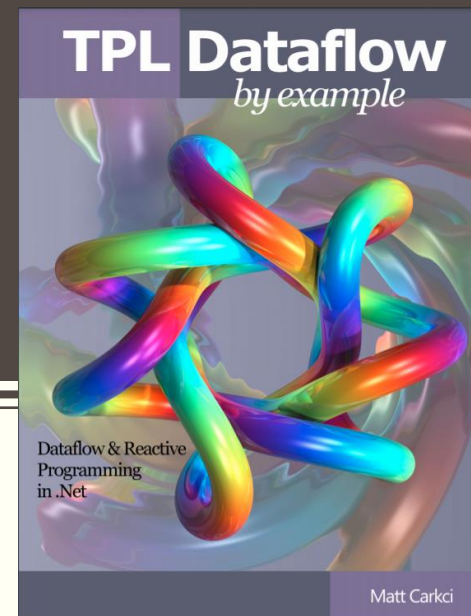


БІБЛІОТЕКА РОЗПАРАЛЕЛЕННЯ ЗАДАЧ TPL DATAFLOW

Питання 12.3.

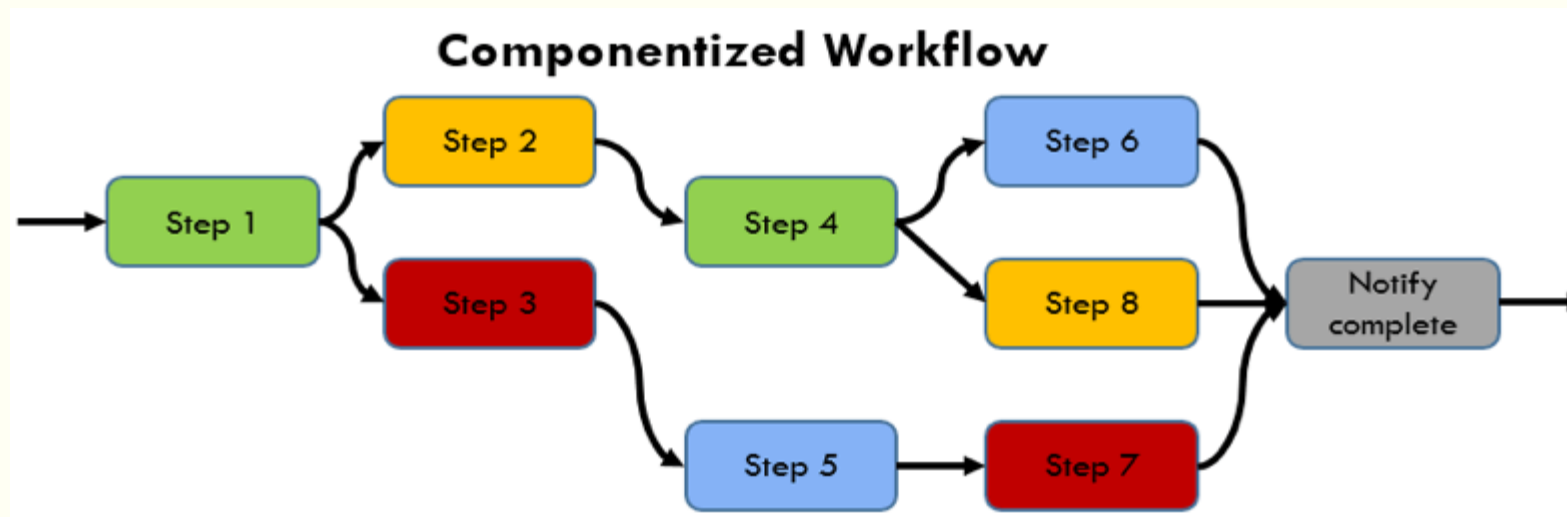


Сітки даних (конвеєри даних, dataflows)

- Термін «сітка даних» (Dataflow) пов'язаний з переміщенням даних та вводився на противагу терміну «control-flow» (виконання на основі управляючих операторів).
- Сітка складається з «блоків» та «зв'язків».
 - Блок – це контейнер для коду, а зв'язок (link) передає дані від одного блоку до іншого.
 - Блоки також мають входи (inputs) для отримання даних та виходи (outputs) для відправки даних.
 - Коли блок має дані на вході, він споживатиме їх та передаватиме внутрішній функції.
 - Результат роботи функції розміщується на виході блоку.
 - Якщо вихід блоку з'єднано зі входом іншого блоку, дані надсилаються по зв'язку на цей вхід, і процес продовжується далі.
- Ключова ідея: дані, що надходять на вхід блоку, спричиняють його виконання.
 - Програмісту не потрібно спеціально опитувати вхід, щоб побачити доступність даних.
 - Блок реагує на існування даних, тому термін «реактивне програмування» давно використовується для опису сітки даних.

Основні поняття для бібліотеки TPL Dataflow

- TPL Dataflow – потужна бібліотека, яка дозволяє створити сітку (конвеєр), а потім (асинхронно) надіслати по ним свої дані.
 - Dataflow використовує декларативний стиль програмування: спочатку повністю визначається сітка, а потім починається обробка даних.
 - Сітка описує структуру, по якій переміщуються дані.
 - Окремі блоки прості, вони відповідають лише за один етап обробки даних.
 - Коли блок завершує роботу над своїми даними, він передає свій результат усім пов'язаним блокам.

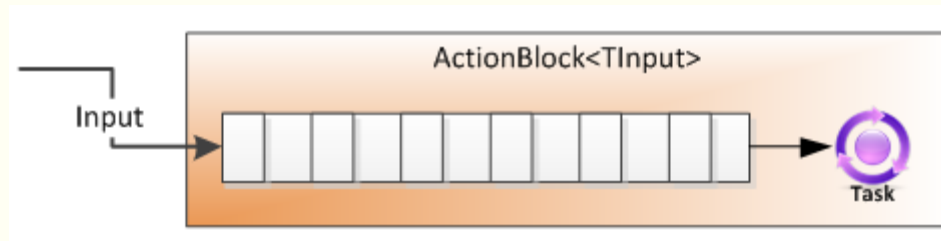


Основні поняття для бібліотеки TPL Dataflow

- За бібліотекою TPL DataFlow стоїть ідея простого створення кількох сценаріїв, зокрема:
 - Конвеєрів пакетної обробки (batch-processing pipelines)
 - Паралельної потокової обробки (parallel stream processing)
 - Буферизація даних (data buffering, joining)
 - Пакетна обробка даних з одного або більше джерел.
- Кожний з цих сценаріїв (patterns) може використовуватись окремо або комбінуватись з іншими сценаріями, дозволяючи розробникам просто виражати складні конвеєри даних.
 - Щоб використовувати TPL Dataflow, встановіть NuGet-пакет System.Threading.Tasks.Dataflow.
- **Блоки (вузли, *blocks*, *nodes*)** – це сутності, які можуть надсилати та отримувати дані, та є базовими композиційними одиницями.
 - Бібліотека TPL Dataflow надає корисні вбудовані типи блоків (predefined blocks), які покривають близько 99% потреб.
 - Вбудовані блоки визначають базові операції dataflow-програм та можуть категоріюватись у 3 групи:
 - блоки, які обробляють дані (виконавчі блоки, execution blocks);
 - блоки, які буферизують чи зберігають дані (буферизуючі блоки, buffer blocks);
 - блоки, які групують дані в колекції (групуючі блоки, grouping blocks).

Виконавчі блоки (Execution Blocks)

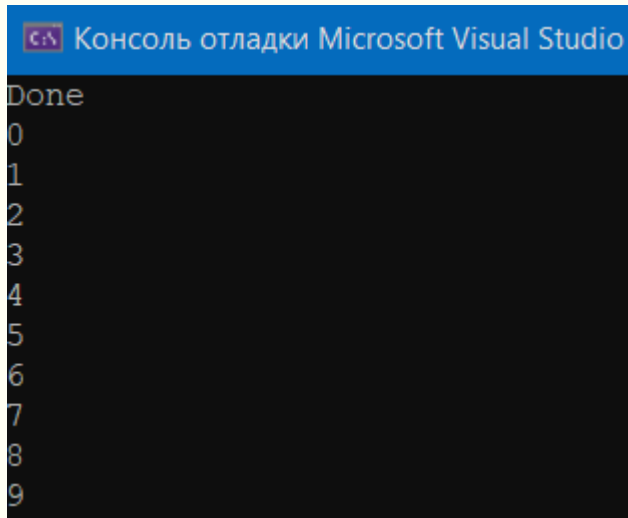
- Обробляють дані подібно до того, як методи приймають дані та, можливо, повертають значення.
 - При створенні Ви передаєте Func- або Action-делегат, який визначає, що виконавчий блок робитиме з даними.
 - Всі виконавчі блоки містять внутрішній буфер, для якого за умовчанням встановлено необмежену місткість (unbounded capacity).
- Блок `ActionBlock<T>` має 1 вхід та не має виходу; еквівалентний класу `Action<T>`.
 - Використовується при потребі обробити вхідні дані, проте не передавати їх далі іншим блокам.
 - У конвеєрі даних такий блок часто називають *стоком* (*sink*).
 - При створенні `ActionBlock<T>` приймає `Action<T>`-об'єкт, який викликається при надходженні даних на вхід.
 - Внутрішній буфер присутній на вході `ActionBlock<T>` та за умовчанням має необмежену місткість, проте її можна змінити за допомогою опцій із перелічення `DataflowBlockOptions`.



Виконавчі блоки (Execution Blocks). `ActionBlock<T>`

```
public static void Run()
{
    var actionBlock = new ActionBlock<int>(
        n => Console.WriteLine(n));
    for (int i = 0; i < 10; i++)
    {
        actionBlock.Post(i);
    }

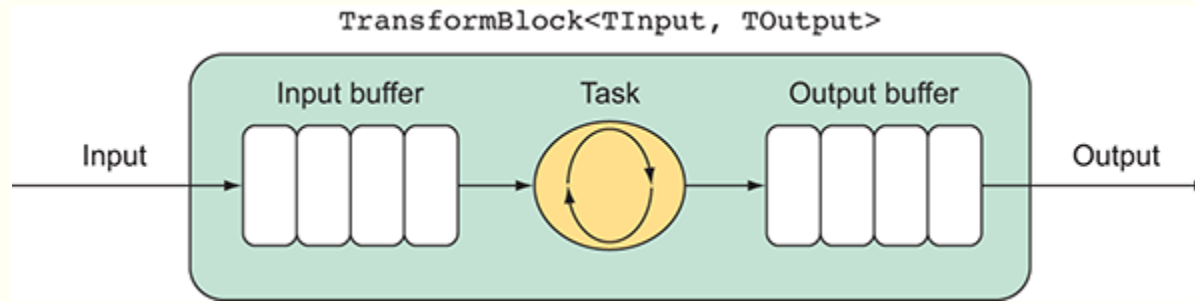
    Console.WriteLine("Done");
}
```



```
Консоль отладки Microsoft Visual Studio
Done
0
1
2
3
4
5
6
7
8
9
```

- Приклад показує базове використання блоку `ActionBlock<T>` та передачу даних усім типам блоків, які мають вхід.
 - Функція `Post()` синхронно надсилає дані блокам та повертає `true`, якщо дані були успішно прийняті.
 - Якщо блок відмовляється приймати дані, функція повертає `false` та не буде намагатись заново відправити їх.
- Тут числа від 0 до 9 проштовхуються (`push`) в `actionBlock`.
 - Блок приймає кожне значення та викликає `Action<T>`-об'єкт, заданий при створенні.
- Можемо імітувати довготривалу дію в блоці, додавши `Thread.Sleep(1000)` у відповідний лямбда-вираз;
 - Вивід буде той же.

