

Najlepszy podręcznik poświęcony C#!

Programowanie

C# 5.0

*Tworzenie
aplikacji Windows 8,
internetowych
oraz biurowych
w .NET 4.5 Framework*



O'REILLY®

Ian Griffiths

Tytuł oryginału: Programming C# 5.0

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-6984-4

© 2013 Helion S.A.

Authorized Polish translation of the English edition Programming C# 5.0 ISBN 9781449320416 © 2013 Ian Griffiths.

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.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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/csh5pr>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/csh5pr.zip>

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	17
1. Prezentacja C#	21
Dlaczego C#?	21
Dlaczego nie C#?	23
Najważniejsze cechy C#	25
Kod zarządzany i CLR	27
Ogólność jest ważniejsza od specjalizacji	29
Programowanie asynchroniczne	30
Visual Studio	31
Anatomia prostego programu	33
Dodawanie projektów do istniejącej solucji	35
Odwołania do innych projektów	35
Pisanie testu jednostkowego	37
Przestrzenie nazw	40
Klasy	44
Punkt wejścia do programu	44
Testy jednostkowe	45
Podsumowanie	47
2. Podstawy stosowania języka C#	49
Zmienne lokalne	50
Zakres	55
Instrukcje i wyrażenia	58
Instrukcje	59
Wyrażenia	60
Komentarze i białe znaki	65
Dyrektywy preprocesora	67
Symbole kompilacji	67
Dyrektywy #error oraz #warning	68
Dyrektywa #line	69
Dyrektywa #pragma	69
Dyrektywy #region i #endregion	70

Wbudowane typy danych	70
Typy liczbowe	71
Wartości logiczne	80
Znaki i łańcuchy znaków	80
Object	81
Operatory	81
Sterowanie przepływem	87
Decyzje logiczne przy użyciu instrukcji if	87
Wielokrotny wybór przy użyciu instrukcji switch	89
Pętle: while oraz do	91
Pętle znane z języka C	92
Przeglądanie kolekcji przy użyciu pętli foreach	93
Podsumowanie	94
3. Typy	95
Klasy	95
Składowe statyczne	98
Klasy statyczne	100
Typy referencyjne	101
Struktury	106
Kiedy tworzyć typy wartościowe?	110
Składowe	115
Pola	115
Konstruktory	117
Metody	125
Właściwości	130
Indeksatory	134
Operatory	135
Zdarzenia	138
Typy zagnieżdżone	138
Interfejsy	140
Typy wyliczeniowe	141
Inne typy	144
Typy anonimowe	145
Typy i metody częściowe	146
Podsumowanie	147
4. Typy ogólne	149
Typy ogólne	150
Ograniczenia	152
Ograniczenia typu	153
Ograniczenia typu referencyjnego	155
Ograniczenia typu wartościowego	157
Stosowanie wielu ograniczeń	158

Wartości przypominające zero	158
Metody ogólne	160
Wnioskowanie typu	160
Tajniki typów ogólnych	161
Podsumowanie	163
5. Kolekcje	165
Tablice	165
Inicjalizacja tablic	168
Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów	169
Przeszukiwanie i sortowanie	171
Tablice wielowymiarowe	178
Kopiowanie i zmiana wielkości	181
List<T>	182
Interfejsy list i sekwencji	185
Implementacja list i sekwencji	189
Iteratory	190
Klasa Collection<T>	194
Klasa ReadOnlyCollection<T>	195
Słowniki	196
Słowniki posortowane	198
Zbiory	200
Kolejki i stosy	201
Listy połączone	202
Kolekcje współbieżne	203
Krotki	204
Podsumowanie	205
6. Dziedziczenie	207
Dziedziczenie i konwersje	208
Dziedziczenie interfejsów	210
Typy ogólne	211
Kowariancja i kontrawariancja	212
System.Object	217
Wszechobecne metody typu object	217
Dostępność i dziedziczenie	218
Metody wirtualne	220
Metody abstrakcyjne	222
Metody i klasy ostateczne	228
Dostęp do składowych klas bazowych	229
Dziedziczenie i tworzenie obiektów	230
Specjalne typy bazowe	234
Podsumowanie	235

7. Cykl życia obiektów	237
Mechanizm odzyskiwania pamięci	238
Określanie osiągalności danych	239
Przypadkowe problemy mechanizmu odzyskiwania pamięci	242
Słabe referencje	244
Odzyskiwanie pamięci	248
Tryby odzyskiwania pamięci	254
Przypadkowe utrudnianie scalania	256
Wymuszanie odzyskiwania pamięci	260
Destruktory i finalizacja	261
Finalizatory krytyczne	264
Interfejs IDisposable	265
Zwalnianie opcjonalne	271
Pakowanie	272
Pakowanie danych typu Nullable<T>	276
Podsumowanie	277
8. Wyjątki	279
Źródła wyjątków	281
Wyjątki zgłaszane przez API	282
Wyjątki w naszym kodzie	284
Błędy wykrywane przez środowisko uruchomieniowe	284
Obsługa wyjątków	285
Obiekty wyjątków	286
Wiele bloków catch	287
Zagnieżdżone bloki try	289
Bloki finally	290
Zgłaszanie wyjątków	292
Powtórne zgłaszanie wyjątków	292
Sposób na szybkie zakończenie aplikacji	295
Typy wyjątków	296
Wyjątki niestandardowe	298
Wyjątki nieobsługiwane	301
Debugowanie i wyjątki	303
Wyjątki asynchroniczne	305
Podsumowanie	308
9. Delegaty, wyrażenia lambda i zdarzenia	309
Typy delegatów	310
Tworzenie delegatów	311
MulticastDelegate — delegaty zbiorowe	314
Wywoływanie delegatów	316
Popularne typy delegatów	318
Zgodność typów	319
Więcej niż składnia	323

Metody inline	326
Przechwytywane zmienne	328
Wyrażenia lambda oraz drzewa wyrażień	335
Zdarzenia	336
Standardowy wzorzec delegatów zdarzeń	338
Niestandardowe metody dodające i usuwające zdarzenia	339
Zdarzenia i mechanizm odzyskiwania pamięci	342
Zdarzenia a delegaty	344
Delegaty a interfejsy	345
Podsumowanie	345
10. LINQ	347
Wyrażenia zapytań	348
Jak są rozwijane wyrażenia zapytań	351
Obsługa wyrażień zapytań	353
Przetwarzanie opóźnione	357
LINQ, typy ogólne oraz interfejs IQueryable<T>	359
Standardowe operatory LINQ	361
Filtrowanie	364
Selekcja	366
Operator SelectMany	369
Określanie porządku	371
Testy zawierania	373
Konkretne elementy i podzakresy	375
Agregacja	379
Operacje na zbiorach	384
Operatory działające na całych sekwencjach z zachowaniem kolejności	384
Grupowanie	386
Złączenia	390
Konwersje	392
Generowanie sekwencji	396
Inne implementacje LINQ	397
Entity Framework	397
LINQ to SQL	398
Klient WCF Data Services	398
Parallel LINQ (PLINQ)	399
LINQ to XML	399
Reactive Extensions	399
Podsumowanie	400
11. Reactive Extensions	401
Rx oraz różne wersje .NET Framework	403
Podstawowe interfejsy	405
Interfejs IObservable<T>	406
Interfejs IObservable<T>	407

Publikowanie i subskrypcja z wykorzystaniem delegatów	413
Tworzenie źródła przy wykorzystaniu delegatów	413
Subskrybowanie obserwowalnych źródeł przy użyciu delegatów	417
Generator sekwencji	418
Empty	418
Never	418
Return	419
Throw	419
Range	419
Repeat	419
Generate	420
Zapytania LINQ	421
Operatory grupowania	423
Operatory Join	424
Operator SelectMany	429
Agregacja oraz inne operatory zwracające jedną wartość	430
Operator Concat	431
Operatory biblioteki Rx	431
Merge	432
Operatory Buffer i Window	433
Operator Scan	440
Operator Amb	441
DistinctUntilChanged	442
Mechanizmy szeregujące	442
Określanie mechanizmów szeregujących	443
Wbudowane mechanizmy szeregujące	445
Tematy	447
Subject<T>	447
BehaviorSubject<T>	448
ReplaySubject<T>	449
AsyncSubject<T>	449
Dostosowanie	450
IEnumerable<T>	450
Zdarzenia .NET	452
API asynchroniczne	454
Operacje z uzależnieniami czasowymi	456
Interval	456
Timer	457
Timestamp	458
TimeInterval	459
Throttle	459
Sample	460
Timeout	460
Operatory okien czasowych	460
Delay	461
DelaySubscription	461
Podsumowanie	462

12. Podzespoły	463
Visual Studio i podzespoły	463
Anatomia podzespołu	464
Metadane .NET	465
Zasoby	465
Podzespoły składające się z wielu plików	466
Inne możliwości formatu PE	467
Tożsamość typu	468
Wczytywanie podzespołów	471
Jawne wczytywanie podzespołów	473
Global Assembly Cache	474
Nazwy podzespołów	476
Silne nazwy	476
Numer wersji	480
Identyfikator kulturowy	484
Architektura procesora	487
Przenośne biblioteki klas	488
Wdrażanie pakietów	490
Aplikacje dla systemu Windows 8	490
ClickOnce oraz XBAP	491
Aplikacje Silverlight oraz Windows Phone	492
Zabezpieczenia	493
Podsumowanie	494
13. Odzwierciedlanie	495
Typy odzwierciedlania	495
Assembly	498
Module	502
MemberInfo	503
Type oraz TypeInfo	506
MethodBase, ConstructorInfo oraz MethodInfo	510
ParameterInfo	512
FieldInfo	513
PropertyInfo	513
EventInfo	514
Konteksty odzwierciedlania	514
Podsumowanie	516
14. Dynamiczne określanie typów	517
Typ dynamic	519
Słowo kluczowe dynamic i mechanizmy współdziałania	521
Silverlight i obiekty skryptowe	524
Dynamiczne języki .NET	525

Tajniki typu dynamic	526
Ograniczenia typu dynamic	526
Niestandardowe obiekty dynamiczne	528
Klasa ExpandableObject	531
Ograniczenia typu dynamic	531
Podsumowanie	534
15. Atrybuty	535
Stosowanie atrybutów	535
Cele atrybutów	537
Atrybuty obsługiwane przez kompilator	539
Atrybuty obsługiwane przez CLR	543
Definiowanie i stosowanie atrybutów niestandardowych	551
Typ atrybutu	551
Pobieranie atrybutów	553
Podsumowanie	556
16. Pliki i strumienie	557
Klasa Stream	558
Położenie i poruszanie się w strumieniu	560
Opróżnianie strumienia	561
Kopiowanie	562
Length	562
Zwalnianie strumieni	564
Operacje asynchroniczne	565
Konkretne typy strumieni	565
Windows 8 oraz interfejs IRandomAccessStream	566
Typy operujące na tekstach	569
TextReader oraz TextWriter	570
Konkretne typy do odczytu i zapisu łańcuchów znaków	572
Kodowanie	574
Pliki i katalogi	578
Klasa FileStream	578
Klasa File	581
Klasa Directory	585
Klasa Path	586
Klasy FileInfo, DirectoryInfo oraz FileSystemInfo	588
Znane katalogi	589
Serializacja	590
Klasy BinaryReader oraz BinaryWriter	590
Serializacja CLR	591
Serializacja kontraktu danych	594
Klasa XmlSerializer	597
Podsumowanie	598

17. Wielowątkowość	599
Wątki	599
Wątki, zmienne i wspólny stan	601
Klasa Thread	607
Pula wątków	609
Powinowactwo do wątku oraz klasa SynchronizationContext	614
Synchronizacja	618
Monitory oraz słowo kluczowe lock	619
Klasa SpinLock	625
Blokady odczytu i zapisu	627
Obiekty zdarzeń	628
Klasa Barrier	631
Klasa CountdownEvent	632
Semaforey	632
Muteksy	633
Klasa Interlocked	634
Leniwa inicjalizacja	637
Pozostałe klasy obsługujące działania współbieżne	639
Zadania	640
Klasy Task oraz Task<T>	640
Kontynuacje	643
Mechanizmy szeregujące	645
Obsługa błędów	647
Niestandardowe zadania bezwątkowe	648
Związki zadanie nadrzędne — zadanie podrzędne	649
Zadania złożone	650
Inne wzorce asynchroniczne	651
Anulowanie	652
Równoległość	653
Klasa Parallel	653
Parallel LINQ	654
TPL Dataflow	654
Podsumowanie	655
18. Asynchroniczne cechy języka	657
Nowe słowa kluczowe: async oraz await	658
Konteksty wykonania i synchronizacji	662
Wykonywanie wielu operacji i pętli	663
Zwracanie obiektu Task	666
Stosowanie async w metodach zagnieżdżonych	667
Wzorzec słowa kluczowego await	668
Obsługa błędów	672
Weryfikacja poprawności argumentów	674
Wyjątki pojedyncze oraz grupy wyjątków	675
Operacje równoległe i nieobsłużone wyjątki	677
Podsumowanie	678

19. XAML	681
Platformy XAML	682
WPF	683
Silverlight	684
Windows Phone 7	686
Windows Runtime oraz aplikacje dostosowane do interfejsu użytkownika Windows 8	687
Podstawy XAML	688
Przestrzenie nazw XAML oraz XML	689
Generowane klasy i kod ukryty	690
Elementy podrzędne	692
Elementy właściwości	692
Obsługa zdarzeń	694
Wykorzystanie wątków	695
Układ	696
Właściwości	696
Panele	702
ScrollView	712
Zdarzenia związane z układem	712
Kontrolki	713
Kontrolki z zawartością	714
Kontrolki Slider oraz ScrollBar	717
Kontrolki postępów	718
Listy	719
Szablony kontroltek	721
Kontrolki użytkownika	724
Tekst	725
Wyświetlanie tekstów	725
Edycja tekstów	727
Wiązanie danych	729
Szablony danych	732
Grafika	735
Kształty	735
Bitmapy	736
Media	737
Style	738
Podsumowanie	739
20. ASP.NET	741
Razor	742
Wyrażenia	743
Sterowanie przepływem	745
Bloki kodu	746
Jawne wskazywanie treści	747
Klasy i obiekty stron	748
Stosowanie innych komponentów	749

Strony układu	749
Strony początkowe	751
Web Forms	752
Kontrolki serwerowe	752
Wyrażenia	758
Bloki kodu	758
Standardowe obiekty stron	759
Klasy i obiekty stron	759
Stosowanie innych komponentów	760
Strony nadrzędne	760
MVC	762
Typowy układ projektu MVC	763
Pisanie modeli	769
Pisanie widoków	771
Pisanie kontrolerów	772
Obsługa dodatkowych danych wejściowych	774
Generowanie łączy do akcji	776
Trasowanie	777
Podsumowanie	781
21. Współdziałanie	783
Wywoływanie kodu rodzimego	783
Szeregowanie	784
Procesy 32- i 64-bitowe	792
Bezpieczne uchwyt	793
Bezpieczeństwo	794
Mechanizm Platform Invoke	795
Konwencje wywołań	796
Obsługa łańcuchów znaków	797
Nazwa punktu wejścia	797
Wartości wynikowe technologii COM	798
Obsługa błędów Win32	802
Technologia COM	802
Czas życia obiektów RCW	803
Metadane	805
Skrypty	811
Windows Runtime	814
Metadane	815
Typy Windows Runtime	815
Bufory	816
Niebezpieczny kod	818
C++/CLI i Component Extensions	819
Podsumowanie	820
Skorowidz	821

Asynchroniczne cechy języka

Podstawową nowością wprowadzoną w C# 5.0 jest wsparcie języka dla stosowania i implementacji metod asynchronicznych. Metody asynchroniczne są niejednokrotnie najbardziej wydajnym sposobem korzystania z niektórych usług. Na przykład większość operacji wejścia-wyjścia jest wykonywana asynchronicznie przez jądro systemu operacyjnego, gdyż większość urządzeń peryferyjnych, takich jak kontrolery dysków lub karty sieciowe, jest w stanie wykonywać większość operacji autonomicznie. Wymagają użycia procesora wyłącznie podczas rozpoczęcia i zakańczania operacji.

Choć wiele usług dostarczanych przez system Windows ma w rzeczywistości asynchroniczny charakter, to jednak programiści często decydują się na korzystanie z nich przy użyciu metod synchronicznych (czyli takich, które kończą się przed wykonaniem tego, co miały zrobić). Jednak takie postępowanie jest marnowaniem zasobów, gdyż powoduje ono zablokowanie wątku aż do momentu zakończenia operacji wejścia-wyjścia. W systemie Windows wątki są cennym zasobem, dlatego też uzyskuje on najwyższą wydajność, gdy liczba działających w nim wątków systemowych jest stosunkowo niewielka. W idealnym przypadku liczba wątków systemowych będzie odpowiadać liczbie wątków sprzętowych, lecz jest to przypadek optymalny, wyłącznie jeśli możemy zagwarantować, że wątki będą blokowane tylko w sytuacjach, gdy nie mają żadnych innych prac do wykonania. (Różnice pomiędzy wątkami systemowymi oraz wątkami sprzętowymi zostały wyjaśnione w rozdziale 17.) Im więcej wątków będzie blokowanych w wywołaniach metod synchronicznych, tym więcej będziemy potrzebowali wątków do obsługi obciążenia, a to z kolei prowadzi do ograniczenia wydajności. Dlatego też w kodzie, w którym wydajność działania odgrywa bardzo dużą rolę, metody asynchroniczne są użyteczne, gdyż zamiast marnować zasoby poprzez zmuszanie wątku do oczekiwania na zakończenie operacji wejścia-wyjścia, wątek może zainicjować taką operację, a następnie w międzyczasie zająć się czymś innym.

Jednak problem z metodami asynchronicznymi polega na tym, że ich stosowanie jest znacząco bardziej złożone od korzystania z metod synchronicznych, zwłaszcza kiedy w grę wchodzi koordynacja wielu powiązanych ze sobą operacji oraz obsługa błędów. To właśnie z tego powodu programiści bardzo często wybierają mniej wydajne, synchroniczne rozwiązania. Jednak nowe, asynchroniczne możliwości języka C# 5.0 pozwalają na tworzenie kodu, który może korzystać z wydajnych, asynchronicznych API, zachowując przy tym jednocześnie znaczną część prostoty cechującej kod używający prostszych rozwiązań synchronicznych.

Nowe możliwości języka przydają się także w niektórych przypadkach, gdy głównym celem zapewnienia wydajności działania nie jest maksymalizacja przepustowości. W przypadku kodu aplikacji klienckich bardzo ważnym zagadnieniem jest unikanie blokowania wątku obsługi interfejsu użytkownika, a jednym z rozwiązań jest stosowanie metod asynchronicznych. Wsparcie dla kodu asynchronicznego, jakie zapewnia C#, jest w stanie obsługiwać problemy związane z powinowactwem do wątku, co w ogromnym stopniu ułatwia tworzenie kodu obsługi interfejsu użytkownika zapewniającego błyskawiczną reakcję na poczynania użytkownika aplikacji.

