

O'REILLY®

Helion

# C# 12 w pigułce

Kompendium programisty



Joseph Albahari

Tytuł oryginału: C# 12 in a Nutshell: The Definitive Reference

Tłumaczenie: Łukasz Piwko z wykorzystaniem fragmentów „C# 9.0 w pigułce”  
w przekładzie Roberta Górczyńskiego i Jakuba Hubisza

ISBN: 978-83-289-1483-4

© 2024 Helion S.A.

Authorized Polish translation of the English edition of *C# 12 in a Nutshell*  
ISBN 9781098147440 © 2024 Joseph Albahari.

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

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

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

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

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

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/c12wpi>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

---

# Spis treści

<b>Wstęp .....</b>	<b>11</b>
<b>1. Wprowadzenie do C# i .NET.....</b>	<b>17</b>
Obiektowość	17
Bezpieczeństwo typów	18
Zarządzanie pamięcią	19
Platformy	19
CLR, BCL i środowiska wykonawcze	19
Historia C# w pigułce	24
<b>2. Podstawy języka C# .....</b>	<b>46</b>
Pierwszy program w języku C#	46
Składnia	49
Podstawy typów	51
Typy liczbowe	61
Typ logiczny i operatory logiczne	69
Łańcuchy znaków i pojedyncze znaki	71
Tablice	75
Zmienne i parametry	80
Wyrażenia i operatory	90
Operatory null	95
Instrukcje	96
Przestrzenie nazw	106
<b>3. Tworzenie typów w języku C# .....</b>	<b>114</b>
Klasy	114
Dziedziczenie	136
Typ object	147
Struktury	151

Modyfikatory dostępu	154
Interfejsy	156
Wyliczenia	163
Typy zagnieżdżone	166
Typy generyczne	167
<b>4. Zaawansowane elementy języka C# .....</b>	<b>180</b>
Delegaty	180
Zdarzenia	188
Wyrażenia lambda	194
Metody anonimowe	200
Instrukcje try i wyjątki	201
Wyliczenia i iteratory	209
Typy wartościowe dopuszczające wartość null	214
Typy referencyjne dopuszczające wartość null	219
Metody rozszerzające	222
Typy anonimowe	224
Krotki	226
Rekordy	230
Wzorcy	241
Atrybuty	246
Atrybuty informacji wywołującego	248
Wiązanie dynamiczne	250
Przeciążanie operatorów	258
Polimorfizm statyczny	261
Niebezpieczny kod i wskaźniki	264
Dyrektywy preprocesora	270
Dokumentacja XML	273
<b>5. Ogólny zarys platformy .....</b>	<b>277</b>
Docelowe środowiska wykonawcze i TFM	278
.NET Standard	279
Zestawy referencyjne	280
Wersje środowiska i C#	280
CLR i BCL	281
Warstwy aplikacji	285
<b>6. Podstawowe wiadomości o platformie .NET .....</b>	<b>289</b>
Obsługa łańcuchów i tekstu	289
Data i godzina	302
Daty i strefy czasowe	309

Formatowanie i parsowanie obiektów DateTime	314
Standardowe łańcuchy formatu i flagi parsowania	320
Inne mechanizmy konwersji	326
Globalizacja	330
Praca z liczbami	331
Wyliczenia	336
Struktura Guid	339
Porównywanie	340
Określanie kolejności	350
Klasy pomocnicze	353
<b>7. Kolekcje .....</b>	<b>358</b>
Przeliczalność	358
Interfejsy ICollection i IList	365
Klasa Array	368
Listy, kolejki, stosy i zbiory	376
Słowniki	384
Kolekcje i pośredniki z możliwością dostosowywania	390
Niezmiennicze kolekcje	396
Kolekcje zamrożone	399
Dołączanie protokołów równości i porządkowania	400
<b>8. Zapytania LINQ .....</b>	<b>407</b>
Podstawy	407
Składnia płynna	409
Wyrażenia zapytań	415
Wykonywanie opóźnione	419
Podzapytania	425
Tworzenie zapytań złożonych	428
Strategie projekcji	432
Zapytania interpretowane	434
EF Core	440
Budowanie wyrażań zapytań	450
<b>9. Operatory LINQ .....</b>	<b>455</b>
Informacje ogólne	456
Filtrowanie	459
Projekcja	463
Łączenie	475
Porządkowanie	482

Grupowanie	485
Operatory zbiorów	489
Metody konwersji	490
Operatory elementów	493
Metody agregacyjne	495
Kwantyfikatory	500
Metody generujące	501
<b>10. LINQ to XML .....</b>	<b>502</b>
Przegląd architektury	502
Informacje ogólne o X-DOM	503
Tworzenie drzewa X-DOM	506
Nawigowanie i wysyłanie zapytań	509
Modyfikowanie drzewa X-DOM	514
Praca z wartościami	517
Dokumenty i deklaracje	519
Nazwy i przestrzenie nazw	523
Adnotacje	528
Projekcja do X-DOM	529
<b>11. Inne technologie XML i JSON .....</b>	<b>533</b>
Klasa XmlReader	533
Klasa XmlWriter	541
Typowe zastosowania klas XmlReader i XmlWriter	543
Praca z formatem JSON	547
<b>12. Zwalnianie zasobów i mechanizm usuwania nieużytków .....</b>	<b>560</b>
IDisposable, Dispose i Close	560
Automatyczne usuwanie nieużytków	566
Finalizatory	568
Jak działa mechanizm usuwania nieużytków?	573
Wycieki pamięci zarządzanej	579
Słabe odwołania	583
<b>13. Diagnostyka .....</b>	<b>587</b>
Kompilacja warunkowa	587
Debugowanie i klasy monitorowania	591
Integracja z debuggerem	594
Procesy i wątki procesów	595
Klasy StackTrace i StackFrame	596

Dziennik zdarzeń Windows	598
Liczniki wydajności	600
Klasa Stopwatch	604
Międzyplatformowe narzędzia diagnostyczne	605
<b>14. Współbieżność i asynchroniczność .....</b>	<b>609</b>
Wprowadzenie	609
Wątki	610
Zadania	626
Reguły asynchroniczności	634
Funkcje asynchroniczne w języku C#	639
Wzorce asynchroniczności	659
Przestarzałe wzorce	667
<b>15. Strumienie i wejście-wyjście .....</b>	<b>670</b>
Architektura strumienia	670
Użycie strumieni	672
Adapter strumienia	686
Kompresja strumienia	694
Praca z plikami w postaci archiwum ZIP	697
Praca z plikami TAR	698
Operacje na plikach i katalogach	699
Bezpieczeństwo systemu operacyjnego	709
Mapowanie plików w pamięci	711
<b>16. Sieć .....</b>	<b>716</b>
Architektura sieci	716
Adresy i porty	718
Adresy URI	719
Klasa HttpClient	721
Tworzenie serwera HTTP	730
Użycie DNS	733
Wysyłanie poczty elektronicznej za pomocą SmtplibClient	733
Użycie TCP	734
Otrzymywanie poczty elektronicznej POP3 za pomocą TcpClient	738
<b>17. Zestawy .....</b>	<b>740</b>
Co znajduje się w zestawie?	740
Silne nazwy i podpisywanie zestawu	744
Nazwy zestawów	745

Technologia Authenticode	748
Zasoby i zestawy satelickie	750
Ładowanie, znajdowanie i izolowanie zestawów	757
<b>18. Refleksja i metadane .....</b>	<b>777</b>
Refleksja i aktywacja typów	778
Refleksja i wywoływanie składowych	784
Refleksja dla zestawów	799
Praca z atrybutami	799
Generowanie dynamicznego kodu	804
Emitowanie zestawów i typów	811
Emitowanie składowych typów	813
Emitowanie generycznych metod i typów	819
Kłopotliwe cele emisji	821
Parsowanie IL	824
<b>19. Programowanie dynamiczne .....</b>	<b>829</b>
Dynamiczny system wykonawczy języka	829
Dynamiczne wybieranie przeciążonych składowych	831
Implementowanie obiektów dynamicznych	837
Współpraca z językami dynamicznymi	840
<b>20. Kryptografia .....</b>	<b>842</b>
Informacje ogólne	842
Windows Data Protection	843
Obliczanie skrótów	844
Szyfrowanie symetryczne	846
Szyfrowanie kluczem publicznym i podpisywanie	851
<b>21. Zaawansowane techniki wielowątkowości .....</b>	<b>855</b>
Przegląd technik synchronizacji	856
Blokowanie wykluczające	856
Blokady i bezpieczeństwo ze względu na wątki	864
Blokowanie bez wykluczania	869
Sygnalizacja przy użyciu uchwytów zdarzeń oczekiwania	877
Klasa Barrier	884
Leniwa inicjalizacja	885
Pamięć lokalna wątku	887
Zegary	891