Виконавчі блоки (Execution Blocks). TransformBlock<T1, T2>



- `TransformBlock<T1,T2>` дуже схожий на `ActionBlock<T>`, проте також має вихід, який можна з'єднати з іншими блоками.
 - Він еквівалентний `Func<T1,T2>` в тому, що повертає результат.
 - Подібно до `ActionBlock<T>`, він приймає функцію на етапі створення, яка оперує вхідними даними.
- Даний блок має 2 буфери: на вході та на виході; проте краще думати розглядати їх як один.
 - Два буфери потрібні, щоб забезпечити передачу даних у тому ж порядку, в якому вони надійшли.
 - Вихідний буфер використовується для відновлення початкового порядку даних.

Виконавчі блоки (Execution Blocks). TransformBlock<T1, T2>

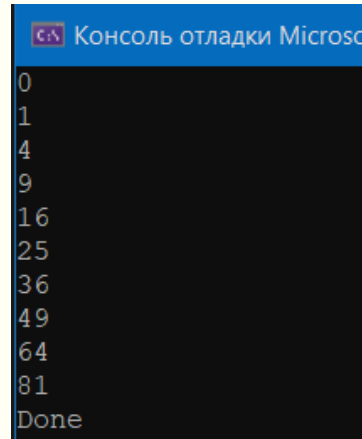
```
public static void Run()
{
    Func<int, int> fn = n => {
        Thread.Sleep(1000);
        return n * n;
    };

    var tfBlock = new TransformBlock<int, int>(fn);

    for (int i = 0; i < 10; i++)
    {
        tfBlock.Post(i);
    }

    for (int i = 0; i < 10; i++)
    {
        int result = tfBlock.Receive();
        Console.WriteLine(result);
    }

    Console.WriteLine("Done");
}
```



```
Консоль отладки Microsof
0
1
4
9
16
25
36
49
64
81
Done
```

- У прикладі створюємо TransformBlock<T1,T2> з функцією, яка підносить до квадрату вхідні значення після секундного очікування (імітуємо тривалий процес).
 - Для виділення даних з TransformBlock<T1,T2> (і будь-якого блоку з виходом) використовують синхронний метод Receive().
 - Якщо доступних даних немає, потік буде призупинено (suspend), поки вони не надійдуть.
 - Цикл for передає числа від 0 до 9 в tfBlock.
- Зауважте, що тут “Done” виводиться після друку всіх вихідних значень: метод Receive() працює синхронно в тому ж потоці, що і цикли for.

Виконавчі блоки (Execution Blocks). TransformBlock<T1, T2> разом з ReceiveAsync(), Task.Result()

```
public static void Run()
{
    Func<int, int> fn = n => {
        Thread.Sleep(1000);
        return n * n;
    };

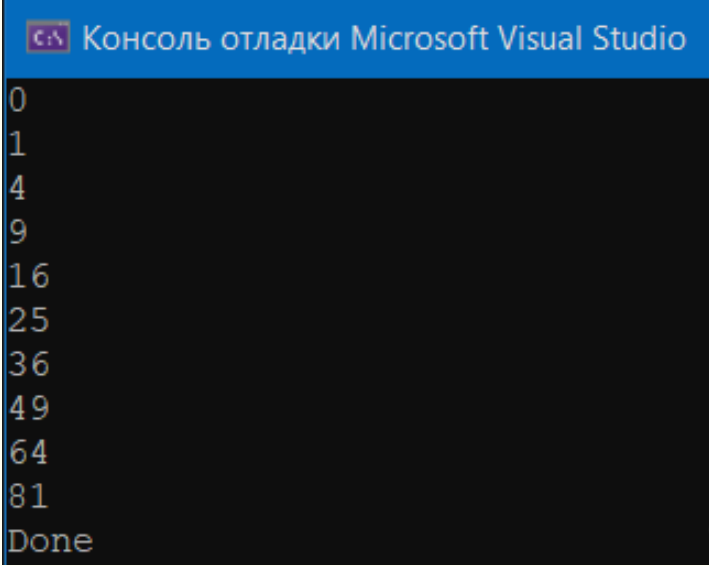
    var tfBlock = new TransformBlock<int, int>(fn);

    for (int i = 0; i < 10; i++)
    {
        tfBlock.Post(i);
    }

    // RecieveAsync повертає Task
    for (int i = 0; i < 10; i++)
    {
        Task<int> resultTask = tfBlock.ReceiveAsync();
        int result = resultTask.Result;
        // виклик Result очікуватиме,
        // поки значення не буде готове
        Console.WriteLine(result);
    }

    Console.WriteLine("Done");
}
```

- Приклад показує, як отримати асинхронно дані з усіх блоків з виходами, використовуючи метод ReceiveAsync().
 - Цей метод повертає не значення (як Receive()), а Task<T>-об'єкт, який представляє операцію надходження.
 - Виклик методу Result() для поверненого Task-об'єкту змушує програму очікувати, поки дані стануть доступними, таким чином роблячи операцію синхронною.



```
Консоль отладки Microsoft Visual Studio
0
1
4
9
16
25
36
49
64
81
Done
```

Виконавчі блоки (Execution Blocks). TransformBlock<T1, T2> разом з ReceiveAsync(), Task.ContinueWith()

```
public static void Run()
{
    Func<int, int> fn = n => {
        Thread.Sleep(1000);
        return n * n;
    };

    var tfBlock = new TransformBlock<int, int>(fn);

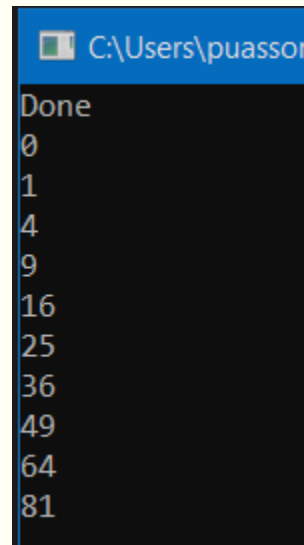
    for (int i = 0; i < 10; i++)
    {
        tfBlock.Post(i);
    }

    Action<Task<int>> whenReady = task => {
        int n = task.Result;
        Console.WriteLine(n);
    };

    for (int i = 0; i < 10; i++)
    {
        Task<int> resultTask = tfBlock.ReceiveAsync();
        resultTask.ContinueWith(whenReady);
        // Коли 'resultTask' виконано,
        // викликаємо 'whenReady' з Task
    }

    Console.WriteLine("Done");
}
```

- Якщо внести зміни в попередній приклад, зможемо отримати дані від блоків асинхронно.
 - Додавання методу ContinueWith() дозволяє головному потоку продовжувати роботу без очікування доступності даних для зчитування.
 - Тут продовження зачіпає дію whenReady, яка виводить результат у консоль.
 - Знову “Done” друкується спочатку, оскільки головному потоку не потрібно очікувати надходження даних.
 - Для того, щоб код спрацював, потрібно застосувати Console.ReadKey() у Main().



```
C:\Users\puasson
Done
0
1
4
9
16
25
36
49
64
81
```

Конфігурування блоку

- Усі вбудовані блоки з TPL Dataflow можна налаштувати, передаючи об'єкт-опції в конструктор блоку.
 - Виконавчі блоки застосовують клас `ExecutionDataflowBlockOptions`, групуючі блоки – `GroupingDataflowBlockOptions`, а буферизуючі блоки – `DataflowBlockOptions`.
 - Класи `ExecutionDataflowBlockOptions` та `GroupingDataflowBlockOptions` породжені від `DataflowBlockOptions`.
- На додачу до опцій з базового класу, `ExecutionDataflowBlockOptions` також включають опції `MaxDegreeOfParallelism` та `SingleProducerConstrained`.

Конфігурування блоку. Опція MaxDegreeOfParallelism

```
public static void Run()
{
    var generator = new Random();
    Action<int> fn = n => {
        Thread.Sleep(generator.Next(1000));
        Console.WriteLine(n);
    };

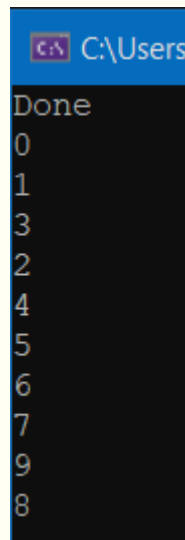
    var opts = new ExecutionDataflowBlockOptions {
        MaxDegreeOfParallelism = 2
    };

    var actionBlock = new ActionBlock<int>(fn, opts);

    for (int i = 0; i < 10; i++)
    {
        actionBlock.Post(i);
    }

    Console.WriteLine("Done");
}
```

- Блоки можна налаштувати для оперування більше, ніж одним шматком даних за раз.
 - За умовчанням обробляється одне значення за раз.
 - Опція MaxDegreeOfParallelism вказує комп'ютеру паралельно оперувати багатьма значеннями.
- Видозмінимо приклад 2 для ActionBlock.
 - Додамо випадкову затримку (delay) в actionBlock, щоб зробити додаток більш реалістичним.
 - Запуск прикладу показує, як порядок вихідних значень відрізняється від порядку вхідних значень у зв'язку із різними затримками.



Конфігурування блоку. Опція SingleProducerConstrained

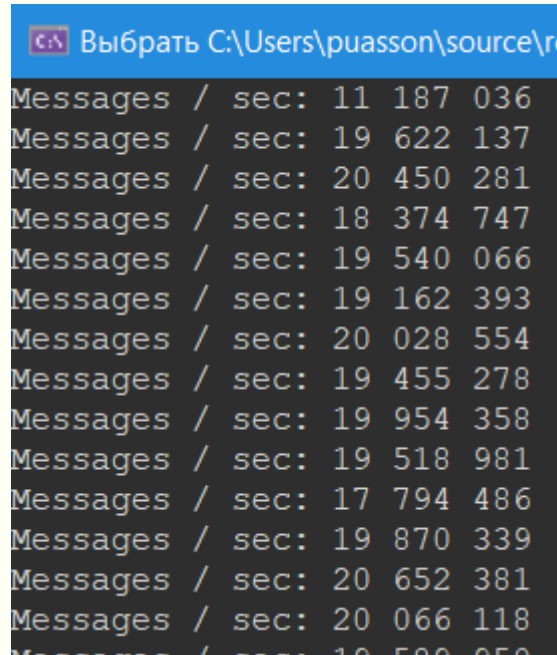
```
static public void Benchmark2()
{
    var sw = new Stopwatch();
    const int ITERS = 6000000;
    var are = new AutoResetEvent(false);

    var ab = new ActionBlock<int>(i => { if (i == ITERS) are.Set(); },
        new ExecutionDataflowBlockOptions {
            SingleProducerConstrained = true
        });

    while (true)
    {
        sw.Restart();
        for (int i = 1; i <= ITERS; i++)
            ab.Post(i);
        are.WaitOne();
        sw.Stop();
        Console.WriteLine("Messages / sec: {0:N0}",
            (ITERS / sw.Elapsed.TotalSeconds));
    }
}
```

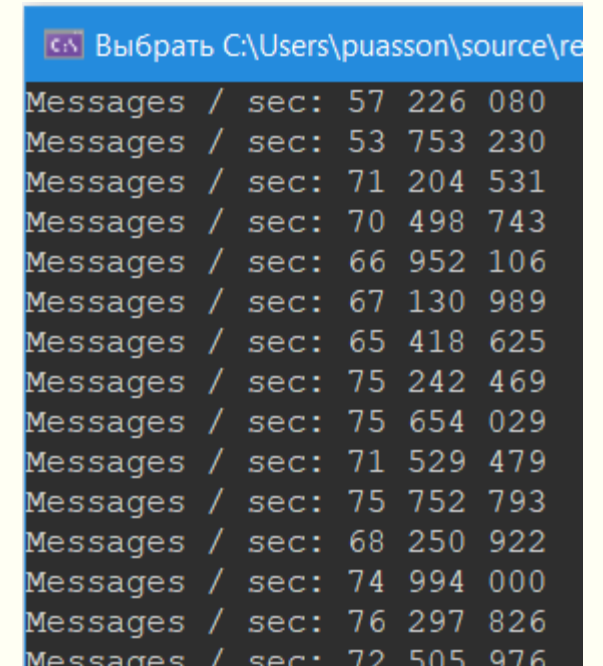
- Оптимізація для ситуацій, коли лише один блок надає дані іншому блоку.
 - Коментар розробника опції

