  - Клас `Parallel` також підтримує `Parallel.For()` - аналог циклу `for` і може використовуватися, якщо обробка даних залежить від індексу.
  - Інший варіант - `PLINQ` (`Parallel LINQ`), що надає метод розширення `AsParallel` для запитів `LINQ`.
- `Parallel` більш ефективно витрачає ресурси, ніж `PLINQ`;
  - `Parallel` краще співіснує з іншими процесами в системі, тоді як `PLINQ` (за умовчанням) буде намагатися поширитися по всьому процесорам.
  - До недоліків `Parallel` слід віднести те, що він вимагає більш явної реалізації; `PLINQ` в багатьох випадках дозволяє писати більш елегантний код.

# Вступ до паралельного програмування

---

- Код, що використовує `Parallel.ForEach`, виглядає приблизно так:

```
abstract class Matrix
{
    public abstract void Rotate(float degrees);
}

void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

- PLINQ виглядає десь так:

```
public abstract bool IsPrime(int value);

IEnumerable<bool> PrimalityTest(IEnumerable<int> values)
{
    return values.AsParallel().Select(value => IsPrime(value));
}
```



# Вступ до паралельного програмування

---

- Для паралельної обробки справедлива рекомендація: блоки роботи повинні бути якомога незалежними один від одного.
  - Це максимізує паралелізм: як тільки доведеться використовувати спільний доступ до стану в різних потоках, потрібно буде синхронізувати його.
- Результати паралельної обробки можуть оброблятися різними способами.
  - Вихід можна помістити в деякий різновид конкурентної колекції або ж провести агрегування результатів для отримання зведеного показника.
  - Агрегування часто застосовується в паралельній обробці; такий різновид функціональності «відображення / згортка» також підтримується перевантаженими версіями методів класу `Parallel`.

# Паралелізм задач

---

- Паралелізм задач орієнтований на виконання роботи.
  - На високому рівні між паралелізмом даних і паралелізмом завдань є багато спільного; «обробка даних» може розглядатися як різновид «роботи».
  - Багато задач паралелізму можуть вирішуватися будь-яким з цих способів; використовуйте той API, який здасться вам більш природним для поточного завдання.
- `Parallel.Invoke()` — різновид метода `Parallel()`, що реалізує різновид паралелізму задач типу «галуження/об'єднання»: ви передаєте набір делегатів, які повинні виконуватись паралельно:

```
void ProcessArray(double[] array) {  
    Parallel.Invoke(  
        () => ProcessPartialArray(array, 0, array.Length / 2),  
        () => ProcessPartialArray(array, array.Length / 2, array.Length)  
    );  
}  
  
void ProcessPartialArray(double[] array, int begin, int end)  
{  
    // CPU-intensive processing...  
}
```

# Паралелізм задач

---

- Тип Task спочатку був розроблений для паралелізму задач, хоча він також використовувався для асинхронного програмування.
  - Екземпляр Task (в контексті паралелізму задач) представляє деяку роботу.
  - Метод Wait() може використовуватися для очікування завершення завдання, а властивості Result і Exception – для отримання результатів цієї роботи.
  - Код, що використовує Task безпосередньо, складніше коду, в якому використовується Parallel, але і він може бути корисним, якщо структура паралелізму невідома до стадії виконання.
  - З цим різновидом динамічного паралелізму кількість необхідних фрагментів роботи невідома до початку обробки; це з'ясовується під час виконання.
- У загальному випадку динамічний фрагмент роботи повинен запускати всі необхідні йому дочірні задачі, а потім очікувати їх завершення.
  - У типу Task є спеціальний прапор TaskCreationOptions.AttachedToParent, який може використовуватися для цієї мети.

# Паралелізм задач

---

- Паралелізм задач повинен прагнути до незалежності складових, як і паралелізм даних.
  - Чим незалежніші ваші делегати, тим ефективніше програма.
