

O'REILLY®



Współbieżność  
w języku C#  
Receptury

Helion 

Stephen Cleary

Tytuł oryginału: Concurrency in C# Cookbook

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2990-4

© 2017 Helion S.A.

Authorized Polish translation of the English edition Concurrency in C# Cookbook, ISBN 9781449367565 © 2014 Stephen Cleary

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:  
<ftp://ftp.helion.pl/przyklady/wspcre.zip>

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
<http://helion.pl/user/opinie/wspcre>  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- **Lubią to!** » Nasza społeczność

---

# Spis treści

<b>Przedmowa .....</b>	<b>9</b>
<b>1. Współbieżność: przegląd .....</b>	<b>15</b>
1.1. Wprowadzenie do współbieżności	15
1.2. Wprowadzenie do programowania asynchronicznego	18
1.3. Wprowadzenie do programowania równoległego	23
1.4. Wprowadzenie do programowania reaktywnego (Rx)	27
1.5. Wprowadzenie do przepływów danych	29
1.6. Wprowadzenie do programowania wielowątkowego	32
1.7. Kolekcje dla aplikacji współbieżnych	33
1.8. Nowoczesne projektowanie	33
1.9. Podsumowanie informacji na temat kluczowych technologii	34
<b>2. Podstawy async .....</b>	<b>37</b>
2.1. Wstrzymanie na określony czas	38
2.2. Zwracanie wykonanych zadań	40
2.3. Raportowanie postępu	42
2.4. Oczekiwanie na wykonanie zestawu zadań	43
2.5. Oczekiwanie na wykonanie jakiegokolwiek zadania	46
2.6. Przetwarzanie wykonanych zadań	48
2.7. Unikanie kontekstu dla kontynuacji	51
2.8. Obsługa wyjątków z metod async typu Task	53
2.9. Obsługa wyjątków z metod async typu void	55

<b>3. Podstawy przetwarzania równoległego .....</b>	<b>59</b>
3.1. Równoległe przetwarzanie danych	60
3.2. Równoległa agregacja	62
3.3. Równoległe wywołanie	63
3.4. Równoległość dynamiczna	65
3.5. Parallel LINQ	67
<b>4. Podstawy przepływu danych .....</b>	<b>69</b>
4.1. Łączenie bloków	70
4.2. Propagowanie błędów	72
4.3. Usuwanie połączeń między blokami	74
4.4. Ograniczanie pojemności bloków	75
4.5. Przetwarzanie równoległe za pomocą bloków przepływu danych	76
4.6. Tworzenie niestandardowych bloków	77
<b>5. Podstawy Rx .....</b>	<b>81</b>
5.1. Konwersja zdarzeń .NET	82
5.2. Wysyłanie powiadomień do kontekstu	85
5.3. Grupowanie danych zdarzeń za pomocą okienek i buforów	87
5.4. Ujzarnianie strumieni zdarzeń za pomocą ograniczania przepływu i próbkowania	90
5.5. Limity czasu	92
<b>6. Testowanie .....</b>	<b>97</b>
6.1. Testy jednostkowe metod async	98
6.2. Testy jednostkowe metod async, które powinny zakończyć się niepowodzeniem	100
6.3. Testy jednostkowe metod async void	102
6.4. Testy jednostkowe siatek przepływu danych	104
6.5. Testy jednostkowe strumieni obserwowalnych Rx	105
6.6. Testy jednostkowe strumieni obserwowalnych Rx za pomocą atrapy harmonogramu	108