■ Без опції:



```
C:\> Выбрать C:\Users\puasson\source\re
Messages / sec: 11 187 036
Messages / sec: 19 622 137
Messages / sec: 20 450 281
Messages / sec: 18 374 747
Messages / sec: 19 540 066
Messages / sec: 19 162 393
Messages / sec: 20 028 554
Messages / sec: 19 455 278
Messages / sec: 19 954 358
Messages / sec: 19 518 981
Messages / sec: 17 794 486
Messages / sec: 19 870 339
Messages / sec: 20 652 381
Messages / sec: 20 066 118
Messages / sec: 19 589 050
```

З опцією:



```
C:\> Выбрать C:\Users\puasson\source\re
Messages / sec: 57 226 080
Messages / sec: 53 753 230
Messages / sec: 71 204 531
Messages / sec: 70 498 743
Messages / sec: 66 952 106
Messages / sec: 67 130 989
Messages / sec: 65 418 625
Messages / sec: 75 242 469
Messages / sec: 75 654 029
Messages / sec: 71 529 479
Messages / sec: 75 752 793
Messages / sec: 68 250 922
Messages / sec: 74 994 000
Messages / sec: 76 297 826
Messages / sec: 72 505 976
```

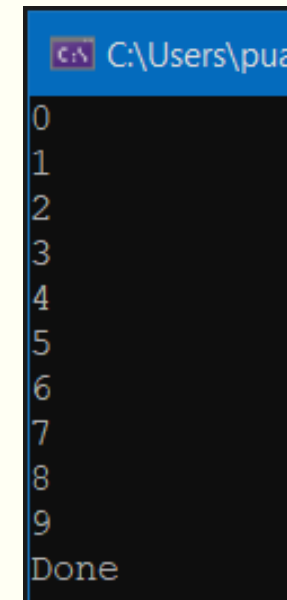
Буферизуючі блоки. `BufferBlock<T>`

```
static public void Run()
{
    var bufferBlock = new BufferBlock<int>();
    for (int i = 0; i < 10; i++)
    {
        bufferBlock.Post(i);
    }

    for (int i = 0; i < 10; i++)
    {
        int result = bufferBlock.Receive();
        Console.WriteLine(result);
    }

    Console.WriteLine("Done");
}
```

- Не змінюють дані, передані через них.
 - Існують лише для збереження чи розповсюдження даних.
- `BufferBlock<T>` - це FIFO-черга даних, подібна до `TransformBlock<T1,T2>`, за винятком того, що вона не змінює дані, що через неї проходять.
 - Приклад показує, що можна прийняти (`Post()`) дані на вхід блоку та, використовуючи `Receive()`, виділити їх з виходу.



Буферизуючі блоки. BroadcastBlock<T>

- BroadcastBlock<T> може надсилати повідомлення багатьом блокам так, щоб вони отримували копію початкового повідомлення.
 - Більшість вбудованих блоків з TPL Dataflow надсилають повідомлення лише одному блоку: першому, який прийме повідомлення.
 - Проте інколи потрібно обробляти повідомлення кількома способами.
- Інша унікальна риса BroadcastBlock<T> - відсутність внутрішнього буфера.
 - Якщо нове повідомлення отримане блоком BroadcastBlock<T>, воно замінить попереднє повідомлення.
 - Якщо подальші потокине приймають повідомлення, BroadcastBlock<T> не намагатиметься повторно його надіслати. Тільки поточне повідомлення буде передаватись.
 - Будь-якому новому блоку, зв'язаному з BroadcastBlock<T>, автоматично надсилатиметься найсвіжіше повідомлення, навіть якщо воно не було негайно отримане BroadcastBlock<T>.
- При конструюванні BroadcastBlock<T> приймає функцію, мета якої дублювати значення.
 - Слід звертати увагу на посилальні типи: кілька блоків, які намагаються змінити об'єкт посилального типу, можуть створити неприємні баги.
 - Дублююча функція має виконувати глибоке клонування всіх посилальних типів для безпеки.
 - Хороша рекомендація: завжди використовувати незмінювані значення в конвеєрах даних.

Буферизуючі блоки. BroadcastBlock<T>

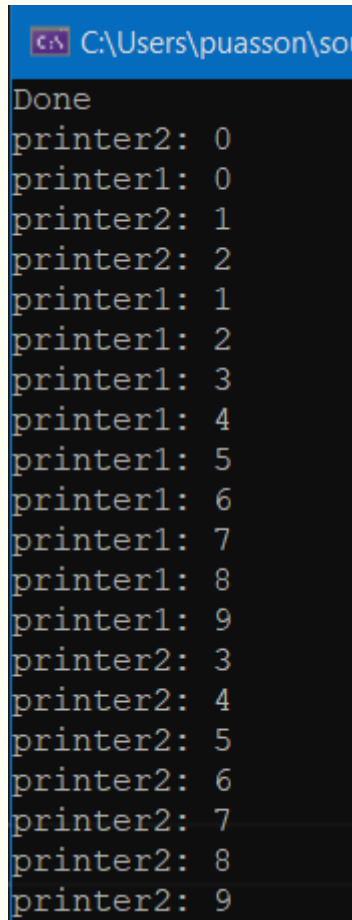
```
static public void Run()
{
    var printer1 = MakePrintBlock("printer1");
    var printer2 = MakePrintBlock("printer2");
    var bcBlock = new BroadcastBlock<int>(n => n);

    bcBlock.LinkTo(printer1);
    bcBlock.LinkTo(printer2);

    for (int i = 0; i < 10; i++)
    {
        bcBlock.Post(i);
    }

    Console.WriteLine("Done");
}

static ActionBlock<int> MakePrintBlock(String prefix)
{
    return new ActionBlock<int>(n =>
        Console.WriteLine(prefix + ": " + n)
    );
}
```



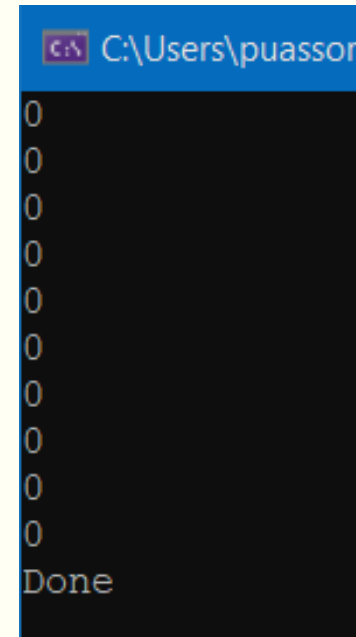
```
Done
printer2: 0
printer1: 0
printer2: 1
printer2: 2
printer1: 1
printer1: 2
printer1: 3
printer1: 4
printer1: 5
printer1: 6
printer1: 7
printer1: 8
printer1: 9
printer2: 3
printer2: 4
printer2: 5
printer2: 6
printer2: 7
printer2: 8
printer2: 9
```

- Для демонстрації з'єднаємо блок з двома іншими блоками.
- printer1 та printer2 виводять префікс, переданий їм при конструюванні та число, яке вони отримують.
 - bcBlock дублює числа від 0 до 9 та передає їх копії обом printer-блокам.
 - Враховуючи використання простого значимого типу, дублююча функція теж проста: n => n.
 - Printer-блоки отримують однакові значення, їх консольний вивід перемежовується.

Буферизуючі блоки. WriteOnceBlock<T>

```
var woBlock = new WriteOnceBlock<int>(n => n);  
for (int i = 0; i < 10; i++)  
{  
    woBlock.Post(i);  
}  
  
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine(woBlock.Receive());  
}  
  
Console.WriteLine("Done");
```

- WriteOnceBlock<T> приймає лише перше передане значення та повертає його при будь-якому зверненні.
 - Блок корисний для зберігання констант.
 - У прикладі приймається та завжди повертається нуль, хоч методи Post() та Receive() викликалися багато разів.



```
C:\Users\puasson  
0  
0  
0  
0  
0  
0  
0  
0  
0  
0  
Done
```

Буферизуючі блоки. Опції DataflowBlockOptions

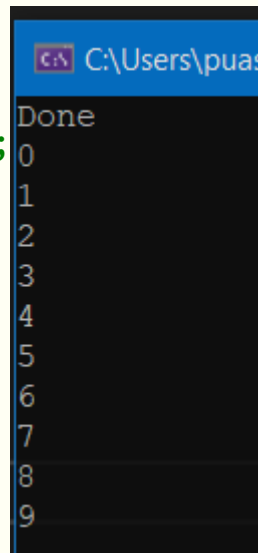
```
static public void Run()
{
    Action<int> fn = n => {
        Thread.Sleep(1000);
        Console.WriteLine(n);
    };
    var opts = new ExecutionDataflowBlockOptions {
        BoundedCapacity = 1
    }; // задає розмір буферу блоку на одне повідомлення

    var actionBlock = new ActionBlock<int>(fn, opts);

    for (int i = 0; i < 10; i++)
    {
        //Console.WriteLine(actionBlock.Post(i));
        actionBlock.SendAsync(i);
    }

    Console.WriteLine("Done");
}
```

- DataflowBlockOptions – батьківський елемент для ExecutionDataflowBlockOptions та GroupingDataflowBlockOptions, тому представлені опції також можуть використовуватись з виконавчими та групуючими блоками.
 - Кожний блок у TPL Dataflow має внутрішній буфер, яким можна керувати.
 - У прикладі встановлено розмір буферу (1) за допомогою властивості BoundedCapacity.
 - Це значить, що лише 1 комірка буферу всередині блоку зберігатиме вхідні дані перед їх обробкою.
 - За умовчанням буфери мають необмежену місткість: BoundedCapacity можна задати значення DataflowBlockOptions.Unbounded (-1), щоб дозволити необмежений розмір.
 - Якщо BoundedCapacity буде 0, в рантаймі викинеться виняток ArgumentOutOfRangeException.



Буферизуючі блоки. Опції DataflowBlockOptions

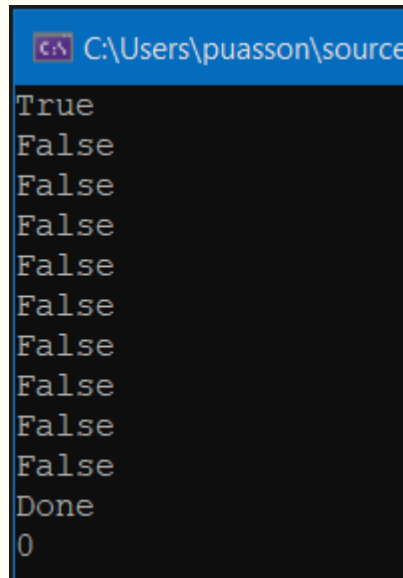
```
static public void Run()
{
    Action<int> fn = n => {
        Thread.Sleep(1000);
        Console.WriteLine(n);
    };
    var opts = new ExecutionDataflowBlockOptions {
        BoundedCapacity = 1
    }; // задає розмір буферу блоку на одне повідомлення

    var actionBlock = new ActionBlock<int>(fn, opts);

    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(actionBlock.Post(i));
        // actionBlock.SendAsync(i);
    }

    Console.WriteLine("Done");
}
```

- Також потрібно замінити метод Post() на SendAsync() унаслідок малого розміру буфера.
 - Щоб побачити причину цього, відкоментуйте рядок з Post() та закоментуйте рядок з SendAsync().
 - Метод Post() прошовхує перше число (нуль) у actionBlock та повертає true.
 - Коли він намагається прошовхнути решту чисел, то не може це зробити, оскільки в буфері блоку не залишилось місця.
 - Таким чином, він перериває передачу та повертає false.



```
C:\Users\puasson\source
True
False
False
False
False
False
False
False
False
Done
0
```