<b>22. Programowanie równoległe .....</b>	<b>895</b>
Dlaczego PFX?	896
PLINQ	899
Klasa Parallel	911
Równoległe wykonywanie zadań	917
Klasa AggregateException	927
Kolekcje współbieżne	929
Klasa BlockingCollection<T>	932
<b>23. Struktury Span&lt;T&gt; i Memory&lt;T&gt; .....</b>	<b>936</b>
Struktura Span i plasterkowanie	937
Struktura Memory<T>	941
Enumeratory działające tylko do przodu	942
Praca z pamięcią alokowaną na stosie i niezarządzaną	944
<b>24. Współdziałanie macierzyste i poprzez COM .....</b>	<b>946</b>
Odwołania do natywnych bibliotek DLL	946
Szeregowanie typów i parametrów	947
Wywołania zwrotne z kodu niezarządzanego	951
Symulowanie unii C	954
Pamięć współdzielona	955
Mapowanie struktury na pamięć niezarządzaną	957
Współpraca COM	961
Wywołanie komponentu COM z C#	963
Osadzanie typów współpracujących	966
Udostępnianie obiektów C# COM	967
<b>25. Wyrażenia regularne .....</b>	<b>969</b>
Podstawy wyrażeń regularnych	969
Kwantyfikatory	974
Asercje o zerowej wielkości	975
Grupy	978
Zastępowanie i dzielenie tekstu	980
Receptury wyrażeń regularnych	981
Leksykon języka wyrażeń regularnych	984
<b>Skorowidz .....</b>	<b>988</b>



---

## Tworzenie typów w języku C#

W tym rozdziale szczegółowo opisujemy typy i składowe typów.

### Klasy

**Klasa** jest najczęściej używanym rodzajem typów referencyjnych. Najprostsza możliwa deklaracja klasy wygląda tak:

```
class NazwaKlasy
{
}
```

Bardziej złożone klasy mogą dodatkowo zawierać następujące składniki:

<b>Przed słowem kluczowym <code>class</code></b>	<i>atrybuty i modyfikatory klasy. Modyfikatory niezagnieżdżonych klas to: <code>public</code>, <code>internal</code>, <code>abstract</code>, <code>sealed</code>, <code>static</code>, <code>unsafe</code> oraz <code>partial</code>.</i>
<b>Za nazwą klasy</b>	<i>generyczne parametry typu i ograniczenia, nazwa klasy bazowej oraz interfejsy.</i>
<b>W klamrach</b>	<i>składowe klasy (metody, własności, indeksatory, zdarzenia, pola, konstruktory, przeciężone operatory, typy zagnieżdżone oraz finalizator).</i>

W tym rozdziale znajduje się opis wszystkich wymienionych składników oprócz atrybutów, funkcji operatorów oraz słowa kluczowego `unsafe`, które są opisane w rozdziale 4. W kilku następnych sekcjach opisujemy po kolei wszystkie składniki klasy.

### Pola

**Pole** to zmienna będąca składową klasy lub struktury. Na przykład:

```
class Octopus
{
    string name;
    public int Age = 10;
}
```

Pola mogą mieć następujące modyfikatory:

Stacyczny	static
Dostępu	public internal private protected
Dziedziczenia	new
Niebezpiecznego kodu	unsafe
Tylko do odczytu	readonly
Wątkowy	volatile

Są dwie popularne szkoły nadawania nazw polom prywatnym: notacja wielbłądzia (np. `drugieImie`) i notacja wielbłądzia ze znakiem podkreślenia (`_drugieImie`). Druga umożliwia natychmiastowe odróżnienie pól prywatnych od parametrów i zmiennych lokalnych.

## Modyfikator `readonly`

Modyfikator `readonly` uniemożliwia zmienianie wartości zmiennej po utworzeniu obiektu. Wartość takiemu polu można przypisać tylko w deklaracji lub konstruktorze zawierającego to pole typu.

## Inicjalizacja pól

Inicjalizacja pól jest opcjonalna. Pole niezainicjalizowane ma wartość domyślną (0, `\0`, `null` lub `false`). Inicjalizatory pól są wykonywane przed konstruktorami:

```
public int Age = 10;
```

Inicjalizator pól może zawierać wyrażenia i wywoływać metody:

```
static readonly string TempFolder = System.IO.Path.GetTempPath();
```

## Deklarowanie wielu pól naraz

Dla wygody można zadeklarować wiele pól tego samego typu za pomocą listy elementów rozdzielanych przecinkami. Jest to wygodny sposób deklarowania pól o takich samych atrybutach i z takimi samymi modyfikatorami. Na przykład:

```
static readonly int legs = 8,  
                 eyes = 2;
```

## Stałe

Wartość **stałej** jest określana statycznie w czasie kompilacji i kompilator dosłownie podstawia jej wartość tam, gdzie została użyta (podobnie jak dzieje się z markami w C++). Stała może mieć dowolny wbudowany typ numeryczny, `bool`, `char`, `string` lub wyliczeniowy.

Stałe deklaruje się za pomocą słowa kluczowego `const` i należy je zainicjalizować wartością. Na przykład:

```
public class Test  
{  
    public const string Message = "Witaj, świecie";  
}
```

Stała może odgrywać podobną rolę, jak statyczne pole tylko do odczytu, ale jest znacznie bardziej restrykcyjna zarówno pod względem dopuszczalnego zestawu typów, jak i semantyki inicjalizacji pól. Ponadto stała różni się od statycznego pola tylko do odczytu tym, że obliczenie jej wartości następuje w czasie kompilacji, a zatem ten kod:

```
public static double Circumference (double radius)
{
    return 2 * System.Math.PI * radius;
}
```

w procesie kompilacji zostanie zamieniony na taki:

```
public static double Circumference (double radius)
{
    return 6.2831853071795862 * radius;
}
```

Zdefiniowanie PI jako stałej jest uzasadnione, ponieważ wartość tej liczby jest znana z góry w czasie kompilacji. Dla porównania wartość statycznego pola tylko do odczytu w każdym uruchomieniu programu może być inna:

```
static readonly DateTime StartupTime = DateTime.Now;
```



Styczne pole tylko do odczytu jest lepszym wyborem, gdy trzeba udostępnić innym zestawom wartość, która może się zmienić w późniejszej wersji. Powiedzmy, że zestaw X udostępnia następującą stałą:

```
public const decimal ProgramVersion = 2.3;
```

Jeśli zestaw Y korzystający z zestawu X użyje tej zmiennej, to wartość 2.3 zostanie wpisana w zestaw Y w czasie kompilacji. W efekcie, jeśli zestaw X zostanie później skompilowany ponownie z wartością stałej zmienioną na 2.4, to zestaw Y wprowadzi ją dopiero *po tym, jak sam zostanie ponownie skompilowany*. Styczne pole tylko do odczytu pozwala pozbyć się tego problemu.

Można też na to spojrzeć w ten sposób, że wartość, która może zmienić się w przyszłości, nie spełnia definicji stałej, w związku z czym nie powinna być tak reprezentowana.

Stałe można też deklarować lokalnie w metodach:

```
void Test()
{
    const double twoPI = 2 * System.Math.PI;
    ...
}
```

Nielokalne stałe mogą mieć następujące modyfikatory:

Modyfikator dostępu	public internal private protected
Modyfikator dziedziczenia	new

## Metody

**Metoda** wykonuje pewną czynność podzieloną na serię instrukcji. Może przyjmować od wywołującego dane *wejściowe* w postaci *parametrów* oraz *zwracać* dane do wywołującego za pomocą *typu zwrotnego*. Metoda może mieć typ zwrotny void, tzn. nie zwraca żadnej wartości do wywołującego. Ponadto metoda może zwracać dane poprzez parametry ref i out.

**Sygnatura** metody nie może się powtarzać w obrębie danego typu. Sygnatura składa się z nazwy metody i typów parametrów (nie należą do niej *nazwy* parametrów ani typ zwrotny).

Metody mogą mieć następujące modyfikatory:

Statyczny	static
Dostępu	public internal private protected
Dziedziczenia	new virtual abstract override sealed
Częściowej metody	partial
Kodu niezarządzanego	unsafe extern
Kodu asynchronicznego	async

## Metody wyrażeniowe

Metody zawierające tylko jedno wyrażenie, takie jak ta:

```
int Foo (int x) { return x * 2; }
```

można zapisywać zwięźlej jako **metody wyrażeniowe** (ang. *expression-bodied method*). W ich składni pozbywamy się klamry i słowa kluczowego return, a dodajemy strzałkę:

```
int Foo (int x) => x * 2;
```

Funkcje wyrażeniowe mogą też mieć typ zwrotny void:

```
void Foo (int x) => Console.WriteLine (x);
```

## Metody lokalne

Metody można definiować w innych metodach:

```
void WriteCubes()  
{  
    Console.WriteLine (Cube (3));  
    Console.WriteLine (Cube (4));  
    Console.WriteLine (Cube (5));  
    int Cube (int value) => value * value * value;  
}
```

Metoda lokalna (w tym przypadku Cube) jest widoczna tylko w zawierającej ją metodzie (WriteCubes). To upraszcza typ nadrzędny i dla programisty czytającego kod stanowi sygnał, że Cube nie używa się nigdzie indziej. Inną zaletą metod lokalnych jest to, że mają dostęp do lokalnych zmiennych i parametrów zawierających je metod. Ten fakt ma poważne implikacje, o których szerzej piszemy w rozdziale 4., w podrozdziale „Przechwytywanie zewnętrznych zmiennych”.