Nowe słowa kluczowe: `async` oraz `await`

C# udostępnia wsparcie dla programowania asynchronicznego, wprowadzając dwa słowa kluczowe: `async` oraz `await`. Pierwsze z nich nie jest przeznaczone do samodzielnego użycia. Umieszcza się je natomiast w deklaracjach metod, a jego zadaniem jest poinformowanie kompilatora, że w metodzie będą używane możliwości asynchroniczne. Jeśli słowo to nie zostanie umieszczone w deklaracji metody, to nie będzie jej można używać wraz ze słowem kluczowym `await`. Jest to prawdopodobnie nieco nadmiarowe — kompilator zgłasza błąd, jeśli spróbujemy użyć słowa kluczowego `await` bez `async`, czyli najwyraźniej jest w stanie określić, czy ciało metody próbuje korzystać z możliwości asynchronicznych. A zatem dlaczego musimy jawnie deklarować asynchroniczność metody? Otóż wynika to z dwóch powodów. Przede wszystkim, jak się niebawem przekonasz, te możliwości drastycznie zmieniają zachowanie kodu generowanego przez kompilator, dlatego też stanowi to wyraźny sygnał informujący wszystkie osoby przeglądające kod, że metoda działa w sposób asynchroniczny. A poza tym słowo `await` nie zawsze było słowem kluczowym języka C#, zatem wcześniej nie stało na przeszkodzie, by używać go jako identyfikatora. Być może firma Microsoft mogła zaprojektować gramatykę słowa `await` w taki sposób, by było ono traktowane jako słowo kluczowe wyłącznie w bardzo specyficznych kontekstach, dzięki czemu we wszystkich innych przypadkach mogłoby wciąż być traktowane jako zwyczajny identyfikator. Niemniej jednak twórcy języka C# zdecydowali się zastosować nieco bardziej ogólne podejście: otóż słowa `await` nie można używać jako identyfikatora wewnątrz metod, w których deklaracji zastosowano modyfikator `async`, natomiast we wszystkich pozostałych miejscach kodu może ono służyć za identyfikator.



Słowo kluczowe `async` nie zmienia sygnatury metody. Determinuje ono sposób kompilacji metody, a nie jej używania.

A zatem modyfikator `async` jedynie deklaruje chęć używania słowa kluczowego `await`. (Choć nie wolno nam używać słowa kluczowego `await` bez `async`, to jednak nie jest błędem umieszczenie modyfikatora `async` w deklaracji metody, która nie wykorzystuje słowa kluczowego `await`. Niemniej jednak takie rozwiązanie nie ma żadnego sensu, dlatego też jeśli wystąpi, kompilator wygeneruje ostrzeżenie). Listing 18.1 przedstawia dosyć typowy przykład metody asynchronicznej. Używa ona klasy `HttpClient`¹, by poprosić jedynie o nagłówki konkretnego zasobu (używając w tym celu standardowego żądania `HEAD`, które istnieje w protokole `HTTP`

¹ Została ona tutaj zastosowana zamiast prostszej klasy `WebClient`, której używaliśmy w przykładach przedstawianych w poprzednich rozdziałach, gdyż zapewnia większą kontrolę nad szczegółami wykorzystania protokołu `HTTP`.

właśnie do tego celu). Uzyskane wyniki są następnie wyświetlane w polu tekstowym stanowiącym element interfejsu użytkownika aplikacji — metoda ta stanowi fragment kodu ukrytego, obsługującego interfejs użytkownika aplikacji, który zawiera pole `TextBox` o nazwie `headerListTextBox`.

Listing 18.1. Stosowanie słów kluczowych `async` i `await` podczas pobierania nagłówków HTTP

```
private async void FetchAndShowHeaders(string url)
{
    using (var w = new HttpClient())
    {
        var req = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead);

        var headerStrings =
            from header in response.Headers
            select header.Key + ": " + string.Join(", ", header.Value);

        string headerList = string.Join(Environment.NewLine, headerStrings);
        headerListTextBox.Text = headerList;
    }
}
```

Powyższy kod zawiera jedno wyrażenie używające słowa kluczowego `await`, które zostało wyróżnione pogrubioną czcionką. Słowo to jest używane w wyrażeniach, które mogą być wykonywane przez dłuższy czas, zanim zwrócą wynik; oznacza ono, że dalsza część metody nie powinna być wykonana, dopóki operacja się nie zakończy. Wygląda to zatem jak zwyczajny, blokujący kod synchroniczny, jednak różnica polega na tym, że słowo kluczowe `await` nie powoduje zablokowania wątku.

Gdybyśmy chcieli zablokować wątek i poczekać na wyniki, to nic nie stoi na przeszkodzie, by to zrobić. Metoda `SendAsync` klasy `HttpClient` zwraca obiekt `Task<HttpResponseMessage>`, więc można by zastąpić wyrażenie z listingu 18.1 używające słowa kluczowego `await` wyrażeniem przedstawionym na listingu 18.2. Pobiera ono wartość właściwości `Result` zadania, a zgodnie z tym, czego dowiedzieliśmy się w rozdziale 17., jeśli zadanie nie zostało zakończone, to próba odczytu tej właściwości spowoduje zablokowanie wątku do czasu wygenerowania wyników (bądź do momentu, gdy zadanie zakończy się niepowodzeniem, lecz w takim przypadku wyrażenie zgłosi wyjątek).

Listing 18.2. Blokujący odpowiednik wyrażenia ze słowem kluczowym `await`

```
HttpResponseMessage response =
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead).Result;
```

Choć wyrażenie `await` zastosowane w kodzie z listingu 18.1 robi coś, co jest pozornie podobne do powyższej instrukcji, to jednak działa zupełnie inaczej. Jeśli wynik zadania nie będzie dostępny od razu, to niezależnie od tego, co sugeruje jego nazwa, słowo kluczowe `await` sprawi, że wątek będzie czekał. Zamiast tego spowoduje ono zakończenie wykonywanej metody. Można użyć debugera, by przekonać się, że wywołanie metody `FetchAndShowHeaders` kończy się natychmiast. Na przykład: jeśli wywołałyśmy tę metodę w procedurze obsługi kliknięcia przycisku, przedstawionej na listingu 18.3, to możemy ustawić jeden punkt przerwania w wierszu zawierającym wywołanie `Debug.WriteLine`, oraz drugi w kodzie z listingu 18.1, w wierszu zawierającym instrukcję aktualizującą wartość właściwości `headerListTextBox.Text`.

Listing 18.3. Wywołanie metody asynchronicznej

```
private void fetchHeadersButton_Click(object sender, RoutedEventArgs e)
{
    FetchAndShowHeaders("http://helion.pl/");
    Debug.WriteLine("Wywołanie metody zostało zakończone.");
}
```

Jeśli uruchomimy taki program w debuggerze, przekonamy się, że najpierw zatrzymamy się w punkcie przerwania umieszczonym w wierszu z listingu 18.3, a dopiero później w punkcie przerwania z listingu 18.1. Innymi słowy, fragment kodu z listingu 18.1 umieszczony za wyrażeniem ze słowem kluczowym `await` zostaje wykonany *po* tym, gdy sterowanie zostanie przekazane z metody do kodu, który ją wywołał. Najwyraźniej kompilator jakoś zmienia dalszą część metody w taki sposób, aby została wykonana przy użyciu wywołania zwrotnego, realizowanego po zakończeniu operacji asynchronicznej.



Debugger Visual Studio stosuje różne sztuczki podczas debugowania metod asynchronicznych, aby zapewnić nam możliwość analizowania ich krok po kroku jak normalnych metod. Zazwyczaj jest to całkiem przydatne, jednak czasami ukrywa prawdziwy przebieg realizacji programu. Opisany powyżej przykład został uważnie zaprojektowany w taki sposób, aby przekreślić starania Visual Studio i pokazać faktyczny sposób realizacji kodu.

Warto zauważyć, że kod z listingu 18.1 oczekuje, że będzie wykonywany w wątku obsługi interfejsu użytkownika, gdyż pod koniec metody modyfikuje wartość właściwości `Text` pola tekstowego. Metody asynchroniczne nie dają gwarancji, że powiadomienia o zakończeniu operacji będą generowane w tym samym wątku, w którym operacja została rozpoczęta — w większości przypadków będą one generowane w innych wątkach. Pomimo to kod z listingu 18.1 działa zgodnie z zamierzeniami. Oznacza to, że słowo kluczowe `await` nie tylko spowodowało przeniesienie połowy kodu metody do wywołania zwrotnego, lecz także zadbało za nas o prawidłową obsługę powinowactwa do wątku.

To wszystko wyraźnie pokazuje, że użycie słowa kluczowego `await` zmusza kompilator do przeprowadzenia drastycznych zmian w naszym kodzie. W C# 4.0, chcąc użyć tego asynchronicznego API, a następnie zaktualizować interfejs użytkownika, konieczne było zastosowanie kodu podobnego do tego z listingu 18.4. Wykorzystuje on technikę opisaną w rozdziale 17.: przygotowuje kontynuację dla zadania zwróconego przez metodę `SendAsync`, wykorzystując przy tym obiekt `TaskScheduler`, by zapewnić, że kod kontynuacji zostanie wykonany w wątku obsługi interfejsu użytkownika.

Listing 18.4. Samodzielne tworzenie odpowiednika metody asynchronicznej

```
private void OldSchoolFetchHeaders(string url)
{
    var w = new HttpClient();
    var req = new HttpRequestMessage(HttpMethod.Head, url);

    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead)
        .ContinueWith(sendTask =>
        {
            try
            {
                HttpResponseMessage response = sendTask.Result;
            }
        });
}
```

```

var headerStrings =
    from header in response.Headers
    select header.Key + ": " + string.Join(", ", header.Value);

string headerList =
    string.Join(Environment.NewLine, headerStrings);
headerListTextBox.Text = headerList;
}
finally
{
    w.Dispose();
}
},
uiScheduler);
}

```

Jest to przykład bardzo dobrego, bezpośredniego wykorzystania TPL i zapewnia podobne efekty jak kod z listingu 18.1, choć nie stanowi on dokładnej reprezentacji sposobu, w jaki kompilator C# przekształca kod. Jak dowiesz się z dalszej części rozdziału, słowo kluczowe `await` używa wzorca, który jest obsługiwany przez klasy `Task` oraz `Task<T>`, lecz który ich nie wymaga. Dodatkowo gwarantuje ono wygenerowanie kodu, który obsługuje wcześniejsze zakończenie (czyli sytuacje, gdy zadanie zostanie wykonane, zanim będziemy gotowi rozpocząć oczekiwanie na jego zakończenie) znacznie bardziej efektywnie niż kod z listingu 18.4. Jednak zanim poznasz wszelkie szczegóły tego, co robi kompilator, warto się dowiedzieć, jakie problemy kompilator za nas rozwiązuje — a to najlepiej zrobić, pokazując kod, który musielibyśmy napisać w C# 4.0.

Nasz aktualny przykład jest całkiem prosty, gdyż realizuje tylko jedną asynchroniczną operację; jednak oprócz dwóch opisanych wcześniej czynności — czyli utworzenia jakiegoś wywołania zwrotnego obsługującego zakończenie oraz zapewnienia, że zostanie ono wykonane w odpowiednim wątku — musimy także zadbać o odpowiednią obsługę instrukcji `using` zastosowanej w kodzie z listingu 18.1. Kod z listingu 18.4 nie może używać instrukcji `using`, gdyż obiekt `HttpClient` chcemy zwolnić dopiero w momencie, gdy nie będzie już nam potrzebny. Wywołanie metody `Dispose` tuż przed zakończeniem metody zewnętrznej nie zda egzaminu, gdyż musimy mieć możliwość użycia obiektu w kodzie kontynuacji, a to zazwyczaj nastąpi trochę po zakończeniu metody. A zatem musimy utworzyć obiekt w jednej metodzie (zewnętrznej) i zwolnić go w innej (wewnętrznej). A ponieważ sami wywołujemy przy tym metodę `Dispose`, zatem sami musimy zadbać o obsługę wyjątków. Dlatego też konieczne było umieszczenie całego kodu przeniesionego do metody zwrotnej w bloku `try` i wywołanie metody `Dispose` w bloku `finally`. (W rzeczywistości zastosowane rozwiązanie nie jest kompletne i niezawodne — gdyby konstruktor klasy `HttpRequestMessage` lub metoda pobierająca mechanizm szeregowania zadań, co jest raczej mało prawdopodobne, zgłosiły wyjątek, to używany obiekt `HttpClient` nie zostałby prawidłowo zwolniony. Innymi słowy, nasz kod obsługuje jedynie tę sytuację, gdy problemy pojawią się w samej operacji sieciowej).

Kod z listingu 18.4 używa mechanizmu szeregowania zadań, by wykonać kontynuację przy wykorzystaniu obiektu `SynchronizationContext`, aktywnego w momencie rozpoczynania operacji. Dzięki temu zapewniamy, że wywołanie zwrotne zostanie wykonane w wątku umożliwiającym aktualizację interfejsu użytkownika. Choć to w zupełności wystarcza do zapewnienia poprawnego działania naszego przykładu, to jednak słowo kluczowe `await` robi dla nas nieco więcej.

Konteksty wykonania i synchronizacji

Jeśli realizacja kodu dociera do wyrażenia zawierającego słowo kluczowe `await` oraz operację, której wykonanie nie zakończyło się od razu, to wygenerowany przez kompilator kod reprezentujący `await` zapewni pobranie aktualnego kontekstu wykonania. (Może się zdarzyć, że nie będzie to wymagało wielkiego zachodu — jeśli nie jest to pierwszy blok `await` w danej metodzie oraz jeśli używany kontekst nie został zmieniony, to będzie on już pobrany). Po zakończeniu operacji asynchronicznej dalsza część kodu metody zostanie wykonana przy wykorzystaniu kontekstu wykonania².

Zgodnie z informacjami podanymi w rozdziale 17., kontekst wykonania obsługuje pewne kontekstowe informacje o bezpieczeństwie oraz lokalny stan wątku, które muszą być przekazywane, gdy jedna metoda wywołuje drugą (i to nawet jeśli robi to bezpośrednio). Niemniej jednak istnieje jeszcze inny rodzaj kontekstu, który może nas interesować, a zwłaszcza jeśli tworzymy kod obsługi interfejsu użytkownika; chodzi o kontekst synchronizacji.

Choć wszystkie wyrażenia `await` pobierają kontekst wykonania, to decyzja o tym, czy wraz z nim należy pobrać także kontekst synchronizacji, zależy od typu, na który oczekujemy. Jeśli oczekujemy na daną typu `Task`, to domyślnie kontekst synchronizacji także zostanie pobrany. Zadania nie są jedynymi obiektami, na jakie można oczekiwać, informacje dotyczące sposobu, w jaki można dostosować typy do obsługi słowa kluczowego `await`, zostały podane w dalszej części rozdziału, w punkcie pt. „Wzorzec słowa kluczowego `await`”.

Czasami mogą się zdarzyć sytuacje, w których nie będziemy chcieli używać kontekstu synchronizacji. Jeśli chcemy wykonać jakąś operację asynchroniczną, rozpoczynając ją w wątku obsługi interfejsu użytkownika, a jednocześnie nie ma konieczności dalszego pozostawania w tym wątku, to planowanie wykonania wszystkich kontynuacji przy użyciu kontekstu synchronizacji będzie jedynie niepotrzebnym obciążeniem. Jeśli operacja asynchroniczna jest reprezentowana przez obiekt `Task` lub `Task<T>`, to używając zdefiniowanej w tych klasach metody `ConfigureAwait` możemy zadeklarować, że nie chcemy używać kontekstu synchronizacji. W takim przypadku zwracana jest nieznacznie zmieniona reprezentacja operacji asynchronicznej, a jeśli jej użyjemy w wyrażeniu `await` zamiast oryginalnego zadania, to bieżący kontekst synchronizacji zostanie zignorowany (oczywiście o ile w ogóle będzie dostępny). (Nie można natomiast zrezygnować z wykorzystania kontekstu wykonania). Listing 18.5 pokazuje, jak można korzystać z metody `ConfigureAwait`.

Listing 18.5. Stosowanie metody `ConfigureAwait`

```
private async void OnFetchButtonClick(object sender, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    using (Stream f = File.Create(fileTextBox.Text))
    {
        Task<Stream> getStreamTask = w.GetStreamAsync(urlTextBox.Text);
        Stream getStream = await getStreamTask.ConfigureAwait(false);

        Task copyTask = getStream.CopyToAsync(f);
        await copyTask.ConfigureAwait(false);
    }
}
```

² Okazuje się, że to samo dzieje się w przykładzie z listingu 18.4, gdyż TPL pobiera kontekst wykonywania za nas.

Powyższy kod reprezentuje procedurę obsługi kliknięć przycisku, dlatego też jest wykonywany w wątku obsługi interfejsu użytkownika. Pobiera on wartości właściwości `Text` kilku pól tekstowych, a następnie wykonuje pewną operację asynchroniczną — pobiera zawartość adresu URL i kopiuje pobrane dane do pliku. Po pobraniu zawartości dwóch właściwości `Text` powyższy kod nie używa już żadnych elementów interfejsu użytkownika, a zatem jeśli wykonanie operacji asynchronicznej trochę zajmuje, to nie będzie miało żadnego znaczenia, że jej pozostała część zostanie wykonana w innym wątku. Poprzez przekazanie wartości `false` w wywołaniu metody `ConfigureAwait` oraz poczekanie na zwróconą przez nie wartość informujemy TPL, że do zakończenia operacji może zostać wykorzystany dowolny wątek, przy czym w tym przypadku będzie to najprawdopodobniej jeden z wątków dostępnych w puli. Dzięki temu operacja będzie mogła zostać wykonana szybciej i bardziej efektywnie, gdyż nie będzie musiała bez potrzeby korzystać z wątku obsługi interfejsu użytkownika po każdym słowie kluczowym `await`.

Kod przedstawiony na listingu 18.1 zawiera tylko jedno wyrażenie ze słowem kluczowym `await`, lecz nawet ten kod trudno jest odtworzyć, wykorzystując klasyczny model programowania z użyciem TPL. Przykład z listingu 18.5 zawiera dwa takie wyrażenia, a odtworzenie sposobu jego działania bez pomocy `await` wymagałoby użycia dosyć rozbudowanego kodu, gdyż wyjątki mogłyby być zgłaszane przed pierwszym wyrażeniem `await`, po drugim wyrażeniu oraz pomiędzy nimi; oprócz tego w każdym z tych przypadków (jak również w sytuacji, gdy nie zostały zgłoszone żadne wyjątki) musielibyśmy zadbać o wywołanie metody `Dispose` w celu zwolnienia używanych obiektów `HttpClient` oraz `Stream`. Niemniej jednak sytuacja staje się znacząco bardziej skomplikowana, kiedy w grę zaczyna dodatkowo wchodzić sterowanie przepływem.

Wykonywanie wielu operacji i pętli

Załóżmy, że zamiast pobierać nagłówki lub kopiować zawartość odpowiedzi HTTP do pliku, chcemy tę zawartość przetworzyć. Jeśli jest ona bardzo duża, to pobranie jej jest operacją, która może wymagać wykonania wielu czasochłonnych kroków. Przykład przedstawiony na listingu 18.6 pobiera całą stronę WWW wiersz po wierszu.