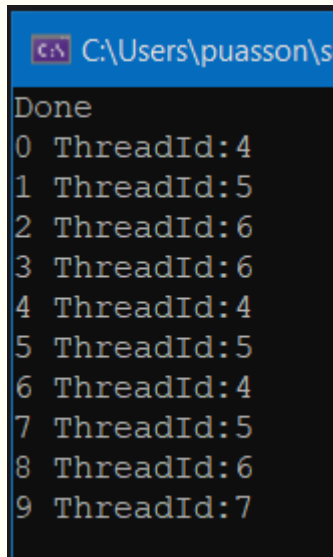
Буферизуючі блоки. Опції DataflowBlockOptions

```
static public void Run()
{
    Action<int> fn = n => {
        Thread.Sleep(1000);
        Console.WriteLine(n + " ThreadId:"
            + Thread.CurrentThread.ManagedThreadId);
    };
    var opts = new ExecutionDataflowBlockOptions {
        MaxMessagesPerTask = 1
    }; // кожна задача оброблятиме лише 1 повідомлення
    // Нова задача буде створюватись для кожного нового повідомлення

    var actionBlock = new ActionBlock<int>(fn, opts);

    for (int i = 0; i < 10; i++)
    {
        actionBlock.SendAsync(i);
    }

    Console.WriteLine("Done");
}
```



```
C:\Users\puasson\s
Done
0 ThreadId:4
1 ThreadId:5
2 ThreadId:6
3 ThreadId:6
4 ThreadId:4
5 ThreadId:5
6 ThreadId:4
7 ThreadId:5
8 ThreadId:6
9 ThreadId:7
```

- Приклад використовує опцію `MaxMessagesPerTask`, щоб визначити кількість повідомлень, яку оброблятиме кожний Task-об'єкт.
 - Тут Task, створений блоком `actionBlock`, оброблятиме тільки одне повідомлення, а новий Task буде створюватись для кожного повідомлення.
 - За умовчанням – `DataflowBlockOptions.Unbounded (-1)`.
 - У той час, як потоки повторно використовуються, лише одне повідомлення обробляється за раз.
- Обмеження кількості повідомлень на Task може допомогти прискорити додатки в певних випадках.
 - З міркувань ефективності фреймворк має пул потоків, з якого він «черпає» при створенні нової задачі.
 - Якщо блок безперервно тримається за потік, оскільки він обробляє постійний потік даних, менше потоків можна використовувати для інших цілей.
 - Створення нового потоку займає більше часу, ніж повторне використання старого, тому обмежуючи кількість повідомлень для задач, ми відпускаємо пов'язаний потік назад до пулу для використання в інших областях додатку.

Буферизуючі блоки. Опції DataflowBlockOptions

```
static public void Run()
{
    var block1 = new BufferBlock<int>(new DataflowBlockOptions {
        NameFormat = "Fu"
    });

    var block2 = new BufferBlock<int>(new DataflowBlockOptions {
        NameFormat = "Bar, Class: {0}, Id: {1}"
    });

    Debug.Assert(false);
}
```

- Опція NameFormat дозволяє визначати debug-time назву та відображати формат для блоків.
 - Присвоєний в опції текст буде відображатись у відладнику.
 - Можна використовувати форматований рядок.

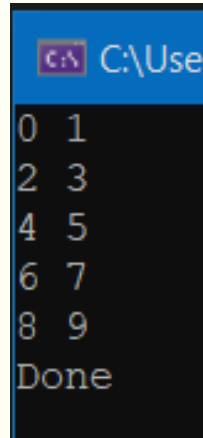
Контрольные значения 1			▼	🔍	✕
Поиск (Ctrl+E)			🔍	↶ ↷	Глубина поиска: 3
Имя	Значение	Тип			
▶ block1	Fu, Count=0	System.Threading.Tasks.Dataflow.BufferBlock<int>			
▶ block2	Bar, Class: BufferBlock`1, Id: 3, Count=0	System.Threading.Tasks.Dataflow.BufferBlock<int>			

Групуючі (grouping) блоки. BatchBlock<T>

```
static public void Run()
{
    var batchBlock = new BatchBlock<int>(2);
    for (int i = 0; i < 10; i++)
    {
        batchBlock.Post(i);
    }

    for (int i = 0; i < 5; i++)
    {
        int[] result = batchBlock.Receive();
        foreach (var r in result)
        {
            Console.Write(r + " ");
        }
        Console.WriteLine("\n");
    }
    Console.WriteLine("Done");
}
```

- Групуючі блоки можуть комбінувати частини даних у контейнер, зокрема List чи Tuple.
 - Інколи окремий елемент даних сам по собі беззмістовний, а має сенс лише в контексті колекції.
 - Групуючі блоки дозволяють збирати елементи даних разом та працювати з колекцією як єдиним цілим.
- Групування даних також може підвищити продуктивність, оскільки зменшуватиме синхронізаційні витрати на передачу елементів.
- BatchBlock<T> приймає потік (stream) даних та групує їх у список.
 - Розмір пакету (batch) задається при конструюванні та має бути 1 або вище; інакше в рантаймі викинеться виняток ArgumentException.



```
C:\Use
0 1
2 3
4 5
6 7
8 9
Done
```

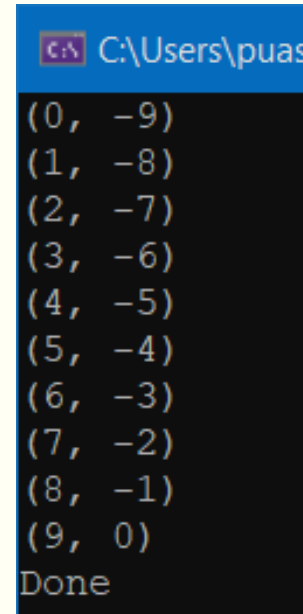
Групуючі (grouping) блоки. JoinBlock<T1,T2>

```
static public void Run()
{
    var jBlock = new JoinBlock<int, int>();
    for (int i = 0; i < 10; i++)
    {
        jBlock.Target1.Post(i);
    }

    for (int i = -9; i < 1; i++)
    {
        jBlock.Target2.Post(i);
    }

    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(jBlock.Receive());
    }
    Console.WriteLine("Done");
}
```

- JoinBlock<T1,T2> має 2 входи (Target1 і Target2), які комбінує в Tuple<T1,T2>.
 - Також існує JoinBlock<T1,T2,T3> на 3 входи та вихід у кортеж Tuple<T1,T2,T3>.



```
C:\Users\puass
(0, -9)
(1, -8)
(2, -7)
(3, -6)
(4, -5)
(5, -4)
(6, -3)
(7, -2)
(8, -1)
(9, 0)
Done
```

Групуючі (grouping) блоки. BatchedJoinBlock<T1,T2>

- BatchedJoinBlock<T1,T2> - комбінація JoinBlock<T1,T2> та BatchBlock<T>.
 - Оскільки JoinBlock повертає кортеж із 2х чи 3х входів, а BatchBlock<T> - погруповані у список вхідні значення, BatchedJoinBlock комбінує 2 чи 3 входи у Tuple<T1[],T2[]> чи Tuple<T1[],T2[],T3[]> відповідно.
- У той час, як JoinBlock повинен мати всі вхідні значення перед утворенням виходу, BatchedJoinBlock може формувати вихідні значення навіть при недоступності вхідних.
 - Все ж, він не генеруватиме вихід, поки не буде досягнуто заданий розмір пакету (batch) – загальної кількості елементів у **всіх** списках.
 - Розмір пакету повинен бути 1+, інакше в рантаймі викидатиметься виняток `ArgumentOutOfRangeException`.

Групуючі (grouping) блоки. BatchedJoinBlock<T1,T2>

```
static public void Run()
{
    var ListToString = new Func<IList<int>, string>(lst =>
    {
        if (lst.Count == 0)
        {
            return "[]";
        }

        var result = new StringBuilder("[");

        foreach (int n in lst)
        {
            result.AppendFormat("{0},",
                                n.ToString(CultureInfo.InvariantCulture));
        }

        result.Remove(result.Length - 1, 1);
        result.Append("]");

        return result.ToString();
    });
```

```
var TupleListToString = new Func<Tuple<IList<int>, IList<int>>,
                                string>(tup => {
    var result = new StringBuilder();

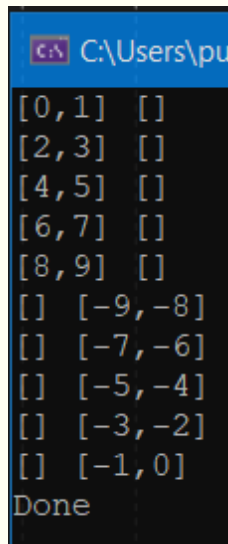
    result.Append(ListToString(tup.Item1));
    result.Append(" ");
    result.Append(ListToString(tup.Item2));

    return result.ToString();
});

var jBlock = new BatchedJoinBlock<int, int>(2);
for (int i = 0; i < 10; i++)
    jBlock.Target1.Post(i);

for (int i = -9; i < 1; i++)
    jBlock.Target2.Post(i);

for (int i = 0; i < 10; i++)
    Console.WriteLine(TupleListToString(jBlock.Receive()));
Console.WriteLine("Done");
}
```



```
C:\Users\pu
[0,1] []
[2,3] []
[4,5] []
[6,7] []
[8,9] []
[] [-9,-8]
[] [-7,-6]
[] [-5,-4]
[] [-3,-2]
[] [-1,0]
Done
```

- У прикладі дані синхронно надсилаються в один блок за один раз.
 - Делегати ListToString, TupleListToString додано з метою виводу кортежу списків, утвореного викликом jBlock.Receive().

Групуючі (grouping) блоки. BatchedJoinBlock<T1,T2>

```
var MakeDelayBlock = new Func<int, TransformBlock<int, int>>(maxDelay =>
{
    var generator = new Random();

    return new TransformBlock<int, int>(n => {
        Thread.Sleep(generator.Next(maxDelay));
        return n;
    });
});
```

```
var bjBlock = new BatchedJoinBlock<int, int>(2);
var delayBlock1 = MakeDelayBlock(1000);
var delayBlock2 = MakeDelayBlock(1000);
for (int i = 0; i < 10; i++)
{
    delayBlock1.SendAsync(i);
    delayBlock2.SendAsync(i - 2 * i); // те ж число, але від'ємне
}
```

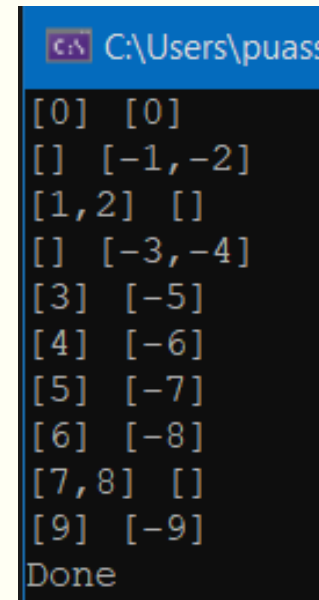
```
delayBlock1.LinkTo(bjBlock.Target1);
delayBlock2.LinkTo(bjBlock.Target2);
```

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(TupleListToString(bjBlock.Receive()));
```

```
Console.WriteLine("Done");
```

■ Приклад імітує асинхронне надходження даних у bjBlock.

- Створюємо 2 TransformBlock-об'єкта, які передають незмінені дані після деякої випадкової затримки.
- Обидва прив'язані (linked) до bjBlock.