<b>7. Interoperacyjność .....</b>	<b>113</b>
7.1. Metody opakowujące async dla metod „async” ze zdarzeniami „Completed”	113
7.2. Metody opakowujące async dla metod „Begin/End”	116
7.3. Metody opakowujące async dla dowolnych operacji lub zdarzeń	117
7.4. Metody opakowujące async dla kodu równoległego	119
7.5. Metody opakowujące async dla strumieni obserwowalnych Rx	120
7.6. Metody opakowujące strumieni obserwowalnych Rx dla kodu asynchronicznego	122
7.7. Strumienie obserwowalne Rx i siatki przepływu danych	124
<b>8. Kolekcje .....</b>	<b>127</b>
8.1. Niemutowalne stopy i kolejki	130
8.2. Listy niemutowalne	133
8.3. Zbiory niemutowalne	135
8.4. Słowniki niemutowalne	137
8.5. Słowniki bezpieczne wątkowo	140
8.6. Kolejki blokujące	142
8.7. Stopy i multizbiory blokujące	145
8.8. Kolejki asynchroniczne	147
8.9. Stopy i multizbiory asynchroniczne	150
8.10. Kolejki blokujące/asynchroniczne	153
<b>9. Anulowanie .....</b>	<b>159</b>
9.1. Wysyłanie żądań anulowania	160
9.2. Reagowanie na żądania anulowania poprzez odpytywanie	163
9.3. Anulowanie z powodu przekroczenia limitu czasu	165
9.4. Anulowanie kodu async	167
9.5. Anulowanie kodu równoległego	168
9.6. Anulowanie kodu reaktywnego	170
9.7. Anulowanie siatek przepływu danych	172
9.8. Wstrzykiwanie żądań anulowania	173
9.9. Współdziałanie z innymi systemami anulowania	175

<b>10. Przyjazne funkcjnie programowanie obiektowe .....</b>	<b>179</b>
10.1. Interfejsy async i dziedziczenie	180
10.2. Konstruowanie async: fabryki	181
10.3. Konstruowanie async: wzorzec inicjowania asynchronicznego	184
10.4. Właściwości async	188
10.5. Zdarzenia async	192
10.6. Usuwanie async	195
<b>11. Synchronizacja .....</b>	<b>201</b>
11.1. Blokady	207
11.2. Blokady async	209
11.3. Sygnały blokujące	212
11.4. Sygnały async	213
11.5. Ograniczanie współbieżności	215
<b>12. Planowanie .....</b>	<b>219</b>
12.1. Planowanie pracy dla puli wątków	219
12.2. Wykonywanie kodu za pomocą dyspozytora zadań	221
12.3. Planowanie kodu równoległego	224
12.4. Synchronizacja przepływu danych z wykorzystaniem dyspozytorów	225
<b>13. Scenariusze .....</b>	<b>227</b>
13.1. Inicjowanie współdzielonych zasobów	227
13.2. Odroczone ewaluacja Rx	229
13.3. Asynchroniczne wiązanie danych	231
<b>Skorowidz .....</b>	<b>237</b>

# Podstawy przetwarzania równoległego

W tym rozdziale omówimy wzorce dla programowania równoległego. Programowanie równoległe jest stosowane do podzielenia ograniczonych możliwości obliczeniowymi fragmentów pracy i rozdzielenia ich na wiele wątków. Przedstawione poniżej receptury przetwarzania równoległego uwzględniają jedynie zadania uzależnione od wydajności procesora. Jeśli masz naturalnie asynchroniczne operacje (takie jak zadania ograniczone możliwościami we-wy), które chcesz wykonywać równoległe, zapoznaj się z rozdziałem 2., a w szczególności z recepturą 2.4.

Abstrakcje przetwarzania równoległego omówione w tym rozdziale są częścią biblioteki zadań równoległych (ang. *Task Parallel Library* — **TPL**). Jest ona wbudowana we framework .NET, ale nie jest dostępna na wszystkich platformach (zobacz tabela 3.1):

Tabela 3.1. Wsparcie platform dla biblioteki TPL

Platforma	Wsparcie dla przetwarzania równoległego
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	×
Windows Phone SL 7.1	×
Silverlight 5	×

## 3.1. Równoległe przetwarzanie danych

### Problem

Mamy kolekcję danych i musimy wykonać tę samą operację na każdym elemencie tych danych. Ta operacja jest ograniczona możliwościami obliczeniowymi procesora i może zająć trochę czasu.

### Rozwiązanie

Typ `Parallel` zawiera metodę `ForEach` specjalnie przeznaczoną do tego celu. Kod przedstawiony w tym przykładzie przyjmuje kolekcję macierzy i odwraca je wszystkie:

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

W niektórych sytuacjach będziemy chcieli zatrzymać pętlę wcześniej, na przykład przy napotkaniu nieprawidłowej wartości. Poniższy kod odwraca każdą macierz, ale jeśli zostanie napotkana nieprawidłowa macierz, pętla zostanie przerwana:

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
    Parallel.ForEach(matrices, (matrix, state) =>
    {
        if (!matrix.IsInvertible)
            state.Stop();
        else
            matrix.Invert();
    });
}
```

Bardziej powszechną sytuacją jest potrzeba posiadania możliwości anulowania równoległej pętli. Różni się to od zatrzymania pętli. Pętla jest *zatrzymywana* z wewnątrz pętli, a *anulowanie* odbywa się z zewnątrz pętli. Przykładowo `CancellationTokenSource` może być anulowany za pomocą przycisku anulowania, co spowoduje anulowanie równoległej pętli, takiej jak ta:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
```



```

        new ParallelOptions { CancellationToken = token },
        matrix => matrix.Rotate(degrees));
    }

```

Należy pamiętać o tym, że każde zadanie równoległe może działać w innym wątku, dlatego każdy współdzielony stan musi być chroniony. Poniższy kod odwraca każdą macierz i zlicza macierze, które nie mogły zostać odwrócone:

```

// Uwaga: to nie jest najbardziej efektywna implementacja.
// To tylko przykład użycia blokady do ochrony współdzielonego stanu.
int InvertMatrices(IEnumerable<Matrix> matrices)
{
    object mutex = new object();
    int nonInvertibleCount = 0;
    Parallel.ForEach(matrices, matrix =>
    {
        if (matrix.IsInvertible)
        {
            matrix.Invert();
        }
        else
        {
            lock (mutex)
            {
                ++nonInvertibleCount;
            }
        }
    });
    return nonInvertibleCount;
}

```

## Dyskusja

Metoda `Parallel.ForEach` umożliwia przetwarzanie równoległe na sekwencji wartości. Podobnym rozwiązaniem jest technologia **PLINQ** (ang. *Parallel LINQ*). PLINQ zapewnia prawie te same możliwości za pomocą składni podobnej do LINQ. Jedną z różnic pomiędzy `Parallel` i PLINQ jest to, że PLINQ zakłada możliwość wykorzystania wszystkich rdzeni na komputerze, podczas gdy `Parallel` będzie dynamicznie reagować na zmieniające się warunki procesora.

`Parallel.ForEach` to równoległa pętla `foreach`. Jeśli trzeba wykonać równoległą pętlę `for`, klasa `Parallel` obsługuje również metodę `Parallel.For`. Ta metoda jest szczególnie użyteczna, jeśli masz wiele tablic danych, z których wszystkie wykorzystują ten sam indeks.

## Zobacz również

Receptura 3.2 omawia równoległą agregację szeregu wartości, w tym sum i średnich.

Receptura 3.5 opisuje podstawy PLINQ.

Rozdział 9. omawia anulowanie.

## 3.2. Równoległa agregacja

### Problem

Na zakończenie równoległej operacji musimy zagregować wyniki. Przykładami agregacji są sumy, średnie itp.

### Rozwiązanie

Klasa `Parallel` obsługuje agregację poprzez koncepcję **wartości lokalnych**, które są zmiennymi występującymi lokalnie w ramach równoległej pętli. Oznacza to, że ciało pętli może po prostu uzyskać dostęp do tej wartości bezpośrednio, bez konieczności przejmowania się synchronizacją. Gdy pętla jest gotowa do agregacji wszystkich swoich lokalnych wyników, robi to za pomocą delegata `localFinally`. Należy zwrócić uwagę, że delegat `localFinally` *potrzebuje* zsynchronizować dostęp do zmiennej, która przechowuje wynik końcowy. Oto przykład sumy równoległej:

```
// Uwaga: to nie jest najbardziej efektywna implementacja.  
// To tylko przykład użycia blokady do ochrony współdzielonego stanu.  
static int ParallelSum(IEnumerable<int> values)  
{  
    object mutex = new object();  
    int result = 0;  
    Parallel.ForEach(source: values,  
        localInit: () => 0,  
        body: (item, state, localValue) => localValue + item,  
        localFinally: localValue =>  
        {  
            lock (mutex)  
                result += localValue;  
        });  
    return result;  
}
```

Parallel LINQ ma bardziej naturalną możliwość obsługi agregacji niż klasa Parallel:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