Listing 18.6. Wykonywanie wielu operacji asynchronicznych

```
private async void FetchAndShowBody(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                headerListTextBox.AppendText(line);
                headerListTextBox.AppendText(Environment.NewLine);
                await Task.Delay(TimeSpan.FromMilliseconds(10));
            }
        }
    }
}
```

W powyższym kodzie zostały użyte trzy wyrażenia `await`. Pierwsze z nich powoduje wykonanie żądania HTTP GET, a operacja ta zakończy się w momencie odebrania pierwszej części odpowiedzi, choć w tym momencie odpowiedź może jeszcze nie być kompletna — może zawierać jeszcze kilka megabajtów danych, które trzeba będzie jeszcze przekazać. Powyższy przykład zakłada, że zawartość odpowiedzi będzie tekstowa, dlatego też przekazuje zwrócony obiekt `Stream` jako argument wywołania konstruktora strumienia `StreamReader`, który udostępnia bajty stanowiące zawartość strumienia jako tekst³. Następnie przykład używa metody `ReadLineAsync`, by wiersz po wierszu odczytywać zawartość odpowiedzi. Ponieważ dane są przesyłane fragmentami, zatem odczytanie pierwszego wiersza tekstu może trochę zająć, jednak kilka kolejnych wywołań metody zostanie zapewne wykonanych momentalnie, gdyż każdy odebrany pakiet sieciowy zazwyczaj zawiera więcej wierszy. Jeśli jednak nasz kod może odczytywać dane szybciej, niż są przesyłane siecią, to w końcu odczyta wszystkie wiersze, które były dostępne w pierwszym pakiecie, i pewnie minie trochę czasu, zanim pojawią się kolejne. Dlatego też wywołania metody `ReadLineAsync` będą zwracały zarówno zadania, których wykonanie zajmuje więcej czasu, jak i takie, które zostaną zakończone błyskawicznie. Trzecią operacją asynchroniczną jest wywołanie metody `Task.Delay`. W powyższym przykładzie została ona użyta po to, by nieco zwolnić odczyt danych i aby kolejne wiersze tekstu pojawiały się w interfejsie użytkownika stopniowo. Metoda `Task.Delay` zwraca obiekt `Task`, który zostanie zakończony po upływie określonego czasu; stanowi ona zatem asynchroniczny odpowiednik metody `Thread.Sleep`. (Metoda `Thread.Sleep` blokuje wątek, w którym została wywołana, natomiast wyrażenie `await Task.Delay` wprowadza opóźnienie bez blokowania wątku).



W powyższym przykładzie każde z wyrażen `await` zostało umieszczone w odrębnej instrukcji; takie rozwiązanie nie jest jednak konieczne. Nic nie stoi na przeszkodzie, by użyć wyrażenia o następującej postaci: `(await t1) + (await t2)`. (W razie potrzeby można pominąć nawiasy, gdyż operator `await` ma wyższy priorytet niż operator dodawania, ja jednak preferuję wizualny porządek i hierarchię, jaką one zapewniają).

Nie przedstawię tu pełnego odpowiednika kodu z listingu 18.6, który należałoby napisać w języku C# 4.0, gdyż jest on zbyt duży. Ograniczę się jedynie do przedstawienia kilku problemów. Przede wszystkim w powyższym kodzie używamy pętli, wewnątrz której zostały umieszczone dwa wyrażenia `await`. Odtworzenie analogicznego kodu z użyciem obiektów `Task` i wywołań zwrotnych oznaczałoby konieczność stworzenia własnego odpowiednika pętli, gdyż jej zawartość musi zostać rozdzielona na trzy metody: pierwsza z nich rozpoczynałaby działanie pętli (byłaby ona metodą zagnieżdżoną, działającą jako kontynuacja metody `GetStreamAsync`), a pozostałe dwie byłyby wywołaniami zwrotnymi obsługującymi zakończenie operacji `ReadLineAsync` oraz `Task.Delay`. Takie rozwiązanie można by zaimplementować, tworząc metodę zagnieżdżoną służącą do rozpoczynania kolejnych iteracji i wywołując ją z dwóch miejsc: w miejscu, w którym chcemy rozpocząć działanie pętli, oraz w kontynuacji zadania `Task.Delay` w celu rozpoczęcia kolejnej iteracji pętli. Ta technika została zaprezentowana na listingu 18.7, choć przedstawia on tylko jeden aspekt działań, które wykonuje za nas kompilator — nie jest on kompletnym odpowiednikiem kodu z listingu 18.6.

³ Precyzyjnie rzecz ujmując, powinniśmy sprawdzić nagłówki odpowiedzi HTTP, by określić użyty sposób kodowania i w odpowiedni sposób skonfigurować obiekt `StreamReader`. Jednak w tym przykładzie pozwalamy, by obiekt strumienia sam określił sposób kodowania, co na potrzeby przykładu powinno działać wystarczająco dobrze.

Listing 18.7. Niekompletna samodzielna implementacja pętli asynchronicznej

```
private void IncompleteOldSchoolFetchAndShowBody(string url)
{
    var w = new HttpClient();
    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.GetStreamAsync(url).ContinueWith(getStreamTask =>
    {
        Stream body = getStreamTask.Result;
        var bodyTextReader = new StreamReader(body);

        Action startNextIteration = null;
        startNextIteration = () =>
        {
            if (!bodyTextReader.EndOfStream)
            {
                bodyTextReader.ReadLineAsync()
                    .ContinueWith(readLineTask =>
                    {
                        string line = readLineTask.Result;

                        headerListTextBox.AppendText(line);
                        headerListTextBox.AppendText(Environment.NewLine);

                        Task.Delay(TimeSpan.FromMilliseconds(10))
                            .ContinueWith(delayTask =>
                                startNextIteration(), uiScheduler);
                    },
                    uiScheduler);
            }
        };
        startNextIteration();
    },
    uiScheduler);
}
```

Ten kod działa jako tako, jednak nawet nie podejmuje próby zwolnienia któregośkolwiek z używanych zasobów. Występuje w nim kilka miejsc, w których potencjalnie może dojść do awarii, dlatego nie wystarczy umieścić w kodzie jednej instrukcji `using` lub pary bloków `try/finally`, aby zabezpieczyć działanie kodu. A nawet bez tego dodatkowego utrudnienia działanie kodu ledwie można zrozumieć — wcale nie jest oczywiste, że próbuje on wykonać te same operacje co przykład z listingu 18.6. Po dodaniu odpowiedniej obsługi błędów ten kod byłby całkowicie niezrozumiały. W praktyce zapewne łatwiej by było zastosować całkowicie inne rozwiązanie, polegające na napisaniu klasy implementującej maszynę stanów i na tej podstawie określającej czynności, jakie ma wykonywać. Takie rozwiązanie zapewne ułatwiłoby napisanie prawidłowo działającego kodu, jednak wcale nie ułatwiłoby osobie analizującej kod zorientować się, że to, na co patrzy, jest w rzeczywistości niewiele więcej niż pętla.

Nic zatem dziwnego, że tak wielu programistów preferuje stosowanie rozwiązań synchronicznych. Jednak C# 5.0 pozwala nam pisać kod asynchroniczny, który ma niemal taką samą strukturę co jego synchroniczny odpowiednik, bezboleśnie zapewniając nam przy tym wszystkie korzyści związane z większą wydajnością działania i sprawnym reagowaniem na poczynania użytkownika. Najprościej rzecz ujmując, właśnie te korzyści zapewniają nam słowa kluczowe `async` oraz `await`.

Każda metoda wykorzystująca słowo kluczowe `await` sama będzie wykonywana przez jakiś określony czas. A zatem oprócz korzystania z asynchronicznych API możemy uznać za stosowne, by stworzyć dla niej jakąś asynchroniczną reprezentację. Oba przedstawione słowa kluczowe pomagają nam to zrobić.

Zwracanie obiektu Task

Kompilator C# narzuca pewne ograniczenia na typy wartości wynikowych, które mogą zwracać metody oznaczone modyfikatorem `async`. Jak już się dowiedzieliśmy, mogą one zwracać `void`, jednak oprócz tego istnieją dwie inne możliwości: można zwracać instancję typu `Task` bądź typu `Task<T>`, gdzie `T` jest dowolnym typem. Dzięki temu kod wywołujący naszą asynchroniczną metodę może uzyskiwać informacje o statusie wykonywanych przez nią prac, a oprócz tego dysponuje możliwością dołączania do niej kontynuacji, a także pobierania wyniku (jeśli zwracany jest obiekt `Task<T>`). Oznacza to oczywiście, że jeśli nasza metoda jest wywoływana wewnątrz innej metody asynchronicznej (oznaczonej modyfikatorem `async`), to jej wynik będzie można pobrać, używając słowa kluczowego `await`.

Zwracanie zadań jest zazwyczaj bardziej preferowanym rozwiązaniem niż zwracanie typu `void`, gdyż w tym drugim przypadku kod wywołujący nie dysponuje tak naprawdę możliwością określenia, kiedy metoda została zakończona oraz czy należy zgłosić wyjątek. (Metody asynchroniczne mogą działać nawet po przekazaniu sterowania do kodu wywołującego — w końcu właśnie o to w nich chodzi — a zatem w momencie, kiedy nasza metoda zgłosi wyjątek, metody, która ją wywołała, może już w ogóle nie być na stosie). Zwracając obiekt `Task` lub `Task<T>`, zapewniamy kompilatorowi możliwość udostępniania wyjątków oraz w razie potrzeby zwracania wyników.



Oprócz ograniczenia nakazującego stosowanie modyfikatora `async` wyłącznie w metodach zwracających wynik typu `void`, `Task` bądź `Task<T>` nie można go także używać w metodzie stanowiącej punkt wejścia do programu, czyli w metodzie `Main`.

Zwrócenie zadania jest tak trywialnie proste, że nie ma żadnego powodu, by tego nie robić. Aby zmodyfikować metodę z listingu 18.6 tak, by zwracała zadanie, trzeba wprowadzić tylko jedną zmianę. Wystarczy zmienić typ wartości wynikowej z `void` na `Task`, jak pokazuje listing 18.8 — reszta kodu może pozostać bez zmian.

Listing 18.8. Zwracanie zadania

```
private async Task FetchAndShowBody(string url)
... jak wcześniej
```

Kompilator automatycznie generuje kod wymagany do utworzenia obiektu `Task` i w zależności od tego, czy metoda zwróci wynik, czy zgłosi wyjątek, ustawia jego status na `zakończony` lub `zakończony niepowodzeniem`. Także zwracanie wyniku z zadania jest bardzo łatwe. Wystarczy użyć typu `Task<T>`, a w kodzie metody umieścić instrukcję `return`, jak gdyby zwracała ona wartość typu `T`. Przykład takiej metody został przedstawiony na listingu 18.9.

Listing 18.9. Zwracanie zadania `Task<T>`

```
public static async Task<string> GetServerHeader(string url)
{
    using (var w = new HttpClient())
    {
        var request = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(request, HttpCompletionOption.ResponseHeadersRead);

        string result = null;
        IEnumerable<string> values;
```



```

        if (response.Headers.TryGetValues("Server", out values))
        {
            result = values.FirstOrDefault();
        }
        return result;
    }
}

```

Powyzsza metoda asynchronicznie pobiera naglowki HTTP, tak samo jak przyklad z listingu 18.1, jednak zamiast je wyswietlac, pobiera i zwraca wartosc pierwszego naglowka `Server`. Jak widać, instrukcja `return` zwraca lancuch znakow, choc zadeklarowanym typem wartosci wynikowej metody jest `Task<string>`. Kompilator generuje kod, który konczy wykonywanie zadania i uzywa zwróconego lancucha znakow jako wyniku. W przypadku uzycia typu `Task` lub `Task<T>` wygenerowany kod zwraca zadanie bardzo podobne do tego, które można uzyć, uzywając klasy `TaskCompletionSource<T>`, opisanej w rozdziale 17.



Choc slowo kluczowe `await` może operować na dowolnej metodzie asynchronicznej pasującej do określonego wzorca (opisanego w dalszej części rozdziału), to jednak C# nie zapewnia równie wielkiej elastyczności, jeśli chodzi o możliwości implementacji metod asynchronicznych. Jedynymi typami, jakie mogą zwracać metody z modyfikatorem `async` są: `Task`, `Task<T>` oraz `void`.

Jednak zwracanie zadań ma pewną wadę. Otóż kod wywołujący nie ma obowiązku robić czegokolwiek z tak zwróconym zadaniem, zatem nasza metoda może być równie łatwa w użyciu co metoda zwracająca typ `void`, a jednocześnie ma tę zaletę, że udostępnia zadanie, które kod wywołujący może wykorzystać. Chyba jedynym powodem zwracania typu `void` mogłoby być narzucenie przez kod zewnętrzny konieczności użycia metody o określonej sygnaturze. Na przykład większość procedur obsługi zdarzeń musi używać typu `void`. Jednak oprócz sytuacji, gdy jesteśmy do tego zmuszeni, stosowanie w metodach asynchronicznych typu `void` nie jest zalecane.

Stosowanie `async` w metodach zagnieżdżonych

W przykładach przedstawionych do tej pory używaliśmy słowa kluczowego `async` tylko w zwyczajnych metodach. Jednak można je także stosować w metodach zagnieżdżonych — zarówno metodach anonimowych, jak i w wyrażeniach `lambda`. Na przykład: jeśli piszemy program, który tworzy elementy interfejsu użytkownika programowo, wygodnym rozwiązaniem może być dołączanie procedur obsługi zdarzeń w formie wyrażeń `lambda`, możemy się przy tym zdecydować, by niektóre z nich zostały zaimplementowane jako asynchroniczne, jak pokazano na listingu 18.10.

Listing 18.10. *Asynchroniczne wyrażenie lambda*

```

okButton.Click += async (s, e) =>
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};

```

Składnia asynchronicznej metody anonimowej jest bardzo podobna, jak widać w przykładzie przedstawionym na listingu 18.11.

Listing 18.11. *Asynchroniczna metoda anonimowa*

```
okButton.Click += async delegate (object s, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};
```

Żeby wszystko było jasne — powyższy kod nie ma nic wspólnego z asynchronicznym wywoływaniem delegatów, czyli techniką, o której wspominałem w rozdziale 9., służącą do korzystania z puli wątków i popularną, zanim metody anonimowe i TPL stały się lepszą alternatywą. Asynchroniczne wywoływanie delegatów jest rozwiązaniem, na które może się zdecydować kod korzystający z delegatu — jednak w takim przypadku asynchroniczność nie jest ani cechą delegatu, ani metody, która go wywołuje. Jest to jedynie rozwiązanie zastosowane przez kod używający delegatu. Jednak zastosowanie modyfikatora `async` w metodzie anonimowej lub wyrażeniu lambda pozwala nam na korzystanie wewnątrz nich ze słowa kluczowego `await`, zmieniając w ten sposób kod metody generowany przez kompilator.

Wzorzec słowa kluczowego `await`

Większość metod asynchronicznych, których będziemy używać wraz ze słowem kluczowym `await`, zwraca jakieś zadania TPL. Niemniej jednak C# wcale tego nie wymaga. Kompilator pozwala na stosowanie ze słowem kluczowym `await` dowolnych obiektów, implementujących określony wzorzec. Choć klasy `Task` i `Task<T>`, obsługują ten wzorzec, to jednak sposób jego działania oznacza, że kompilator będzie używał zadań w nieco inny sposób, niż to robimy, korzystając z biblioteki TPL bezpośrednio — po części właśnie z tego powodu napisałem wcześniej, że kod wykorzystujący zadania i stanowiący odpowiednik kodu używającego słowa kluczowego `await` nie stanowi dokładnego odpowiednika kodu generowanego przez kompilator. W tym podrozdziale wyjaśnię, jak kompilator używa zadań oraz innych typów, które mogą być stosowane wraz ze słowem kluczowym `await`.

W dalszej części tego podrozdziału stworzymy własną implementację wzorca słowa kluczowego `await`, aby pokazać, czego oczekuje kompilator. (Tak się składa, że Visual Basic rozpoznaje i obsługuje dokładnie ten sam wzorzec). Listing 18.12 przedstawia metodę asynchroniczną o nazwie `UseCustomAsync`, która korzysta z naszej asynchronicznej implementacji. Metod ta zapisuje wynik wyrażenia `await` w zmiennej typu `string`, a zatem najwyraźniej oczekuje, że nasza asynchroniczna operacja zwróci łańcuch znaków. Wywołuje ona metodę `CustomAsync`, zwracającą tę implementację wzorca. Jak widać, nie jest to wcale `Task<string>`.

Listing 18.12. *Wywoływanie niestandardowej implementacji typu współpracującego z `await`*

```
static async Task UseCustomAsync()
{
    string result = await CustomAsync();
    Console.WriteLine(result);
}

public static MyAwaitableType CustomAsync()
{
    return new MyAwaitableType();
}
```

Kompilator oczekuje, że typ operandu słowa kluczowego `await` będzie udostępniał metodę o nazwie `GetAwaiter`. Może to być zwyczajna metoda składowa bądź metoda rozszerzenia. (A zatem definiując odpowiednią metodę rozszerzenia, można sprawić, że słowo kluczowe `await` będzie współpracowało z typem, który sam z siebie go nie obsługuje). Metoda ta musi zwracać obiekt lub wartość zapewniającą trzy możliwości.

Przede wszystkim musi udostępniać właściwość typu `bool` o nazwie `IsCompleted`, którą kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` będzie sprawdzał w celu określenia, czy operacja już się zakończyła. W przypadku gdy operacja została już zakończona, przygotowywanie wywołania zwrotnego byłoby stratą czasu. A zatem kod obsługujący słowo kluczowe `await` nie będzie tworzyć niepotrzebnego delegatu, jeśli właściwość `IsCompleted` zwróci wartość `true`, a zamiast tego od razu wykona dalszą część metody.

Oprócz tego kompilator wymaga jakiegoś sposobu pobrania wyniku, kiedy operacja zostanie już zakończona. Dlatego też obiekt lub wartość zwracana przez metodę `GetAwaiter` musi udostępniać metodę `getResult`. Typ wyniku zwracanego przez tę metodę definiuje typ wyniku operacji — a zatem będzie to typ całego wyrażenia `await`. W przykładzie z listingu 18.12 wynik wyrażenia `await` jest zapisywany w zmiennej typu `string`, a zatem wynik zwracany przez metodę `getResult` obiektu zwróconego przez metodę `GetAwaiter` klasy `MyAwaitableType` musi być typu `string` (bądź jakiegoś innego typu, który niejawnie można skonwertować na `string`).

I w końcu ostatnią możliwością, której potrzebuje kompilator, jest dostarczenie metody zwrotnej. Jeśli właściwość `IsCompleted` zwróci wartość `false`, informując tym samym, że operacja jeszcze się nie zakończyła, to kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` wygeneruje delegat, który wykona pozostałą część kodu metody. (Przypomina to nieco przekazywanie delegatu do metody `ContinueWith` zadania). Kompilator wymaga w tym celu nie metody, lecz całego interfejsu. Musimy zatem zaimplementować interfejs `INotifyCompletion`, lecz oprócz niego istnieje jeszcze jeden interfejs, `ICriticalNotifyCompletion`, którego implementacja jest zalecana, o ile tylko jest to możliwe. Oba te interfejsy są podobne: każdy z nich definiuje jedną metodę (`OnCompleted` oraz `UnsafeOnCompleted`), która pobiera jeden delegat typu `Action`, który klasa implementująca interfejs musi wywołać w momencie zakończenia operacji. Oba te interfejsy oraz ich metody różnią się tym, że pierwszy z nich wymaga od klasy implementującej przekazania kontekstu wykonania do metody docelowej, natomiast w przypadku drugiego interfejsu nie jest to konieczne. Kompilator C# zawsze przekazuje kontekst wykonania za nas, a zatem jeśli metoda `UnsafeOnCompleted` będzie dostępna, to kompilator wywoła ją, by uniknąć dwukrotnego przekazywania kontekstu. (Jeśli kompilator wywoła metodę `OnCompleted`, to także obiekt zwrócony przez metodę `GetAwaiter` przekaże kontekst wykonania). Niemniej jednak skorzystanie z metody `UnsafeOnCompleted` może być niemożliwe ze względów bezpieczeństwa. Ponieważ metoda ta nie przekazuje kontekstu wykonania, zatem kod niedysponujący pełnym zaufaniem nie może jej wywoływać, gdyż pozwoliłoby to na ominięcie pewnych mechanizmów zabezpieczeń. Metoda `UnsafeOnCompleted` jest oznaczona atrybutem `SecurityCriticalAttribute`, co oznacza, że może ją wywoływać tylko kod dysponujący pełnym zaufaniem. A zatem metoda `OnCompleted` jest potrzebna, by także kod, który nie dysponuje pełnym zaufaniem, mógł korzystać z obiektu zwracanego przez metodę `GetAwaiter`.

