O'REILLY®

C# 6.0

Leksykon kieszonkowy

Joseph Albahari,
Ben Albahari

Helion

Tytuł oryginału: C# 6.0 Pocket Reference: Instant Help for C# 6.0 Programmers

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-283-2446-6
© 2016 Helion S.A.

Authorized Polish translation of the English edition C# 6.0 Pocket Reference ISBN 9781491927410 © 2015 Joseph Albahari, Ben Albahari.

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.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ch6lek
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Funkcje asynchroniczne

Zaczynając od wersji 5.0 języka C# na bazie słów kluczowych `await` i `async` wprowadzono mechanizmy asynchronicznego wywołania funkcji; *programowanie asynchroniczne* polega na tym, że funkcje o długim czasie wykonania jak najszybciej zwracają sterowanie do miejsca wywołania, a swoją właściwą pracę wykonują w tle. Dla porównania wywołanie *synchroniczne* oznacza, że powrót z wywołania funkcji następuje dopiero po wykonaniu całości operacji realizowanych przez funkcję. Programowanie asynchroniczne promuje *współbieżność*: długotrwałe zadania są wykonywane *współbieżnie* z podstawowym przebiegiem wykonania programu wywołującego. Implementacja wywołań asynchronicznych odbywa się albo na bazie osobnego wątku, w którym wykonuje się właściwa część zadań funkcji asynchronicznych (w przypadku zadań wymagających obliczeniowo), albo na bazie mechanizmu wywołań zwrotnych (w przypadku operacji wejścia-wyjścia).

> **Uwaga**
>
> Programowanie wielowątkowe, współbieżne i asynchroniczne to same w sobie obszerne zagadnienia: w książce *C# 6.0 in a Nutshell* poświęcono im całe dwa rozdziały; szerokie omówienie znajduje się też pod adresem *http://albahari.com/threading*.

Weźmy za przykład poniższą metodę synchroniczną, wykonującą długotrwałe obliczenia:

```
int ComplexCalculation()
{
  double x = 2;
  for (int i = 1; i < 100000000; i++)
    x += Math.Sqrt (x) / i;
  return (int)x;
}
```

Wywołanie powyższej metody blokuje wywołującego na co najmniej kilka sekund, aż do obliczenia i zwrócenia wyniku:

```
int result = ComplexCalculation();
// Jakiś czas później:
Console.WriteLine (result); // 116
```

Środowisko uruchomieniowe CLR definiuje (w `System.Threading.Tasks`) klasę o nazwie `Task<TResult>`, reprezentującą operację, która kończy się w przyszłości. Instancję `Task<TResult>` dla zadania obliczeniowego można utworzyć za pośrednictwem metody `Task.Run`, która nakazuje środowisku CLR uruchomienie wskazanego delegatu w osobnym wątku, wykonywanym współbieżnie z wątkiem wywołującego:

Kup książkę Poleć książkę

```
Task<int> ComplexCalculationAsync()
{
  return Task.Run (() => ComplexCalculation());
}
```

Powyższa metoda staje się *asynchroniczna*, ponieważ zwraca sterowanie do wywołującego natychmiast po wywołaniu `Task.Run`. Potrzebny jest tu jednak jakiś sposób poinstruowania środowiska przez wywołującego, co zrobić po zakończeniu współbieżnej operacji i udostępnieniu jej wyniku. Służy do tego metoda `GetAwaiter` klasy `Task<TResult>`, która z asynchronicznym wywołaniem kojarzy tzw. *kontynuację* (ang. *continuation*):

```
Task<int> task = ComplexCalculationAsync();
var awaiter = task.GetAwaiter();
awaiter.OnCompleted (() =>         // Kontynuacja
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);      // 116
});
```

Powyższe oznacza: „po zakończeniu wywołaj podany delegat”. Nasza kontynuacja najpierw pobiera wynik operacji asynchronicznej (`GetResult`; wywołanie to może też przerzucić wyjątek rzucony przez wywołanie asynchroniczne). Pobrany wynik jest następnie wypisywany na wyjście programu za pomocą wywołania `Console.WriteLine`.

Słowa kluczowe await i async

Słowo `await` upraszcza kojarzenie kontynuacji z wywołaniami asynchronicznymi. Począwszy od najprostszego scenariusza, kompilator rozwija blok:

```
var wynik = await wyrażenie;
instrukcja/instrukcje;
```

do czegoś w rodzaju:

```
var awaiter = wyrażenie.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var wynik = awaiter.GetResult();
  instrukcja/instrukcje;
);
```

> **Uwaga**
>
> Kompilator dodatkowo emituje kod optymalizujący scenariusz synchronicznego (natychmiastowego) zakończenia operacji; operacja asynchroniczna kończy się od razu najczęściej w wyniku działania wewnętrznego mechanizmu cache, kiedy to wynik operacji jest gotowy w momencie wywołania.