  - Крім того, якщо делегати залежні один від одного, їх доведеться синхронізувати, що ускладнить написання правильного коду.
  - При паралелізмі задач слід особливо уважно стежити за змінними, збереженими в замиканнях (closures).
  - Пам'ятайте, що в замиканнях зберігаються посилання (а не значення), і це може привести до неочевидних ситуацій зі спільним використанням даних.

# Обробка помилок при всіх типах паралелізму організується аналогічно

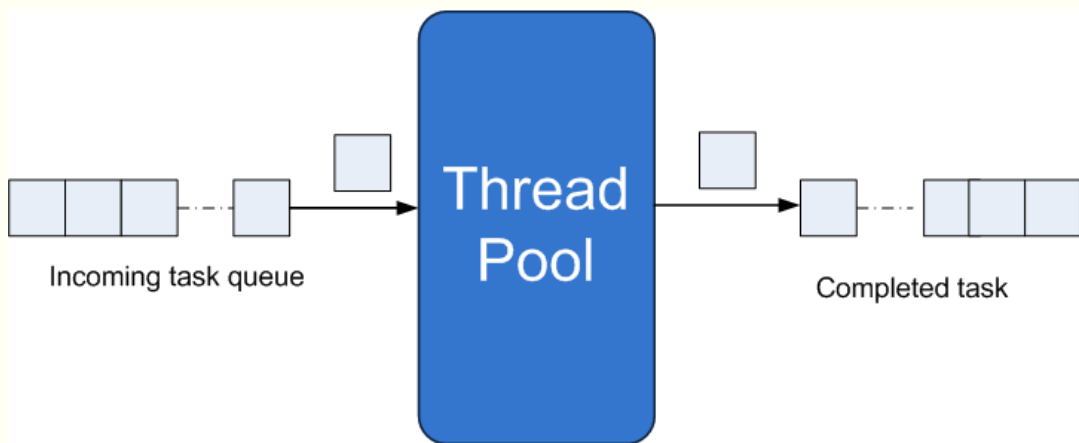
---

- Оскільки операції виконуються паралельно, в програмі можуть виникнути множинні винятки, тому вони упаковуються в виняток `AggregateException`, що запускається в ваш код.
  - Ця поведінка послідовно реалізується для `Parallel.ForEach()`, `Parallel.Invoke()`, `Task.Wait()` і т. д.
  - Тип `AggregateException` містить корисні методи `Flatten()` і `Handle()`, що спрощують код обробки помилок.

```
void Test()
{
    try
    {
        Parallel.Invoke(() => { throw new Exception(); },
            () => { throw new Exception(); });
    }
    catch (AggregateException ex)
    {
        ex.Handle(exception =>
        {
            Trace.WriteLine(exception);
            return true; // "handled"
        });
    }
}
```

# Паралелізм задач

---



- Зазвичай не доводиться турбуватися про те, як пул потоків організовує виконання роботи.
  - Паралелізм даних і задач використовують динамічно регульовані розподільники (partitioners) для розподілу роботи між робочими потоками.
  - Пул потоків збільшує кількість потоків при необхідності.
  - Він має одну робочу чергу, і кожен потік з пулу потоків використовує власну робочу чергу.
  - Коли потік з пулу ставить в чергу додаткову роботу, то спочатку відправляє її в свою чергу, так як робота зазвичай зв'язується з поточним робочим елементом (work item); така поведінка змушує потоки займатися своєю власною частиною роботи і максимізує відсоток влучень в кеш.
  - Якщо у іншого потоку немає роботи, він забирає роботу з черги іншого потоку.
  - Компанія Microsoft витратила багато сил на те, щоб пул потоків по можливості працював ефективно; існує багато налаштувань, які можна змінювати для досягнення максимальної швидкодії.
  - Якщо ваші задачі не дуже малі, вони повинні добре працювати з налаштуваннями за умовчанням.