Listingu 18.13 przedstawia minimalną, nadającą się do użycia implementację wzorca słowa kluczowego `await`. Przedstawiony kod jest jednak bardzo uproszczony, gdyż zawsze kończy się synchronicznie, a zatem jego metoda `OnCompleted` nic nie robi. W rzeczywistości, jeśli nasza przykładowa klasa zostanie użyta w taki sposób, w jaki wzorzec `await` ma być używany,

to jej metoda `OnCompleted` w ogóle nie zostanie wywołana — właśnie dlatego zgłasza wyjątek. Niemniej jednak choć przedstawiony przykład jest nierealistycznie prosty, to jednak całkiem dobrze pokazuje sposób działania słowa kluczowego `await`.

Listing 18.13. Wyjątkowo prosta implementacja wzorca słowa kluczowego `await`

```
public class MyAwaitableType
{
    public MinimalAwaiter GetAwaiter()
    {
        return new MinimalAwaiter();
    }

    public class MinimalAwaiter : INotifyCompletion
    {
        public bool IsCompleted { get { return true; } }

        public string GetResult()
        {
            return "Oto wynik!";
        }

        public void OnCompleted(Action continuation)
        {
            throw new NotImplementedException();
        }
    }
}
```

Po przedstawieniu tego kodu możemy już zobaczyć, jak działa przykład z listingu 18.12. Wywoła on metodę `GetAwaiter` instancji typu `MyAwaitableType` zwróconej przez metodę `CustomAsync`. Następnie sprawdzi wartość właściwości `IsCompleted` uzyskanego obiektu i jeśli okaże się, że ma ona wartość `true` (co też się stanie), to bezzwłocznie zostanie wykonana reszta metody. Kompilator nie wie o tym, że właściwość `IsCompleted` zawsze ma wartość `true`, dlatego też wygeneruje kod pozwalający na prawidłowe obsłużenie przypadku, gdyby właściwość ta przyjęła wartość `false`. Kod ten utworzy delegat, który kiedy zostanie wywołany, wykona pozostałą część metody, po czym przekaże ten delegat do metody `OnComplete`. (Nasz przykładowy kod nie implementuje metody `UnsafeOnCompleted`, zatem zostanie użyta metoda `OnCompleted`). Kod, który wykonuje te wszystkie operacje, został przedstawiony na listingu 18.14.

Listing 18.14. Bardzo ogólne przybliżenie działania słowa kluczowego `await`

```
static void ManualUseCustomAsync()
{
    var awaiter = CustomAsync().GetAwaiter();
    if (awaiter.IsCompleted)
    {
        TheRest(awaiter);
    }
    else
    {
        awaiter.OnCompleted(() => TheRest(awaiter));
    }
}

private static void TheRest(MyAwaitableType.MinimalAwaiter awaiter)
{
    string result = awaiter.GetResult();
    Console.WriteLine(result);
}
```

Metoda została podzielona na dwie części, gdyż kompilator unika tworzenia delegatu, jeśli właściwość `IsCompleted` przyjmie wartość `true`, a my chcemy, by nasz kod działał podobnie. Niemniej jednak nie jest to dokładnie to samo, co robi kompilator C# — potrafi on także uniknąć tworzenia dodatkowych metod dla poszczególnych instrukcji `await`, choć oznacza to, że generowany przez niego kod jest znacząco bardziej skomplikowany. W rzeczywistości w przypadku metod zawierających tylko jedno słowo kluczowe `await` generowany przez kompilator narzut jest znacząco większy do tego z listingu 18.14. Niemniej jednak wraz ze wzrostem liczby używanych wyrażeń `await` ta dodatkowa złożoność zaczyna się opłacać, gdyż kompilator nie musi dodawać kolejnych metod. Listing 18.15 przedstawia kod, który w nieco większym stopniu przypomina to, co faktycznie generuje kompilator.

Listing 18.15. Nieco lepsze przybliżenie sposobu działania słowa kluczowego `await`

```
private class ManualUseCustomAsyncState
{
    private int state;
    private MyAwaitableType.MinimalAwaiter awaiter;

    public void MoveNext()
    {
        if (state == 0)
        {
            awaiter = CustomAsync().GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                state = 1;
                awaiter.OnCompleted(MoveNext);
                return;
            }
        }
        string result = awaiter.GetResult();
        Console.WriteLine(result);
    }
}

static void ManualUseCustomAsync()
{
    var s = new ManualUseCustomAsyncState();
    s.MoveNext();
}
```

Powyższy kod i tak jest prostszy do tego, który kompilator generuje w rzeczywistości, jednak pokazuje ogólną strategię działania: kompilator generuje zagnieżdżony typ działający jako maszyna stanów. Definiuje on pole (`state`) przechowujące informacje o tym, do którego miejsca dotarła realizacja metody, oraz pola reprezentujące zmienne lokalne metody. (W powyższym przykładzie jest to jedynie zmienna `awaiter`). Kiedy operacja asynchroniczna nie zostaje zablokowana (czyli gdy właściwość `IsCompleted` natychmiast zwróci wartość `true`), od razu można wykonać następną część kodu. Jednak w przypadku, gdy wykonanie operacji wymaga nieco czasu, aktualizowana jest wartość zmiennej `state` w celu zapamiętania aktualnego stanu, po czym wywoływana jest metoda `OnCompleted` obiektu zwróconego przez metodę `GetAwaiter`. Należy zwrócić uwagę, że metodą, którą chcemy wywołać po zakończeniu operacji, jest aktualnie wykonywana metoda, czyli `MoveNext`. Takie rozwiązanie jest stosowane zawsze, niezależnie od ilości zastosowanych słów kluczowych `await` — każde wywołanie zwrotne wykonywane po zakończeniu operacji asynchronicznej powoduje wywołanie tej samej metody — po prostu klasa pamięta, dokąd dotarła realizacja metody, i wznowia jej działanie od tego miejsca.

Nie pokażę tu faktycznego kodu generowanego przez kompilator. Jest on skrajnie nieczytelny, gdyż zawiera mnóstwo całkowicie *niewymawialnych* identyfikatorów. (Zapewne pamiętasz z rozdziału 3., że kiedy kompilator C# musi wygenerować identyfikatory, które nie mogą kolidować z naszym kodem ani być w nim widoczne, to tworzy nazwy, które są prawidłowe, lecz które język C# uznaje za niedozwolone; to właśnie takie nazwy są określane jako *niewymawialne*, ang. *unspeakable*). Co więcej, kod generowany przez kompilator używa różnych klas pomocniczych należących do przestrzeni nazw `System.Runtime.CompilerServices`, które są przeznaczone wyłącznie do użycia w metodach asynchronicznych i służą do zarządzania takimi aspektami ich działania jak określanie, które interfejsy są dostępne w danym obiekcie, oraz obsługa przekazywania kontekstu wykonania. Co więcej, jeśli metoda zwraca zadanie, to używane są dodatkowe klasy pomocnicze, które to zadanie tworzą i aktualizują. Niemniej jednak jeśli chodzi o możliwość zrozumienia natury związku pomiędzy typem współpracującym ze słowem kluczowym `await` oraz kodem generowanym przez kompilator w celu obsługi tego słowa kluczowego, to kod z listingu 18.15 jest stosunkowo dobrym przybliżeniem.

Obsługa błędów

Słowo kluczowe `await` obsługuje wyjątki tak, jak byśmy sobie tego życzyli: jeśli wykonanie operacji asynchronicznej zakończy się niepowodzeniem, to wyjątek zostanie zgłoszony przez wyrażenie `await` realizujące tę operację. Ogólna zasada, zgodnie z którą kod asynchroniczny może mieć taką samą strukturę co zwyczajny kod synchroniczny, obowiązuje także w obliczu zgłaszanych wyjątków, kompilator robi wszystko co niezbędne, by zapewnić taką możliwość.

Listing 18.16 przedstawia dwie asynchroniczne operacje, z których jedna jest wykonywana wewnątrz pętli. Przypomina to nieco przykład z listingu 18.6. Poniższy przykład wykonuje jednak nieco inne operacje na pobieranych danych, jednak co jest najważniejsze — zwraca zadanie. Dzięki temu istnieje miejsce, do którego mogą trafić informacje o błędzie, w przypadku gdyby wykonanie operacji się nie udało.

Listing 18.16. Kilka potencjalnych źródeł niepowodzenia

```
private static async Task<string> FindLongestLineAsync(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            string longestLine = string.Empty;
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                if (longestLine.Length < line.Length)
                {
                    longestLine = line;
                }
            }
            return longestLine;
        }
    }
}
```

Obsługa wyjątków jest potencjalnie sporym wyzwaniem dla operacji asynchronicznych, gdyż w momencie wystąpienia problemu metoda, która zapoczątkowała wykonywanie operacji, zazwyczaj będzie już zakończona. Przedstawiona w powyższym przykładzie metoda `FindLongestLineAsync` zazwyczaj kończy działanie w momencie wykonania pierwszego wyrażenia `await`. (Może się zdarzyć, że będzie inaczej — jeśli analizowany zasób będzie dostępny w lokalnej pamięci podręcznej HTTP, to operacja asynchroniczna może się natychmiast zakończyć sukcesem. Jednak zazwyczaj jej wykonanie zajmie nieco czasu, a to oznacza, że metoda zostanie zakończona). Załóżmy, że wykonanie operacji zakończy się pomyślnie i zacznie być wykonywana dalsza część metody, jednak gdzieś wewnątrz pętli pobierającej zawartość odpowiedzi zostanie przerwane połączenie sieciowe. W efekcie jedna z operacji rozpoczętych przez wywołanie metody `ReadLineAsync` zakończy się niepowodzeniem.

Wyjątek dla tej operacji zostanie zgłoszony przez wyrażenie `await`. Wewnątrz tej metody nie ma żadnego kodu obsługującego wyjątki, co zatem powinno stać się potem? Zazwyczaj oczekiwaliśmy, że wyjątek zacznie być przekazywany w górę stosu wywołań, powstaje jednak pytanie, co znajduje się na stosie powyżej tej metody? Niemal na pewno nie będzie to ten sam kod, który ją wywołał — pamiętamy zapewne, że metoda zazwyczaj jest kończona w chwili, gdy dotrze do pierwszego wyrażenia `await`, a zatem na tym etapie nasz kod zazwyczaj będzie działał jako efekt wywołania zwróconego przez obiekt zwrócony przez metodę `GetAwaiter` na potrzeby zadania zwróconego przez metodę `ReadLineAsync`. Istnieje pewne prawdopodobieństwo, że nasz kod będzie jeszcze wykonywany w tym samym wątku z puli, a kod znajdujący się na stosie bezpośrednio nad nim będzie elementem obiektu zwróconego przez metodę `GetAwaiter`. Jednak ten kod nie będzie wiedział, co należy zrobić z wyjątkiem.

Jednak wyjątek nie jest przekazywany w górę stosu. Kiedy wyjątek zgłoszony w metodzie asynchronicznej zwracającej zadanie nie zostanie obsłużony, jest on przechwytywany przez kod wygenerowany przez kompilator, który następnie zmienia stan zwracanego zadania, informując, że jego wykonanie zakończyło się niepowodzeniem. Jeśli kod wywołujący metodę `FindLongestLineAsync` korzysta bezpośrednio z możliwości TPL, to będzie on w stanie wykryć fakt zgłoszenia wyjątku, sprawdzając stan zadania, oraz pobrać obiekt wyjątku, korzystając z właściwości `Exception` zadania. Ewentualnie można także wywołać metodę `Wait` lub odczytać wartość właściwości `Result` zadania, a każda z tych operacji spowoduje zgłoszenie wyjątku `AggregatedException` zawierającego obiekt oryginalnego wyjątku. Jeśli jednak kod wywołujący metodę `FindLongestLineAsync` użyje zwróconego obiektu zadania wraz ze słowem kluczowym `await`, to wyjątek zostanie ponownie zgłoszony w tym wyrażeniu. Z punktu widzenia kodu wywołującego wygląda to tak, jak gdyby wyjątek został zgłoszony w normalny sposób, co też pokazuje przykład z listingu 18.17.

Listing 18.17. Obsługa wyjątków zgłaszanych w wyrażeniu `await`

```
try
{
    string longest = await FindLongestLineAsync("http://192.168.22.1/");
    Console.WriteLine("Najdłuższy wiersz: " + longest);
}
catch (HttpRequestException x)
{
    Console.WriteLine("Błąd podczas pobierania strony: " + x.Message);
}
```

Powyższy kod jest niemal zwodniczo prosty. Pamiętajmy, że kompilator przeprowadza bardzo głęboką restrukturyzację naszego kodu wokół każdego wystąpienia słowa kluczowego `await` oraz że wykonanie kodu, który z pozoru może się wydawać jedną metodą, w rzeczywistości może wymagać wykonania kilku wywołań. Dlatego też zachowanie semantyki nawet tak prostego kodu obsługi błędów (lub podobnych konstrukcji, takich jak instrukcja `using`) jak ten z powyższego przykładu nie jest zadaniem trywialnym. Jeśli kiedykolwiek próbowałeś napisać analogiczny kod obsługi błędów w operacjach asynchronicznych bez korzystania z pomocy kompilatora, to zapewne docenisz, jak bardzo pomaga w tym C#.



Słowo kluczowe `await` pobiera oryginalny wyjątek z obiektu `AggregatedException` i ponownie go zgłasza. To dzięki temu metody asynchroniczne mogą obsługiwać błędy w taki sam sposób jak zwyczajny kod synchroniczny.

Weryfikacja poprawności argumentów

Sposób, w jaki C# automatycznie zgłasza błędy za pośrednictwem obiektu zadania zwracanego przez naszą asynchroniczną metodę (i to bez względu na ilość używanych wywołań zwrotnych), ma jedną wadę. Powoduje on bowiem, że kod taki jak ten z listingu 18.18 nie robi tego, na czym mogłoby nam zależeć.

Listing 18.18. W jaki sposób nie należy sprawdzać poprawności argumentów

```
public async Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    ...
}
```

Wewnątrz metody asynchronicznej kompilator traktuje wszystkie wyjątki w taki sam sposób: żaden z nich nie może zostać przekazany w górę stosu, jakby się to stało w normalnej metodzie, i każdy zostanie zasygnalizowany poprzez odpowiednią zmianę stanu zwracanego zadania — zostanie ono oznaczone jako zakończone niepowodzeniem. Dotyczy to nawet tych wyjątków, które są zgłaszane przed wykonaniem słowa kluczowego `await`. W naszym przykładzie weryfikacja argumentów jest wykonywana, zanim metoda wykona jakiegokolwiek inne operacje, zatem w tym przypadku nasz kod będzie wykonywany w tym samym wątku, w którym metoda została wywołana. Można by pomyśleć, że wyjątek zgłoszony w tym miejscu kodu zostanie przekazany bezpośrednio do kodu wywołującego. W rzeczywistości jednak zauważysz on jedynie zwyczajne zakończenie metody zwracającej obiekt zadania, przy czym stan tego zadania będzie informował, że zakończyło się ono niepowodzeniem.

Jeśli metoda wywołująca od razu wykona zwrócone zadanie, używając do tego celu słowa kluczowego `await`, to nie będzie to miało większego znaczenia — wyjątek i tak zostanie przez nią zauważony. Jednak może się zdarzyć, że kod nie będzie chciał od razu wykonać zadania, a w takim przypadku wyjątek nie zostanie zauważony tak szybko. Ogólnie przyjęta konwencja związana ze stosowaniem i obsługą prostych wyjątków związanych z weryfikacją argumentów zaleca, by w przypadkach, gdy nie ma wątpliwości, że problem został spowodowany kodem przez kod wywołujący, wyjątek należy zgłosić natychmiast. A zatem w powyższym przykładzie naprawdę powinniśmy wymyślić coś innego.



Jeśli nie ma możliwości sprawdzenia poprawności argumentów bez wykonywania jakichś długotrwałych operacji, to chcąc napisać naprawdę asynchroniczną metodę, nie będziemy w stanie zastosować się do powyższej konwencji. W takim przypadku trzeba będzie podjąć decyzję, czy wolimy, by metoda została zablokowana do momentu, gdy będzie w stanie sprawdzić poprawność argumentów, czy też by wyjątki dotyczące argumentów metody były zgłaszane za pośrednictwem zadania, a nie bezpośrednio.

Standardowym rozwiązaniem jest napisanie normalnej metody, która sprawdzi poprawność argumentów przed wywołaniem metody asynchronicznej, wykonującej zamierzone operacje. (Okazuje się, że w podobny sposób należałoby postępować w przypadku przeprowadzania natychmiastowej wersyfikacji argumentów iteratora. Iteratory zostały opisane w rozdziale 5.). Listing 18.19 przedstawia właśnie taką metodę publiczną oraz początek samej metody asynchronicznej.

Listing 18.19. Weryfikacja argumentów metody asynchronicznej

```
public Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    return FindLongestLineCore(url);
}

private async Task<string> FindLongestLineCore(string url)
{
    ...
}
```

Ponieważ metoda publiczna nie została oznaczona jako asynchroniczna (nie dodano do niej modyfikatora `async`), zatem wszelkie zgłaszane przez nią wyjątki będą przekazywane bezpośrednio do kodu wywołującego. Natomiast wszelkie problemy, które nastąpią po rozpoczęciu prywatnej metody asynchronicznej, będą zgłaszane za pośrednictwem obiektu zadania.

Wyjątki pojedyncze oraz grupy wyjątków

Z rozdziału 17. można się było dowiedzieć, że TPL definiuje model pozwalający na raportowanie wielu błędów — właściwość `Exception` zadania zwraca obiekt `AggregatedException`. Nawet jeśli pojawi się tylko jeden problem, to i tak informacje o nim trzeba będzie pobierać z obiektu `AggregateException`. Niemniej jednak w razie stosowania słowa kluczowego `await` to pobranie wyjątku jest wykonywane automatycznie za nas — jak mieliśmy się okazję przekonać w przykładzie z listingu 18.17, pobiera on pierwszy wyjątek zapisany w tablicy `InnerExceptions`, a następnie ponownie go zgłasza.

Takie rozwiązanie jest wygodne, jeśli w operacji może wystąpić tylko jeden problem — nie musimy bowiem pisać żadnego dodatkowego kodu, który by obsługiwał ten grupowy wyjątek i pobierał jego zawartość. (Jeśli korzystamy z zadania zwróconego przez metodę asynchroniczną, to nigdy nie będzie ono zawierać więcej niż jednego wyjątku). Jednak takie rozwiązanie przysparza problemów, kiedy posługujemy się złożonymi zadaniami, w których jednocześnie może się pojawić kilka wyjątków. Na przykład do metody `Task.WhenAll` przekazywana jest kolekcja zadań, a metoda ta zwraca jedno zadanie, które zostanie zakończone

wyłącznie w przypadku, gdy wszystkie zadania podrzędne zostaną prawidłowo zakończone. Jeśli któryś z nich zakończy się niepowodzeniem, to uzyskamy obiekt `AggregateException` zawierający wiele błędów. W razie użycia słowa kluczowego `await` do obsługi takiego zadania zgłosi ono wyłącznie pierwszy z wyjątków.