No dobrze, to był cios poniżej pasa, ponieważ PLINQ ma wbudowaną obsługę wielu typowych operatorów (na przykład Sum). PLINQ oferuje również ogólną obsługę agregacji poprzez operator Aggregate:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Aggregate(
        seed: 0,
        func: (sum, item) => sum + item
    );
}
```

## Dyskusja

Jeśli używasz już klasy Parallel, możesz chcieć skorzystać z jej wsparcia dla agregacji. W przeciwnym razie wsparcie PLINQ jest w większości scenariuszy bardziej ekspresyjne i ma krótszy kod.

## Zobacz również

Receptura 3.5 omawia podstawy PLINQ.

## 3.3. Równoległe wywołanie

### Problem

Mamy wiele metod wywoływanych równoległe; metody te są (w większości) niezależne od siebie.

### Rozwiązanie

Klasa Parallel zawiera prostą składową Invoke, która jest przeznaczona dla tego scenariusza. Oto przykład kodu, który dzieli tablicę na pół i przetwarza każdą połówkę niezależnie:

```

static void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
    );
}

static void ProcessPartialArray(double[] array, int begin, int end)
{
    // Przetwarzanie wymagające obliczeniowo...
}

```

Do metody `Parallel.Invoke` można również przekazać tablicę delegatów, jeśli liczba wywołań nie jest znana aż do czasu wykonywania:

```

static void DoAction20Times(Action action)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(actions);
}

```

Metoda `Parallel.Invoke` obsługuje anulowanie podobnie jak inne składowe klasy `Parallel`:

```

static void DoAction20Times(Action action, CancellationToken token)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(new ParallelOptions { CancellationToken = token },
        actions);
}

```

## Dyskusja

Metoda `Parallel.Invoke` jest doskonałym rozwiązaniem dla prostych wywołań równoległych. Nie sprawdzi się jednak zbyt dobrze, jeśli chcesz wywołać akcję dla każdego elementu danych wejściowych (zamiast niej należy użyć metody `Parallel.ForEach`) lub jeśli każda akcja generuje jakieś dane wyjściowe (wtedy należy skorzystać z `Parallel.LINQ`).

## Zobacz również

Receptura 3.1 opisuje metodę `Parallel.ForEach`, która wywołuje akcję dla każdego elementu danych.

Receptura 3.5 omawia `Parallel.LINQ`.

## 3.4. Równoległość dynamiczna

### Problem

Mamy bardziej złożoną sytuację równoległą, w której struktura i liczba równoległych zadań zależą od informacji znanych dopiero w czasie wykonywania programu.

### Rozwiązanie

Biblioteka zadań równoległych jest skoncentrowana wokół typu `Task`. Klasa `Parallel` i technologia `Parallel LINQ` zapewniają tylko wygodne funkcje opakowujące dla tego typu. Gdy potrzebna jest równoległość dynamiczna, najłatwiej jest użyć typu `Task` bezpośrednio.

Oto jeden z przykładów, w którym dla każdego węzła binarnego drzewa trzeba wykonać kosztowne przetwarzanie. Struktura drzewa nie będzie znana aż do czasu wykonywania, dlatego jest to dobry scenariusz dla równoległości dynamicznej. Metoda `Traverse` przetwarza bieżący węzeł, a następnie tworzy dwa zadania podrzędne, po jednym dla każdej gałęzi pod węzłem (w tym przykładzie zakładamy, że węzły nadrzędne muszą być przetwarzane przed podrzędnymi). Metoda `ProcessTree` rozpoczyna przetwarzanie poprzez utworzenie zadania nadrzędnego najwyższego poziomu i czekanie na jego zakończenie:

```
void Traverse(Node current)
{
    DoExpensiveActionOnNode(current);
    if (current.Left != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Left),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
    if (current.Right != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Right),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
}
```

```

public void ProcessTree(Node root)
{
    var task = Task.Factory.StartNew(() => Traverse(root),
        CancellationToken.None,
        TaskCreationOptions.None,
        TaskScheduler.Default);
    task.Wait();
}

```

Jeśli nie masz sytuacji typu zadanie nadrzędne – podrzędne, możesz zaplanować, aby każde zadanie było uruchamiane po innym, stosując **kontynuację** zadań. Kontynuacja jest osobnym zadaniem, które jest wykonywane, gdy pierwotne zadanie zostanie zakończone:

```

Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Zadanie zostało wykonane" ),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// Argument "t" dla kontynuacji jest taki sam jak "task".

```

## Dyskusja

Powyższy przykładowy kod wykorzystuje właściwości `CancellationToken.None` oraz `TaskScheduler.Default`. Tokeny anulowania zostały opisane w recepturze 9.2, a dyspozytory zadań w recepturze 12.3. Zawsze dobrym pomysłem jest bezpośrednio określić `TaskScheduler` używany przez metody `StartNew` i `ContinueWith`.

Ten układ zadań nadrzędnych i podrzędnych jest typowy dla równoległości dynamicznej. Nie jest jednak wymagany. Można również przechowywać każde nowe zadanie w wątkowo bezpiecznej kolekcji, a następnie za pomocą `Task.WaitAll` poczekać na zakończenie ich wszystkich.



Używanie typu `Task` dla przetwarzania równoległego całkowicie różni się od wykorzystania `Task` dla przetwarzania asynchronicznego. Zobacz poniżej.

W programowaniu równoległym typ `Task` służy dwóm celom: może być zadaniem równoległym lub zadaniem asynchronicznym. Zadania równoległe mogą wykorzystywać składowe blokujące, takie jak `Task.Wait`, `Task.Result`, `Task.Wait` ↪ `All` i `Task.WaitAny`. Często używają również `AttachedToParent` do tworzenia między zadaniami relacji nadrzędne – podrzędne. Zadania równoległe powinny być tworzone za pomocą metod `Task.Run` lub `Task.Factory.StartNew`.

W przeciwieństwie do tego, w przypadku zadań asynchronicznych powinno się unikać składowych blokujących i preferować użycie `await`, `Task.WhenAll` i `Task.WhenAny`. Zadania asynchroniczne nie używają `AttachedToParent`, ale mogą formować dorozumiany rodzaj relacji nadrzędne – podrzędne poprzez oczekiwanie na kolejne zadanie.

## Zobacz również

Receptura 3.3 omawia równoległe wywoływanie sekwencji metod, gdy wszystkie te metody są znane przy rozpoczynaniu pracy równoległej.

## 3.5. Parallel LINQ

### Problem

Mamy do wykonania przetwarzanie równoległe na sekwencji danych, tworzące kolejną sekwencję danych lub podsumowanie tych danych.

### Rozwiązanie

Większość programistów jest zaznajomiona z technologią LINQ, którą można wykorzystać do napisania opartych na modelu *pull* obliczeń na sekwencjach. `Parallel LINQ (PLINQ)` rozszerza to wsparcie LINQ o przetwarzanie równoległe.

PLINQ dobrze sprawdza się w scenariuszach strumieniowych, gdy mamy sekwencję danych wejściowych i generujemy sekwencję danych wyjściowych. Oto prosty przykład, w którym po prostu mnożymy przez dwa każdy element w sekwencji (rzeczywiste scenariusze będą znacznie bardziej wymagające obliczeniowo niż proste mnożenie):

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
```

```
        return values.AsParallel().Select(item => item * 2);
    }
```

Ten kod może generować swoje dane wyjściowe w dowolnej kolejności. Jest to ustawienie domyślne dla Parallel LINQ. Można także określić kolejność, która ma być zachowana. Poniższy kod jest nadal przetwarzany równoległe, ale zachowuje oryginalną kolejność:

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().AsOrdered().Select(item => item * 2);
}
```

Innym naturalnym wykorzystaniem PLINQ jest równoległe agregowanie lub podsumowywanie danych. Poniższy kod wykonuje równoległe sumowanie:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

## Dyskusja

Klasa Parallel jest dobra dla wielu scenariuszy, ale kod PLINQ jest prostszy, gdy wykonujemy agregację lub przekształcamy jedną sekwencję w drugą. Należy pamiętać, że klasa Parallel jest bardziej przyjazna dla innych procesów w systemie niż PLINQ. Jest to szczególnie istotne, jeżeli przetwarzanie równoległe odbywa się na serwerze.

PLINQ zapewnia równoległe wersje wielu różnych operatorów, w tym filtrowania (Where), rzutowania (Select) oraz wielu agregacji, takich jak Sum, Average i bardziej ogólna Aggregate. Zasadniczo wszystko, co można zrobić za pomocą zwykłego LINQ, można zrobić równoległe za pomocą PLINQ. To sprawia, że PLINQ jest doskonałym wyborem, jeśli mamy istniejący kod LINQ, który skorzystałby na wykonywaniu równoległym.

## Zobacz również

Receptura 3.1 opisuje sposób użycia klasy Parallel do wykonywania kodu dla każdego elementu w sekwencji.

Receptura 9.5 omawia sposób anulowania zapytań PLINQ.



## A

agregacja, 62, 68  
anulowanie, 159, 165  
    kodu async, 167  
    kodu reaktywnego, 170  
    kodu równoległego, 168  
    siatek przepływu danych, 172  
APM, Asynchronous Programming  
    Model, 116  
async, 37  
    anulowanie kodu, 167  
    blokady, 209  
    interfejsy, 180  
    sygnały, 213  
    testy jednostkowe, 98  
    usuwanie, 195  
    zdarzenia, 192  
asynchroniczne wiązanie danych, 231

## B

biblioteka  
    AsyncEx, 231  
    Microsoft.Bcl.Async, 166  
    Nito.AsyncEx, 114, 192  
    przepływu danych, 29, 69  
    rozszerzeń reaktywnych, 81, 223  
    Rx, 81  
    TPL, 70

blok

    przepływu danych, 30  
    try-catch, 53

blokada, 207

    Monitor, 208  
    ReaderWriterLockSlim, 208  
    SpinLock, 208

blokady async, 209, 211

bloki

    niestandardowe, 77  
    przepływu danych, 70, 74

bufor, 87

## C

CancellationToken, 172, 175

## D

delegat localFinally, 62

dyspozytor, scheduler, 109, 219

    Rx, 223

    TaskScheduler, 222

dyspozytory zadań, 221

dziedziczenie, 180

## E

EAP, Event-based Asynchronous  
Pattern, 113  
ewaluacja Rx, 229

## F

fabryka, 182  
FIFO, first-in, first-out, 132  
framework .NET, 34  
funkcja callback, 118

## G

gorący strumień obserwowalny, 28  
grupowanie danych zdarzeń, 87

## I

inicjowanie współdzielonych zasobów,  
227  
instrukcja lock, 208  
interfejs IObservable<T>, 105  
interfejsy async, 180  
interoperacyjność, 113

## K

klasa  
CancellationTokenSource, 162  
ExpectedExceptionAttribute, 100  
Parallel, 62, 68  
ParallelOptions, 168  
kolejka, 130  
FIFO, 132  
kolejki  
asynchroniczne, 147  
blokujące, 142  
blokujące/asynchroniczne, 153  
kolekcje, 127  
bezpieczne wątkowo, 128, 207  
dla aplikacji współbieżnych, 33

niemutowalne, 35, 127  
producent-konsument, 128  
współbieżne, 35

komunikacja, 201  
konstruktory, 182  
kontekst synchronizacji, 86  
kontynuacja zadań, 66  
konwersja właściwości synchronicznej,  
190  
konwersja zdarzeń .NET, 82  
koordynowanie działania bloków, 226

## L

LIFO, last-in, first-out, 131  
limity czasu, 92  
LINQ, 27, 67  
lista, 133

## Ł

łączenie bloków, 70  
przepływu danych, 74

## M

metoda  
AddOrUpdate, 140  
AdvanceBy, 111  
AdvanceTo, 111  
AggregateException.Flatten, 73  
Assert.ThrowsException, 101  
ConfigureAwait, 52  
CreateLinkedTokenSource, 174  
DataflowBlock.Encapsulate, 78  
Delay, 38  
DownloadStringTaskAsync, 114, 115  
Encapsulate, 78, 79  
Execute, 55  
ForEach, 60  
FromAsync, 117, 123  
InitializeAsync, 182

- Link, 71
- LinkTo, 74
- LogicalGetData, 234
- LogicalSetData, 234