Metody lokalne mogą występować także w innych rodzajach funkcji, takich jak metody dostępu do własności, konstruktory itd. Można nawet definiować je w innych lokalnych metodach i w wyrażeniach lambda używających bloków instrukcji (rozdział 4.). Lokalne metody mogą być iteratorami (rozdział 4.) lub mogą działać asynchronicznie (rozdział 14.).

## Statyczne metody lokalne

Dodanie modyfikatora `static` (od C# 8) do lokalnej metody odbiera jej dostęp do lokalnych zmiennych i parametrów zawierającej ją metody. Pomaga to ograniczyć gęstość powiązań i umożliwia deklarowanie dowolnych nazw zmiennych w metodzie wewnętrznej bez ryzyka wystąpienia kolizji z nazwami w metodzie zewnętrznej.

## Metody lokalne i instrukcje najwyższego poziomu

Wszystkie metody zadeklarowane w instrukcjach najwyższego poziomu są traktowane jako metody lokalne. To znaczy, że mają dostęp (jeśli nie zdefiniowano ich jako statycznych) do zmiennych w instrukcjach najwyższego poziomu:

```
int x = 3;
Foo();
void Foo() => Console.WriteLine(x);
```

## Przeciążanie metod



Metod lokalnych nie można przeciążać, co znaczy, że nie można tego robić z metodami zadeklarowanymi w instrukcjach najwyższego poziomu (które są traktowane jako metody lokalne).

Typ może **przeciążać** metody (zawierać kilka metod o takiej samej nazwie), pod warunkiem że każda z nich ma inną sygnaturę. Na przykład wszystkie poniższe metody mogą się znajdować w jednym typie:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

Natomiast poniższe pary metod nie mogą koegzystować w jednym typie, ponieważ typ zwrotny i modyfikator `params` nie wchodzi w skład sygnatury:

```
void Foo (int x) {...}
float Foo (int x) {...} // błąd kompilacji

void Goo (int[] x) {...}
void Goo (params int[] x) {...} // błąd kompilacji
```

Informacja, czy parametr jest przekazywany przez wartość, czy przez referencję, należy do sygnatury. Na przykład `Foo(int)` może współistnieć z `Foo(ref int)` lub `Foo(out int)`. Natomiast `Foo(ref int)` i `Foo(out int)` nie mogą się znajdować w tym samym typie:

```
void Foo (int x) {...}
void Foo (ref int x) {...} // na razie OK
void Foo (out int x) {...} // błąd kompilacji
```

## Konstrukторы egzemplarzy

Konstruktor wykonuje kod inicjalizacyjny na klasie lub strukturze. Definiuje się go jak metodę, tylko nazwę i typ zwrotny redukuje się do postaci nazwy typu, do którego konstruktor należy:

```
Panda p = new Panda ("Petey"); // wywołanie konstruktora

public class Panda
{
    string name;           // definicja pola
    public Panda (string n) // definicja konstruktora
    {
        name = n;         // kod inicjalizacyjny (ustawienie wartości pola)
    }
}
```

Konstruktory egzemplarzy mogą mieć następujące modyfikatory:

Dostępu	public internal private protected
Kodu niezarządzanego	unsafe extern

Konstruktory zawierające tylko jedną instrukcję można także pisać jako składowe będące wyrażeniami:

```
public Panda (string n) => name = n;
```



Jeśli nazwa parametru (lub jakiegokolwiek zmiennej) koliduje z nazwą pola, to problem ten można rozwiązać przez dodanie do nazwy pola przedrostka w postaci referencji `this`:

```
public Panda (string name) => this.name = name;
```

## Przeciążanie konstruktorów

Klasy i struktury mogą przeciążać konstruktory. Aby uniknąć powielania kodu, jeden konstruktor może wywoływać inny za pomocą słowa kluczowego `this`:

```
public class Wine
{
    public decimal Price;
    public int Year;
    public Wine (decimal price) { Price = price; }
    public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

Gdy jeden konstruktor wywołuje inny, to ten *wywoływany konstruktor* zostaje wykonany pierwszy.

Do drugiego konstruktora można przekazać *wyrażenie*:

```
public Wine (decimal price, DateTime year) : this (price, year.Year) { }
```

W wyrażeniu można używać składowych statycznych klasy, ale nie składowych egzemplarza. (Ta zasada została wprowadzona, ponieważ na tym etapie obiekt nie jest jeszcze zainicjalizowany przez konstruktor, więc wykonywanie metod na tym obiekcie nie może się udać).



Ten konkretny przykład można by było lepiej zaimplementować przy użyciu pojedynczego konstruktora z `year` jako parametru opcjonalnego:

```
public Wine (decimal price, int year = 0)
{
    Price = price; Year = year;
}
```

W podrozdziale „Inicjalizatory obiektów” przedstawiamy jeszcze inne rozwiązanie.

## Niejawne konstruktory bez parametrów

Dla klas kompilator C# automatycznie generuje publiczny konstruktor bez parametrów, ale wtedy i tylko wtedy, gdy programista sam nie zdefiniuje żadnego konstruktora. Jeśli programista zdefiniuje jakikolwiek konstruktor, automatyczne generowanie zostaje wstrzymane.

## Konstruktor i kolejność inicjalizowania pól

Wiemy już, że polom można nadać wartości domyślne w deklaracji:

```
class Player
{
    int shields = 50; // pierwsza inicjalizacja
    int health = 100; // druga inicjalizacja
}
```

Inicjalizacja pól następuje *przed* wykonaniem konstruktora i w kolejności występowania deklaracji.

## Konstruktory niepubliczne

Konstruktor nie musi być publiczny. Niepubliczne konstruktory najczęściej tworzy się w celu kontrolowania procesu tworzenia egzemplarzy przez wywołania metody statycznej. Metoda ta może zwracać obiekt z puli, zamiast za każdym razem tworzyć nowy egzemplarz, albo na podstawie otrzymanych na wejściu argumentów może zwracać obiekty różnych podklas:

```
public class Klasa1
{
    Klasa1() {} // konstruktor prywatny
    public static Klasa1 Create (...)
    {
        // kod zwracający egzemplarzy klasy Klasa1
        ...
    }
}
```

## Dekonstrukторы

Dekonstruktor (zwany także **metodą dekonstrukcji**) działa mniej więcej w sposób odwrotny do konstruktora, tzn. podczas gdy konstruktor za pośrednictwem parametrów pobiera zestaw wartości i przypisuje je do pól, dekonstruktor przypisuje pola z powrotem do zmiennych.

Metoda dekonstrukcji musi nazywać się `Deconstruct` i zawierać przynajmniej jeden parametr wyjściowy, jak w poniższym przykładzie:

```
class Rectangle
{
    public readonly float Width, Height;
    public Rectangle (float width, float height)
    {
        Width = width;
        Height = height;
    }
}
```



```

public void Deconstruct (out float width, out float height)
{
    width = Width;
    height = Height;
}

```

Do wywołania dekonstruktora służy specjalna składnia:

```

var rect = new Rectangle (3, 4);
(float width, float height) = rect;    // dekonstrukcja
Console.WriteLine (width + " " + height); // 3 4

```

Drugi wiersz zawiera wywołanie dekonstrukcyjne. Tworzy dwie lokalne zmienne, po czym wywołuje metodę Deconstruct. To wywołanie jest równoważne z następującym:

```

float width, height;
rect.Deconstruct (out width, out height);

```

a także z tym:

```

rect.Deconstruct (out var width, out var height);

```

W wywołaniach dekonstrukcji dozwolone jest korzystanie z niejawnego typizowania, więc nasze wywołanie możemy skrócić do następującej postaci:

```

(var width, var height) = rect;

```

albo po prostu:

```

var (width, height) = rect;

```



Jeśli nie interesuje Cię jedna lub więcej zmiennych, możesz użyć operatora odrzucenia (\_):

```

var (_, height) = rect;

```

To lepiej wyraża intencję programisty niż deklaracja zmiennej, która nigdy nie jest używana.

Jeśli zmienne używane w procesie dekonstrukcji są już zdefiniowane, typy można opuścić:

```

float width, height;
(width, height) = rect;

```

Taka konstrukcja nazywa się **przypisaniem dekonstrukcyjnym**. Dzięki niej można uprościć konstruktor klasy:

```

public Rectangle (float width, float height) =>
    (Width, Height) = (width, height);

```

Przeciążając metodę Deconstruct, można dostarczyć kilka różnych opcji dekonstrukcji do wyboru.



Metoda Deconstruct może być metodą rozszerzenia (zobacz „Metody rozszerzeń” w rozdziale 4.). Z możliwości tej można skorzystać, gdy ktoś chce dokonać dekonstrukcji typu, którego autorem jest ktoś inny.

Od C# 10 podczas dekonstrukcji można mieszać istniejące i nowe zmienne:

```

double x1 = 0;
(x1, double y2) = rect;

```

## Inicjalizatory obiektów

Aby uprościć inicjalizację obiektów, wszystkie jego dostępne pola i własności można ustawiać przez **inicjalizator obiektu** bezpośrednio po zakończeniu procesu konstrukcji. Spójrz na poniższą przykładową klasę:

```
public class Bunny
{
    public string Name;
    public bool LikesCarrots, LikesHumans;

    public Bunny () {}
    public Bunny (string n) { Name = n; }
}
```

Przy użyciu inicjalizatorów obiektów można tworzyć obiekty klasy Bunny w następujący sposób:

```
// jeśli konstruktor nie ma parametrów, można opuścić pusty nawias
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo") { LikesCarrots=true, LikesHumans=false };
```

Te instrukcje utworzenia obiektów b1 i b2 są równoznaczne z poniższym kodem:

```
Bunny temp1 = new Bunny(); // temp1 to nazwa wygenerowana przez kompilator
temp1.Name = "Bo";
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

Zmienne tymczasowe są wykorzystywane po to, by w razie wystąpienia wyjątku podczas inicjalizacji nie pozostał nam częściowo niezainicjalizowany obiekt.