Standardowe mechanizmy TPL — metoda `Wait` oraz właściwość `Result` — udostępniają pełen zbiór błędów, jednak blokują wykonywanie wątku, jeśli zadanie jeszcze nie zostało zakończone. A co moglibyśmy zrobić, gdybyśmy chcieli skorzystać z wydajnego, asynchronicznego działania słowa kluczowego `await`, które wykonuje coś w wątkach, wyłącznie jeśli znajdzie się dla nich coś do zrobienia, a jednocześnie chcielibyśmy zauważać wszystkie wyjątki? Jedno z potencjalnych rozwiązań zostało przedstawione na listingu 18.20.

Listing 18.20. Zastosowanie słowa kluczowego `await` i metody `Wait`

```
static async Task CatchAll(Task[] ts)
{
    try
    {
        var t = Task.WhenAll(ts);
        await t.ContinueWith(
            x => {},
            TaskContinuationOptions.ExecuteSynchronously);
        t.Wait();
    }
    catch (AggregateException all)
    {
        Console.WriteLine(all);
    }
}
```

Powyższa metoda używa słowa kluczowego `await`, by skorzystać z wydajności asynchronicznych metod C#, jednak zamiast stosować je wraz z samym zadaniem złożonym, używa go wraz z zadaniem, do którego została dodana kontynuacja. Kontynuacja może zostać pomyślnie zakończona, jeśli zostanie zakończona poprzedzająca ją operacja, niezależnie od tego, czy zakończy się ona pomyślnie, czy też nie. Zastosowana kontynuacja jest pusta, więc wewnątrz niej nie mogą wystąpić żadne problemy, a to oznacza, że w tym miejscu nie zostaną zgłoszone żadne wyjątki. Natomiast jeśli wykonanie którejkolwiek z operacji zakończyło się niepowodzeniem, to wywołanie metody `Wait` spowoduje zgłoszenie wyjątku `AggregatedException` — dzięki temu blok `catch` będzie w stanie zauważyć wszystkie wyjątki. Co więcej, ponieważ metoda `Wait` jest wykonywana dopiero po zakończeniu realizacji wyrażenia `await`, zatem wiemy, że zadanie zostało zakończone, a zatem wywołanie to nie spowoduje zablokowania metody.

Jedną z wad takiego rozwiązania jest to, że tworzy ono dodatkowe zadanie tylko po to, byśmy mogli poczekać bez narażania się na napotkanie wyjątku. W powyższym przykładzie kontynuacja została skonfigurowana w taki sposób, że jest wykonywana synchronicznie, dzięki czemu unikamy realizacji drugiego fragmentu kodu przy użyciu puli wątków; niemniej jednak i tak stanowi to marnowanie zasobów. Nieco bardziej zagmatwane, lecz jednocześnie bardziej wydajne rozwiązanie mogłoby polegać na użyciu słowa kluczowego `await` w standardowy sposób i napisaniu kodu obsługi wyjątków w taki sposób, by sprawdzał on, czy nie zostało zgłoszonych więcej wyjątków. Takie rozwiązanie przedstawia listing 18.21.

Listing 18.21. Poszukiwanie dodatkowych wyjątków

```
static async Task CatchAll(Task[] ts)
{
    Task t = null;
    try
    {
        t = Task.WhenAll(ts);
        await t;
    }
    catch (Exception first)
    {
        Console.WriteLine(first);
        if (t != null && t.Exception.InnerExceptions.Count > 1)
        {
            Console.WriteLine("Znaleziono więcej wyjątków:");
            Console.WriteLine(t.Exception);
        }
    }
}
```

To rozwiązanie pozwala uniknąć tworzenia drugiego zadania, jednak jego wadą jest to, że wygląda nieco dziwnie.

Operacje równoległe i nieobsłużone wyjątki

Najprostszym sposobem używania słowa kluczowego `await` jest wykonywanie kolejnych operacji jedna po drugiej dokładnie w taki sam sposób, jaki robimy to w kodzie synchronicznym. Choć wydaje się, że działanie całkowicie sekwencyjne nie pozwala wykorzystywać całego potencjału kodu asynchronicznego, to jednak zapewnia możliwość znacznie wydajniejszego wykorzystania dostępnych wątków niż użycie analogicznego kodu synchronicznego, a dodatkowo w aplikacjach klienckich doskonale współpracuje z kodem obsługi interfejsu użytkownika. Jednak można pójść jeszcze dalej.

Istnieje możliwość jednoczesnego uruchomienia kilku różnych operacji. Można wywołać metodę asynchroniczną, a następnie zamiast od razu skorzystać ze słowa kluczowego `await`, można zapisać wynik w zmiennej i w podobny sposób uruchomić drugą operację asynchroniczną, po czym zaczekać na zakończenie obu. Choć takie rozwiązanie jest możliwe do wykonania, to jednak kryje ono w sobie pewną pułapkę na nieostrożnych programistów; przedstawia ją przykład z listingu 18.22.

Listing 18.22. W jaki sposób nie należy wykonywać wielu współbieżnych operacji

```
static async Task GetSeveral()
{
    using (var w = new HttpClient())
    {
        w.MaxResponseContentBufferSize = 2000000;

        Task<string> g1 = w.GetStringAsync("http://helion.pl/");
        Task<string> g2 =
            w.GetStringAsync("http://helion.pl/kategorie/programowanie/c-sharp");

        // BŁĄD!

        Console.WriteLine((await g1).Length);
        Console.WriteLine((await g2).Length);
    }
}
```

Powyższa metoda pobiera równocześnie zawartość dwóch stron WWW. Po uruchomieniu obu operacji metoda używa słowa kluczowego `await`, by pobrać ich wyniki i wyświetlić długości zwróconych łańcuchów znaków. Jeśli operacje zakończą się pomyślnie, to powyższy kod zadziała prawidłowo, jednak nie zapewnia on prawidłowej obsługi błędów. Jeśli wykonanie pierwszej operacji zakończy się niepowodzeniem, to powyższy kod nigdy nie wykona drugiego wyrażenia `await`. To oznacza, że jeśli także druga operacja zakończy się niepowodzeniem, to nie będzie kodu, który mógłby sprawdzić zgłoszone wyjątki. W końcu TPL wykryje, że wyjątki nie zostały zauważone, co spowoduje zgłoszenie wyjątku `UnobservedTaskException`, a on najprawdopodobniej doprowadzi do awarii programu. (Zagadnienia związane z obsługą niezauważonych wyjątków zostały opisane w rozdziale 17.). Problem polega na tym, że takie sytuacje zdarzają się bardzo rzadko — konieczne jest bowiem, by obie operacje zakończyły się niepowodzeniem i to w bardzo krótkim odstępie czasu — a zatem bardzo łatwo będzie je przegapić podczas testowania aplikacji.

Takich problemów można uniknąć dzięki uważnej obsłudze błędów — na przykład można przechwytywać wszystkie wyjątki zgłaszane przez pierwsze wyrażenie `await` przed wykonaniem drugiego z nich. Ewentualnie można także skorzystać z metody `Task.WhenAll`, by poczekać na wyniki obu operacji wykonywanych w formie jednego zadania — w takim przypadku, gdyby nie udało się wykonać którejkolwiek z operacji, uzyskalibyśmy zadanie zakończone niepowodzeniem, z informacjami o błędach zapisanymi w obiekcie `AggregatedException`, dzięki czemu moglibyśmy sprawdzić wszystkie zgłoszone wyjątki. Oczywiście jak mogliśmy się przekonać, obsługa wielu błędów w przypadku korzystania ze słowa kluczowego `await` może być dosyć kłopotliwa. Jeśli jednak chcemy uruchamiać wiele asynchronicznych operacji i pozwolić, by wszystkie były wykonywane jednocześnie, to kod niezbędny do koordynacji uzyskiwanych wyników będzie bardziej złożony niż w przypadku wykonywania tych samych operacji sekwencyjnie. Niemniej jednak słowa kluczowe `await` oraz `async` i tak znacznie ułatwiają nam życie.

Podsumowanie

Operacje asynchroniczne nie blokują wątku, w którym zostały rozpoczęte; dzięki temu są bardziej wydajne od zwyczajnych metod synchronicznych, co ma szczególnie duże znaczenie na bardzo obciążonych komputerach. Ta cecha sprawia również, że z powodzeniem można z nich korzystać w aplikacjach klienckich, gdyż pozwalają na wykonywanie długotrwałych operacji bez obniżania szybkości reakcji interfejsu aplikacji na działania użytkownika. Jednak wadą operacji asynchronicznych zawsze była ich wysoka złożoność, dotyczy to w szczególności obsługi błędów w przypadku stosowania wielu powiązanych ze sobą operacji. W języku C# 5.0 wprowadzono słowo kluczowe `await`, które pozwala na pisanie kodu asynchronicznego w sposób bardzo zbliżony do zwyczajnego kodu synchronicznego. Sprawy się nieco komplikują, jeśli chcemy, by jedna metoda zarządzała kilkoma operacjami wykonywanymi równoległe, jednak nawet jeśli napiszemy tę metodę w taki sposób, że poszczególne operacje asynchroniczne będą wykonywane w ściśle określonej kolejności, to i tak uzyskamy korzyści. W przypadku aplikacji serwerowej tą korzyścią będzie znacznie bardziej wydajne wykorzystanie wątków, dzięki czemu taka aplikacja będzie w stanie obsłużyć większą liczbę jednocześnie działających użytkowników, gdyż każda z operacji będzie zużywać mniej zasobów. Natomiast w przypadku aplikacji klienckich tą korzyścią będzie działający sprawniej interfejs użytkownika.

Metody korzystające ze słowa kluczowego `await` muszą być oznaczone przy użyciu modyfikatora `async` i powinny zwracać wynik typu `Task` lub `Task<T>`. (C# pozwala także na zwracanie wyniku typu `void`, jednak zazwyczaj jest on stosowany wyłącznie w ostateczności, gdy nie ma innego wyboru). Kompilator zadba o to, by zadanie zostało zakończone pomyślnie, jeśli nasza metoda zostanie prawidłowo wykonana, oraz by zakończyło się niepowodzeniem, jeśli w trakcie wykonywania metody pojawią się jakiegokolwiek problemy. Ponieważ słowo kluczowe `await` może operować na każdym obiekcie `Task` lub `Task<T>`, zatem ułatwia ono rozdzielenie logiki asynchronicznej na wiele metod, gdyż metoda nadrzędna może używać go do wykonywania metod podrzędnych. Zazwyczaj faktyczne operacje są wykonywane przez jakieś metody wykorzystujące zadania, jednak nie jest to regułą, gdyż słowo kluczowe `await` wymaga jedynie użycia określonego wzorca — można w nim podać dowolne wyrażenie pozwalające na wywołanie metody `GetAwaiter` w celu uzyskania obiektu odpowiedniego typu.

.NET Core Profile, 490, 497, 607
.NET Framework, 22

A

- abstrakcja wątków, 600
- abstrakcyjna implementacja interfejsu, 223
- abstrakcyjny typ bazowy, 234
- adnotacje do danych, 772
- adres URL, 766
- agregacja, 379
- akcesor
 - get, 130
 - set, 130
- akumulator, accumulator, 380
- anatomia podzespołu, 464
- animacje zmiany stanu, 724
- ANSI, 576
- anulowanie długotrwałych operacji, 652
- apartament wielowątkowy, MTA, 550
- API, 22
 - asynchroniczne, 454, 492
 - odzwierciedlania, 495, 502
- aplikacje
 - GUI, 467
 - internetowe, 741
 - konsolowe, 467, 557
 - OOB, 686
 - Windows 8, 490
 - Windows Phone, 492
 - Windows Runtime, 713
 - XBAP, 492
- APM, Asynchronous Programming Model, 454, 565, 651
- architektura
 - procesora, 487
 - wspólnego języka, 26, 819
- argument typu, 149, 179
- argumenty opcjonalne, 127
- ASCII, 51, 574, 787
- ASP.NET, 741
 - MVC, 762
 - Razor, 742
 - trasowanie, 777
 - Web Forms, 752
- asynchroniczna metoda
 - anonimowa, 668
- asynchroniczne wyrażenie
 - lambda, 667
- atak XSS, 743
- atrybut
 - [CallerMemberName], 543
 - [ComImport], 806
 - [DataContract], 594
 - [DataMember], 594
 - [MTAThread], 550
 - [NonSerialized], 594
 - [Obsolete], 578
 - [Serializable], 591
 - [STAThread], 550
 - [TestClass], 535, 536
 - [ThreadStatic], 604, 606
 - AggressiveInlining, 549
 - AssemblyFileVersion, 482
 - AssemblyKeyFileAttribute, 540
 - DebuggableAttribute, 549
 - DllExport, 785
 - DllImport, 787, 807, 819
 - ExpectedExceptionAttribute, 537
 - Flags, 143
 - InternalsVisibleToAttribute, 543, 544
 - LoaderOptimizationAttribute, 549
 - MarshalAs, 784
 - NoOptimization, 549
 - OnClick, 753
 - SecurityCriticalAttribute, 669
 - SecuritySafeCriticalAttribute, 548
 - SerializableAttribute, 546
 - StructLayout, 791
 - TestCategoryAttribute, 536
 - ThreadStaticAttribute, 605
 - TypeIdentifier, 810
 - x:Class, 690
 - x:Name, 691
 - xmlns:x, 690
- atrybuty, 46, 535–556
 - określanie celu, 552
 - stosowanie, 535
 - ustawienia opcjonalne, 537
 - wartości opcjonalne, 553
- atrybuty
 - metody, 538
 - modułu, 538
 - niestandardowe, 547, 551
 - obsługiwane przez CLR, 543
 - kompilator, 539
 - określające numer wersji, 539
 - podzespołu, 538
 - pola zdarzenia, 539
 - własne, custom attributes, 167
 - z informacjami o kodzie wywołującym, 541
 - podzespole, 540
- automatyzacja COM, 522, 524

B

bezpieczeństwo, 546, 794
 pod względem
 wielowątkowym, 603
 typów, 28, 162, 216, 818
bezpieczne uchwyty, 793
białe znaki, whitespace, 66
biblioteka
 advapi32.dll, 787
 jQuery, 764
 Knockout, 764
 Modernizr, 764
 Moq, 157
 mscorlib.dll, 464
 mscorlib.dll, 403
 ole32.dll, 798, 800
 Reactive Extensions, 401
 System.Observable.dll, 404
 System.Web.dll, 42
 System.Windows.
 Controls.dll, 492
 TPL, 263, 325, 446, 610, 640
biblioteki
 DLL, 464
 klas, 22, 40, 488
 typów, 809
blok, 55
 catch, 286, 287
 CER, 307
 finally, 290, 623
 lock, 620
 try, 289
blokady odczytu i zapisu, 627
blokowanie, 621
błąd, error, 225
 kompilacji, 563
 obserwatora, 409
BOM, byte order mark, 573

C

C++/CLI, 819
CCW, COM-callable wrapper,
 788
cechy C#, 25
cel, target, 31, 463
cel atrybutu, 537
CER, constrained execution
 region, 306
ciasteczka, cookies, 743
ciąg Fibonacciego, 358

CLI, Common Language
 Infrastructure, 26, 819
ClickOnce, 491
CLR, Common Language
 Runtime, 22, 238, 465, 495, 803
CLS, Common Language
 Specification, 26, 72
COM, Component Object
 Model, 521, 798
COM Automation, 522
Component Extensions, 820
CTS, Common Type System, 22
czas
 trwania zdarzenia, 424
 życia obiektów, 237, 252
 COM, 803
 RCW, 803
częściowe deklaracje klas, 146

D

debuger Visual Studio, 45, 660
debugowanie, 281, 303
 aplikacji, 293
 wyjątków, 304
definiowanie
 klasy, 44, 150
 konstruktor, 117
deklaracja
 indeksatora, 134
 przestrzeni nazw, 42
 typu ogólnego, 150
 zmiennej, 50
delegaty, delegate, 144, 310, 416
 Action, 318
 Func, 318
 typu Predicate<int>, 312
 typu Predicate<T>, 312
 zbiorowe, 314, 316
deserializacja, 592
deserializacja wyjątku, 300
destruktor, destructors, 237,
 261
diagram aktywności Rx, 406
DirectX, 24
disassembler kodu .NET, 191
DLL, Dynamic Link Library, 33
DLR, Dynamic Language
 Runtime, 521
długość strumienia, 562
dodatek Service Pack, 295
dodawanie projektów do solucji,
 35

dokumentacja MSDN, 547
DOM, Document Object Model,
 399
domena aplikacji, appdomain,
 300, 549
domyślna implementacja
 zdarzeń, 340
domyślne wartości
 argumentów, 128
domyślny konstruktor
 bezargumentowy, 230
dopisywanie łańcucha do pliku,
 584
dostawca
 CustomLinqProvider, 356
 LINQ to Entities, 364
 LINQ to Objects, 364
 LINQ to SQL, 367
 SillyLinqProvider, 356
dostawcy LINQ, 347
dostęp do
 bufora, 818
 elementu tablicy, 167
 obiektu, 240, 602
 obiektu Request, 748
 pola, 130
 prywatnego stanu, 621
 składowych, 115
 składowych klas bazowych,
 229
 systemu plików, 529
 węzła listy, 202
dostępność, 218
 metod, 130
 obiektu, 245
drzewo
 elementów interfejsu
 użytkownika, 688
 wyrażenia, 336
dynamiczne
 języki .NET, 525
 określanie typów, 517
 tworzenie delegatów, 314
dyrektywa
 #define, 67
 #endregion, 70
 #error, 68
 #line, 69
 #pragma, 69
 #region, 70
 #warning, 68
 using, 40, 129, 354
dyrektywy preprocesora, 67

działanie
konwersji, 529
operatorów, 362
słowa kluczowego await,
670
dziedziczenie, inheritance, 207,
230
interfejsów, 210
typów odzwierciedlania, 496

E

EAP, Event-based
Asynchronous Pattern, 651
edycja tekstu, 727
edytor plików zasobów, 485
efektywne wykorzystanie
pamięci, 339
element
@RenderBody, 750
@RenderSection, 750
CheckBox, 714
ContentPresenter, 722
Ellipse, 702
Grid, 692
ItemsPresenter, 722
MediaElement, 737
ResourceDictionary, 739
RowDefinition, 709
StackPanel, 692
TextBlock, 725, 738
TextBox, 694, 725
elementy
listy, 734
podrzędne, 692
właściwości, property
elements, 693
Entity Framework, 367, 397
etykiety ekranowe, ToolTip, 716

F

FIFO, first-in, first-out, 201
filtrowanie
danych, 364
elementów, 421
finalizacja, finalization, 237, 261
finalizator, 220, 263, 270
finalizatory krytyczne, 264
flaga
AttachedToParent, 650
ExecuteSynchronously, 645
LongRunning, 641

newslet, 227
OnlyOnRanToCompletion,
644
PreferFairness, 641
useAsync, 581
format
JSON, 30
PE, 464, 466
resx, 484
XML, 31
formatowanie tekstu, 728
funkcja
BackEventLogA, 787
BackupEventLog, 793
EnumWindows, 789
GetVersionEx, 791
OpenEventLog, 794
funkcje anonimowe, 326

G

GAC, Global Assembly Cache,
474, 482, 494
GC, garbage collector, 237
generowanie
elementów, 420
kodu, 329
nazwy klasy, 330
sekwencji, 396
widoków, 741
główny podzespół
współdziałania, 809
gniazda układu, layout slot, 698
grafika
bitmapy, 736
kształty, 735
media, 737
grupowanie, 386
grupowanie zdarzeń, 423
grupujące wyrażenie zapytania,
386
grupy wyjątków, 675