```
C:\Users\puass
[0] [0]
[] [-1,-2]
[1,2] []
[] [-3,-4]
[3] [-5]
[4] [-6]
[5] [-7]
[6] [-8]
[7,8] []
[9] [-9]
Done
```

Групуючі блоки. Опції GroupingDataflowBlockOptions

- Існує 2 режими операції для JoinBlock: жадібний (greedy) та нежадібний (non-greedy).
- У **жадібному режимі** блок приймає всі вхідні значення, які йому пропонуються, навіть якщо він не може утворити кортеж.
 - Якщо блоку пропонується значення лише для Target1, він прийме це значення та очікуватиме на надходження даних для Target2 перед тим, як сформувати вихід Tuple<T1,T2>.
 - Якщо інше значення знову пропонується для Target1, воно знову буде прийняте, незважаючи на відсутність значення для Target2.
- У **нежадібному режимі** блок приймає значення за умови очікування на прийняття даних для Target1 і Target2.
 - Нежадібний режим може застосовуватись для уникнення взаємоблокувань: як тільки блок приймає повідомлення, жодний інший блок не може отримати це повідомлення.
 - Тому для двох JoinBlocks з їх Target1, зв'язаними з одним виходом та їх Target2, зв'язаними з одним виходом (проте не таким, як для Target1) може виникнути дедлок.
 - Один JoinBlock може отримати один з виходів, а інший JoinBlock – інший вихід: жодному з цих блоків не вистачатиме даних для вироблення вихідних значень у кортеж.
 - Налаштування нежадібного режиму для обох JoinBlock-ів забезпечить, щоб лише один отримав обидва виходи та зміг прогресувати.

Групуючі блоки. Опції GroupingDataflowBlockOptions

```
static public void Run()
{
    var opts = new GroupingDataflowBlockOptions { Greedy = false };
    var jBlock = new JoinBlock<int, int>(opts);

    for (int i = 0; i < 10; i++)
    {
        if (jBlock.Target1.Post(i))
            Console.WriteLine("Target1 accepted: " + i);
        else
            Console.WriteLine("Target1 REFUSED: " + i);
    }

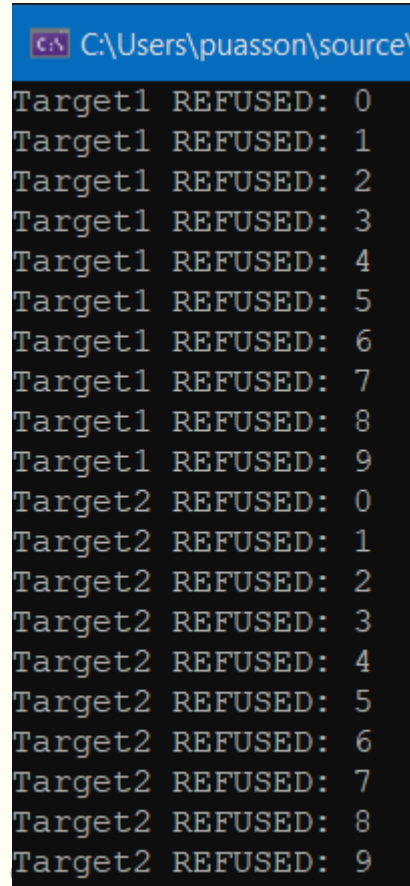
    for (int i = 0; i < 10; i++)
    {
        if (jBlock.Target2.Post(i))
            Console.WriteLine("Target2 accepted: " + i);
        else
            Console.WriteLine("Target2 REFUSED: " + i);
    }

    for (int i = 0; i < 10; i++)
        Console.WriteLine(jBlock.Receive());

    Console.WriteLine("Done");
}
```

- Приклад демонструє нежадібну поведінку.

- Оскільки використовуємо синхронний метод Post(), при спробі надіслати дані для Target1 у першому циклі for буде відмова – jBlock не має доступних даних і для Target2.
- Те ж і з другим циклом for: JoinBlock відмовляє всім даним, які ми намагались передати в нього у першому циклі for.



```
C:\Users\puasson\source\
Target1 REFUSED: 0
Target1 REFUSED: 1
Target1 REFUSED: 2
Target1 REFUSED: 3
Target1 REFUSED: 4
Target1 REFUSED: 5
Target1 REFUSED: 6
Target1 REFUSED: 7
Target1 REFUSED: 8
Target1 REFUSED: 9
Target2 REFUSED: 0
Target2 REFUSED: 1
Target2 REFUSED: 2
Target2 REFUSED: 3
Target2 REFUSED: 4
Target2 REFUSED: 5
Target2 REFUSED: 6
Target2 REFUSED: 7
Target2 REFUSED: 8
Target2 REFUSED: 9
```

Групуючі блоки. Опції GroupingDataflowBlockOptions

```
var opts = new GroupingDataflowBlockOptions { Greedy = false };
var jBlock = new JoinBlock<int, int>(opts);

for (int i = 0; i < 10; i++)
{
    Task<bool> task = jBlock.Target1.SendAsync(i);

    // needed to capture 'i' so we can use it in `ContinueWith`
    int iCopy = i;

    task.ContinueWith(t => {
        if (t.Result)
        {
            Console.WriteLine("Target1 accepted: " + iCopy);
        }
        else
        {
            Console.WriteLine("Target1 REFUSED: " + iCopy);
        }
    });
}
```

■ Для запуску додатку необхідно внести кілька змін.

- Спочатку замінимо синхронний метод Post() на асинхронний SendAsync().
- Щоб визначити, чи приймаються дані блоком jBlock, візьмемо Task-об'єкт, повернений SendAsync(), та додамо продовжуючу задачу після завершення Task.

Продовження коду

```
for (int i = 0; i < 10; i++)
{
    Task<bool> task = jBlock.Target2.SendAsync(i);
    // needed to capture 'i' so we can use it in `ContinueWith`
    int iCopy = i;

    task.ContinueWith(t => {
        if (t.Result)
        {
            Console.WriteLine("Target2 accepted: " + iCopy);
        }
        else
        {
            Console.WriteLine("Target2 REFUSED: " + iCopy);
        }
    });
}

for (int i = 0; i < 10; i++)
{
    Console.WriteLine(jBlock.Receive());
}

Console.WriteLine("Done");
```

- Тепер все ще можемо надсилати дані одному target-у за раз у двох циклах for.

- Проте оскільки SendAsync() очікуватиме, поки jBlock не прийме дані, вивід у консолі буде приблизно такий:

```
C:\Users\puasson\source\rep
Target1 accepted: 0
Target2 accepted: 1
Target1 accepted: 1
Target1 accepted: 9
(0, 0)
Target2 accepted: 2
Target2 accepted: 8
Target1 accepted: 3
Target1 accepted: 2
(1, 1)
Target2 accepted: 4
Target2 accepted: 3
(2, 2)
Target1 accepted: 8
(3, 3)
Target2 accepted: 7
Target1 accepted: 4
Target1 accepted: 5
Target2 accepted: 6
Target1 accepted: 7
(4, 4)
(5, 5)
(6, 6)
(7, 7)
(8, 8)
(9, 9)
Done
Target2 accepted: 5
Target1 accepted: 6
```

Завершення роботи блоку (Block Completion)

```
static public void Run()
{
    Action<int> fn = n => {
        Thread.Sleep(1000);
        Console.WriteLine(n);
    };

    var actionBlock = new ActionBlock<int>(fn);

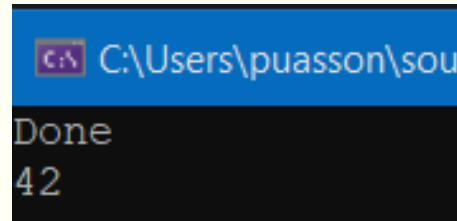
    actionBlock.Post(42);
    actionBlock.Complete();

    for (int i = 0; i < 10; i++)
    {
        actionBlock.Post(i);
    }

    // Хоч ми надсилаємо блоку всі дані, він виконається
    // лише раз, оскільки вказуємо йому "Complete"

    Console.WriteLine("Done");
}
```

- Можна вказати блоку зупинити обробку даних за допомогою методу Complete().
 - У прикладі значення 42 надсилається блоку, а потім він зупиняється, оскільки в наступному рядку викликається actionBlock.Complete().



Завершення роботи блоку (Block Completion)

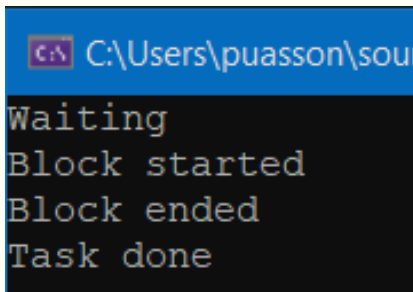
```
static public void Run()
{
    var block = new ActionBlock<bool>(_ => {
        Console.WriteLine("Block started");
        Thread.Sleep(5000);
        Console.WriteLine("Block ended");
    });

    block.Post(true);

    Console.WriteLine("Waiting");

    block.Complete();
    block.Completion.Wait();

    Console.WriteLine("Task done");
}
```



```
C:\Users\puasson\source>
Waiting
Block started
Block ended
Task done
```

- Крім цього, також можемо очікувати завершення обробки даних блоком.
 - Метод Completion() повертає Task, який представляє стан завершення обробки в блоці.
 - Поширений спосіб призупинити поточний потік, поки блок не завершить обробку, - викликати метод Wait() для поверненої задачі.
- У прикладі блок виведе “Block started” upon entry, очікуватиме 5 секунд, а потім виведе “Block ended”.
 - Ми надсилаємо значення true блоку, щоб активувати його, а потім негайно викликаємо block.Complete(), щоб вказати, що блоку слід перейти у стан «complete», як тільки він завершить поточну роботу.
 - Головний потік призупиняється, поки блок не завершить роботу за допомогою інструкції block.Completion.Wait().

Зв'язки (links)

- Зв'язки – це засіб комунікації між блоками.
 - Блок може з'єднуватись з іншими блоками за допомогою методу `LinkTo()` як статичний член класу `DataflowBlock` або метод екземпляру довільного блоку, що реалізує інтерфейс `ISourceBlock`.
 - TPL Dataflow передає дані, застосовуючи передачу повідомлень (message passing).
- Коли кілька блоків з'єднані з виходом, повідомлення надсилається кожному блоку в порядку їх зв'язування.
 - Перший блок, який приймає повідомлення, призводить до видалення цього повідомлення з вихідної черги (output queue) блоку-джерела, тому іншим блокам це ж повідомлення не пропонується.
 - Виняток з правила – `BroadcastBlock<T>` - спроектований для надсилання повідомлення всім пов'язаним блокам.
- Коли кілька блоків зв'язано зі входом, повідомлення з усіх зв'язків об'єднуються (merge) in a time-order fashion та представляються входу блоку як єдиний вхідний зв'язок.
- Повідомлення передаються впорядковано.
 - Для блоку неможливо випадково виділити повідомлення зі зв'язку – лише наступне доступне повідомлення.
 - Якщо повідомлення неможливо відправити, тоді жодне інше повідомлення теж не можна передати по цьому зв'язку, тому можливе взаємоблокування.
 - У такому випадку використовуйте `BroadcastBlock<T>`: всередині блоку повідомлення можуть оброблятися неупорядковано через паралелізм чи by design, проте рекомендується впорядкована обробка.

Зв'язки (links). Метод LinkTo()

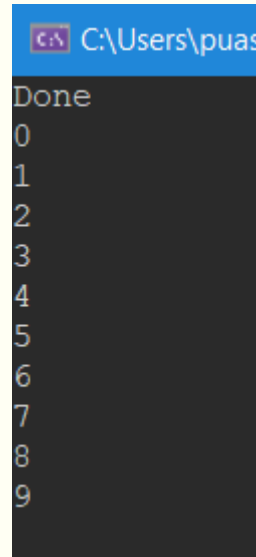
```
static public void Run()
{
    var bufferBlock = new BufferBlock<int>();
    var printBlock = new ActionBlock<int>(n =>
        Console.WriteLine(n));

    for (int i = 0; i < 10; i++)
    {
        bufferBlock.Post(i);
    }

    bufferBlock.LinkTo(printBlock);

    Console.WriteLine("Done");
}
```

- Оскільки зв'язки не є окремими сутностями від блоків у TPL Dataflow, протокол передачі повідомлень контролюється блоками.
 - Певні блоки, зокрема BroadcastBlock<T>, можуть мати різні процедури передачі повідомлень.
 - Розглянутий раніше протокол є загальним способом обробки повідомлень для більшості блоків.
- Тут показано найбільш базовий та поширений спосіб зв'язування 2 блоків: всі значення з bufferBlock передаються в printBlock через зв'язок.



Зв'язки (links). Кілька отримувачів (receivers)

```
static public void Run()
{
    var MakePrintBlock = new Func<string,
        ActionBlock<int>>(prefix => {
        return new ActionBlock<int>(n =>
            Console.WriteLine(prefix + ": " + n));
    });

    var printBlock1 = MakePrintBlock("printBlock1");
    var printBlock2 = MakePrintBlock("printBlock2");

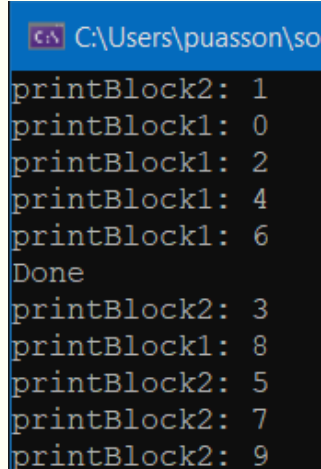
    var bufferBlock = new BufferBlock<int>();

    bufferBlock.LinkTo(printBlock1, n => n % 2 == 0);
    bufferBlock.LinkTo(printBlock2);

    for (int i = 0; i < 10; i++)
    {
        bufferBlock.Post(i);
    }

    Console.WriteLine("Done");
}
```

- Приклад демонструє, як типовий блок намагається доставити повідомлення кільком отримувачам: блоки printBlock1 та printBlock2 зв'язані з bufferBlock – джерелом даних.
 - Додано фільтр на зв'язок з printBlock1, щоб пропускати лише парні числа.
 - Оскільки першим встановлено зв'язок з printBlock1, bufferBlock спочатку спробує доставити повідомлення йому.
 - Предикат-фільтр блокує всі непарні числа від надсилання блоку printBlock1, тому bufferBlock потім намагається надіслати їх у printBlock2 та наступні зв'язані блоки.