### Inicjalizatory obiektów a parametry opcjonalne

Zamiast używać inicjalizatorów obiektów, konstruktor klasy Bunny można by zdefiniować w następujący sposób z jednym obowiązkowym parametrem i z dwoma parametrami opcjonalnymi:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
    Name = name;
    LikesCarrots = likesCarrots;
    LikesHumans = likesHumans;
}
```

Teraz obiekt klasy Bunny moglibyśmy utworzyć w następujący sposób:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

Z historycznego punktu widzenia zaletą polegania na konstruktorach w kwestii inicjalizacji obiektów jest to, że w razie potrzeby pola (albo *własności*, o czym piszemy nieco dalej) klasy Bunny moglibyśmy zdefiniować jako tylko do odczytu. Definiowanie pól tylko do odczytu jest dobrym pomysłem, jeśli nie ma żadnego powodu do ich zmiany przez cały okres istnienia obiektu.

Ten cel w odniesieniu do inicjalizatorów obiektów możemy natomiast osiągnąć przy użyciu wprowadzonego w C# 9 modyfikatora `init`, o którym jest mowa w podrozdziale o własnościach.

Parametry opcjonalne mają dwie wady. Pierwsza jest taka, że choć umożliwiają korzystanie z typów tylko do odczytu w metodach niebędących konstruktorami, nie pozwalają na (łatwą) *mutację bez destrukcji*. (Mutacja bez destrukcji — i rozwiązanie tego problemu — jest opisana w rozdziale 4. w podrozdziale „Rekordy (C# 9)”).

Druga wada parametrów opcjonalnych polega na tym, że w bibliotekach publicznych utrudniają utrzymanie zgodności wstecznej. Wynika to z tego, że dodanie opcjonalnego parametru w późniejszym czasie godzi w *zgodność binarną* zestawu z istniejącymi konsumentami. (Jest to szczególnie istotne w przypadku, gdy biblioteka zostanie opublikowana w NuGet: gdy konsument korzysta z pakietów *A* i *B*, które korzystają z niekompatybilnych wersji pakietu *L*, powstaje trudny do rozwiązania problem).

Problemem jest to, że wartość każdego parametru opcjonalnego jest wbudowana w *miejsce wywołania*. Innymi słowy — C# przetłumaczy nasz konstruktor na taką postać:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

Może to być problemem, gdy utworzymy egzemplarz klasy `Bunny` z innego zestawu, a później zmodyfikujemy tę klasę przez dodanie kolejnego parametru opcjonalnego — np. `likesCats`. Jeśli zestaw używający klasy nie zostanie ponownie skompilowany, będzie nadal wywoływał już nieistniejący konstruktor z trzema parametrami, czego skutkiem będzie błąd wykonawczy. (Trudniejszy do wykrycia błąd może wystąpić, gdy zmienimy wartość jednego z parametrów opcjonalnych, a w innych zestawach nadal będzie używana stara wartość opcjonalna, dopóki zestawu te nie zostaną skompilowane ponownie).

Na koniec należy jeszcze zastanowić się nad wpływem konstruktorów na tworzenie podklas (o czym piszemy w podrozdziale „Dziedziczenie”). Obecność wielu konstruktorów z długimi listami parametrów utrudnia tworzenie podklas. Dlatego dobrym pomysłem może być ograniczenie ich liczby i poziomu złożoności do minimum oraz wstawianie szczegółów za pomocą inicjalizatorów.

## Referencja `this`

Referencja `this` odnosi się do samego egzemplarza. W poniższym przykładzie metoda `Marry` za pomocą referencji `this` ustawia pole `Mate` partnera:

```
public class Panda
{
    public Panda Mate;

    public void Marry (Panda partner)
    {
        Mate = partner;
        partner.Mate = this;
    }
}
```

Referencja `this` pozwala też odróżnić zmienną lokalną lub parametr od pola. Na przykład:

```
public class Test
{
    string name;
    public Test (string name) { this.name = name; }
}
```

Referencji `this` można używać tylko w niestatycznych składowych klas i struktur.

## Własności

Własności na zewnątrz wyglądają jak pola, ale wewnątrz zawierają logikę, tak jak metody. Na przykład patrząc na poniższy kod, nie da się stwierdzić, czy `CurrentPrice` jest polem, czy własnością:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Własności deklaruje się tak jak pola, ale z dodatkiem bloku `get/set`. Oto przykładowa implementacja własności `CurrentPrice`:

```
public class Stock
{
    decimal currentPrice;           // prywatne pole pomocnicze

    public decimal CurrentPrice    // własność publiczna
    {
        get { return currentPrice; }
        set { currentPrice = value; }
    }
}
```

Elementy `get` i `set` to **metody dostępne** własności. Metoda `get` jest wykonywana, gdy ktoś odczytuje własność i musi zwracać wartość takiego samego typu, jakiego jest własność. Metoda `set` jest uruchamiana, gdy ktoś przypisuje wartość własności. Ma ona tego samego typu co własność niejawni parametr o nazwie `value`, który zazwyczaj przypisuje się do prywatnego pola (w tym przypadku `currentPrice`).

Choć własności używa się tak samo jak pól, różnią się od nich tym, że dają programiście pełną kontrolę nad procesem sprawdzania i ustawiania wartości. Dzięki temu programista może dowolnie zdefiniować wewnętrzną implementację, nie ujawniając jej użytkownikom własności. W tym przykładzie metoda `set` mogłaby zgłaszać wyjątek, gdyby wartość `value` nie mieściła się w określonym przedziale.



W tej książce dla uproszczenia często używamy pól publicznych, ale w prawdziwej aplikacji raczej używa się własności publicznych, które pozwalają zachować hermetyczność obiektów.

Własności mogą mieć następujące modyfikatory:

Statyczny	<code>static</code>
Dostępu	<code>public internal private protected</code>
Dziedziczenia	<code>new virtual abstract override sealed</code>
Kodu niezarządzanego	<code>unsafe extern</code>

## Własności tylko do odczytu i obliczane

Własność jest tylko do odczytu, gdy ma zdefiniowaną tylko metodę dostępową `get`, a tylko do zapisu, gdy ma zdefiniowaną tylko metodę dostępową `set`. Własności tylko do zapisu są rzadko spotykane.

Typowa własność zawiera specjalne pole pomocnicze do przechowywania potrzebnych jej danych, ale jej wartość może też być obliczana na podstawie innych informacji. Na przykład:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
    get { return currentPrice * sharesOwned; }
}
```

## Własności wyrażeniowe

Własności tylko do odczytu, takie jak przedstawiona w poprzednim punkcie, można deklarować w zwięźlejszy sposób. Służąca do tego składnia nazywa się **własnością wyrażeniową** (ang. *expression-bodied property*). Klamrę oraz słowa kluczowe get i return zastępuje strzałka:

```
public decimal Worth => currentPrice * sharesOwned;
```

Dodatkowo wyrażeniami mogą być metody ustawiające, co definiuje się za pomocą dodatkowej składni:

```
public decimal Worth
{
    get => currentPrice * sharesOwned;
    set => sharesOwned = value / currentPrice;
}
```

## Własności automatyczne

Większość własności zawiera metodę pobierającą (get) i/lub ustawiającą (set), które służą do odczytywania i zapisywania wartości w polach prywatnych tego samego typu co własność. Deklaracja **własności automatycznej** stanowi dla kompilatora sygnał, że ma sam dostarczyć implementację. Pierwszy przykład z tej sekcji można poprawić przez zadeklarowanie CurrentPrice jako własności automatycznej:

```
public class Stock
{
    ...
    public decimal CurrentPrice { get; set; }
}
```

Kompilator automatycznie wygeneruje prywatne pole pomocnicze o pewnej nazwie, do którego nie będzie można się odnosić. Jeśli własność ma być tylko do odczytu dla innych typów, metodę set można oznaczyć jako prywatną (private) lub chronioną (protected). Własności automatyczne wprowadzono w C# 3.0.