- Observable.FromEvent, 84
- OnNext, 82
- OrderByCompletion, 51
- OutputAvailableAsync, 149
- Parallel.Invoke, 225
- ProcessTree, 65
- StartAsync, 123
- Task.Delay, 38, 39
- Task.FromResult, 41
- Task.Run, 120, 220
- Task.Wait, 99
- Task.WhenAll, 44
- Task.WhenAny, 46
- TaskScheduler.FromCurrentSynchronizationContext, 223
- ThrowIfCancellationRequested, 165
- ThrowsException, 101
- ToObservable, 123
- TryGetValue, 141
- metody
  - async, 98
  - async typu Task, 53
  - async typu void, 55, 102
  - opakowujące async, 113, 116–120
  - opakowujące strumieni obserwowalnych Rx, 122
  - rozszerzające, 51
  - testów jednostkowych, 98
  - typu Parallel, 168
- migawki, 33
- model
  - pull, 28
  - push, 28
- multizbiory
  - asynchroniczne, 150
  - blokujące, 145

## N

- niemutowalne
  - kolejki, 130
  - listy, 133
  - słowniki, 137
  - stosy, 130
  - zbiory, 135

## O

- obiekty oczekiwalne, 20
- obietnica, 17
- obsługa
  - przepływu danych, 37, 69
  - wyjątków, 53, 55
- ochrona danych, 201
- oczekiwane zadania, 48
- oczekiwanie na wykonanie, 43, 46
- odpytywanie, 163
- odroczone ewaluacja Rx, 229
- ograniczanie
  - pojemności bloków, 75
  - przepływu, 90
  - współbieżności, 215
- okienko, 87
- opakowywanie APM, 116
- opcja
  - BoundedCapacity, 77
  - MaxDegreeOfParallelism, 76, 77
- operacje asynchroniczne, 17, 122
- operator
  - Buffer, 88
  - FromAsync, 171
  - Observable.Defer, 230
  - ObserveOn, 85
  - Return, 106
  - Sample, 90
  - SelectMany, 171
  - SingleAsync, 106
  - StartAsync, 171

- operator
  - Throttle, 90
  - Throw, 107
  - Timeout, 93
  - Window., 88

## P

- Parallel LINQ, 67, 169
- pętla foreach, 134
- planowanie, 219
  - kodu równoległego, 224
  - pracy dla puli wątków, 219
- platformy, 35, 211
- PLINQ, Parallel LINQ, 61, 67
- połączone tokeny anulowania, 174
- powiadomienia do kontekstu, 85
- powiadomienie, 20
- problem z wydajnością, 52
- programowanie
  - asynchroniczne, 17, 18
  - obiektywne, 179
  - reaktywne, Rx, 17, 27
  - równoległe, 23
  - wielowątkowe, 32
- projektowanie nowoczesne, 33
- propagowanie błędów, 72
- próbkiowanie, 90
- przeciążenie metody, 55
- przekroczenie limitu czasu, 165
- przepływ danych, 29, 35, 69
- przetwarzanie
  - równoległe, 16, 35, 59, 76, 119
  - wykonanych zadań, 48
- pula wątków, 120, 219

## R

- raportowanie postępu, 42
- równoległa agregacja, 62

- równoległe
  - przetwarzanie danych, 60
  - wywołanie, 63
- równoległość dynamiczna, 65
- równoważenie obciążenia, 75
- Rx, Reactive Extensions, 27, 35, 81

## S

- scenariusze, 227
- siatka przepływu danych, 104, 124
- słownik HttpContext.Current.Items, 235
- słowniki
  - bezpieczne wątkowo, 140
  - niemutowalne, 137
- słowo kluczowe async, 37
- stan niejawny, 233
- stos, 130, 145, 150
  - LIFO, 131
- struktura danych
  - kolejka, 131
  - lista, 133
  - stos, 131
- strumień obserwowalny Rx, 105, 120, 124
  - gorący, 29
  - zimny, 29
- strumień zdarzeń, 90
- subskrypcja strumienia obserwowalnego, 170
- sygnał ManualResetEventSlim, 212
- sygnały
  - async, 213
  - blokujące, 212
- symulowanie zależności
  - asynchronicznych, 99
- synchronizacja, 201
  - przepływu, 225

## T

TAP, Task-based Asynchronous Pattern, 113

TDD, test-driven development, 97

technologia

LINQ, 67

PLINQ, 61

testowanie, 97

obsługi błędów, 102

testy jednostkowe, 98

metod async, 100

metod async void, 102

siatek przepływu danych, 104

strumieni obserwowalnych Rx, 105, 108

tokeny anulowania, 66

TPL, Task Parallel Library, 34, 59

TPL Dataflow, 29, 69

tworzenie niestandardowych bloków, 77

typ

ActionBlock<T>, 153

Async

ProducerConsumerQueue<T>, 150

AsyncCollection<T>, 151, 153

AsyncContext, 103

AsyncLazy<T>, 229

AsyncManualResetEvent, 215

AsyncProducerConsumerQueue<T>, 145, 148, 155, 156

BlockingCollection<T>, 143, 145, 146, 152