# Задачі повинні бути ні надто короткими, ні надто довгими

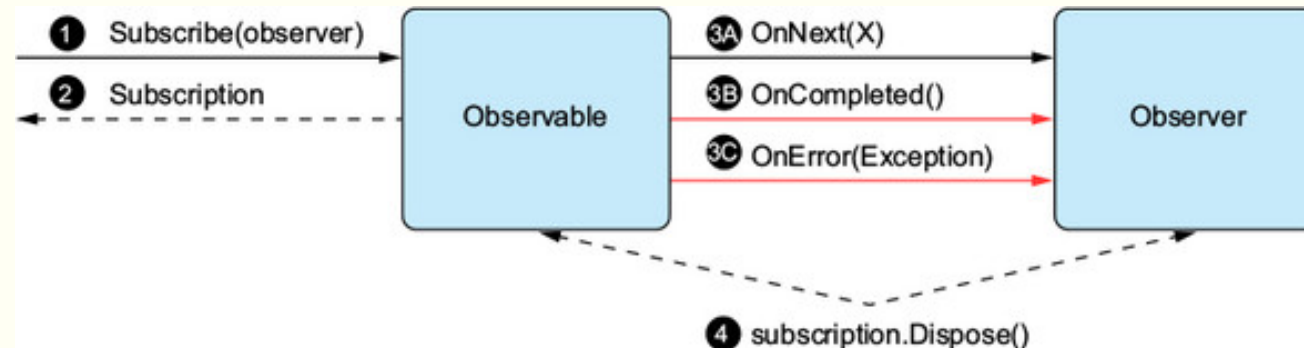
---

- Якщо задачі виходять занадто короткими, то витрати ресурсів на розбиття даних на задачі та планування цих задач у пулі потоків починають відігравати значну роль.
  - Якщо задачі надто довгі, то пул потоків не може динамічно регулювати рівномірний розподіл роботи.
  - Важко заздалегідь визначити, яку задачу слід вважати «занадто короткою» або «занадто довгою»; це залежить від розв'язуваної задачі і приблизних можливостей обладнання.
  - Як правило, краще робити свої задачі якомога коротшими без створення проблем швидкодії (якщо швидкодія раптово падає, значить задачі занадто короткі).
  - Ще краще не працювати з задачами безпосередньо, а скористатися типом Parallel або PLINQ.
  - Ці високорівневі форми паралелізму містять вбудовані засоби розподілу роботи, які вирішують цю задачу за вас (і вносять необхідні корективи під час виконання).

# Вступ до реактивного програмування

---

- Реактивне програмування дозволяє розглядати потік подій як потік даних.
  - Як правило, якщо події передаються будь-які аргументи, то в коді краще використовувати `System.Reactive` замість звичайного обробника подій.
  - Раніше пакет `System.Reactive` називався `Reactive Extensions`; ця назва часто скорочувалася до «Rx.». Всі три терміни відносяться до однієї технології.
- Реактивне програмування базується на концепції спостережуваних потоків (`observable streams`).
  - Підписавшись на спостережуваний потік, ви будете отримувати довільну кількість елементів даних (`OnNext`);
  - Потік може завершитися однією помилкою (`OnError`) або повідомленням «кінець потоку» (`OnCompleted`).
  - Деякі спостерігаються потоки ніколи не завершуються.





# Вступ до реактивного програмування

---

- Реальні інтерфейси виглядають так:

```
interface IObservable<in T>
{
    void OnNext(T item);
    void OnCompleted();
    void OnError(Exception error);
}
interface IObservable<out T>
{
    IDisposable Subscribe(IObservable<TResult> observer);
}
```

- Бібліотека System.Reactive (Rx) компанії Microsoft містить всі реалізації, які можуть знадобитися.
  - Код Reactive дуже схожий на LINQ; його можна розглядати як «LINQ to Events».
  - System.Reactive містить всі можливості LINQ, а також додає велику кількість власних операторів — особливо призначених для роботи з часом.