```
C:\Users\puasson\so
printBlock2: 1
printBlock1: 0
printBlock1: 2
printBlock1: 4
printBlock1: 6
Done
printBlock2: 3
printBlock1: 8
printBlock2: 5
printBlock2: 7
printBlock2: 9
```

Зв'язки (links). Кілька джерел (sources)

```
static public void Run()
{
    var MakeDelayBlock = new Func<int,
        TransformBlock<int, int>>(delay => {
        var generator = new Random();

        return new TransformBlock<int, int>(n =>
        {
            Thread.Sleep(generator.Next(delay));

            return n;
        });
    });

    var source1 = MakeDelayBlock(1000);
    var source2 = MakeDelayBlock(800);

    var printBlock = new ActionBlock<int>(n =>
        Console.WriteLine(n));

    for (int i = 0; i < 10; i++)
    {
        source1.Post(i);
        source2.Post(i - 2 * i); // negate i
    }

    source1.LinkTo(printBlock);
    source2.LinkTo(printBlock);
}
```

15.03.2021

■ Приклад показує, що робиться, коли вхід блоку зв'язаний з кількома блоками-джерелами.

- sourceBlock1 та sourceBlock2 передаватимуть дані **through them unaltered** після випадкової затримки.
- Заповнимо sourceBlock1 невід'ємними числами від 0 до 9, а sourceBlock2 – недодатніми числами від 0 до -9, щоб уточнити, звідки надходять дані.
- Обидва блоки-джерела зв'язані зі входом printBlock, який просто виводить значення входу в консоль.
- З виводу видно, що, що значення від обох блоків-джерел перемішуються згідно з часом надходження на вхід printBlock.

Зв'язки (links). Фільтр у LinkTo()

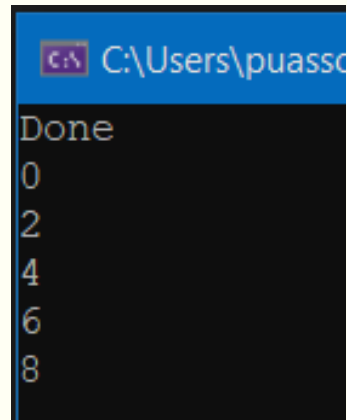
```
static public void Run()
{
    var sourceBlock = new BroadcastBlock<int>(n => n);

    // використовуємо BroadcastBlock, щоб відкидати
    // невикористані повідомлення
    var printBlock = new ActionBlock<int>(n =>
        Console.WriteLine(n));

    // надсилає лише парні числа в printBlock
    sourceBlock.LinkTo(printBlock, n => n % 2 == 0);

    for (int i = 0; i < 10; i++)
    {
        sourceBlock.SendAsync(i);
    }

    Console.WriteLine("Done");
}
```



- Зв'язки також фільтрують повідомлення, які по них проходять.
 - Метод розширення `LinkTo(target, predicate)` дозволяє присвоїти зв'язку функцію-предикат.
 - Кожне повідомлення перевіряється на відповідність предикату та при задоволенні умови передається зв'язаному блоку.
- TPL Dataflow забезпечує, щоб повідомлення оброблялись впорядковано.
 - Якщо зв'язок фільтрує повідомлення, а інших зв'язків для відправки немає, система взаємоблокується.
 - Для уникнення такої ситуації тут застосовуємо `BroadcastBlock<T>` у якості джерела: він надсилає лише найбільш свіже повідомлення, а відкинуті повідомлення просто ігноруються.

Зв'язки (links). Фільтр у LinkTo() та NullTarget<T>

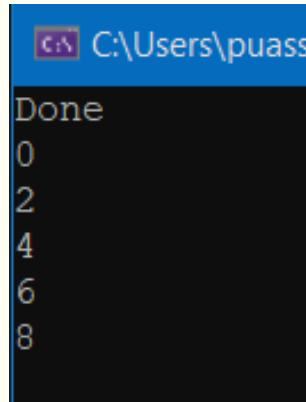
```
static public void Run()
{
    var sourceBlock = new BufferBlock<int>();
    var printBlock = new ActionBlock<int>(n =>
        Console.WriteLine(n));

    // надсилає лише парні числа в printBlock
    sourceBlock.LinkTo(printBlock, n => n % 2 == 0);

    // NullTarget відкидає всі отримані повідомлення
    sourceBlock.LinkTo(DataflowBlock.NullTarget<int>());

    for (int i = 0; i < 10; i++)
    {
        sourceBlock.SendAsync(i);
    }

    Console.WriteLine("Done");
}
```



- Тут застосовуємо BroadcastBlock<T>, щоб відкинути повідомлення, які неможливо передати через фільтр зв'язка.
 - Часто потрібно використовувати фільтровані зв'язки від блоків, які не є BroadcastBlock.
 - Хоч ми завжди можемо вставляти BroadcastBlock<T>, додаткова обробка додаватиметься до накладних витрат і сповільнюватиме програму.
- Приклад показує кращий спосіб: додаємо зв'язок, пов'язаний з NullTarget<T>-блоком зі статичного класу DataflowBlock, після фільтрованого зв'язку.
 - NullTarget<T> відкидає всі повідомлення, які йому надходять.
 - Оскільки ми додаємо зв'язок після фільтрованого зв'язку, sourceBlock намагатиметься надіслати всі повідомлення спочатку блоку printBlock, а решту повідомлень – блоку NullTarget<T>.

Зв'язки (links). Опції DataflowLinkOptions

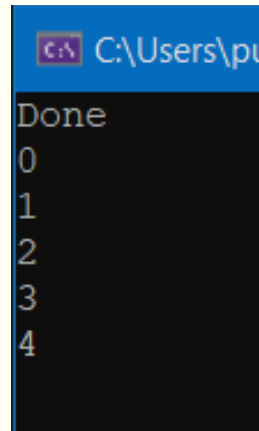
```
static public void Run()
{
    var bufferBlock = new BufferBlock<int>();
    var printBlock = new ActionBlock<int>(n => Console.WriteLine(n));

    for (int i = 0; i < 10; i++)
    {
        bufferBlock.Post(i);
    }

    var opts = new DataflowLinkOptions { MaxMessages = 5 };
    bufferBlock.LinkTo(printBlock, opts);

    Console.WriteLine("Done");
}
```

- Зв'язки можна налаштувати, щоб передавати лише визначену кількість повідомлень протягом існування цих зв'язків.
 - У прикладі задається поле MaxMessages об'єкту DataflowLinkOptions.
 - Тоді при спробі передати 10 значень по зв'язку (числа від 0 до 9) будуть надіслані лише перші п'ять.



Зв'язки (links). Опції DataflowLinkOptions

```
static public void Run()
{
    var MakePrintBlock = new Func<string,
        ActionBlock<int>>(prefix => {
        return new ActionBlock<int>(n =>
            Console.WriteLine(prefix + ": " + n));
    });

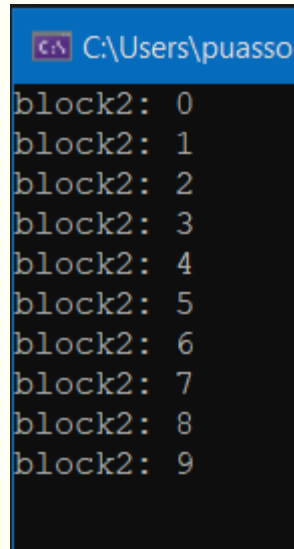
    var block1 = MakePrintBlock("block1");
    var block2 = MakePrintBlock("block2");

    var source = new BufferBlock<int>();

    source.LinkTo(block1);

    var opt = new DataflowLinkOptions { Append = false };
    source.LinkTo(block2, opt);

    for (int i = 0; i < 10; i++)
    {
        source.SendAsync(i);
    }
}
```



```
C:\Users\puasso
block2: 0
block2: 1
block2: 2
block2: 3
block2: 4
block2: 5
block2: 6
block2: 7
block2: 8
block2: 9
```

- За умовчанням блоки намагаються передавати повідомлення по першому доданому зв'язку.
 - Якщо це неможливо, зв'язки перевіряються по черзі, поки повідомлення не буде прийняте.
 - Кожний виклик LinkTo() додає новий зв'язок до відповідної колекції.
 - Задаючи Append = false в об'єкті DataflowLinkOptions та передаючи його методу LinkTo(), можемо здійснити реверс та додавати зв'язок на початок колекції.
 - Майбутні повідомлення будуть надсилатись спершу через цей зв'язок.
- У коді block1 зв'язаний першим, а block2 – другим.
 - Проте при виклику LinkTo() з append-опцією block2 додається на початок колекції та отримуватиме всі повідомлення.

Рецепт 1: зв'язування блоків

- За умовчанням зв'язані блоки лише розповсюджують дані; вони не розповсюджують завершення або помилки.
 - Якщо ваш потік даних лінійний (наприклад, у конвеєрі), то, скоріше за все, ви захочете розповсюджувати завершення / помилки. Для цього встановіть параметр `PropagateCompletion` для зв'язку:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

var options = new DataflowLinkOptions { PropagateCompletion = true };
multiplyBlock.LinkTo(subtractBlock, options);

...

// Завершение первого блока автоматически распространяется во второй блок.
multiplyBlock.Complete();
await subtractBlock.Completion;
```

Рецепт 1: зв'язування блоків

- Після зв'язування дані будуть автоматично переміщатись від блоку-джерела до блоку-приймача.
 - Параметр `PropagateCompletion` переміщує не лише дані, але й завершення; проте на кожному етапі конвеєру збійний блок буде розповсюджувати в наступний блок свій виняток, упакований в `AggregateException`.
 - Таким чином, якщо маємо довгий конвеєр, що розповсюджує завершення, початкова помилка може вкладатись у кілька екземплярів `AggregateException`.
 - `AggregateException` має кілька методів (наприклад, `Flatten`), які спрощують обробку помилок у таких ситуаціях.
- Блоки потоку даних можуть зв'язуватись один з одним різними способами; сітка може містити галуження, з'єднання та навіть цикли.
 - Для більшості сценаріїв зазвичай вистачає простого лінійного конвеєра.
- Тип `DataflowLinkOptions` надає ряд параметрів, які можуть встановлюватись для зв'язків (наприклад, вже використаний `PropagateCompletion`), а перевантажена версія `LinkTo()` також може отримувати фільтруючий предикат.