## Inicjatory własności

Programista może używać **inicjatorów własności** automatycznych, których używa się podobnie jak w przypadku pól:

```
public decimal CurrentPrice { get; set; } = 123;
```

Własność `CurrentPrice` otrzyma wartość początkową 123. Własności z inicjalizatorami mogą być tylko do odczytu:

```
public int Maximum { get; } = 999;
```

Tak jak pola tylko do odczytu, własności automatyczne tylko do odczytu mogą mieć przypisywane wartości w konstruktorze typu. Dzięki temu można tworzyć **typy niezmiennicze** (ang. *immutable type*), czyli tylko do odczytu.

## Dostępność metod `get` i `set`

Metody dostępne `get` i `set` mogą mieć ustawione różne poziomy dostępności. Najczęściej tworzy się publiczną własność z metodą `set` opatrzoną modyfikatorem `internal` lub `private`:

```
public class Foo
{
    private decimal x;
    public decimal X
    {
        get { return x; }
        private set { x = Math.Round (value, 2); }
    }
}
```

Zwróć uwagę, że sama własność ma mniej restrykcyjny modyfikator dostępu (w tym przypadku `public`), a wybrana metoda dostępowa ma modyfikator *bardziej* restrykcyjny.

## Metody ustawiające tylko do inicjalizacji

Od C# 9 można deklarować metody dostępu do własności przy użyciu słowa kluczowego `init` zamiast `set`:

```
public class Note
{
    public int Pitch { get; init; } = 20; // własność „tylko do inicjalizacji”
    public int Duration { get; init; } = 100; // własność „tylko do inicjalizacji”
}
```

Te własności *tylko do inicjalizacji* są jak własności tylko do odczytu, z tą różnicą, że dodatkowo można ustawiać ich wartość za pomocą inicjalizatora obiektu:

```
var note = new Note { Pitch = 50 };
```

Potem takiej własności już nie można zmienić:

```
note.Pitch = 200; // Błąd — metoda ustawiająca tylko do inicjalizacji!
```

Własności tylko do inicjalizacji nie można ustawiać nawet wewnątrz klasy, chyba że za pomocą inicjalizatora własności, konstruktora lub innej konstrukcji dostępowej wykonującej tylko inicjalizację.

Alternatywą dla własności tylko do inicjalizacji jest tworzenie własności tylko do odczytu ustawianych przez konstruktor:

```
public class Note
{
    public int Pitch { get; }
```

```

public int Duration { get; }
public Note (int pitch = 20, int duration = 100)
{
    Pitch = pitch; Duration = duration;
}
}

```

Gdyby ta klasa znajdowała się w publicznej bibliotece, takie podejście utrudniałoby obsługę wersji, ponieważ dodanie opcjonalnego parametru do konstruktora w późniejszym czasie niweczy zgodność binarną z konsumentami (natomiast dodanie nowej własności tylko do inicjalizacji w naszym nie szkodzi).



Ponadto własności tylko do inicjalizacji mają jeszcze jedną ważną zaletę — w połączeniu z rekordami (podrozdział „Rekordy” w rozdziale 4.) umożliwiają niedestrukcyjną mutację.

Tak samo jak zwykle metody ustawiające, metody ustawiające tylko do inicjalizacji mogą dostarczać implementację:

```

public class Note
{
    readonly int _pitch;
    public int Pitch { get => _pitch; init => _pitch = value; }
    ...
}

```

Należy zauważyć, że pole `_pitch` jest tylko do odczytu: metody ustawiające tylko do odczytu mogą modyfikować pola tylko do odczytu w swojej klasie. (Bez tej cechy pole `_pitch` musiałoby dopuszczać możliwość zapisu i klasa nie byłaby wewnętrznie niezmienna).



Zmiana metody dostępowej własności z `init` na `set` (lub odwrotnie) powoduje *utrata zgodności binarnej*: każdy korzystający z zestawu będzie musiał ponownie skompilować swój zestaw.

To nie powinno stanowić problemu przy tworzeniu całkowicie niezmiennych typów, ponieważ taki typ nigdy nie potrzebuje własności z (zapisywalną) metodą ustawiającą.

## Implementacja własności w CLR

Metody dostępne w C# są wewnętrznie kompilowane do postaci metod o nazwach `get_XXX` i `set_XXX`:

```

public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}

```

Metoda dostępowa `init` jest przetwarzana jak `set`, tylko ma dodatkową flagę w metadanych „modreq” (podrozdział „Własności tylko do inicjalizacji” w rozdziale 18.).

Proste niewirtualne metody dostępne własności są przez kompilator JIT *rozwijane*, co eliminuje różnice wydajnościowe między dostępem do własności i pól. Rozwijanie (ang. *inlining*) to technika optymalizacyjna polegająca na zastąpieniu wywołania metody treścią tej metody.

## Indeksatory

Indeksatory zapewniają naturalną składnię dostępu do elementów klas i struktur zawierających listy lub słowniki wartości. Indeksatory są podobne do własności, ale dostęp do nich odbywa się za pośrednictwem argumentu indeksowego, a nie nazwy własności. Klasa `string` ma np. indeksator dający dostęp do każdej jej wartości char poprzez indeks typu `int`:

```
string s = "cześć";
Console.WriteLine (s[0]); // 'c'
Console.WriteLine (s[3]); // 'ś'
```

Składnia indeksatorów jest taka sama jak w tablicach, z tym że argument indeksowy może być każdego typu.

Indeksatory mają takie same modyfikatory jak własności (zob. sekcję „Własności”) i mogą być wywoływane warunkowo w zależności od wyniku testu na wartość `null` przez wstawienie znaku zapytania przed nawiasem kwadratowym (zob. sekcję „Operatory null” w rozdziale 2.):

```
string s = null;
Console.WriteLine (s?[0]); // nic nie drukuje; brak błędu
```

## Implementowanie indeksatora

Aby napisać indeksator, zdefiniuj własność o nazwie `this` z argumentami w nawiasie kwadratowym. Na przykład:

```
class Sentence
{
    string[] words = "Nosił wilk razy kilka".Split();

    public string this [int wordNum] // indeksator
    {
        get { return words [wordNum]; }
        set { words [wordNum] = value; }
    }
}
```

Oto przykład użycia tego indeksatora:

```
Sentence s = new Sentence();
Console.WriteLine (s[1]); // wilk
s[1] = "kangur";
Console.WriteLine (s[1]); // kangur
```

Typ może mieć kilka indeksatorów, każdy z innym zestawem parametrów. Ponadto indeksator może przyjmować więcej niż jeden parametr:

```
public string this [int arg1, string arg2]
{
    get { ... } set { ... }
}
```

Gdyby usunięto metodę `set`, indeksator byłby narzędziem tylko do odczytu. Ponadto można stosować składnię wyrażeniową, która jest bardziej zwięzła:

```
public string this [int wordNum] => words [wordNum];
```



## Implementacja indeksatorów w CLR

Podczas kompilacji indeksatory są zamieniane na metody o nazwach `get_Item` i `set_Item`, jak w przykładzie:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value) {...}
```

## Używanie indeksów i zakresów z indeksatorami

Jeśli programista chce, aby jego klasy obsługiwały indeksy i zakresy (zobacz podrozdział „Indeksy i zakresy” w rozdziale 2.), to powinien zdefiniować indeksator z parametrem typu `Index` lub `Range`. Poprzedni przykład możemy rozszerzyć przez dodanie następujących indeksatorów do klasy `Sentence`:

```
public string this [Index index] => words [index];
public string[] this [Range range] => words [range];
```

Teraz można pisać taki kod:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]); // kilka
string[] firstTwoWords = s [..2]; // (Nosił, wilk)
```

## Konstruktory podstawowe (C# 12)

Od C# 12 listę parametrów można umieszczać bezpośrednio po deklaracji klasy (lub struktury):

```
class Person (string firstName, string lastName)
{
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

To jest znak dla kompilatora, aby automatycznie utworzył **konstruktor podstawowy** (ang. *primary constructor*) przy użyciu **parametrów konstruktora podstawowego** (`firstName` i `lastName`), dzięki czemu obiekt naszej klasy możemy utworzyć w następujący sposób:

```
Person p = new Person ("Alicja", "Jackowska");
p.Print(); // Alicja Jackowska
```

Konstruktory podstawowe są przydatne podczas tworzenia prototypów i w innych prostych sytuacjach. Alternatywną opcją byłoby zdefiniowanie pól i samodzielne napisanie konstruktora:

```
class Person // (bez konstruktorów podstawowych)
{
    string firstName, lastName; // Deklaracje pól

    public Person (string firstName, string lastName) // Konstruktor
    {
        this.firstName = firstName; // Przypisanie do pola
        this.lastName = lastName; // Przypisanie do pola
    }

    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

Konstruktor tworzony przez C# nazywa się podstawowym, ponieważ wszystkie dodatkowe konstruktory napisane przez programistę muszą go wywoływać:

```
class Person (string firstName, string lastName)
{
    public Person (string firstName, string lastName, int age)
        : this (firstName, lastName) // Musi wywołać konstruktor podstawowy
    {
        ...
    }
}
```

Dzięki temu jest pewne, że parametry konstruktora podstawowego **zawsze otrzymają wartości**.