# Вступ до реактивного програмування

---

```
void Test() {  
    Observable.Interval(TimeSpan.FromSeconds(1))  
        .Timestamp()  
        .Where(x => x.Value % 2 == 0)  
        .Select(x => x.Timestamp)  
        .Subscribe(x => Trace.WriteLine(x));  
}
```

==

```
void Test2() {  
    IObservable<DateTimeOffset> timestamps =  
        Observable.Interval(TimeSpan.FromSeconds(1))  
            .Timestamp()  
            .Where(x => x.Value % 2 == 0)  
            .Select(x => x.Timestamp);  
    timestamps.Subscribe(x => Trace.WriteLine(x));  
}
```

- Код зліва починається з запуску лічильника з періодичним таймером (Interval) і додавання часової мітки для кожної події (Timestamp).
  - Потім події фільтруються так, щоб включалися тільки парні значення лічильника (Where), вибираються значення часової мітки (Timestamp), після чого кожне значення часової мітки, що надійшло, записується в відкладник (Subscribe).
  - Головна відмінність: LINQ to Objects і LINQ to Entities використовують *модель витягування (pull model)*, при якій перерахування запиту LINQ «витягує» дані з запиту, тоді як LINQ to Events (System.Reactive) використовує *модель проттовхування (push model)*, при якій події надходять і переміщаються по запиту самі по собі.
- Визначення спостережуваного потоку не залежить від його підписок. Для типу нормально визначати спостережувані потоки та робити їх доступними в вигляді ресурсу IObservable<TResult>.
  - Потім інші типи можуть підписуватись на ці потоки або об'єднувати їх з іншими операторами для створення іншого спостережуваного потоку.
- Підписка System.Reactive також є ресурсом.
  - Оператори Subscribe повертають реалізацію IDisposable, яка представляє підписку.
  - Коли ваш код завершить прослуховування спостережуваного потоку, він повинен припинити свою підписку.

# Вступ до реактивного програмування

---

- Підписки поведуть себе по-різному з холодними та гарячими спостережуваними об'єктами.
  - Гарячий (hot) спостережуваний об'єкт представляє собою потік подій, який завжди знаходиться в русі; якщо при появі події немає жодного підписника, вона втрачається. Наприклад, переміщення миші.
  - У холодного (cold) спостережуваного об'єкта події не надходять постійно. Такий об'єкт реагує на підписку, починаючи послідовність подій. Наприклад, завантаження HTTP; підписка ініціює відправку запиту HTTP.
- Оператор `Subscribe` також завжди повинен отримувати параметр обробки помилок.
  - Більш правильний приклад, який буде правильно реагувати, якщо спостережуваний потік завершується з помилкою:

```
void Test3() {  
    Observable.Interval(TimeSpan.FromSeconds(1))  
        .Timestamp()  
        .Where(x => x.Value % 2 == 0)  
        .Select(x => x.Timestamp)  
        .Subscribe(x => Trace.WriteLine(x),  
            ex => Trace.WriteLine(ex));  
}
```

# Бібліотека TPL Dataflow — цікаве поєднання асинхронних і паралельних технологій

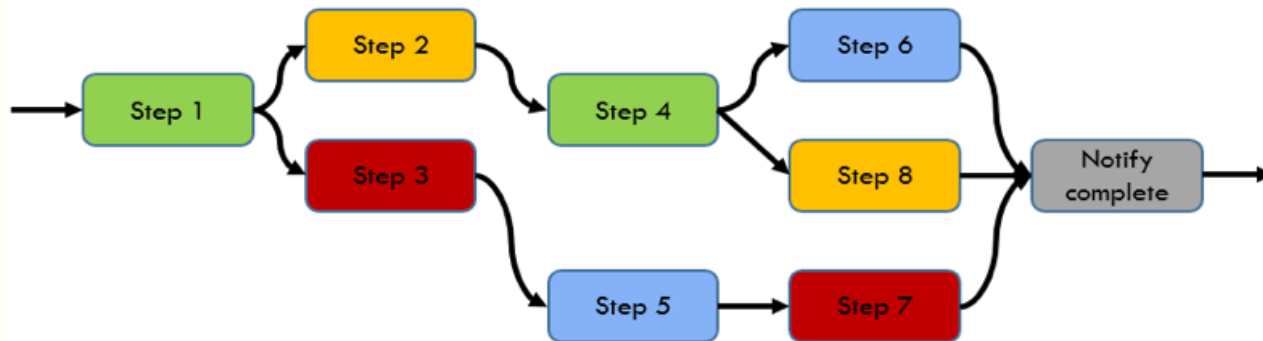
---

- Ця бібліотека може бути корисною для послідовності процесів, які повинні застосовуватись до ваших даних.