Рецепт 2: розповсюдження помилок

- **Задача:** знайти спосіб реагувати на помилки, які можуть відбуватись у сітці потоку даних.
- **Вирішення:** якщо делегат, переданий блоку, викидає виняток, то цей блок входить у стан відмови: втрачає всі свої дані та перестає приймати нові.
 - У коді блок не виробляє вихідних даних; перше значення видає виняток, а друге - втрачається:

```
var block = new TransformBlock<int, int>(item =>
{
    if (item == 1)
        throw new InvalidOperationException("Blech.");

    return item * 2;
});
block.Post(1);
block.Post(2);
```

Рецепт 2: розповсюдження помилок

- Щоб перехоплювати винятки від блоку, необхідно очікувати його властивість `Completion`.
 - Воно повертає `Task`-об'єкт, який завершується при завершенні блоку.
 - Якщо у блоці відбувається відмова, то і в задачі `Completion` вона теж стається:

```
try
{
    var block = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
    });
    block.Post(1);
    await block.Completion;
}
catch (InvalidOperationException)
{
    // Здесь перехватывается исключение.
}
```

Рецепт 2: розповсюдження помилок

```
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);
```

100 Глава 5. Основы Dataflow

```
multiplyBlock.LinkTo(subtractBlock,
    new DataflowLinkOptions { PropagateCompletion = true });
multiplyBlock.Post(1);
await subtractBlock.Completion;
}
catch (AggregateException)
{
    // Здесь перехватывается исключение.
}
```

- При розповсюдженні завершення за допомогою параметра зв'язку `PropagateCompletion` помилки теж розповсюджуються.
 - Проте виняток передається наступному блоку, упакованому в `AggregateException`.
 - У коді виняток перехоплюється в кінці конвеєра, тому він перехопить `AggregateException`, якщо виняток розповсюджувався з попередніх блоків.
- Кожний блок упаковує вхідні помилки в `AggregateException`, навіть якщо наступна помилка вже являє собою `AggregateException`.
 - Якщо помилка відбувається на ранній стадії конвеєра та встигає переміститись на кілька зв'язків уперед до її виявлення, початкова помилка буде упакована в `AggregateException` на кількох рівнях.
 - Метод `AggregateException.Flatten()` спрощує обробку помилок у такому сценарії.

Рецепт 2: розповсюдження помилок

- Будуючи сітку (або конвеєр), подумайте над тим, як будуть оброблятися помилки.
 - У простих ситуаціях може бути краще розповсюджувати помилки та перехоплювати їх усі в кінці.
 - У більш складних, можливо, доведеться переірити кожний блок при завершенні потоку даних.
- Можливий інший варіант: для збереження працездатності блоків розглядати винятки як інший різновид даних та дати їм проходи по сітці з елементами даних, що правильно обробляються.
 - Це дозволить зберегти працездатність сітки потоку даних, оскільки блоки не будуть переходити в стан відмови та продовжуватимуть обробку наступного елементу даних.

Рецепт 3: видалення зв'язків між блоками

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
IDisposable link = multiplyBlock.LinkTo(subtractBlock);
multiplyBlock.Post(1);
multiplyBlock.Post(2);
// Удаление связей между блоками.
// Данные, отправленные выше, могут быть уже переданы
// или не переданы по связи. В реальном коде стоит рассмотреть
// возможность блока using вместо простого вызова Dispose.
link.Dispose();
```

- Сценарій корисний, коли неможливо змінити фільтр для зв'язку — доведеться видалити старий зв'язок та створити новий з новим фільтром (можливо, з `DataflowLinkOptions.Append = false`).
- Також може використовуватись в стратегічній точці для призупинки сітки потоку даних.

- **Задача:** у ході обробки необхідно динамічно змінити структуру потоку даних. Це нетиповий сценарій.
- **Вирішення:** зв'язки між блоками потоку даних можна створювати або видаляти в будь-який момент;
 - Створення та видалення зв'язків є повністю потокобезпечними.
 - При створенні зв'язку між блоками збережіть об'єкт `IDisposable`, повернений методом `LinkTo()` та знищіть його, коли потрібно буде розірвати зв'язок між блоками.
- Якщо немає гарантії бездіяльності зв'язку, при його видаленні може виникнути стан гонитви.
 - Зазвичай він не створює проблем; дані або проходять по зв'язку перед його видаленням, або не проходять.
 - Немає умов гонитви, що призведуть до дублювання чи втрати даних.

Рецепт 4: створення власних блоків

- **Задача:** маємо деяку логіку, яку слід розмістити в нестандартному блоці потоку даних.
 - Це дозволить створювати великі блоки, що містять складну логіку.
- **Вирішення:** можна виділити довільну частину сітки потоку даних, що містить один вхідний та один вихідний блок за допомогою методу `Encapsulate()`.
 - Розповсюдження даних та завершення між цими кінцевими точками залишаються на совісті розробника.
 - Створимо з 2 блоків нестандартний блок потоку даних з розповсюдженням даних і завершення:

```
IPropagatorBlock<int, int> CreateMyCustomBlock()
{
    var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
    var addBlock = new TransformBlock<int, int>(item => item + 2);
    var divideBlock = new TransformBlock<int, int>(item => item / 2);

    var flowCompletion = new DataflowLinkOptions { PropagateCompletion = true };
    multiplyBlock.LinkTo(addBlock, flowCompletion);
    addBlock.LinkTo(divideBlock, flowCompletion);

    return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```


Рецепт 4: створення власних блоків

- Інкапсулюючи сітку в нестандартний блок, подумайте про те:
 - які параметри потрібно надати користувачам.
 - як кожний параметр блоку повинен (чи ні) передаватись вашій внутрішній мережі; часто деякі параметри блоків незастосовні або не мають сенсу, тому нестандартні блоки зазвичай визначають власні нестандартні параметри замість того, щоб отримувати параметр `DataflowBlockOptions`.
- `DataflowBlock.Encapsulate` інкапсулює лише сітку з одним вхідним та одним вихідним блоками.
 - Якщо якихось блоків більше, слід інкапсулювати цю сітку в спеціальному об'єкті та надати доступ до входів і виходів як до властивостей типу `ITargetBlock<T>` (для входів) і `IReceivableSourceBlock<T>` (для виходів).
 - Також можливо реалізувати інтерфейси потоку даних самотійно, проте це набагато складніше.
 - Компанія Microsoft опублікувала [статтю](#) з описом нетривіальних прийомів створення нестандартних блоків потоку даних.

Рецепт 5: модульне тестування сіток потоків даних

```
[TestMethod]
public async Task MyCustomBlock_AddsOneToDataItems()
{
    var myCustomBlock = CreateMyCustomBlock();
    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Complete();
    Assert.AreEqual(4, myCustomBlock.Receive());
    Assert.AreEqual(14, myCustomBlock.Receive());
    await myCustomBlock.Completion;
}
```

- **Задача:** в додатку існує сітка потоку даних. Потрібно переконатись у правильності її роботи.
- **Вирішення:** Сітки потоків даних незалежні, мають власний термін життя й асинхронні за своєю природою.
 - Таким чином, найпростіший підхід їх тестування — асинхронні модульні тести.
 - Представлений модульний тест перевіряє нестандартний блок потоку даних з попереднього рецепту.

Рецепт 7: модульне тестування сіток потоків даних

```
[TestMethod]
public async Task MyCustomBlock_Fault_DiscardsDataAndFaults()
{
    var myCustomBlock = CreateMyCustomBlock();
    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    (myCustomBlock as IDataflowBlock).Fault(new
        InvalidOperationException());
    try
    {
        await myCustomBlock.Completion;
    }
    catch (AggregateException ex)
    {
        AssertExceptionIs<InvalidOperationException>(
            ex.Flatten().InnerException, false);
    }
}

public static void AssertExceptionIs<TException>(Exception ex,
    bool allowDerivedTypes = true)
{
    if (allowDerivedTypes && !(ex is TException))
        Assert.Fail($"Exception is of type {ex.GetType().Name}, but " +
            $"{typeof(TException).Name} or a derived type was expected.");
    if (!allowDerivedTypes && ex.GetType() != typeof(TException))
        Assert.Fail($"Exception is of type {ex.GetType().Name}, but " +
            $"{typeof(TException).Name} was expected.");
}
```

- Модульне тестування відмов не таке прямолінійне.
 - Винятки в сітках потоків даних упаковуються в нову обертку `AggregateException` кожен раз, коли вони розпространяються в наступний блок.
 - В прикладі використовується допоміжний метод для перевірки того, чи виключення відкидає дані і розпространяється через нестандартний блок.
 - Пряме модульне тестування мереж потоків даних можливо, але кілька незручно.
 - Якщо ваша мережа є частиною більшого компонента, можливо, буде простіше організувати модульне тестування для більшого компонента (тоді мережа буде тестуватися неявно).
 - Але якщо ви розробляєте нестандартний блок або мережу для повторного використання, використовуйте модульні тести на зразок описаних у цьому рецепті.



ДЯКУЮ ЗА УВАГУ!

Наступна тема: Додаткові питання конкурентного виконання коду

Використання TPL Dataflow. Як передаються повідомлення?

- Many factors affect how, when and if messages are transmitted.
 - For the most part, all predefined blocks transmit messages in the same way but some, like `BroadCastBlock` may do things slightly different.
- Blocks attempt to transmit messages through links in the order they were added.
 - Most blocks attempt to send messages to only one target block.
 - For those cases, the block must decide in what order the target blocks are offered the message.
 - The default is to offer blocks the message in the order they were added.
 - Using the `DataflowLinkOptions.Append` configuration can force that a new link is prepended and not appended to the collection of links.
 - `BroadcastBlocks` and `WriteOnceBlocks` are designed to send messages to all connected blocks so the order they were added is unimportant.
- Link filters can block a message before it gets to the block.
 - If a link was added with a filter predicate and the message causes the predicate to return true, then the message is blocked and next link in line is tried.

Використання TPL Dataflow. Як передаються повідомлення?

- Links that exceed MaxMessages block messages.
 - The DataflowLinkOptions.MaxMessages property can limit the total number of messages that a link will transmit over its lifetime.
 - If this limit has been reached, the message is blocked.
- Blocks can refuse messages.
 - If blocks are in the completed state, error state or its internal buffer is full, the block will refuse the message.
 - A WriteOnceBlock will refuse all messages except for the first it receives.
- Blocks can postpone messages.
 - Grouping blocks have the option to operate in non-greedy mode – it will only accept a message when all inputs have a message waiting.
 - In this case the block postpones the message.
 - Either another block will accept the message or the block that postponed it can accept it later.
- A non-deliverable message can cause a deadlock.
 - TPL Dataflow will always transmit messages in order. For all blocks except for BroadcastBlock and WriteOnceBlock a message that can't be transmitted causes the block deadlock and wait until it can be delivered.
 - A DataflowBlock.NullTarget is useful in this case.

Внесення змін у ході виконання

- Blocks can be added and removed at runtime.
 - Adding a link at runtime is just calling the LinkTo() method as normal.
 - Unlinking a block can be accomplished by using the IDisposable object that is returned when you initially linked the block.
 - Simply call Dispose() on the IDisposable to remove the link.
- Use this ability with caution.
 - With extensive use in various parts of your application you can never be sure what is connected to what and makes debugging frustratingly difficult.
 - Modifying the delegate used in execution blocks at runtime is not possible.
 - However it is possible to design a delegate that chooses among a few different functions based upon some criteria.
 - Another option is to just replace the whole block by unlinking the old one and linking a new one.
 - But this also changes the order of links and thus messages transmittal will happen in a different order.
 - You'll need to determine on a case by case basis if this will cause problems.

Підтримка стану всередині виконавчого блоку

- In dataflow literature it is often recommended that blocks be stateless.
 - While you should strive for this goal, sometimes it is not possible.
 - Take for example an ActionBlock whose only purpose is to take its incoming data and add it to a collection.
 - The most obvious way to implement this is for the block to retain a reference to the collection from one execution to the next.
- At creation time, all execution blocks require a function or procedure that is called when the block receives data.
 - Most of the examples in this book use lambdas (a.k.a. anonymous functions) in the creation statement but a lambda cannot retain state from one invocation to the next.
 - The fix is to create a class that contains both the state and the lambda as a method of the class.
 - Before creating the block you will create an instance of the class you just defined and pass the method as a parameter of the block creation statement.
 - The method can access the state held within the object and so the block is able to maintain state.
 - For safety, this block should not set MaxDegreeOfParallelism greater than the default of 1 or you run the risk of multiple threads attempting to update the state at the same time.