W C# można używać także **rekordów**, których opis znajduje się w podrozdziale „Rekordy” w rozdziale 4. One też obsługują konstruktory podstawowe, ale w ich przypadku kompilator wykonuje dodatkową czynność, którą jest wygenerowanie (domyślnie) publicznej własności tylko do inicjalizacji dla każdego parametru konstruktora podstawowego. Jeśli zależy Ci na tego typu funkcjonalności, rozważ użycie rekordu.

Ze względu na poniższe ograniczenia konstruktory podstawowe najlepiej sprawdzają się w prostych zastosowaniach:

- Do konstruktora podstawowego nie można wstawić dodatkowego kodu inicjalizującego.
- Mimo że parametr konstruktora podstawowego łatwo można udostępnić jako własność publiczną, nie da się w prosty sposób dodać logiki weryfikacji, chyba że własność jest tylko do odczytu.

Konstruktory podstawowe zastępują domyślny konstruktor bezparametrowy, który zostałby wygenerowany przez C#, gdyby nie one.

## Semantyka konstruktora podstawowego

Aby zrozumieć sposób działania konstruktorów podstawowych, najpierw przyjrzyj się działaniu zwykłego konstruktora:

```
class Person
{
    public Person (string firstName, string lastName)
    {
        ... jakieś operacje na firstName, lastName
    }
}
```

Kiedy kod wewnątrz tego konstruktora zakończy działanie, parametry `firstName` i `lastName` znikną z zakresu dostępności i nie będzie już można ich używać. Natomiast parametry konstruktora podstawowego nie znikają z zakresu dostępności i mogą być używane w dowolnym miejscu klasy przez cały czas istnienia obiektu.



Parametry konstruktora podstawowego są specjalnymi konstrukcjami C#, nie **polami**, choć w razie potrzeby kompilator i tak generuje ukryte pola do przechowywania ich wartości.

## Konstruktory podstawowe oraz inicjalizatory pól i własności

Dostępność parametrów konstruktora podstawowego obejmuje także inicjalizatory pól i własności. W poniższym przykładzie za pomocą inicjalizatorów pól i własności przypisujemy `firstName` do pola publicznego, a `lastName` — do własności publicznej:

```
class Person (string firstName, string lastName)
{
    public readonly string FirstName = firstName; // Pole
    public string LastName { get; } = lastName; // Własność
}
```

## Maskowanie parametrów konstruktorów podstawowych

Pola (lub własności) mogą mieć nazwy pokrywające się z nazwami parametrów konstruktora podstawowego:

```
class Person (string firstName, string lastName)
{
    readonly string firstName = firstName;
    readonly string lastName = lastName;

    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

W takiej sytuacji większą wagę ma pole lub własność, czego skutkiem jest zamaskowanie parametru konstruktora podstawowego. **Wyjątkiem** jest prawa strona inicjalizatora pola i własności (pogrubione).



Parametry konstruktora podstawowego, tak jak zwykle parametry, można zapisywać. Zamaskowanie ich za pomocą pola tylko do odczytu (jak w naszym przykładzie) chroni je przed późniejszą modyfikacją.

## Weryfikacja parametrów konstruktora podstawowego

Czasami trzeba wykonać pewne obliczenia w inicjalizatorach pól:

```
new Person ("Alicja", "Jackowska").Print(); // Alicja Jackowska

class Person (string firstName, string lastName)
{
    public readonly string FullName = firstName + " " + lastName;
    public void Print() => Console.WriteLine (FullName);
}
```

W następnym przykładzie zapisujemy wersję `lastName` pisaną wielkimi literami w polu o tej samej nazwie (czyli maskujemy pierwotną wartość):

```
new Person ("Alicja", "Jackowska").Print(); // Alicja JACKOWSKA

class Person (string firstName, string lastName)
{
    readonly string lastName = lastName.ToUpper();
    public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

W rozdziale 4. w podrozdziale „Wyrażenia throw” pokazujemy, jak zgłaszać wyjątki w razie napotkania niepoprawnych danych. Poniżej przedstawiamy, jak to wykorzystać w połączeniu z konstruktorami podstawowymi w celu upewnienia się podczas tworzenia obiektu, że `lastName` nie ma wartości `null`:

```
new Person ("Alicja", null); // zgłasza ArgumentNullException

class Person (string firstName, string lastName)
{
    readonly string lastName = (lastName == null)
        ? throw new ArgumentNullException ("lastName")
        : lastName;
}
```

(Pamiętaj, że kod w inicjalizatorze pola lub własności jest wykonywany podczas tworzenia obiektu, a nie w chwili użycia pola lub własności). W następnym przykładzie udostępniamy parametr konstruktora podstawowego jako własność do odczytu i zapisu:

```
class Person (string firstName, string lastName)
{
    public string LastName { get; set; } = lastName;
}
```

Dodanie weryfikacji do tego przykładu nie jest proste, ponieważ należy to zrobić w dwóch miejscach: w (ręcznie zaimplementowanej) metodzie ustawiającej własności i w inicjalizatorze własności. (Ten sam problem istnieje, gdy własność jest zdefiniowana jako tylko do inicjalizacji). W tym momencie lepiej jest zrezygnować z konstruktorów podstawowych oraz samodzielnie zdefiniować konstruktor i odpowiednie pola.

## Konstruktory statyczne

Konstruktor statyczny jest wykonywany tylko raz dla *typu*, a nie raz dla każdego *egzemplarza*. W typie może być zdefiniowany tylko jeden konstruktor statyczny, który nie może przyjmować parametrów i musi mieć taką samą nazwę jak typ:

```
class Test
{
    static Test() { Console.WriteLine ("Typ zainicjalizowany."); }
}
```

System wykonawczy automatycznie wywołuje konstruktor statyczny przed użyciem typu. Może to nastąpić w dwóch przypadkach:

- przy tworzeniu egzemplarza typu,
- przy dostępie do statycznej składowej typu.

Jedynie modyfikatory, jakich można używać w definicjach konstruktorów statycznych, to `unsafe` i `extern`.



Jeżeli statyczny konstruktor zgłosi nieobsługiwany wyjątek (rozdział 4.), zawierający go typ staje się *bezużyteczny* do końca czasu działania aplikacji.



Od C# 9 można także definiować *inicjalizatory modułowe*, które są wykonywane po jeden raz na moduł (po pierwszym załadowaniu zestawu). Aby zdefiniować inicjalizator modułowy, należy napisać statyczną metodę `void` i przypisać jej atrybut `[ModuleInitializer]`:

```
[System.Runtime.CompilerServices.ModuleInitializer]
internal static void InitAssembly() {}
```

## Konstruktory statyczne i kolejność inicjalizacji pól

Inicjalizatory pól statycznych są wykonywane bezpośrednio *przed* wywołaniem konstruktora statycznego. Jeżeli typ nie ma konstruktora statycznego, statyczne inicjalizatory pól są wykonywane bezpośrednio przed użyciem typu — lub *kiedyś wcześniej* w zależności od kaprysu systemu wykonawczego.

Inicjalizatory pól statycznych są wykonywane w kolejności deklaracji tych pól. Ilustruje to poniższy przykład — pole `X` jest inicjalizowane wartością `0`, a `Y` — `3`:

```
class Foo
{
    public static int X = Y; //0
    public static int Y = 3; //3
}
```

Jeżeli zamienimy miejscami te dwa inicjalizatory, oba pola zostaną zainicjalizowane wartością `3`. Poniższy przykładowy program drukuje `0`, a następnie `3`, ponieważ inicjalizator pola tworzący egzemplarz typu `Foo` jest wykonywany przed inicjalizacją `X` wartością `3`:

```
Console.WriteLine (Foo.X); } //3

class Foo
{
    public static Foo Instance = new Foo();
    public static int X = 3;
    Foo() => Console.WriteLine (X); //0
}
```

Jeżeli zamienimy miejscami pogrubione wiersze, program wydrukuje dwie trójki.

## Klasy statyczne

Klasę można oznaczyć modyfikatorem `static` na znak, że może zawierać tylko statyczne składowe oraz że nie można stworzyć jej podklas. Dobrymi przykładami tego rodzaju klas są klasy `System.Console` i `System.Math`.

## Finalizatory

Finalizatory to metody klasowe wywoływane, zanim system usuwania nieużytków odzyska pamięć zajmowaną przez nieużywany obiekt. Składnia finalizatora składa się z nazwy klasy z przedrostkiem `~`:

```
class Class1
{
```

```

~Class1()
{
    ...
}
}

```

W istocie jest to składnia do przesłaniania metody `Finalize` klasy `Object`. Kompilator rozwinie tę deklarację do następującej postaci:

```

protected override void Finalize()
{
    ...
    base.Finalize();
}

```

Szczegółowy opis usuwania nieużywanych obiektów i finalizatorów znajduje się w rozdziale 12.