Kup książkę Poleć książkę

Dzięki temu możemy swoją asynchroniczną metodę `ComplexCalculationAsync` wywołać następująco:

```
int result = await ComplexCalculationAsync();
Console.WriteLine (result);
```

Żeby całość skompilować, trzeba jeszcze dodać modyfikator `async` do deklaracji funkcji zawierającej wywołanie z `await`:

```
async void Test()
{
  int result = await ComplexCalculationAsync();
  Console.WriteLine (result);
}
```

Modyfikator `async` informuje kompilator, aby słowo `await` traktować jako słowo kluczowe, a nie identyfikator widoczny wewnątrz metody (służy to zabezpieczeniu kodu przed kompilacją kompilatorem starszym niż C# 5.0, w którym uznanie słowa `await` za zwyczajny identyfikator spowodowałoby niepoprawną kompilację). Modyfikator `async` można stosować tylko wobec metod (i wyrażeń lambda) bez wartości zwracanych albo zwracających wartości typu `Task` bądź `Task<TResult>`.

Uwaga

Modyfikator `async` działa podobnie jak modyfikator `unsafe` w tym sensie, że nie ma wpływu na sygnaturę czy widoczność metody; wpływa wyłącznie na to, jak metoda jest interpretowana i realizowana *wewnętrznie*.

Metody z modyfikatorem `async` są nazywane *funkcjami asynchronicznymi*, bo też zazwyczaj są w istocie asynchroniczne. Aby to uwidocznić, zobaczmy, jak przebiega wykonanie funkcji asynchronicznej.

Po napotkaniu wyrażenia `await` sterowanie (zazwyczaj) wraca do wywołującego (podobnie jak w przypadku instrukcji `yield return` w iteratorach), ale jeszcze zanim to nastąpi, środowisko wykonawcze kojarzy z oczekującym na wykonanie zadaniem kontynuację, aby już po zakończeniu wykonania zadania możliwe było podjęcie wykonania od określonego miejsca programu. Jeśli zadanie asynchroniczne wykona się błędnie, rzucony przez nie wyjątek zostanie przerzucony do kontynuacji (jak po wywołaniu `GetResult`); jeśli zadanie zwróci wartość, zostanie ona przypisana do wyrażenia `await`.

Uwaga

Wbudowana implementacja metody `OnComplete` kontynuacji oczekującej na zakończenie wykonania asynchronicznego gwarantuje zachowanie bieżącego *kontekstu synchronizacji* (jeśli takowy istnieje). W praktyce oznacza

Kup książkę Poleć książkę

to, że w aplikacjach z rozbudowanym interfejsem użytkownika (WPF, WinRT, Silverlight czy Windows Forms), przy oczekiwaniu na operację na interfejsie użytkownika, kontynuacja będzie się wykonywać w tym samym wątku co sama operacja; jest to bardzo pomocne w zachowaniu bezpieczeństwa wątkowego aplikacji.

Wyrażenie występujące za `await` to zazwyczaj zadanie, czyli egzemplarz klasy `Task`; może to być jednak dowolny obiekt z metodą `GetAwaiter` zwracającą obiekt implementujący interfejs `INotify Completion.OnCompleted` i z metodą `GetResult` o odpowiedniej sygnaturze (oraz z właściwością `IsCompleted`, służącą do sprawdzania możliwości wykonania synchronicznego).

Zauważmy, że wyrażenie `await` jest obliczane jako wartość typu `int`; spowodowane jest to tym, że oczekiwaliśmy na asynchroniczne wykonanie zadania typu `Task<int>` (którego metoda `GetAwaiter().GetResult()` zwraca wartość typu `int`).

Możliwe jest również oczekiwanie na wykonanie zadania nieuogólnionej klasy `Task`; wyrażenie `await` jest wtedy wyrażeniem pustym:

```
await Task.Delay (5000);
Console.WriteLine ("Minęło pięć sekund!");
```

Metoda `Task.Delay` jest statyczną metodą zwracającą obiekt `Task` kończący wykonanie po odczekaniu określonej liczby milisekund. Synchronicznym odpowiednikiem wywołania `Task.Delay` jest wywołanie `Thread.Sleep`.