Перетворення stateful-блоку в stateless-блок

- It is always possible to convert a stateful block to one that is stateless but it requires a block with multiple inputs and outputs.
 - Since the predefined blocks do not offer this ability you will also have to implement a custom block before you can use this technique.
- The concept is simple.
 - Instead of retaining state, you send the current state on an output that is linked to an input of the same block.
 - The block will still have whatever inputs and outputs needed for its normal operation but will now have one additional input and output.
 - Upon execution the block will get the current state and use it for the current set of input values possibly updating the state.
 - Once the block is done, it will send the new state to the output that is linked back to the input so it can use the new state with a new set of values.

Поради щодо проєктування програм з Dataflow

- Проєктуйте блоки для повторного використання.
 - One of the key aspects of dataflow is the ability to reuse existing blocks.
 - While reusable code has been preached for decades, it was the programming languages themselves that lead to non-reusable code.
 - You don't know if code is truly reusable until you have reused it at least three times.
 - Yet, re-usability is still something we should strive for and dataflow makes it easier.
 - Dataflow blocks should be free standing units.
 - Do not reference other blocks or anything outside of a block.
 - In one design I have seen, the author needed to ensure that both of the connected block were completed before the subject block performed some operation.
 - He directly set a continuation on the tasks of the two blocks to accomplish his goal.
 - While it solved his problem this time, there is no way he could reuse the block elsewhere.
 - If you find there is no way to solve your problem without referencing external entities, then maybe they belong inside the block you are designing.

Поради щодо проєктування програм з Dataflow

- Створюйте блоки з нуля лише при необхідності.
 - While it is certainly possible to build a new block simply from the core interfaces of TPL Dataflow (IDataflowBlock, ISourceBlock and ITargetBlock) it is rarely necessary.
 - Building new block by using the predefined blocks is much safer.
 - When building from scratch, you must ensure that the protocols are identical to the predefined blocks.
 - You can think of the predefined blocks to be equivalent to keywords in C#.
 - You use the keywords to build C# applications.
 - Similarly, you use the predefined blocks to build dataflow applications.
 - Creating a new block from scratch is akin to adding a new keyword to C# and as much care must be taken to ensure it operates correctly.
 - I recommend using the `DataflowBlock.Encapsulate<T1,T2>` method to build a new block with the same interface as the predefined blocks or to create your own, application specific, interface that all of your blocks will conform to.
 - The key is that we don't have to concern ourselves with the details of message passing, threading issues and other details when we don't have to.

Поради щодо проєктування програм з Dataflow

- Розробляйте власний Block-інтерфейс.
 - I often find myself developing my own interface that all of the blocks in my application implement.
 - Since the TPL Dataflow library makes no allowances for blocks that have multiple inputs and/or outputs, my interface mostly deals with making it easy to get access to the internal blocks that act as the input and output connections.
 - Usually I use a dictionary with a key of type string or GUID to store the 'BufferBlock's that are the "ports" of my application specific blocks.
 - All of my blocks implement this interface so that it is consistent throughout the application.
 - I will also wrap predefined blocks in this interface whenever I need to use them alongside my other application specific blocks, again for consistency.
 - The idea is to create a layer above what is offered by TPL Dataflow to assist in having blocks with multiple inputs and outputs and any other requirements I need for that specific application.
 - Keep it simple. Don't create a new messaging protocol or a new way to link blocks.
 - Reuse the basic operations that TPL Dataflow offers like LinkTo or SendAsync.
- Будьте обережними зі збереженням (Retaining) стану в блоках.
 - Blocks that retain state information from one activation to the next should be used carefully when MaxDegreeOfParallelism is set to anything more than the default of 1.
 - This is because a single block will have a single state for all threads.
 - If the block is used in a multithreaded manner, there is a danger of multiple threads updating the state at the same time.
 - Additionally, this makes the block less reusable because you have to ensure that all future users of the block understand the dangers of using the block in a multithreaded manner

Поради щодо проєктування програм з Dataflow

- **Favor Application Specific Blocks Over Predefined Blocks.**
 - For anything more than a trivial use of the TPL Dataflow library, your application should only use blocks that implement your custom interface (as detailed above).
 - Even when you only need a BufferBlock you should wrap it in your own interface to maintain consistency.
 - The TPL Dataflow Library should be thought of as the tools to build your house not the doors and windows that comprise your house.
 - The library was designed to offer the needed abstractions for a variety of dataflow situations. It wasn't designed with your application in mind.
 - Many uses of dataflow require blocks to have many inputs and output, there may be application requirements that some blocks take priority over others or you may want to make it easy to compose new block by using other blocks.
- **Avoid Excess Synchronization and Blocks.**
 - Every link requires some overhead to send/receive messages.
 - Eliminate unnecessary blocks to reduce the synchronization requirements.
 - A BufferBlock feeding into an ActionBlock could be redundant depending on the situation.
 - Remember that most predefined blocks contain their own buffer.
 - While I have preached making small reusable blocks, sometimes it is better to combine the functionality of a few connected blocks into one for the sake of performance.
 - This reduces the amount of synchronization needed to transmit data between the blocks and the number of Tasks needed.
 - First design for re-usability, then if you need to speed things up, only then consider combining blocks.
 - TPL Dataflow is not designed for very "small" blocks (e.g. a single addition operation).
 - Medium to large blocks (about the size of a typical procedure or larger) give much better performance.

Поради щодо проєктування програм з Dataflow

- **Dealing with Loops and Cycles.**
 - Anytime you have a link from an output of a downstream block to the input of an upstream block, there is a possibility that the dataflow program could run forever without stopping.
 - This is not always a bad thing. Think of dataflow as being similar to electronic circuits.
 - Circuits are designed to operate continuously until the power is turned off.
 - In most engineering disciplines, the best creations are the one that run continuously for long periods of time without faults.
 - Don't be too concerned about designing dataflow applications that will terminate. If you need to stop them then you can use the `Complete()` method or a `CancellationToken`.
 - This takes a change of attitude for most developers who were taught that programs that do not terminate are wrong. Dataflow is just a different way to think about programming.
- **Prevent Large Buffers.**
 - You must be mindful of the rate of incoming data to the rate of processing that data in a block.
 - A high rate of incoming data combined with a slow process could create a large backlog of data in the buffer.
 - To reduce the likelihood of this situation either increase the `MaxDegreeOfParallelism` or add more blocks to process the incoming data.
 - But remember that these solutions could also cause the data to be returned in a different order than they arrived.
 - If maintaining order is required, consider using a `BatchBlock` to group together multiple data items to be processed at one time.

Поради щодо проєктування програм з Dataflow

- **Data Should be Immutable.**
 - Mutable data and parallelism go together like oil and water.
 - When using mutable data, anytime you have to split data between two blocks you will have to perform a deep-copy of the object. This is very costly in terms of time and memory.
 - Of course sometimes it is not possible to use immutable data. In those cases you should design your classes to be thread safe.
 - Use blocking collections, never refer to singletons or global variables and ensure that the platform functions you are using are also thread safe.
- **Use SingleProducerConstrained if Possible.**
 - In the previous chapter we covered how using the SingleProducerConstrained option can drastically speed up your applications when a block will only ever have one source block.
 - The creator of the TPL Dataflow Library showed how, in his benchmarks, it was able to increase performance by a factor of 3.
 - As there is no downside to applying this option, you should use it whenever possible.

Рецепт 4: регулювання блоків

- **Задача:** маємо сітку потоку даних із галуженням. Потрібно організувати передачу даних з розподілом навантаження.
- **Вирішення:** По умовчанию блок, генерирующий выходные данные, проверяет все свои связи (в порядке их создания) и пытается последовательно передавать данные по каждому каналу.
 - Кроме того, по умолчанию каждый блок поддерживает входной буфер и принимает произвольное количество данных до того, как он будет готов их обработать.
- Это создает проблему в сценарии с ветвлением, в котором один блок-источник связан с двумя блоками-приемниками: в этом случае второй блок будет простаивать.
 - Первый блок-приемник всегда будет принимать данные и буферизовать их, так что блок-источник никогда не будет пытаться передавать данные второму блоку-приемнику.
 - Проблему можно решить регулировкой (throttling) блоков-приемников с использованием параметра блока `BoundedCapacity`.
 - По умолчанию `BoundedCapacity` присваивается значение `DataflowBlockOptions.Unbounded`, при котором первый блок-приемник буферизует все данные, даже если еще не готов к их обработке.

Рецепт 4: регулювання блоків

- BoundedCapacity можно присвоить любое значение больше нуля (или, конечно, DataflowBlockOptions.Unbounded).
- Если блоки-приемники успевают обрабатывать данные, поступающие от блоков-источников, простого значения 1 будет достаточно:

```
var sourceBlock = new BufferBlock<int>();  
var options = new DataflowBlockOptions { BoundedCapacity = 1 };  
var targetBlockA = new BufferBlock<int>(options);  
var targetBlockB = new BufferBlock<int>(options);  
  
sourceBlock.LinkTo(targetBlockA);  
sourceBlock.LinkTo(targetBlockB);
```

- Регулировка полезна для распределения нагрузки в конфигурациях с ветвлением, но она также может применяться везде, где возникнет необходимость в регулировании.
 - Например, если сеть потока данных заполняется данными от операции ввода/вывода, можно применить BoundedCapacity к блокам своей сети.
 - В этом случае вы не прочтаете слишком много данных ввода/вывода до того, как сеть будет к этому готова, а все входные данные не будут буферизованы сетью до того, как она сможет его обработать.

Рецепт 5: паралельна обробка з блоками потоку даних

- **Задача:** потрібно виконати паралельну обробку в сітці потоку даних.
- **Вирішення:** По умовчанию каждый блок потока данных не зависит от других блоков.
 - Когда вы связываете два блока, они будут выполнять обработку независимо друг от друга. Таким образом, в каждую сеть потока данных встроена некоторая степень естественного параллелизма.
- Если требуется выйти за эти рамки, — например, если один конкретный блок выполняет интенсивные вычисления на процессоре, — вы можете дать команду этому блоку работать параллельно с входными данными, устанавливая параметр `MaxDegreeOfParallelism`.
 - По умовчанию этому параметру тоже присваивается значение 1, поэтому каждый блок потока данных обрабатывает только один фрагмент данных за раз.

Рецепт 5: паралельна обробка з блоками потоку даних

- BoundedCapacity можно присвоить DataflowBlockOptions.Unbounded или любое значение, большее 0.
 - Следующий пример позволяет любому количеству задач умножать данные одновременно:

```
var multiplyBlock = new TransformBlock<int, int>(  
    item => item * 2,  
    new ExecutionDataflowBlockOptions  
    {  
        MaxDegreeOfParallelism = DataflowBlockOptions.Unbounded  
    });  
var subtractBlock = new TransformBlock<int, int>(item => item - 2);  
multiplyBlock.LinkTo(subtractBlock);
```

- Параметр MaxDegreeOfParallelism упрощает организацию параллельной обработки в блоке.
- Сложнее определить, каким блокам это потребуется.
- Один из способов заключается в том, чтобы приостановить выполнение потока данных в отладчике и проанализировать количество элементов данных в очереди (т. е. элементов, которые еще не были обработаны блоком).
- Неожиданно высокое количество элементов данных может указывать на то, что реорганизация или параллелизация могли бы принести пользу.

Рецепт 5: паралельна обробка з блоками потоку даних

- MaxDegreeOfParallelism також працює і в тому випадку, якщо блок потоку даних виконує асинхронну обробку.
 - В цьому випадку параметр MaxDegreeOfParallelism задає *рівень паралелізму* — визначене кількість слотів.
 - Кожен елемент даних займає слот, коли блок приступає до його обробки, і покидає цей слот тільки при повному завершенні асинхронної обробки.