H

hermetyzacja, encapsulation, 95
heurystyki tworzenia wątków,
612
hiperwątkowość,
hyperthreading, 600
HPC, High-Performance
Computing, 369

I

IANA, 577
identyfikator
assemblyName, 779
CLSID, 809
GUID, 113, 799, 806
IID, 805
kulturowy, 484
ProdID, 809
IIS, Internet Information Server,
741
IL, intermediate language, 27
implementacja
interfejsu, 141
IEnumerable<T>, 450
INotifyPropertyChanged,
542
IObservable<T>, 407, 414
list, 189
operatora +, 136
pętli asynchronicznej, 665
wzorca await, 670
źródła ciepłych, 410
źródła zimnych, 407
importowanie
funkcji, 789
przestrzeni nazw, 760
wartości wynikowej, 798
indeksatory, 134
informacje
o atrybutach, 554
o klasach, 809
o kodzie wywołującym, 541
o metodzie, 541
o pliku, 588
o podzespole, 540
o typach, 776, 779, 809
o wtyczkach, 554
o wyjątku, 283
o zdarzeniu, 339
inicjalizacja statyczna, 124
inicjalizatory, 51, 92
list, 184
pól, 120, 232
pól niestatycznych, 117
pól statycznych, 123
słownika, 198
tablic, 168, 179
instrukcja
break, 90
continue, 91
fixed, 819

- instrukcja
 - goto, 90
 - if, 87
 - lock, 622
 - switch, 89
 - unchecked, 563
 - using, 331
 - yield return, 358
 - instrukcje, statements, 59
 - blokowe, 88
 - deklaracji, 59
 - iteracyjne, 59
 - sprawdzone, 78
 - wyboru, 59
 - wyrażeń, 59, 62
 - interfejs
 - COM, 805
 - IBuffer, 816
 - IClickHandler, 310
 - ICloneable, 102
 - ICollection, 183
 - ICollection<T>, 185, 188, 214
 - IComparer<T>, 153, 214, 345
 - ICustomAttributeProvider, 553
 - IDictionary<TKey, TValue>, 197
 - IDisposable, 237, 265–274, 406, 411, 564, 571, 794
 - IDynamicMetaObjectProvider, 528
 - IEnumerable<T>, 93, 140, 185, 212, 402, 450
 - IHomeGroup, 806
 - IList<T>, 185, 187
 - INotifyPropertyChanged, 542, 731
 - IObservable<T>, 344, 401–406, 450
 - IObserver<T>, 402, 405, 450
 - IQueryable, 360
 - IQueryable<T>, 359
 - IQueryProvider, 360
 - IRandomAccessStream, 566
 - IReadOnlyList<T>, 189, 217
 - ISerializable, 300
 - ISet<T>, 200, 226
 - interfejsy, 131, 140, 345
 - interfejsy Rx, 403
 - IPC, interprocess
 - communication, 566
 - iterator, 92, 190
 - iterator nieskończony, 191
- ## J
- jawna implementacja interfejsu, 141
 - jawne
 - delegaty instancji, 313
 - operatory konwersji, 137
 - przekazywanie
 - argumentów, 170
 - mechanizmów szeregujących, 445
 - wartości typu
 - wyliczeniowego, 142
 - wczytywanie podzespołów, 473
 - wywoływanie konstruktora, 231
 - język
 - C#, 21
 - C++, 22
 - F#, 22
 - IronPython, 22, 525
 - IronRuby, 22, 525
 - Java, 23
 - JavaScript, 23
 - Objective-C, 23
 - pośredni, IL, 27
 - VBA, 522
 - Visual Basic, 22, 29
 - XAML, 287, 492, 681
 - języki
 - dynamiczne, 517
 - skryptowe, 811
 - JIT, just in time, 27
 - JSON, 30
- ## K
- katalog
 - App_Code, 749
 - App_Data, 764
 - App_Start, 777
 - AppData, 589
 - ApplicationData, 589
 - Biblioteka, 789
 - Common, 739
 - Content, 764
 - Controllers, 772
 - Models, 769
 - Reflection, 771
 - Scripts, 764
 - Views, 767, 771
 - klasa, 44, 95
 - ActionResult, 766
 - AfterYou, 125
 - AppDomain, 302
 - ApplicationData, 589
 - ArgumentException, 296
 - ArgumentOutOfRangeException
 - ↳Exception, 296
 - Assembly, 465, 473, 496, 769
 - AssemblyModel, 771
 - AsyncSubject<T>, 449
 - Attribute, 552
 - AutoResetEvent, 631
 - Barrier, 631
 - Base, 208
 - BaseWithVirtual, 220
 - bazowa object, 158
 - BehaviorSubject<T>, 448
 - BinaryWriter, 591
 - BinaryReader, 591
 - Block, 727
 - BlockingCollection<T>, 203
 - BlockUIContainer, 727
 - BufferedStream, 566
 - Capture, 41
 - Collection<T>, 194
 - CollectionView, 344
 - COM, 807
 - Complex, 168
 - ConcurrentQueue<T>, 203
 - ConcurrentStack<T>, 203
 - ConstructorInfo, 512
 - ContentControl, 714
 - Control, 713, 714
 - Controller, 767
 - ControlScheduler, 442
 - CoreDispatcherScheduler, 442
 - CountdownEvent, 632
 - Counter, 101
 - CourseChoice, 390
 - CriticalFinalizerObject, 265
 - CryptoStream, 566
 - CultureInfo, 348
 - CustomerDerived, 224, 227
 - CustomLinqProvider, 355
 - DataContractJsonSerialization, 596
 - DataContractJsonSerializer, 597
 - DataContractSerialization, 596
 - DateTimeOffset, 442
 - Debug, 68
 - DeflateStream, 566

Delegate, 317
 Derived, 208
 Dictionary<TKey, TValue>, 196, 618
 Directory, 268, 585
 DirectoryInfo, 588
 Dispatcher, 696
 DispatcherObservable, 444
 DispatcherScheduler, 442
 DynamicFolder, 530
 DynamicObject, 528
 Encoding, 572
 Environment, 296
 EventInfo, 514
 EventLoopScheduler, 445
 EventPattern<T>, 452
 Exception, 286, 298
 ExceptionDispatchInfo, 295
 ExecutionContext, 617
 ExpandoObject, 531
 Expression, 335
 FieldInfo, 513
 File, 581, 583
 FileInfo, 588
 FileNotFoundException, 288
 FileStream, 271, 562–565, 578
 FileSystemInfo, 588
 FileSystemWatcher, 789
 FlowDocument, 726
 FrameworkElement, 690, 698, 725, 738
 GCHandle, 240
 Grid, 707
 GZipStream, 566
 HashSet, 200
 HashSet<T>, 319
 HomeController, 765
 ImageBrush, 737
 Interlocked, 100, 634, 636
 InternalsVisibleToAttribute, 39
 IOException, 288
 ItemsControl, 719, 722
 KeyWatcher, 412
 Lazy<T>, 152, 637
 LazyInitializer, 100, 638
 LibraryBase, 225
 LinkedList<T>, 202
 List<T>, 162, 182, 619
 ManualResetEvent, 628, 632
 ManualResetEventSlim, 631
 ManualResetState, 628
 Marshal, 802
 MemberInfo, 503, 511
 MemoryStream, 564
 MethodBase, 511
 MethodBody, 511
 MethodInfo, 512
 Mock<T>, 157
 ModelSource, 770, 773, 775
 ModeSource, 781
 Module, 502
 Monitor, 619, 622
 MoreDerived, 208
 MulticastDelegate, 314, 319, 323
 Mutex, 633
 NewThreadScheduler, 446
 NoAfterYou, 125
 Observable, 414, 418, 452
 Page, 692, 761
 Parallel, 653
 ParameterInfo, 512
 Path, 42, 578, 586
 PipeStream, 566
 Predicate<T>, 319
 ProgressBar, 718
 PropertyInfo, 513
 publiczna, 97
 Queue<T>, 202
 Random, 174
 ReaderWriterLock, 627
 ReaderWriterLockSlim, 621, 627
 ReadOnlyCollection<T>, 188, 195
 ReflectionController, 776
 ReplySubject<T>, 449
 ResourceManager, 484, 486
 RuntimeHelpers, 307
 SafeHandle, 265, 307, 793
 SaleLog, 620
 ScarceEventSource, 342
 Semaphore, 632
 SemaphoreSlim, 633
 Shape, 215
 SmtClient, 629, 648
 SortedDictionary<TKey, TValue>, 199
 SortedSet, 200
 Source<T>, 359
 SpinLock, 625
 Stack<T>, 202
 StateStore, 567
 Stopwatch, 77
 StorageFile, 568
 Stream, 258, 558–570
 StreamReader, 283, 572
 StreamWriter, 572
 StringBuilder, 81
 StringComparer, 198, 471
 StringReader, 573
 StringWriter, 573
 Subject<T>, 447
 SynchronizationContext, 615
 SynchronizationContext
 ↳ Scheduler, 442
 System.Array, 234
 System.Attribute, 535
 System.Exception, 234
 System.GC, 260
 System.Object, 234
 System.String, 776
 TabControl, 721
 Task, 610, 640, 647, 650
 Task<T>, 640
 TaskCompletionSource<T>, 648
 TaskPoolScheduler, 446
 TaskScheduler, 446, 645
 testu jednostkowego, 37
 TextReader, 570, 571
 TextWriter, 570, 571
 Thread, 606, 607
 ThreadLocal<T>, 605
 ThreadPool, 611, 613, 630
 ThreadPoolScheduler, 446
 ThresholdComparer, 313, 329
 Type, 497, 499, 506
 TypeDescriptor, 514
 TypeInfo, 497, 507, 508
 TypModel, 774
 UnicodeEncoding, 574
 VariableSizedWrapGrid, 709
 ViewResult, 766
 VisualStateManager, 723
 WaitHandle, 630
 WeakReference, 247
 WebClient, 455, 640, 646
 WindowsObservable, 454
 WindowsRuntimeStreamExtensions, 568
 WrapPanel, 711
 XmlReader, 573
 XmlSerializer, 597
 klasy
 kolekcji współbieżnych, 639
 konkretne, 222

- klasy
 - ogólne, 150
 - ostateczne, 228
 - statyczne, 100
 - wewnętrzne, 97
- klauzula
 - else, 87
 - from, 349
 - group, 350, 386
 - join, 390
 - let, 352
 - orderby, 372
 - select, 350, 366, 422
 - where, 349, 354, 421
- klient WCF Data Services, 398
- klip wideo, 737
- klucz
 - grupowania, 389
 - InprocServer32, 809
 - PageHeaderTextStyl, 739
- klucze silnych nazw, 478
- kod
 - formularza, 755
 - IL, 465
 - kontrolera HomeController, 765
 - maszynowy, machin code, 27
 - niezarządzany, 240, 258
 - rodzimy, 783
 - ukryty, codebehind, 690
 - uwierzytelniania
 - komunikatu, MAC, 755
 - widoku głównego, 780
 - widoku Index, 768
 - zarządzany, 27
- kodeki, 738
- kodowanie, 574
 - ASCII, 51, 574, 787
 - ISO/IEC 8859-5, 569
 - UTF-8, 569
 - Windows-1252, 569
- kody mieszające, 198
- kolejka, 201
 - FIFO, 611, 612
 - LIFO, 611
- kolejność
 - inicjalizacji, 122
 - przetwarzania operandów, 64
 - tworzenia obiektów, 232
- kolekcja, 93, 165
 - commaCultures, 359
 - wyjątków, 647
- kolekcje
 - bezpieczne, 603
 - leniwe, lazy collections, 94
 - współbieżne, 203
- kolizja
 - kodów mieszających, 109
 - nazw, 100, 225
- komentarze
 - jednowierszowe, 65
 - oddzielone, delimited
 - comments, 65
 - wielowierszowe, 66
- kompilacja
 - JIT, 548
 - warunkowa, 67
- kompilator XAML, 690
- komponent, 463, 493
- komponent programowy, 463
- kommunikacja pomiędzy
 - procesami, ICP, 566
- kommunikat o błędzie, 55
- konfiguracja budowania
 - Debug, 68
 - Release, 68
- konflikt nazw, 56, 470
- konkatenacja, 54
- konstruktor, 117
 - bezargumentowy, 107, 119, 231
 - domyślny, 118
 - klasy bazowej, 231
 - klasy pochodnej, 231
 - statyczny, 121
- konstruktory
 - klasy FileStream, 580
 - klasy Thread, 608
- kontekst
 - odzwierciedlania, reflection
 - contexts, 514
 - odzwierciedlania
 - niestandardowy, 515
 - synchronizacji, 662
 - wykonywania, execution
 - context, 607, 662
- konteksty sprawdzane, 77
- kontener StackPanel, 699
- kontrawariancja, 151, 212
- kontrawariantny parametr typu, 215
- kontroler
 - HomeController, 765, 778
 - ReflectionController, 773, 781
- kontrolka
 - Button, 714, 717, 722
 - CheckBox, 715
 - ComboBox, 709, 715, 719
 - ListBox, 715, 719, 733
 - ProgressBar, 719
 - RadioButton, 715
 - RichTextBlock, 727
 - RichTextBlockOverflow, 727
 - RichTextBox, 728
 - ScrollBar, 718
 - ScrollView, 712, 716
 - Slider, 717, 719
 - TabControl, 721
 - TextBox, 727, 731
 - ToolTip, 716
 - WebBrowser, 811
- kontrolki
 - list, 720
 - postępów, 718
 - serwerowe, server-side
 - controls, 752
 - serwerowe HTML, 756
 - użytkownika, user controls, 303, 724
 - z zawartością, 714
- kontynuacja, 643
- konwencja
 - cdecl, 796
 - stdcall, 796
- konwencje nazewnicze, 96
- konwersja, 392
 - delegatów, 320
 - delegatów zbiorowych, 322
 - jawna liczb, 75
 - niedozwolona delegatów, 320
 - niejawna liczb, 74
 - niestandardowa, 529
 - wyrażenia zapytania, 351
- kończenie operacji wejścia-wyjścia, 614
- kopiowanie
 - instancji, 102
 - podzespołów, 525
 - referencji, 101
 - strumienia, 562
- koszt
 - alokacji, 249
 - blokowania, 622
 - utworzenia zasobu, 266
- kowariancja, 151, 212
- kowariancja delegatów, 320

kowariantny parametr typu, 213
krotki, tuples, 204
kształtowanie danych, data
 shaping, 368
kwalifikator
 ascending, 372
 descending, 372
 out, 127
 ref, 127
kwantyfikatory
 istnienia, Any, 375, 430
 ogólny, All, 375, 430

L

lambda, 327
leniwa inicjalizacja, lazy
 initialization, 637
liczba
 argumentów, 169
 operacji, 176
 taktów, tick count, 77
 wątków, 613
 wymiarów tablic, 179
liczby losowe, 174
LIFO, last-in, first-out, 202
LINQ, Language Integrated
 Query, 21, 335
 operatory grupowania, 423
 operatory Join, 424
LINQ operators, 347
LINQ provider, 347
LINQ to Entities, 347, 362, 397
LINQ to Events, 401
LINQ to HPC, 369
LINQ to Objects, 177, 347–386,
 393–400
LINQ to SQL, 347, 362, 398
LINQ to XML, 399
lista, 185, 189, 719
 FIFO, 201
 iVector<T>, 566
 List<T>, 183
listy połączone, 202
literały, 60
logiczna wartość wynikowa, 798
logowanie, 811

Ł

łańcuch
 dziedziczenia, 208
 znaków, 786

łącza do widoku typów, 776
łączenie
 delegatów, 315
 list, 385
 obserwowalnych źródeł, 432

M

MAC, message authentication
 code, 755
magazyn podzespołów, 474
manifest
 podzespołu, assembly
 manifest, 466
 wdrożenia, deployment
 manifest, 466
mechanizm
 DLR, 521
 dynamicznych
 pośredników, 157
 generowania widoków,
 view engines, 741
 odzwierciedlania, 495, 520
 odzyskiwania pamięci, 28,
 237, 241–244, 248–255, 342,
 793
Platform Invoke,
 Patrz P/Invoke
Razor, *Patrz* Razor
 serializacji, 558, 590
szeregujący, scheduler, 442,
 645, 661
 ControlScheduler, 442, 445
 CoreDispatcherScheduler,
 442, 445
 CurrentThreadScheduler,
 443
 DispatcherScheduler, 442,
 445
 ImmediateScheduler, 443
 przekazywanie, 445
 sposoby określania, 443
 SynchronizationContextSc
 heduler, 442, 445
trasowania, 777
wczytywania podzespołów,
 463
 współdziałania, 521
metadane, metadata, 465, 805,
 815
metadane .NET, 465
metoda, 125
 _Foo@12, 796