Klasa `Task` to nieuogólniona klasa bazowa dla `Task<TResult>` i funkcjonalnie jest równoważna klasie `Task<TResult>` we wszystkim poza wartością zwracaną.

Zachowanie lokalnego kontekstu

Największą siłą wyrażeń `await` jest to, że mogą występować w niemal dowolnym miejscu kodu; w szczególności wyrażenie *await* (w obrębie funkcji asynchronicznej) może się pojawiać w miejscu, gdzie dozwolone jest wystąpienie dowolnego wyrażenia za wyjątkiem wyrażenia blokady, kontekstu nienadzorowanego, wnętrza bloku *catch* lub bloku *finally*, kodu nienadzorowanego czy, wreszcie, punktu wejścia do wykonania (czyli głównej metody programu).

W poniższym przykładzie stosujemy wyrażenie `await` w ciele pętli:

```
async void Test()
{
  for (int i = 0; i < 10; i++)
  {
    int result = await ComplexCalculationAsync();
```


Kup książkę Poleć książkę

```
    Console.WriteLine (result);
  }
}
```

Po pierwszym wywołaniu `ComplexCalculationAsync` sterowanie powraca do miejsca wywołania za wyrażeniem `await`. Kiedy metoda zakończy działanie (albo rzuci wyjątek), wykonanie zostanie wznowione w przerwanym miejscu, z zachowaniem wartości zmiennych lokalnych, w tym licznika pętli. Kompilator uzyskuje ten efekt poprzez translację powyższego kodu na maszynę stanową, podobnie jak w przypadku iteratorów.

Nieobecność słowa kluczowego `await` oznaczałaby, że to programista musiałby jawnie oprogramować taką maszynę stanową ze starannym ręcznym stosowaniem kontynuacji; tradycyjnie to właśnie ten element programowania asynchronicznego przysparza najwięcej trudności.

Pisanie funkcji asynchronicznych

W każdej funkcji synchronicznej można zastąpić pusty typ zwracany instancją klasy `Task`, a funkcja ta stanie się w pełni *użyteczną* funkcją asynchroniczną; żadne inne zmiany nie są konieczne:

```
async Task PrintAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Zauważmy, że w ciele metody nie ma jawnej instrukcji zwracającej obiekt zadania (`Task`). Zajmuje się tym kompilator, to on również obsługuje sygnalizację zakończenia bądź przerwania wykonania metody (w przypadku wyjątku). Dzięki temu można łatwo tworzyć łańcuchy wywołań asynchronicznych:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Gotowe");
}
```

(A ponieważ metoda `Go` zwraca `Task`, `Go` sama w sobie jest również asynchroniczna). Kompilator rozwinie funkcję asynchroniczną zwracającą zadanie współbieżne `Task` do postaci kodu, który (pośrednio) wykorzystuje klasę `TaskCompletionSource` do utworzenia zadania współbieżnego wraz z sygnalizacją zakończenia bądź błędu.

Kup książkę Poleć książkę

Uwaga

`TaskCompletionSource` to typ środowiska uruchomieniowego CLR, pozwalający na tworzenie zadań współbieżnych pozostających pod ręczną kontrolą programisty, a także zajmujący się sygnalizacją zakończenia wykonania współbieżnego albo błędu wykonania. W przeciwieństwie do `Task.Run`, `TaskCompletionSource` nie wiąże wątku, w którym występuje wywołanie, na czas wykonania operacji. `TaskCompletionSource` znajduje też zastosowanie przy pisaniu metod obsługujących wejście-wyjście i zwracających zadania współbieżne (jak `Task. Delay`).

Chodzi o zapewnienie, by kiedy metoda asynchroniczna zwracająca zadanie współbieżne zakończy wykonanie, sterowanie mogło zostać przekazane (za pośrednictwem kontynuacji) do miejsca, w którym ktoś na to oczekuje.