BufferBlock<T>, 145, 147, 150, 155

CallContext, 234

CancellationTokenSource, 160

ConcurrentDictionary<TKey, TValue>, 140

future, 17

ImmutableDictionary<TK,TV>, 139

ImmutableHashSet<T>, 136

ImmutableList<T>, 134

ImmutableSortedDictionary<TK,TV>, 139

ImmutableSortedSet<T>, 136

IMyFundamentalType, 185

IProgress<T>, 42

Lazy<T>, 227

List<T>, 134

ManualResetEvent, 214

NotifyTaskCompletion, 231

Progress<T>, 42

SemaphoreSlim, 209

SynchronizationContext, 57

System.Timers.Timer, 83

Task, 67

Task<T>, 220

TaskCompletionSource<T>, 118, 213

TaskCompletionSource<TResult>, 114

TestScheduler, 110

typy synchronizacji

komunikacja, 201

ochrona danych, 201

## U

unikanie kontekstu dla kontynuacji, 51

usuwanie

async, 195

połączeń między blokami, 74

## W

wątek wywołujący, 120

wątki

konsumentów, 143

producentów, 143

wielowątkowość, 16

właściwości async, 188, 190

właściwość

CancellationToken, 172

TaskScheduler.Default, 222

- wsparcie platform
  - dla blokad asynchronicznych, 211
  - dla współbieżności, 35
- współbieżność, 15
- wstrzykiwanie żądań anulowania, 173
- wstrzymanie na określony czas, 38
- wydajność, 52
- wyjątek
  - AggregateException, 71, 73
  - InvalidOperationException, 148
  - TrySetException, 115
- wyjątki, 53
- wykonywanie kodu
  - dyspozytor zadań, 221
- wysyłanie
  - powiadomień, 85
  - żądań anulowania, 160
- wznawianie w kontekście, 52
- wzorzec
  - BeginOperacja, 116
  - EndOperacja, 116
  - inicjowania asynchronicznego, 184
  - OperacjaAsync, 113
  - OperacjaCompleted, 113

## Z

- zadania
  - asynchroniczne, 67
  - równoległe, 67
- zakończenie asynchroniczne, 195, 196, 199
- zbiory niemutowalne, 135
- zdarzenia
  - .NET, 82
  - async, 17, 192
  - OperacjaCompleted, 113
  - poleceń, 194
  - powiadomień, 194
- zimny strumień obserwowalny, 29

## Ż

- żądanie anulowania, 160, 163

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



**Helion SA**

## Kod współbieżny – responsywność, skalowalność, nowoczesność!

Współbieżność jest jednym z wymogów nowoczesnych aplikacji, niezależnie od ich rodzaju i platformy. Tworzenie współbieżnych aplikacji jest co prawda dość złożone, jednak cecha ta czyni interfejs użytkownika bardziej responsywnym, a systemy – skalowalnymi. Zrozumienie istoty współbieżności i umiejętność jej zaimplementowania jest już koniecznością dla profesjonalnego dewelopera, nawet jeśli zagadnienia te wydają się trudne i skomplikowane.

Ta książka przedstawia nowoczesne podejście do współbieżności. Jest przeznaczona dla programistów, którzy chcą poszerzyć wiedzę i umiejętności. Omówiono tu zagadnienia wielowątkowości i przedstawiono kilka różnych rodzajów współbieżności, w tym programowanie równoległe, asynchroniczne i reaktywne. Opisano biblioteki, dzięki którym programowanie współbieżnych aplikacji staje się znacznie łatwiejsze poprzez podniesienie poziomu abstrakcji. Zawarte w książce receptury uzupełniono o działający kod i przedyskutowano sposób ich działania.

**Stephen Cleary** – jest doświadczonym programistą. Od samego początku tworzy oprogramowanie *open source*, między innymi bibliotek Boost C++. Opublikował również kilka własnych bibliotek i narzędzi.

W książce między innymi:

- omówienie różnych rodzajów współbieżności
- reaktywność i przepływ zdarzeń
- biblioteka zadań równoległych (TPL)
- biblioteki rozszerzeń reaktywnych w LINQ
- asynchroniczne programowanie obiektowe
- synchronizacja wątków

sięgnij po WIĘCEJ



KOD KORZYŚCI

**Helion**

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

Sprawdź najnowsze promocje:  
● <http://helion.pl/promocje>  
Książki najchętniej czytane:  
● <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
● <http://helion.pl/nowości>

ISBN 978-83-283-2990-4



9 788328 329904

Informatyka w najlepszym wydaniu

cena: 49,00 zł