  - Уявіть, що потрібно завантажити дані за URL-адресою, розібрати їх, а потім обробити паралельно з іншими даними.
  - TPL Dataflow зазвичай використовується як простий конвеєр: дані входять з одного кінця та переміщуються, поки не вийдуть з іншого кінця.
- Бібліотека здатна впоратися з сітчастими (mesh) структурами будь-якого типу.
  - У сітках можна визначати галуження, об'єднання і цикли.
  - У більшості випадків мережі TPL Dataflow використовуються як конвеєри.
- Базовим структурним елементом мережі потоку даних (dataflow mesh) є *блок потоку даних (dataflow block)*.
  - Блок може бути *блоком-приймачем (отримання даних)*, *блоком-джерелом (виробництво даних)* або їх поєднанням.
  - Блоки-джерела можуть зв'язуватись з блоками-приймачами для формування мережі.

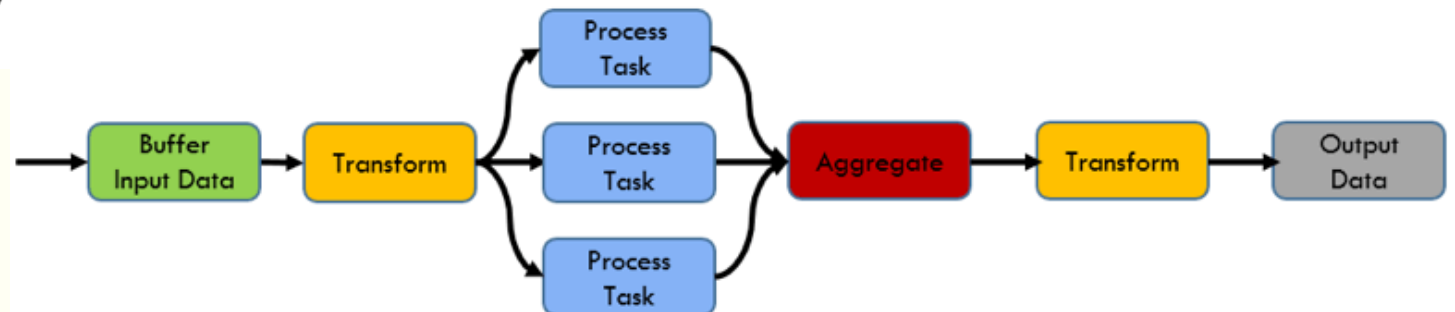
# Бібліотека TPL Dataflow

- Блоки є напівнезалежними; вони обробляють дані по мірі надходження і передають результат далі.
  - Зазвичай ви створюєте всі блоки, встановлюєте зв'язки між ними, а потім починаєте подавати дані з одного кінця.
  - Дані після цього виходять з іншого кінця самі собою.
  - Можливості потоків даних цим не обмежуються; можна створювати зв'язки і додавати їх у мережу в той час, коли по них переміщуються дані; проте це досить нетривіальний сценарій.