Finalizatory zawierające tylko jedną instrukcję można definiować przy użyciu składni wyrażień:

```

~Class1() => Console.WriteLine ("Finalizowanie");

```

## Metody i typy częściowe

Typy częściowe to technika pozwalająca dzielić definicje typów na części — najczęściej kilka różnych plików. Często spotykaną sytuacją jest wygenerowanie części klasy przez jakiś automat (np. z szablonu Visual Studio) i dodanie do niej przez programistę pozostałych potrzebnych składników. Na przykład:

```

// PaymentFormGen.cs - wygenerowany automatycznie
partial class PaymentForm { ... }
// PaymentForm.cs - napisany ręcznie
partial class PaymentForm { ... }

```

Każda część musi zawierać w deklaracji słowo kluczowe `partial`, tzn. poniższy kod jest nieprawidłowy:

```

partial class PaymentForm {}
class PaymentForm {}

```

Części nie mogą zawierać składowych kolidujących ze składowymi innych części. Na przykład nie może się powtórzyć konstruktor przyjmujący takie same parametry. Kompilator pracuje na typach częściowych w całości, tzn. wszystkie części muszą być dostępne w czasie kompilacji i muszą one znajdować się w tym samym zestawie.

Klasę bazową można określić w jednej deklaracji części lub w większej liczbie deklaracji, ale pod warunkiem że w każdym przypadku jest to ta sama klasa. Ponadto każda część może niezależnie od pozostałych określać interfejsy, które muszą być zaimplementowane. Szczegółowy opis klas bazowych i interfejsów znajduje się w sekcji „Dziedziczenie” i w sekcji „Interfejsy”.

Kompilator niczego nie gwarantuje w odniesieniu do kolejności inicjalizowania pól w częściowych deklaracjach typów.

## Metody częściowe

Typ częściowy może zawierać **metody częściowe**, które są dla programisty punktami zaczepienia do wpisania własnej implementacji. Na przykład:

```
partial class PaymentForm // w pliku wygenerowanym automatycznie
{
    ...
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm // w pliku pisanym ręcznie
{
    ...
    partial void ValidatePayment (decimal amount)
    {
        if (amount > 100)
            ...
    }
}
```

Metoda częściowa składa się z dwóch części: **definicji i implementacji**. Definicja jest zazwyczaj automatycznie tworzona przez generator kodu, a implementację pisze ręcznie programista. Jeżeli implementacja nie zostanie dostarczona, wygenerowana definicja metody częściowej zostaje usunięta (wraz z wywołującym ją kodem). Dzięki temu można pozwolić automatowi generować większą liczbę punktów zaczepienia bez obaw, że doprowadzi to do rozdęcia kodu. Metody częściowe muszą mieć typ zwrotny void i są niejawnie definiowane jako prywatne. Nie mogą zawierać parametrów out.

## Rozszerzone metody częściowe

**Rozszerzone metody częściowe** (z C# 9) są przeznaczone do użytku w przypadku odwrotnego generowania kodu, gdy programista definiuje uchwyt implementowane przez generator kodu. Znajdują np. zastosowanie w generatorach kodu źródłowego kompilatora Roslyn, które przyjmują zestaw i na jego podstawie automatycznie generują fragmenty kodu.

Deklaracja metody częściowej jest **rozszerzona**, gdy jest opatrzona modyfikatorem dostępu:

```
public partial class Test
{
    public partial void M1(); // rozszerzona metoda częściowa
    private partial void M2(); // rozszerzona metoda częściowa
}
```

Obecność modyfikatora dostępu nie tylko zmienia dostępność, ale dodatkowo nakazuje kompilatorowi odmiennie traktować daną deklarację.

Rozszerzone metody częściowe *muszą* mieć implementacje, tzn. nie znikają nigdzie, jeśli nie zostaną zaimplementowane. W tym przykładzie zarówno M1, jak i M2 muszą mieć implementacje, ponieważ obie są opatrzone modyfikatorem dostępu (public i private).

Ze względu na swoje właściwości rozszerzone metody częściowe mogą zwracać dowolny typ i mogą zawierać parametry out:

```
public partial class Test
{
    public partial bool IsValid (string identifier);
    internal partial bool TryParse (string number, out int result);
}
```

## Operator nameof

Operator nameof zwraca nazwę symbolu (typu, składowej, zmiennej itd.) w postaci łańcucha:

```
int count = 123;
string name = nameof (count); // nazwa to "count"
```

Zaletą tego operatora w porównaniu z podaniem po prostu łańcucha jest statyczna kontrola typów. Takie narzędzia, jak np. Visual Studio rozpoznają odwołanie do symbolu, więc jeśli zmienimy interesujący nas symbol, to zmienią się także wszystkie odwołania do niego.

Jeśli trzeba określić nazwę składowej typu, np. pola lub własności, należy dodać nazwę tego typu. Ta technika działa zarówno dla składowych statycznych, jak i dla egzemplarza:

```
string name = nameof (StringBuilder.Length);
```

Wartością tego wyrażenia jest łańcuch "Length". Aby został zwrócony łańcuch "StringBuilder.Length", należałoby napisać:

```
nameof (StringBuilder) + "." + nameof (StringBuilder.Length);
```

## Dziedziczenie

Klasa może **dziedziczyć** zawartość innej klasy, aby ją rozszerzyć lub dostosować do indywidualnych potrzeb. Dziedziczenie umożliwia wielokrotne wykorzystanie funkcjonalności klasy, dzięki czemu nie trzeba za każdym razem pisać wszystkiego od nowa. Klasa może dziedziczyć tylko po jednej innej klasie, ale sama może być wykorzystywana w tej roli przez wiele innych klas. W ten sposób powstaje hierarchia klas. Poniżej znajduje się przykład definicji klasy Asset:

```
public class Asset
{
    public string Name;
}
```

Następnie definiujemy klasy Stock i House dziedziczące po klasie Asset. Będą one zawierać wszystko to, co klasa Asset, oraz dodatkowe własne składowe:

```
public class Stock : Asset // dziedziczy po Asset
{
    public long SharesOwned;
}

public class House : Asset // dziedziczy po Asset
{
    public decimal Mortgage;
}
```



Oto przykład użycia tych klas:

```
Stock msft = new Stock { Name="MSFT",  
                        SharesOwned=1000 };  
Console.WriteLine (msft.Name);           // MSFT  
Console.WriteLine (msft.SharesOwned);    // 1000  
  
House mansion = new House { Name="Mansion",  
                             Mortgage=250000 };  
Console.WriteLine (mansion.Name);        // Mansion  
Console.WriteLine (mansion.Mortgage);    // 250000
```

**Klasy pochodne** (ang. *derived class*) Stock i House dziedziczą pole Name po klasie bazowej Asset.



Klasy pochodne nazywa się też **podklasami** (ang. *subclass*).

Klasy bazowe nazywa się też **nadklasami** (ang. *superclass*).

## Polimorfizm

Referencje są **polimorficzne**. Oznacza to, że zmienna typu *x* może się odnosić do obiektu typu będącego podklasą klasy *x*. Spójrz np. na poniższą metodę:

```
public static void Display (Asset asset)  
{  
    System.Console.WriteLine (asset.Name);  
}
```

Ta metoda może wyświetlać zarówno wartość obiektu klasy Stock, jak i House, ponieważ te klasy są podklasami klasy Asset:

```
Stock msft = new Stock ... ;  
House mansion = new House ... ;  
  
Display (msft);  
Display (mansion);
```

Działanie polimorfizmu opiera się na fakcie, że podklasy (Stock i House) mają wszystkie właściwości klasy bazowej (Asset). Ale twierdzenie odwrotne nie jest prawdziwe. Gdyby metoda Display przyjmowała obiekty klasy House, nie można by było do niej przekazywać obiektów klasy Asset:

```
Display (new Asset()); // błąd kompilacji  
public static void Display (House house) // nie przyjmuje jako argumentu obiektów klasy Asset  
{  
    System.Console.WriteLine (house.Mortgage);  
}
```

## Rzutowanie i konwertowanie referencji

Referencja do obiektu może być:

- niejawnie **rzutowana w górę** do referencji typu klasy bazowej;
- jawnie **rzutowana w dół** do referencji typu podklasy.

Rzutowanie w górę i w dół między zgodnymi typami referencyjnymi powoduje **konwersję referencji**: tworzona jest (logicznie) nowa referencja wskazująca *ten sam* obiekt. Rzutowanie w górę zawsze się udaje. Natomiast rzutowanie w dół udaje się tylko wtedy, gdy obiekt ma odpowiedni typ.