Zwracanie Task<TResult>

Jeśli ciało funkcji synchronicznej zwraca wartość uogólnionego typu `TResult`, można ją zamienić na asynchroniczną poprzez zamianę wartości zwracanej na `Task<TResult>`:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  // answer to int, więc nasza metoda zwraca Task<int>
  return answer;
}
```

Działanie metody `GetAnswerToLife` można zademonstrować poprzez wywołanie jej wewnątrz wcześniej zdefiniowanej metody `PrintAnswerToLife` (która z kolei jest wywoływana we wnętrzu `Go`):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Gotowe");
}
async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```


Kup książkę Poleć książkę

Funkcje asynchroniczne upodobniają programowanie asynchroniczne do klasycznego programowania synchronicznego. Poniżej zaprezentowany jest synchroniczny równoważnik powyższego łańcucha wywołań, w którym wywołanie `Go()` daje taki sam efekt po blokującym (synchronicznym) odczekaniu pięciu sekund:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Gotowe");
}
void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}
int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

Powyższy przykład ilustruje też podstawową zasadę projektowania funkcji asynchronicznych w C#, czyli pisanie metod najpierw jako synchronicznych, a następnie zastępowanie wywołań *synchronicznych* wywołaniami *asynchronicznymi* z oczekiwaniem na ich wykonanie współbieżne.

Współbieżność

Poznaliśmy najprostszy schemat programowania asynchronicznego, w którym tuż po wywołaniu funkcji asynchronicznych oczekujemy na zakończenie odpowiadających im zadań współbieżnych. W efekcie otrzymujemy program z uszeregowanym wykonaniem, z logicznego punktu widzenia równoważny programowi w pełni synchronicznemu.

Natomiast prawdziwie współbieżne wykonanie implementujemy wtedy, kiedy wywołujemy funkcje asynchroniczne bez momentalnego oczekiwania na ich zakończenie. Na przykład poniższy kod dokona współbieżnego, dwukrotnego wykonania metody `PrintAnswerToLife`:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

Współbieżny przebieg wykonania programu kończy się w miejscu oczekiwania na zakończenie obu współbieżnych zadań (tam też dochodzi do kolekcjonowania wyników albo rzucenia wyjątków). Klasa `Task` udostępnia statyczną metodę o nazwie `WhenAll`, agregującą oczekiwanie i przez to nieco wydajniejszą. Me-

Kup książkę Poleć książkę

toda `WhenAll` zwraca zadanie współbieżne, które kończy się, kiedy zakończą się wszystkie zadania współbieżne wskazane w wywołaniu:

```
await Task.WhenAll (PrintAnswerToLife(),
                    PrintAnswerToLife());
```

Metoda `WhenAll` to tak zwany agregator zadań (ang. *task combinator*; klasa `Task` udostępnia również agregator o nazwie `WhenAny`, kończący wykonanie w momencie zakończenia dowolnego z przekazanych zadań). Agregatory zadań współbieżnych zostały szczegółowo omówione w książce *C# 6.0 in a Nutshell*.

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

Kup książkę Poleć książkę

L

M

Kup książkę Poleć książkę

N

O

Kup książkę Poleć książkę

P

Kup książkę Poleć książkę

Q

R

S

T

Kup książkę Poleć książkę

U

V

W

Kup książkę Poleć książkę

X

Z

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

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

O'REILLY®

Przekonaj się, jaki jest C# — nowoczesny, prosty, efektywny!

C# został zaprojektowany jako obiektowy język programowania z kontrolą typów. C# w wersji 6.0 jest dojrzałą technologią, narzędziem, dzięki któremu można efektywnie pisać bezpieczny, przejrzysty i wydajny kod. Język ten z założenia ma być prosty w stosowaniu, niekiedy jednak programista potrzebuje pomocy: trzeba szybko coś sprawdzić, upewnić się co do przyjętych rozwiązań, przypomnieć sobie rzadziej używaną konstrukcję.

Niniejsza książka jest zwięzłym i praktycznym kompendium. Zawiera dokładnie to, co powinna — bez nudnych wywodów i niepotrzebnych dywagacji. Może posłużyć jako podręcznik do nauki C# lub jako poręczna ściągawka, pozwalająca na szybkie odnalezienie informacji. Jeśli tylko znasz podstawy programowania w Javie, C++ w poprzednich wersjach C# i chcesz bez większych problemów przystąpić do programowania w C# 6.0, trzymasz w ręku właściwą książkę.

Joseph Albahari — jest autorem kilku książek o programowaniu w języku C#. Jest twórcą LINQPada, popularnego narzędzia do prototypowania zapytań LINQ.

Ben Albahari pracował w firmie Microsoft, gdzie był szefem wielu ważnych projektów. Jest współtwórcą serwisu Auditionist, obsługującego wirtualne castingi dla aktorów w Wielkiej Brytanii.

W leksykonie przedstawiono:

- podstawy języka C#
- stosowanie technologii LINQ w pracy na kolekcjach danych
- wiązania dynamiczne i funkcje asynchroniczne
- wskaźniki, atrybuty, dyrektywy preprocesora i wiele innych zagadnień

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
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-2446-6

cena: 34,90 zł