ActionLink, 777
AddRef, 803
Aggregate, 382
AppednAllLines, 584
Application_Start, 777
Array.BinarySearch, 174
Array.Clear, 182
Array.Copy, 181
Array.FindIndex, 310, 315
Array.IndexOf, 171, 174
Array.Sort, 174, 177
Assembly, 773
Assembly.CreateInstance, 509
Assembly.Load, 474
Assert, 68
AsStreamForRead, 569
AsStreamForWrite, 569
BackupEventLog, 787
BackupEventLogW, 787
Base.Foo, 504
BeginInvoke, 324
BinarySearch, 173, 176
BlockRead, 571
CallDispose, 275
Cancel, 652
Cast<string>, 749
Caught, 332
Close, 564
CoCreateInstance, 806, 815
Compare, 154, 215
CompareExchange, 634
ConfigureAwait, 662
Connect, 416
Console.WriteLine, 174
Console.ReadKey, 62
Console.WriteLine, 170
Contact, 766
Contains, 200
ContainsKey, 197
ContinueWith, 643, 650
CopyTo, 181
Create, 414
CreateDelegate, 314, 322
Derived.Foo, 504
Dispose, 186, 193, 265–274,
 564
DownloadStringTaskAsync,
 641, 646
Encrypt, 582
EndInvoke, 324
EnsureInitialized, 638
Enumerable.Repeat<T>, 397
EnumerateFiles, 268

metoda
 EnumWindows, 789
 Equals, 108, 218
 Exist, 582
 FailFast, 296
 File, 766
 Finalize, 218, 261
 FinAll, 364
 FindAll, 173, 330, 356
 FindIndex, 172, 310
 FindLongestLineAsync, 673
 Flush, 561, 571
 FormatDictionary, 602
 Froblicate, 519
 FromCurrentSynchronization
 ↳Context, 646
 FromEventPattern, 452, 454
 GetAccessControl, 578
 GetAwaiter, 670, 673, 679
 GetCallingAssembly, 499
 GetCultures, 356
 GetData, 606
 GetDetails, 620
 GetDirectoryName, 587
 GetDirectoryRoot, 586
 GetEnumerator, 186, 191
 GetExportedTypes, 500
 GetFileName, 587
 GetFolderPath, 589
 GetHashCode, 109, 198, 218
 GetHashCode.Equals, 108
 GetInfo, 197
 GetInvocationList, 323
 GetIsGreaterThanPredicate,
 313
 GetLastWin32Error, 802
 GetLength, 181
 GetManifestResourceStream,
 465
 GetNames, 234
 GetNextValue, 100, 103
 GetObjectData, 300
 GetPosition, 339
 GetResult, 669
 GetType, 218, 238, 273, 500
 GetTypeFromCLSID, 808
 GetValue, 122
 GroupJoin, 427
 IgnoreRoute, 778
 IndexOf, 177
 Initialize, 46
 InitializeComponent, 692
 inline, 330
 int.TryParse, 280
 Interval, 457, 458
 Invoke, 324, 512
 InvokeMember, 509
 IsDefined, 553
 IsGreaterThan, 313, 329
 IsGreaterThanZero, 172, 310
 IsSupersetOf, 200
 Join, 427
 LoadFile, 499
 LoadFrom, 474
 LogPersistently, 629
 Main, 45, 98, 469
 MapRoute, 778
 Marshal.Release, 804
 Math.Sqrt, 62
 MemberwiseClone, 218
 Monitor.Enter, 623
 Monitor.Pulse, 624
 Monitor.Wait, 623
 Monitr.Exit, 622
 MoveNext, 186
 Notify, 606, 811
 object.ReferenceEquals, 105,
 803
 Observable.Create, 416
 Observable.Empty<T>, 418
 Observable.FromAsync, 455
 Observable.FromEvent
 ↳Pattern, 452
 Observable.Generate
 ↳<TState, TResult>, 420
 Observable.Interval, 456
 Observable.Never<T>, 418
 Observable.Range, 419, 445
 Observable.Repeat<T>, 419
 Observable.Return<T>, 419
 Observable.Throw, 419
 Observable.Timer, 457
 ObserveOn, 444
 OnComplete, 345
 OnCompleted, 405, 410
 OnError, 345, 405, 410
 OnNext, 345, 412
 OrderBy, 372
 OrderByDescending, 372
 Overlap, 200
 Parallel.For, 653
 Parse, 234
 Path.Combine, 586
 Pop, 202
 PrepareConstrainedRegions,
 307
 Publish, 416
 Pulse, 624
 PulseAll, 624
 Push, 202
 QueryInterface, 806
 QueueUserWork, 611
 Read, 558, 560
 ReadLineAsync, 664, 673
 Redirect, 766
 ReferenceEquals, 218
 ReflectionOnlyGetType, 507
 RegisterRoutes, 778
 RegisterWaitForSingleObject,
 630
 ReleaseHandle, 794
 ReleaseMutex, 633
 RemoveFirst, 202
 RemoveLast, 202
 Reset, 628
 Resize, 182
 RoCreateInstance, 815
 RoGetActivationFactory, 815
 Run, 413
 Seek, 561
 Select, 354
 SelectMany, 370
 SendAsync, 659
 Serialize, 593
 SetAccessControl, 578
 SetData, 606
 SetLastError, 802
 SetLength, 562
 SetMaxThreads, 613
 ShowMessage, 221
 SignalAndWait, 631
 Subscribe, 405, 412, 416
 SuppressFinalize, 264
 Task.Factory.StartNew, 641
 ThenBy, 373
 this.GetType, 759
 Thread.Sleep, 664
 ToEventPattern, 454
 ToObservable, 450, 455
 ToString, 217
 TrimExcess, 184
 TryEnter, 625
 TryGetValue, 197, 245, 280
 TryParse, 280
 Union, 382
 UseObjects, 519

- View, 766, 773
 - Wait, 624, 676
 - WaitAll, 630
 - WaitOne, 631
 - WhenAll, 650
 - WhenAny, 650
 - Where, 335, 354
 - Write, 558
 - WriteAsync, 651
 - XmlReader.Create, 573
 - metody
 - abstrakcyjne, 222
 - akcji, 766
 - anonimowe, anonymous
 - method, 326, 819
 - asynchroniczne, 657, 660
 - częściowe, 146
 - globalne, 44
 - inline, inline method, 326
 - klasy
 - Directory, 585
 - File, 582
 - List<T>, 184
 - object, 217
 - ogólne, generic methods, 149, 160
 - ostateczne, sealed methods, 228
 - rozszerzeń, extension
 - methods, 129
 - statyczne, 45, 815
 - ukryte, 225
 - warunkowe, conditional
 - methods, 68
 - wirtualne, 220
 - wytwórcze, factory method, 335
 - z modyfikatorem async, 666
 - zagnieżdżone, 667
 - zwrotne, 669
 - mikropomiary,
 - microbenchmarking, 174
 - model
 - AssemblyModel, 774
 - kodu ukrytego, 753
 - najwyższego poziomu, 780
 - programowania
 - asynchronicznego, APM, 565
 - reprezentujący podzespół, 769
 - wiązania danych, 764
 - widoku, viewmodel, 729
 - model-widok-prezenter, 729
 - moduły, modules, 466
 - modyfikacja
 - przechwyconej zmiennej, 330
 - tablic, 168
 - modyfikator internal, 38
 - modyfikatory dostępności, 97
 - modyfikowanie
 - właściwości, 132
 - wartości obiektu, 133
 - monitory, 622
 - MSDN, Microsoft Developer
 - Network Library, 547
 - MTA, multithreaded apartment, 550
 - muteksy, 633
 - MVC, Model View Controller, 762
 - domyślny układ strony, 768
 - generowanie łączy, 776
 - kontrolery, 765
 - modele, 767
 - obsługa danych
 - wejściowych, 774
 - układ projektu, 763, 765
 - widoki, 767
- ## N
- narzędzie
 - Fakes, 157
 - FxCop, 96
 - ILDASM, 191
 - msbuild, 31
 - nawiasy
 - kątowe, 106, 149, 744
 - klamrowe, 42, 55, 747
 - kwadratowe, 167
 - nazwa
 - klasy, 96
 - punktu wejścia, 797
 - zmiennej, 56
 - podzespołu, 476, 487
 - nazwane potoki, named pipes, 566
 - nazwy
 - niewymawialne, 672
 - zastępcze, 42, 470
 - NET Core Profile, 488
 - niebezpieczna sztuczka, 745
 - niebezpieczny kod, 818
 - niejawna
 - delegacja instancji, 313
 - konwersja, 208
 - konwersja referencji, 214, 319
 - niejawne
 - pakowanie, 273
 - tworzenie delegatu, 312
 - numer wersji podzespołu, 480–483
- ## O
- obiekt, 238
 - AggregateException, 676
 - CancellationTokenSource, 652
 - CCW, 788
 - ExpandoObject, 531
 - FileInfo, 588
 - FileStream, 409, 572
 - IBuffer, 817
 - InputStream, 568
 - IRandomAccessStream, 567
 - KeyWatcher, 412
 - Polyline, 428
 - RCW, 788
 - Request, 748
 - SimpleColdSource, 408
 - StorageFile, 568
 - StringBuilder, 602
 - StringReader, 573
 - Task, 633
 - Thread, 600
 - TypeInfo, 499, 510
 - obiekty
 - COM, 814
 - dynamiczne
 - niestandardowe, 528
 - formatujące, 593
 - obserwowalne, 439
 - programu Excel, 524
 - RCW, 803
 - skryptowe, scriptable
 - objects, 524
 - stron, 748, 759
 - wyjątków, 286
 - zdarzeń, 628
 - obsługa
 - anulowania, 653
 - błędów, 409, 415, 647, 672
 - błędów Win32, 802
 - działań asynchronicznych, 599
 - działań współbieżnych, 639

- obsługa
 - łańcuchów znaków, 786, 797
 - modelu TypeModel, 775
 - operacji asynchronicznych, 429
 - powiadomień ze skryptu, 812
 - przepływu danych, 654
 - Unicode, 787
 - wartości HRESULT, 800
 - widoków, 763
 - wielowątkowości, 618, 634, 637
 - wyjątków, 285, 289, 673
 - wyrażeń zapytań, 353
 - zdarzeń, 337, 340, 632, 694
- oddzielona prezentacja,
 - separated presentation, 729
- odwołanie do projektu, 35
- odwzorowanie, map, 368
- odzwierciedlanie, reflection, 495
- odzyskiwanie pamięci, GC, 28, 237, 241–244, 248–255, 260, 342, 793
- ograniczenia typu, 153
 - dynamic, 526, 531
 - referencyjnego, 155
 - wartościowego, 157
- okno
 - Add View, 771
 - Exceptions, 304
 - New Project, 34
 - projektu MVC, 764
 - przesuwane, 434
 - Reference Manager, 35
- okrajanie, slicing, 207
- określanie
 - klasy bazowej, 207
 - typów, 517
 - typu delegatu, 533
 - właściwości DataContext, 730
- opakowywanie
 - łańcucha znaków, 573
 - typu referencyjnego, 272
 - zdarzeń, 452, 453
 - źródła, 451
- opcja
 - Add Reference, 760
 - Embedded Resource, 465
 - Find All References, 143
 - References, 42, 471
 - Resource File, 485
 - Start Debugging, 45
 - Start Without Debugging, 45
- opcje
 - kontynuacji, 644
 - tworzenia zadań, 641
- operacje
 - asynchroniczne, 565, 663, 678
 - bez blokowania, 636
 - na zbiorach, 384
 - odzwierciedlania, 554
 - równoległe, 677
 - wejścia-wyjścia, 614, 657
 - współbieżne, 653, 677
 - z uzależnieniami czasowymi, 456
- operandy, 60
- operator, 135
 - >=, 136
 - |, 135
 - ||, 135
 - !=, 108
 - &, 135, 818
 - &&, 135
 - ., 138
 - ??, 138
 - +, 54, 136
 - ++, 102
 - +=, 136
 - ==, 105, 108
 - >, 136
- Aggregate, 369, 438, 440
- All, 375
- Amb, 441
- Any, 374
- as, 209
- AsEnumerable<T>, 394
- AsQueryable<T>, 394
- Average, 379
- Buffer, 433, 439
 - dzielenie słów, 440
 - okna czasowe, 434, 460
- wygładzanie wyników, 435
- Concat, 384, 385, 431
- Contains, 374
- Count, 374
- DefaultIfEmpty<T>, 379
- Delay, 461
- DelaySubscription, 461
- Distinct, 384
- DistinctUntilChanged, 442
- ElementAt, 377
- ElementAtOrDefault, 377
- Except, 384
- false, 138
- First, 376
- FirstOrDefault, 376
- GroupBy, 388
- GroupJoin, 392, 423, 428
- Intersect, 384
- is, 209
- Join, 390, 426
- Last, 377
- LastOrDefault, 377
- LongCount, 374
- Max, 380
- Merge, 432
- Min, 380
- new, 98, 103
- null coalescing, 85
- OfType<T>, 365, 393
- Reverse, 385
- Sample, 460
- Scan, 440
- Select, 366, 422, 431
- SelectMany, 369, 371, 429
- SequenceEqual, 385
- Single, 375
- SingleOrDefault, 376
- Skip, 378, 383
- SkipWhile, 378
- Sum, 379
- Take, 378
- TakeWhile, 379
- ThenBy, 361
- Throttle, 459
- Timeout, 460
- Timestamp, 459
- ToArray, 375
- ToDictionary, 394
- ToList, 375
- ToLookup, 394
- trójargumentowy, 84
- true, 138
- typeof, 506, 734
- Union, 384
- Where, 358, 364, 422
- Window, 433
 - sekwencja obserwowalna, 439
 - wygładzanie, 437
- operator Zip, 385
- operatory
 - agregujące, 430
 - arytmetyczne, 82
 - bitowe, 82
 - konwersji, 136
 - LINQ, 186, 347, 395, 422, 749
 - logiczne, 83
 - okien czasowych, 460
 - przypisania złożone, 86

- operatory
 - relacyjne, 84
 - zwracające jedną wartość, 430
 - opóźnione podpisywanie,
 - delay sign, 478
 - opróżnianie strumienia, 561
 - optymalizacja kompilatora JIT, 498
 - ostrzeżenie, warning, 225
- P**
- P/Invoke
 - błędy Win32, 802
 - konwencje wywołań, 796
 - łańcuch znaków, 797
 - punkt wejścia, 797
 - pakiet
 - .xap, 525
 - Office, 523, 809
 - pakowanie, boxing, 141, 272
 - pakowanie danych typu
 - Nullable<T>, 276
 - pamięć
 - lokalna wątku, 604
 - podręczna podzespołów, 474
 - panel
 - Canvas, 703
 - DockPanel, 710
 - Grid, 705, 710
 - Solution Explorer, 32, 35
 - StackPanel, 703, 708, 714
 - Test Explorer, 39
 - Unit Test Explorer, 39, 45
 - WrapPanel, 711
 - panele
 - Windows Runtime, 709
 - WPF, 710
 - XAML, 702
 - Parallel LINQ, 399, 654
 - parametry typu, 149
 - pędzel, brush, 737
 - pętla
 - @foreach, 779
 - do, 91
 - for, 92
 - foreach, 93, 269, 745
 - while, 91, 269
 - piaskownica, sandbox, 22, 491
 - pisanie
 - kontrolerów, 772
 - modeli, 769
 - widoków, 771
 - Platform Invoke, 795
 - platforma
 - MapReduce, 369
 - Windows Forms, 491
 - WPF, 491
 - pliki
 - _Layout.cshtml, 767
 - _PageStart.cshtml, 761
 - _ViewStart.cshtml, 767
 - About.cshtml, 767
 - App.config, 166, 254
 - App.xaml, 733
 - AssemblyInfo.cs, 97, 538, 540
 - Contact.cshtml, 767
 - global.asax, 303
 - Global.asax, 777
 - Global.asax.cs, 762
 - Index.cshtml, 767
 - kodu ukrytego, 691
 - mscorlib, 479
 - Page.aspx.cs, 759
 - RouteConfig.cs, 777
 - StandardStyles.xaml, 739
 - Type.cshtml, 775
 - UnitText1.cs, 37
 - web.config, 254
 - pliki
 - appx, 490
 - appxsym, 490
 - appxupload, 490
 - aspx, 741, 752, 763
 - .cshtml, 741
 - .csproj, 31, 34, 743
 - .dll, 31
 - .exe, 31, 464
 - .msi, 491
 - .resx, 484
 - .sln, 32
 - .suo, 32
 - .vbhtml, 741
 - .vcxproj, 31
 - .winmd, 815
 - .xap, 492
 - .zip, 492
 - bitmap, 736
 - CSS, 751
 - nagłówkowe C++, 819
 - PE, 465
 - XAML, 287
 - XML, 491
 - zasobów, 486
 - pobieranie
 - atrybutów, 553, 555
 - obiektu Type, 506
 - strony WWW, 663
 - wyniku zadania, 643
 - z obiektu podzespołu, 500
 - zasobów, 485
 - podsystem POSIX, 467
 - podzespoły, 463
 - Global Assembly Cache, 474
 - hybrydowe, 488
 - identyfikator kulturowy, 484
 - klucze silnych nazw, 478
 - nazwa prosta, 476
 - nazwa silna, 476
 - numer wersji, 480
 - określanie architektury, 488
 - wczytywanie, 471, 473
 - współdziałania, interop
 - assembly, 809
 - zabezpieczenia, 493
 - podzespół, assembly, 463
 - ComparerLib, 471
 - Microsoft.CSharp, 525
 - mscorlib, 468
 - podział sterty, 250
 - pojemność listy, 183
 - pola, 115
 - pole
 - niestatyczne, 99
 - statyczne, 99
 - porównywanie
 - referencji, 104
 - wartości, 105
 - porządek
 - big-endian, 784
 - little-endian, 784
 - poszukiwanie wyjątków, 677
 - powiadomienia o zmianach
 - właściwości, 731
 - powinowactwo do wątku,
 - thread affinity, 550, 614
 - poziomy dostępu
 - chroniony, protected, 219
 - chroniony wewnętrzny,
 - protected internal, 219
 - prywatny, private, 218
 - publiczny, public, 218
 - wewnętrzny, internal, 219
 - preambuła, 577
 - predykat, predicate, 310
 - priorytet operatorów, 64
 - procedura obsługi zdarzeń, 694
 - procesy, 792
 - program
 - Excel, 523
 - ILDASM, 815

- program
 - sn, 478, 544
 - TLBIMP, 808
 - programowanie
 - asynchroniczne, 30, 325
 - obiektywne, 95
 - w oparciu o testy, 35
 - projekcja
 - elementów, 387
 - lambda, 431
 - projekcje, 368
 - projektant XAML, XAML designer, 698
 - promowanie, 75
 - propagacja zdarzeń, event bubbling, 340
 - protokół OData, 398
 - prywatna klasa zagnieżdżona, 139
 - przechodzenie
 - do końca instrukcji, 90
 - między sekcjami, 90
 - przechwytywanie
 - myszy, 422
 - wartości licznika, 333
 - zmiennych, 332
 - przeciążanie metod, 128
 - przeglądanie kolekcji, 93
 - przekazywanie argumentów
 - przez referencję, 126
 - przekroczenie zakresu, 77
 - przekształcanie
 - sygnatur, 799
 - wyników zapytania, 368
 - przenośne biblioteki klas, 488
 - przepełnienie danych, 77
 - przesłanianie metod
 - wirtualnych, 221
 - przestrzenie nazw, namespace, 40
 - XAML, 689
 - XML, 689
 - zagnieżdżone, 43
 - przestrzeń deklaracji, declaration scope, 57
 - przestrzeń nazw
 - Microsoft.Phone.Reactive, 404
 - System, 40
 - System.Collections.
 - Concurrent, 203, 639
 - System.Collections.Generics, 185
 - System.ComponentModel.
 - DataAnnotations, 772
 - System.Core, 42
 - System.Diagnostics, 68, 174
 - System.Globalization, 296
 - System.IO, 41
 - System.Linq, 354
 - System.Numerics, 79
 - System.Reactive, 404
 - System.Reactive.Linq, 443, 454
 - System.Reflection, 495
 - System.Runtime.
 - CompilerServices, 672
 - System.Text, 41
 - System.Threading.Tasks.
 - Dataflow, 655
 - System.Transactions, 604
 - System.Web, 42
 - System.Web.UI, 759
 - System.Windows, 132
 - Windows.Storage, 589
 - Windows.UI.Xaml.Controls, 689, 714
 - przeszukiwanie tablicy, 171
 - przetwarzanie
 - bloku instrukcji lock, 622
 - elementów, 421
 - kolekcji, 94
 - operandów, 64
 - opóźnione, 357
 - wyrażeń, 64
 - przypisanie, 63
 - pudełko, box, 141, 272
 - pula wątków, 609–616
 - punkt
 - strumienia, 560
 - wejścia do programu, 38, 44, 98
- ## R
- Razor, 741
 - bloki kodu, 746
 - klasy i obiekty stron, 748
 - sterowanie przepływem, 745
 - stosowanie wyrażeń, 743
 - strony początkowe, 751
 - wskazywanie treści, 747
 - RCW, runtime-callable wrapper, 788
 - Reactive Extensions, 401
 - dostosowywanie źródeł, 450
 - funkcje w wersjach .NET, 403
 - generowanie sekwencji, 418
 - grupowanie zdarzeń, 423
 - interfejs IObservable<T>, 405, 450
 - interfejs IOObserver<T>, 406, 450
 - mechanizmy szeregujące, 442
 - model wypychania, 401
 - obsługa subskrybentów, 417
 - operacje z uzależnieniami czasowymi, 456
 - operatory, 431
 - tematy, subjects, 447
 - TPL, 454
 - tworzenie zapytań LINQ, 421
 - tworzenie źródła, 413
 - zdarzenia .NET, 452
 - źródło zdarzeń, 407
 - redukcja, reduce, 369, 383
 - referencja, 101, 112
 - do zmiennej, 126
 - this, 229
 - typu object, 273
 - referencje główne, root references, 239
 - region wymuszonego wykonania, 306
 - reguła precyzyjna, 781
 - reguły
 - dotyczące typów, 779
 - wyraźnego przypisania, 54
 - rodzina, rodziny, 507
 - rozkład tekstu, 726
 - rozszerzenia znaczników, 734
 - Binding, 735
 - TemplateBinding, 734
 - równoległe obliczanie spłotu, 654
 - równoznaczność typów, type equivalence, 810
 - rzutowanie, 75, 273
 - obiektów, 523
 - sekwencji, 393
 - w dół, 208