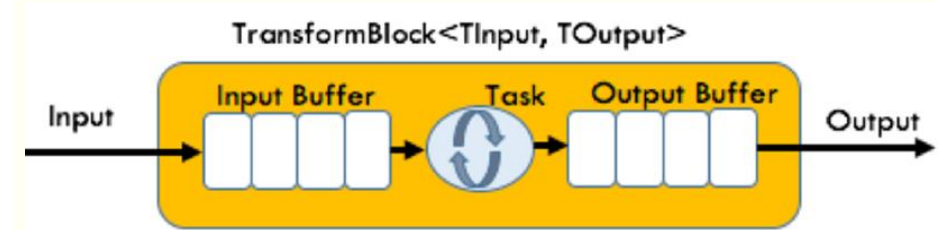
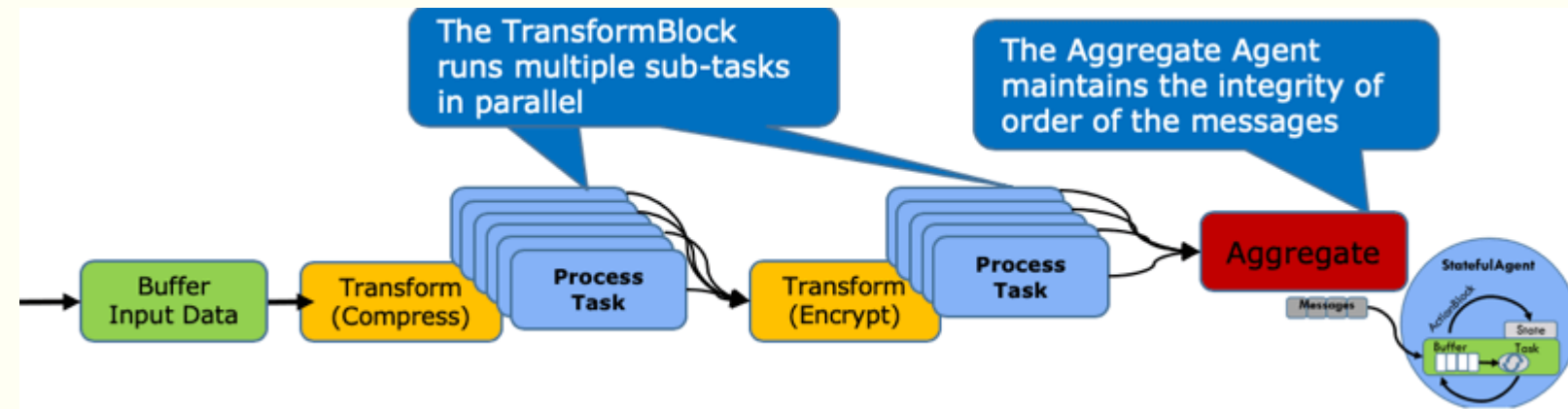
**Componentized Workflow**



**TPL Dataflow workflow**



# Бібліотека TPL Dataflow



- Блоки-приймачі містять буфери для отримуваних даних.
  - Наявність буфера дозволяє їм отримувати нові елементи даних навіть тоді, коли вони ще не готові до їх обробки; це дозволяє даним переміщатись по мережі.
  - Така буферизація може створити проблеми в сценаріях з галуженням, в яких один блок-джерело зв'язується з 2 блоками-приймачами.
  - Якщо у блока-джерела маються дані для відправки по напрямку потоку, він починає пропонувати їх своїм зв'язаним блокам по одному.
  - За умовчанням перший блок-приймач просто отримує дані та буферизує їх, а другий блок-приймач ці дані ніколи не отримає.
  - Проблема вирішується обмеженням буферів блоків-приймачів.

# Бібліотека TPL Dataflow

---

```
try
{
    var multiplyBlock = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
    });
    var subtractBlock = new TransformBlock<int, int>(item => item - 2);
    multiplyBlock.LinkTo(subtractBlock,
        new DataflowLinkOptions { PropagateCompletion = true });

    multiplyBlock.Post(1);
    subtractBlock.Completion.Wait();
}
catch (AggregateException exception)
{
    AggregateException ex = exception.Flatten();
    Trace.WriteLine(ex.InnerException);
}
```

- Якщо щось піде не так, відбувається відмова блоку – наприклад, якщо оброблюючий делегат викидає виняток при обробці елемента даних.