S

satelickie podzespoły zasobów, 486
scalanie sterty, 249, 256
sekwencje, 185, 189, 396, 418
semafory, 632
serializacja, 545, 558, 590
 CLR, 591
 danych, 28
 kontraktu danych, 594–597
 wyjątku, 300
serwer IIS, 741
Silverlight, 403, 446, 488, 524
Silverlight dla Linuksa, 684
składanie, fold, 383
składowe, 97, 115
 klasy Stream, 558
 prywatne, 115
 statyczne, 98
 typu delegatu, 323
skrypty, 811
słabe referencje, 244
 długie, 248
 krótkie, 247
słowniki, 196, 597
słowniki posortowane, 198
słowo kluczowe
 abstract, 222
 async, 415, 658, 667, 678
 await, 415, 430, 567, 643, 658–661, 667–678
 base, 230
 case, 89
 catch, 286
 checked, 77, 162
 class, 44, 97, 100
 const, 116
 default, 159
 delegate, 311, 326
 dynamic, 50, 519–524, 813
 else, 88
 enum, 141
 event, 337
 explicit, 136
 extern, 785, 795
 group, 387
 implicit, 137
 in, 349
 internal, 97, 493
 lock, 619, 622
 new, 165, 169, 226
 null, 55
 operator, 135

out, 126, 213
override, 221, 225
params, 169, 170
partial, 146, 690
private, 97
public, 38, 97
readonly, 101, 115
ref, 126
sealed, 228
static, 66, 98, 125
string, 469
struct, 107, 157
this, 100, 135
throw, 292
try, 286
unchecked, 79
unsafe, 818
using, 564
var, 52, 53
virtual, 220
void, 45
where, 152
while, 92
yield, 190
solucja, solution, 32
sondowanie, probing, 473
sortowanie, 373
sortowanie tablicy, 174
sposoby kodowania, 574
sprawdzanie wartości
 HRESULT, 801
SSCLI, Share Source CLI, 26
stałe, 116
stan widoku, viewstate, 754
standard
 ECMA-335, 26
 IEEE 754, 72
statusy zadań, 642
statyczny typ zmiennej, 51
sterowanie przepływem, 87
sterta, heap, 238, 248, 256
stos, 201, 705
stosowanie
 atributu niestandardowego, 552
 puli wątków, 615
 sposobów kodowania, 577
 stylów, 738
 szeregowania, 784
strona
 _PageStart.cshtml, 751
 nadrzędna, master page, 760
 Page.aspx, 759

strony kodowe, 576
strony układu, layout pages, 749
struktura DisposableValue, 274
struktura wyrażenia, 61
struktury, 102, 106, 790, 791
struktury danych, 204
strumienie, 557
 aktualizowanie położenia, 560
 długość, 562
 kopiowanie, 562
 obsługa APM, 565
 obsługa TAP, 565
 opróżnianie, 561
 typy, 565
 zwalnianie, 564
strumień
 FileStream, 593
 StreamReader, 664
subskrybent, 416, 455
subskrypcja
 obserwowalnych źródeł, 417
 zdarzeń, 405
sygnatura metody delegatu, 338
symbol
 @, 743
 DEBUG, 67
 TRACE, 67
symbole kompilacji, 67
synchronizacja, 618
szablon, template, 721
 projektu MVC, 767
szablony
 C++, 161
 danych, 734
 danych, data template, 732
 kontrolki, 721
szeregowanie, marshaling, 784

T

tablica, 165, 792
 byte[], 558
 CultureInfo[], 356
 Curse.Catalog, 366
tablice
 mieszające, hash table, 109
 nieregularne, 178
 prostokątne, 180
TAP, Task-based Asynchronous Pattern, 565

- technologia
 - ClickOnce, 491
 - COM, 521, 550, 798, 802
 - metadane, 805
 - skrypty, 811
 - IntelliSense, 144
 - LINQ, 130
 - LINQ to Objects, 186
 - Silverlight, 29
 - TPL Dataflow, 653
 - Web Forms, 752
 - tematy, subjects, 447
 - termin
 - ASCII, 576
 - Unicode, 574
 - testy
 - jednostkowe, 37, 45, 729
 - zawierania, 373
 - token
 - funkcji, function token, 323
 - klucza publicznego, 476, 479
 - metadanych, metadata token, 293
 - tożsamość typu, 468
 - TPL, Task Parallel Library, 30, 263, 325, 455, 610
 - TPL Dataflow, 654
 - transakcja otoczenia,
 - ambient transaction, 604
 - trasa domyślna, 781
 - trasowanie, routing, 742, 777
 - tryb serwerowy, 255
 - tryby
 - odzyskiwania pamięci, 254
 - stacji roboczej, 254
 - tworzenie
 - aplikacji internetowych, 741
 - atrybutów, 536
 - delegatu, 311
 - dynamiczne obiektów, 501
 - instancji klasy COM, 806, 808
 - kolekcji asocjacyjnej, 394
 - krotki, 204
 - metod zwrotnych, 309
 - obiektów, 230
 - obiektów COM, 813
 - obiektów Type, 509
 - obiektu IBuffer, 816
 - obiektu StreamWriter, 584
 - odpowiednika metody asynchronicznej, 660
 - okien czasowych, 460
 - plików Windows Inaller, 491
 - projektu, 33
 - pudełka, 273
 - silnej nazwy, 478
 - słabych referencji, 247
 - stron HTML, 742
 - stron WWW, 741
 - tablic, 165
 - tablicy nieregularnej, 178
 - typów wartościowych, 111
 - wątków, 608, 612
 - źródła, 414
 - typ
 - atrybutu, 551
 - BigInteger, 79, 86
 - BindingFlags, 502, 509
 - bool, 80
 - CancellationToken, 652
 - Capture, 41
 - Complex, 107, 114
 - ConstructorInfo, 510
 - EventHandler, 339
 - Dictionary, 597
 - dynamic, 519–527, 813
 - elementy HTML, 525
 - mechanizmy współdziałania, 521
 - ograniczenia, 526, 531–534
 - Enum, 234
 - EventArgs, 338
 - FileAccess, 579
 - FileMode, 579
 - FileOptions, 581
 - IntPtr, 786, 793
 - MethodBase, 510
 - MethodInfo, 510
 - MouseButtonEventArgs, 339
 - Nullable<T>, 106, 158, 276
 - object, 81, 144
 - opakowujący, wrapper type, 106
 - Rect, 383
 - skonstruowany, constructed type, 150
 - string, 80
 - System.Attribute, 235
 - System.MulticastDelegate, 235
 - System.Object, 217, 526
 - System.ValueType, 234
 - UnmanagedType, 788
 - WeakReference, 245
 - WeakReference<T>, 245
 - zmiennej, 51
 - typowanie
 - dynamiczne, 50, 518
 - jawne, 518
 - silne, 518
 - słabe, 518
 - statyczne, 50, 518, 523
 - typy
 - anonimowe, 53, 145, 367
 - bazowe, 234
 - CRT, 235
 - częściowe, 146
 - kopiowalne, 785
 - liczbowe, 71
 - należące do CLS, 72
 - ogólne, generic types, 149, 211, 509
 - referencyjne, 101, 111, 234
 - statyczne, 517
 - tablicowe, 166
 - wartościowe, 106, 114, 234
 - wyliczeniowe, 141, 144, 234, 502, 579, 788
 - zagnieżdżone, 138
 - zmiennoprzecinkowe, 72
- ## U
- układ
 - marginesy, 698
 - tekstu, 726
 - szerokość i wysokość, 700
 - właściwości, 696
 - wypełnienia, 698
 - wyrównanie, 697
 - zdarzenia, 712
 - ukrywanie
 - metod, 225
 - zmiennej, 57
 - Unicode, 51, 575, 787
 - unieruchamianie
 - bloków pamięci, 258
 - tablicy, 819
 - uruchamianie CLR, 465
 - usługi odzwierciedlania, 28
 - ustawienia kulturowe, 486
 - usuwanie
 - delegatów, 315
 - obiektów, 239
 - procedury obsługi zdarzeń, 340

V

VBA, Visual Basic for Applications, 522
VES, Virtual Execution System, 26
Visual Studio, 31, 463

W

warstwy interfejsu użytkownika, 730
wartości, 112

- domyślne, 159
- logiczne, 80
- wynikowe, 798

wartość

- HRESULT, 799
- null, 276, 279
- pusta, nullable types, 138

wątek sprzętowy, hardware thread, 599
wątki

- CLR, 601
- pierwszoplanowe, foreground threads, 609

wbudowane typy danych, 70
WCF, Windows Communication Foundation, 347
WCF Data Services, 397
wczytywanie podzespołów, assembly loader, 463, 471
wdrażanie pakietów, 490
Web Forms, 741, 752

- bloki kodu, 758
- klasy i obiekty stron, 759
- kontrolki serwerowe, 752
- obiekty stron, 759
- strony nadrzędne, 760
- wyrażenia, 758

WER, Windows Error Reporting, 293
weryfikacja

- argumentów, 674
- iteratora, 193
- metody asynchronicznej, 675
- podpisu, 479
- żądania, 744

węzeł References, 36
wiązanie

- danych, data binding, 729
- dwukierunkowe, 731
- szablonów, 722

widok

- Assembly.cshtml, 773
- Index.cshtml, 767

widok-model, View-Model, 764
wielowątkowość, 599
wielowątkowość współbieżna, 600
Windows 8, 490
Windows Azure, 811
Windows Phone, 403, 488, 686
Windows Runtime, 456, 566, 686

- bufor, 816
- działanie klas, 815
- metadane, 815

Windows SDK, 787
Windows Workflow Foundation, 688
WinRT, 23, 30
właściwości, 130

- dołączane, attachable properties, 693
- kontrolki, 723
- obiektu strony, 748

właściwość, 77

- AggregateException, 650
- automatyczna, 131
- CallStack, 287
- Cells, 523
- CurrentPhaseNumber, 632
- DateTime.Now, 458
- DeclaredProperties, 516
- DeclaredType, 504
- DefinedTypes, 496, 500
- EndOfStream, 284
- Environment.TickCount, 77
- Exception, 647
- FieldType, 513
- Filter, 344
- FusionLog, 472, 480
- GenericTypeArguments, 510
- HasDefaultValue, 512
- Headers, 749
- Height, 701
- HorizontalAlignment, 697
- HResult, 563
- InnerException, 294
- InnerExceptions, 647
- IsAlive, 247
- IsCompleted, 669
- IsGenericTypeDefinition, 510
- IsNestedFamANDAssem, 507

IsNestedFamily, 507
IsPasswordRevealButton

- ↳ Enable!, 728

Items, 720
Layout, 750
Length, 562
Length tablicy, 166, 181
LocalFolder, 589
LongLength tablicy, 166
Margin, 697–700
Method, 322
NewLine, 571
ObjectForScripting, 811
Orientation, 698
Page.Title, 750
Position, 558, 560
ReflectedType, 504
Result, 676
Stretch, 736
Target, 322
TargetSite, 287
Text, 726
Timestamp, 458
TotalHours, 380
ViewBag, 767
Width, 701
Worksheets, 523

- zmiennego typu wartościowego, 132

WMF, Windows Media Foundation, 737
wnioskowanie typu, 52, 160
WPF, Windows Presentation Foundation, 42, 683
wskaźnik, 28, 786, 818
wskaźnik niezarządzany, 816
wskaźniki do funkcji, 789
wstrzykiwanie zależności, dependency injection, 132
wtyczka Silverlight, 684
wycieki pamięci, 342
wydajność działania programu, 549
wyjątek, exception, 279

- AggregateException, 673, 676
- AggregateException, 647
- ArgumentOutOfRangeException
 - ↳ Exception, 296
- ArrayTypeMismatch
 - ↳ Exception, 216
- COMException, 801
- DivideByZeroException, 285

- wyjątek, exception,
 - FileNotFoundException, 286, 301, 472, 588
 - FormatException, 280
 - IndexOutOfRangeException, 168, 287
 - InvalidCastException, 209
 - InvalidOperationException, 201, 297, 376
 - IOException, 287, 562
 - KeyNotFoundException, 197
 - MarshalDirectiveException, 801
 - MissingMethodException, 223
 - NotImplementedException, 297
 - NotSupportedException, 297, 560
 - NullReferenceException, 83, 85, 276, 289, 336
 - ObjectDisposedException, 267
 - OutOfMemoryException, 191, 305
 - OverflowException, 78
 - RuntimBinderException, 519
 - StackOverflowException, 305, 608
 - System.OverflowException, 77
 - ThreadAbortException, 305
 - UnobservedTaskException, 678
 - XamlParseException, 287
 - XamlParseException, 303
 - wyjątki
 - asynchroniczne, 285, 305
 - COMException, 799
 - nieobsługiwane, 301, 677
 - niestandardowe, 298
 - niezaobserwowane, 647
 - pojedyncze, 675
 - zgłaszane przez API, 282
 - wymuszanie odzyskiwania pamięci, 260
 - wypychanie, push, 401
 - wyrażenia, 60
 - lambda, lambda
 - expressions, 327, 335, 381, 439, 610, 667
 - zakodowane, 758
 - zapytań, query expression, 347
 - wyrażenie
 - @using, 749
 - this(), 120
 - wyszukiwanie
 - binarne, 173, 176
 - plików, 585
 - wyświetlanie
 - bitmap, 736
 - filmów, 737
 - komunikatu, 45
 - tekstu, 725
 - wywołania zwrotne, 336, 345
 - wywoływanie
 - delegatów, 316, 317
 - finalizatora, 261
 - funkcji Win32, 795
 - metody, 38, 127, 509, 512
 - asynchronicznej, 660
 - ogólnej, 160
 - wirtualnej, 221
 - wzorce asynchroniczne, 456, 651
 - wzorzec
 - APM, 651
 - delegatów zdarzeń, 338
 - słowa kluczowego await, 668
- ## X
- XAML, 681–739
 - elementy podrzędne, 692
 - elementy właściwości, 692
 - grafika, 735
 - klasy, 690
 - kod ukryty, 690
 - kontrolki, 713
 - obsługa zdarzeń, 694
 - panele, 702
 - platformy, 682
 - przestrzenie nazw, 689
 - style, 738
 - tekst, 725
 - układ, 696
 - wątki, 695
 - wiązanie danych, 729
 - XBAP, XAML browser application, 492
 - XML, 30
- ## Z
- zabezpieczenie
 - dostępu do danej, 626
 - stanu, 619
 - zabieranie pracy, work stealing, 612
 - zadania, 640
 - bezwątkowe, 641, 650
 - bezwątkowe
 - niestandardowe, 648
 - nadrzędne, 649
 - podrzędne, 649
 - złożone, 650
 - zadaniowy wzorzec
 - asynchroniczny, TAP, 565
 - zagnieżdżanie
 - elementów, 692
 - klas, 97
 - zagnieżdżone bloki try, 289
 - zakleszczenie, 638
 - zakres zmiennej, 55
 - zamykanie uchwytów, 270, 564
 - zapis do pliku, 572
 - zapisywanie tablicy byte[], 817
 - zapytania
 - LINQ, 348, 357, 421
 - SQL, 336
 - zapytanie
 - grupujące, 386–388
 - opóźnione, 359
 - zasoby, 465
 - zasoby Win32, 467
 - zastosowanie wskaźników, 818
 - zawieranie typów
 - odzwierciedlania, 496
 - zbiory, 200
 - zbiór SortedSet, 201
 - zdarzenia, events, 138, 336, 344, 628
 - zdarzenia .NET, 452
 - zdarzenie
 - Click, 615
 - DispatcherUnhandled
 - ↳Exception, 303
 - LayoutChanged, 712
 - Loaded, 694
 - MouseDown, 425
 - MouseUp, 425
 - PropertyChanged, 542, 732
 - SendCompleted, 629
 - SizeChanged, 713
 - TextChanged, 694
 - UnhandledException, 302
 - UnobservedTaskException, 647
 - zewnętrzna nazwa zastępcza, extern alias, 470

- zgłaszanie
 - powtórne wyjątku, 292
 - wyjątku, 292
- zgodność
 - delegatów, 320
 - łącza z trasą, 779
- ziarno, seed, 380
- złączenia, 390
- złączenie pogrupowane, 392
- zmienna, 50
 - fullUri, 240
 - offset, 333
 - treshold, 329
 - zakresu, range variable, 349
- zmiennie
 - lokalne, 50, 58
 - typu dynamic, 519

- typu referencyjnego, 106
- typy wartościowe, 132
- znacznik kolejności bajtów,
BOM, 573
- znaczniki asp., 753
- znak
 - gwiazdki, 818
 - minusa, 60
- zwalnianie
 - opcjonalne, 271
 - strumieni, 564
 - zasobów, 331
- zwracanie
 - obiektu Task, 666
 - zadania Task<T>, 666

Ż

- źródła wyjątków, 281
- źródło
 - asynchroniczne, 415
 - ciepłe, 407, 415, 429
 - Rx, 405, 407
 - zdarzeń, 402
 - zimne, 407, 415

Ż

- żądanie POST, 753, 755

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄZKI
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**

C# 5.0. Programowanie

Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework



W dzisiejszych czasach szczególną popularnością cieszą się języki programowania pozwalające na pisanie kodu łatwego do przenoszenia między platformami. Nikt nie ma czasu na pisanie kilku wersji jednej aplikacji. C# to uniwersalny język, w którym bez trudu dokonasz tego dzieła. Dzięki swojej elastyczności, wydajności oraz mocnemu wsparciu społeczności zdobył uznanie programistów. Taki wybór to strzał w dziesiątkę!

Ten rewelacyjny podręcznik jest Twoim kluczem do poznania wszystkich niuansów języka C# 5.0. Kolejne wydanie zostało zaktualizowane o wszystkie nowości w C#. Znajdziesz tu kompletny opis języka i platformy .NET. W trakcie lektury oprócz standardowych zagadnień będziesz mógł sprawdzić, jak tworzyć aplikacje dla systemu Windows 8 i interfejsu Metro. Ponadto błyskawicznie opanujesz detale związane z programowaniem obiektowym, dynamicznym i statycznym określaniem typów oraz językiem XAML. Książka ta jest uznanym kompendium wiedzy na temat języka C#. Musisz ją mieć!

Dzięki tej książce:

- przygotujesz interfejs użytkownika zgodny z duchem Windows 8
- wykorzystasz wielowątkowość w platformie .NET
- poznasz podstawy programowania obiektowego
- przekonasz się, jak LINQ może ułatwić Ci życie
- opanujesz język C#

Wykorzystaj potencjał języka C#!

Ian Griffiths jest autorem kursu WPF oraz instruktorem w firmie Pluralsight specjalizującej się w prowadzeniu kursów Microsoft .NET. Pracuje także jako niezależny konsultant. Jest współautorem książek *Windows Forms in a Nutshell*, *Mastering Visual Studio .NET* oraz *Programming WPF*, wydanych przez wydawnictwo O'Reilly.

helion.pl
księgarnia
internetowa

Nr katalogowy: 14522



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900
0 601 339900



Helion

Sprawdź najnowsze promocje:

🔗 <http://helion.pl/promocje>

Książki najchętniej czytane:

🔗 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

🔗 <http://helion.pl/nowosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po WIECEJ



KOD KORZYŚCI

ISBN 978-83-246-6984-4



Cena 129,00 zł