  - Коли в блоці відбувається відмова, він перестає отримувати дані.
  - За умовчанням це не призводить до порушення працездатності всієї мережі, а дозволяє перебудувати цю частину мережі чи перенаправити дані.
  - Проте це нетривіальний сценарій; у більшості випадків зазвичай потрібно, щоб відмови розповсюджувались по зв'язках до цільових блоків.
- Потік даних теж підтримують цей варіант; єдиний неочевидний аспект — виняток, що розповсюджується по зв'язках, упаковується в `AggregateException`.
  - Звідси, при довгому конвеєрі можуть з'являтися винятки з великою глибиною вкладеності;
  - проблему можна обійти за допомогою методу `AggregateException.Flatten()`

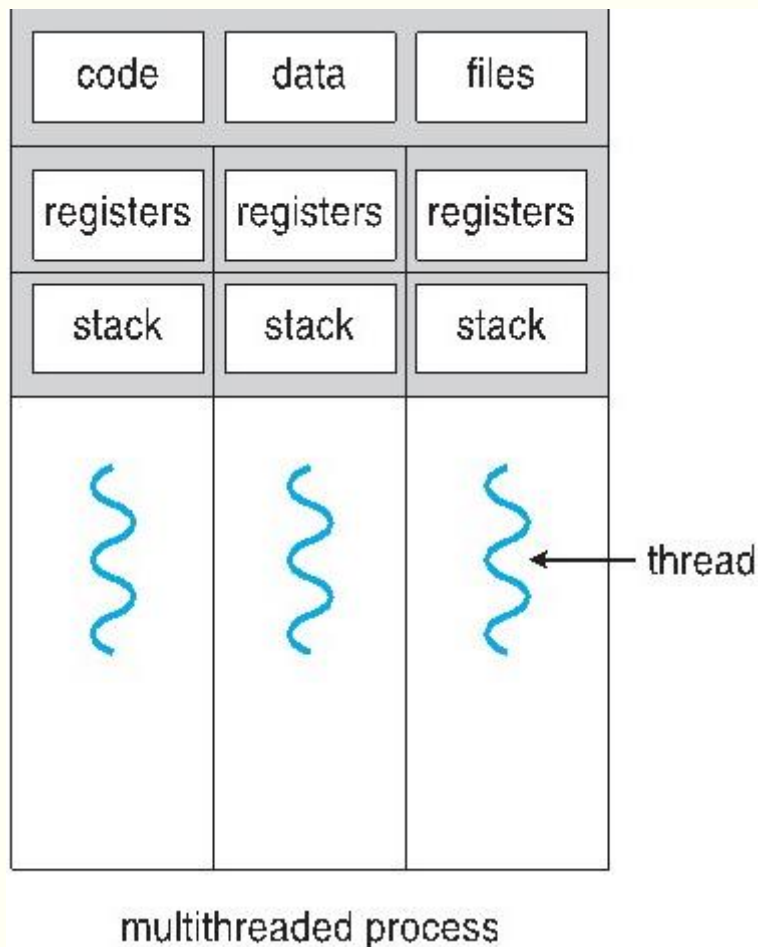
# На перший погляд мережі потоків даних дуже схожі на спостережувані потоки

---

- Як у мереж, так і в потоків існує концепція елементів даних, що в них переміщаються.
  - Крім того, у мереж і потоків є концепції нормального завершення (сповіщення про те, що дані перестали надходити) і завершення з відмовою (сповіщення про те, що протягом обробки даних виникла деяка помилка).
- Проте System.Reactive (Rx) і TPL Dataflow володіють різними можливостями.
  - Спостережувані об'єкти Rx зазвичай краще блоків потоку даних при виконанні будь-яких операцій, пов'язаних з хронометражем.
  - Блоки потоків даних загалом краще спостережуваних об'єктів Rx при виконанні паралельної обробки.
- На концептуальному рівні робота Rx нагадує налаштування зворотних викликів: кожний крок спостережуваного об'єкта напряму викликає наступний крок.
  - З іншого боку, кожний блок у мережі потоку даних практично незалежний від решти блоків.
  - Як Rx, так і TPL Dataflow мають власні галузі застосування, які частково перетинаються. Також вони добре працюють разом.



# Вступ до багатопоточного програмування



- Потік є незалежним виконавцем (executor).
  - Кожний процес складається з кількох потоків, тому вони можуть виконувати різні операції одночасно.
  - Кожний потік має власний незалежний стек, проте він спільно використовує пам'ять з рештою потоків процесу.
  - У деяких додатках передбачено один спеціальний потік. Наприклад, додатки з користувацьким інтерфейсом мають один спеціальний UI-потік, а в консольних додатках існує спеціальний головний потік.
- У кожного додатку .NET присутній пул потоків.
  - Пул потоків містить набір робочих потоків, готових до виконання довільної роботи, що буде їм призначена.
  - Пул потоків відповідає за визначення кількості потоків, що знаходяться в пулі, в будь-який момент часу.
  - Пул потоків було ретельно оптимізовано для більшості реальних ситуацій.
- Створювати нові потоки самостійно вам не потрібно.
  - Єдина ситуація, в якій може виникнути необхідність у створенні екземплярів Thread, — створення потоків STA для COM-взаємодій.
  - Потік відноситься до низькорівневих абстракцій. Пул потоків – на дещо вищому рівні абстракції; коли код ставить роботу в чергу пулу потоків, то сам пул потоків при необхідності позаботиться о создании потока.
  - Розглядатимемо ще вищі абстракції: паралельна обробка і потоки даних ставлять роботу в чергу пулу потоків при необхідності.

# Колекції для конкурентних додатків

---

- Існує кілька видів колекцій, які можуть принести користь при конкурентному програмуванні: конкурентні колекції та незмінювані колекції.
  - Конкурентні колекції дозволяють кільком потокам оновлювати їх одночасно з забезпеченням безпеки.
  - Багато конкурентних колекцій використовують знімки (snapshots) поточного стану, щоб один потік міг перелічити значення, поки інший може додавати чи видаляти значення.
  - Конкурентні колекції зазвичай працюють ефективніше простого захисту звичайної колекції за допомогою блокувань (lock).
- Для модифікації незмінюваної колекції створюється нова колекція, що представляє змінену колекцію.
  - Незмінювані колекції розділяють максимально можливий об'єм пам'яті між екземплярами колекцій.
  - Перевага незмінюваних колекцій – усі їх операції є чистими, тому вони дуже добре працюють у поєднанні з функціональним кодом.



# ДЯКУЮ ЗА УВАГУ!

Наступне запитання: Асинхронні операції на базі синтаксису `async-await`

# Поняття конкурентності

---

- Если вы знакомы с акторскими фреймворками, то увидите, что TPL Dataflow на первый взгляд имеет ряд общих черт с ними.
  - Каждый блок потока данных независим от других — он запускает задачи для выполнения работы по мере необходимости (например, выполнения преобразующего делегата или передачи вывода следующему блоку).
  - Можно также настроить каждый блок для параллельного выполнения, чтобы он запускал несколько задач для обработки дополнительного ввода.
  - Из-за этого поведения каждый блок отчасти напоминает актора в акторских фреймворках.
  - Но TPL Dataflow не является полноценным акторским фреймворком; в частности, отсутствует встроенная поддержка корректного восстановления после ошибок или повторных попыток.
  - TPL Dataflow — библиотека с функциональностью, сходной с функциональностью акторов, но не являющаяся полноценным акторским фреймворком.
- Самые распространенные типы блоков — `TransformBlock<TInput, TOutput>` (аналог LINQ Select), `TransformManyBlock<TInput, TOutput>` (аналог LINQ SelectMany) и `ActionBlock<TResult>`, выполняющий делегата для каждого элемента данных.