## Rzutowanie w górę

Operacja rzutowania w górę powoduje utworzenie referencji typu klasy bazowej z referencji typu podklasy. Na przykład:

```
Stock msft = new Stock();  
Asset a = msft; // rzutowanie w górę
```

Po operacji rzutowania zmienna `a` odwołuje się do tego samego obiektu klasy `Stock` co `msft`. Sam wskazywany obiekt nie jest w żaden sposób zmieniany:

```
Console.WriteLine (a == msft); // prawda
```

Choć zmienne `a` i `msft` odnoszą się do tego samego obiektu, zmienna `a` ma do niego bardziej ograniczony dostęp:

```
Console.WriteLine (a.Name); // OK  
Console.WriteLine (a.SharesOwned); // Błąd kompilacji
```

Drugi z tych wierszy kodu spowoduje błąd kompilacji, ponieważ zmienna `a` jest typu `Asset`, mimo że wskazuje obiekt typu `Stock`. Aby uzyskać dostęp do pola `SharesOwned` tego obiektu, należy dokonać **rzutowania w dół** z typu `Asset` na `Stock`.

## Rzutowanie w dół

Operacja rzutowania w dół powoduje utworzenie referencji typu podklasy z referencji typu klasy bazowej. Na przykład:

```
Stock msft = new Stock();  
Asset a = msft; // rzutowanie w górę  
Stock s = (Stock)a; // rzutowanie w dół  
Console.WriteLine (s.SharesOwned); // nie ma błędu  
Console.WriteLine (s == a); // prawda  
Console.WriteLine (s == msft); // prawda
```

Podobnie jak w rzutowaniu w górę, w tym przypadku operacji również poddawane są tylko referencje, a sam obiekt pozostaje bez zmian. Rzutowanie w dół musi być wykonywane jawnie, ponieważ istnieje ryzyko niepowodzenia operacji w czasie działania programu:

```
House h = new House();  
Asset a = h; // rzutowanie w górę zawsze się udaje  
Stock s = (Stock)a; // rzutowanie w dół się nie uda: a nie jest typu Stock
```

Nieudana próba wykonania rzutowania w dół jest zgłaszana w formie wyjątku `InvalidCastException`. Jest to przykład działania mechanizmu **kontroli typów podczas działania programu** (rozwijamy ten temat w sekcji „Statyczna i dynamiczna kontrola typów”).

## Operator as

Operator `as` wykonuje rzutowanie w dół, ale w razie niepowodzenia operacji zamiast zgłaszać wyjątek, zwraca wartość `null`:

```
Asset a = new Asset();  
Stock s = a as Stock; // s ma wartość null; nie zostanie zgłoszony wyjątek
```

Po wykonaniu rzutowania można sprawdzić, czy wynik operacji to `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```



Gdyby nie ten test, lepszym rozwiązaniem byłoby zwykłe rzutowanie, ponieważ w razie niepowodzenia zostalibyśmy poinformowani o porażce za pomocą wyjątku. Porównaj np. dwa poniższe wiersze kodu:

```
long shares = ((Stock)a).SharesOwned; // technika 1  
long shares = (a as Stock).SharesOwned; // technika 2
```

Jeśli zmienna `a` nie jest typu `Stock`, to pierwszy wiersz spowoduje wyjątek `InvalidCastException`, który wyraźnie wskazuje, co poszło nie tak. Drugi wiersz spowoduje wyjątek `NullReferenceException`, który już nie jest jednoznaczny. Czy zmienna `a` nie była typu `Stock`, czy miała wartość `null`?

Innymi słowy: używając operatora rzutowania, „mówimy” do kompilatora, że jesteśmy *pewni* typu wartości, więc jeśli jest ona inna, w kodzie jest błąd, więc należy zgłosić wyjątek! Natomiast używając operatora `as`, stwierdzamy, że nie jesteśmy pewni co do typu, i chcemy wykonać różne ścieżki kodu w zależności od wyniku testu w czasie działania programu.

Operator `as` nie wykonuje **konwersji niestandardowych** (zob. sekcję „Przeciążanie operatorów” w rozdziale 4.) ani numerycznych:

```
long x = 3 as long; // błąd kompilacji
```



Operatory `as` i rzutowania wykonują także rzutowanie w górę, chociaż niewiele z tego pożytku, ponieważ tego rodzaju konwersje są wykonywane niejawnie.

## Operator is

Operator `is` sprawdza, czy zmienna pasuje do określonego **wzorca**. C# obsługuje kilka rodzajów wzorców, z których najważniejszy jest **wzorec typu**, polegający na tym, że po nazwie typu stawia się słowo kluczowe `is`.

W tym kontekście operator `is` sprawdza, czy konwersja referencyjna by się udała. Innymi słowy, sprawdza, czy obiekt pochodzi od określonej klasy (lub implementuje określony interfejs). Często jest używany przed rzutowaniem w dół.

```
if (a is Stock)  
    Console.WriteLine (((Stock)a).SharesOwned);
```

Operator `is` zwraca prawdę także w przypadku przewidywanego powodzenia operacji *konwersji rozpakowującej* (zob. podrozdział „Typ object”). Nie działa natomiast w odniesieniu do konwersji niestandardowych oraz numerycznych.



Operator `is` współpracuje z wieloma innymi wzorcami wprowadzonymi w paru ostatnich wersjach C#. Więcej informacji znajduje się w rozdziale 4., w punkcie „Wzorce”.

## Wprowadzanie zmiennej wzorcowej

Zmienne można tworzyć za pomocą operatora `is`:

```
if (a is Stock s)
    Console.WriteLine (s.SharesOwned);
```

Ten sposób jest równoważny z tym:

```
Stock s;
if (a is Stock)
{
    s = (Stock) a;
    Console.WriteLine (s.SharesOwned);
}
```

Tworzona w ten sposób zmienna jest od razu dostępna do użycia, więc poniższy kod jest poprawny:

```
if (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine ("Bogaty");
```

Ponadto zmienna taka pozostaje dostępna także poza wyrażeniem `is`, dzięki czemu można pisać taki kod:

```
if (a is Stock s && s.SharesOwned > 100000)
    Console.WriteLine ("Wealthy");
else
    s = new Stock(); // s jest dostępna
    Console.WriteLine (s.SharesOwned); // nadal dostępna
```

## Wirtualne składowe funkcyjne

Funkcja oznaczona słowem kluczowym `virtual` może być *przesłonięta* przez podklasę, w której potrzebna jest jej specjalna implementacja. Wirtualne mogą być metody, własności, indeksatory oraz zdarzenia:

```
public class Asset
{
    public string Name;
    public virtual decimal Liability => 0; // własność wyrażeniowa
}
```

(Zapis `Liability => 0` to skrót od `{ get { return 0; } }`). Więcej o tej składni napisaliśmy w sekcji „Własności wyrażeniowe”.

Aby w podklasie przesłonić metodę wirtualną, należy użyć modyfikatora `override`:

```
public class Stock : Asset
{
    public long SharesOwned;
}
```

```
public class House : Asset
{
    public decimal Mortgage;
    public override decimal Liability => Mortgage;
}
```

Domyślnie własność `Liability` w klasie `Asset` ma wartość 0. W klasie `Stock` nie trzeba tego zmieniać. Natomiast w klasie `House` potrzebna jest specjalna własność `Liability` zwracająca wartość składowej `Mortgage`:

```
House mansion = new House { Name="McMansion", Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability); // 250000
Console.WriteLine (a.Liability);      // 250000
```

Sygnatury, typy zwrotne oraz dostępność metody wirtualnej i jej przesłonięcia muszą być identyczne. Przesłonięta metoda może wywoływać swoją implementację z klasy bazowej za pomocą słowa kluczowego `base` (szerzej piszemy o nim w sekcji „Słowo kluczowe `base`”).



Wywoływanie metod wirtualnych w konstruktorze może być niebezpieczne, ponieważ programiści piszący podklasy często podczas przesłaniania metody nie wiedzą, że pracują z częściowo zainicjalizowanym obiektem. Innymi słowy: metoda przesłaniająca może korzystać z metod lub własności wykorzystujących pola, które nie zostały jeszcze zainicjalizowane przez konstruktor.

## Kowariantne typy zwrotne

Od C# 9 metodę (lub metodę dostępową własności `get`) można przesłonić tak, aby zwracała obiekt *węższego* typu (podklasy). Na przykład:

```
public class Asset
{
    public string Name;
    public virtual Asset Clone() => new Asset { Name = Name };
}

public class House : Asset
{
    public decimal Mortgage;
    public override House Clone() => new House
        { Name = Name, Mortgage = Mortgage };
}
```

Jest to dopuszczalne, ponieważ nie łamie umowy mówiącej, że metoda `Clone` musi zwracać typ `Asset`: zwraca obiekt typu `House`, który także jest typu `Asset` (i nie tylko).

Przed C# 9 konieczne byłoby przesłanianie metod z identycznym typem zwrotnym:

```
public override Asset Clone() => new House { ... }
```

To także spełni swoje zadanie, ponieważ przesłonięta metoda `Clone` tworzy obiekt typu `House`, a nie `Asset`. Aby jednak móc traktować zwrócony obiekt jako obiekt typu `House`, należy wykonać rzutowanie w dół:

```
House mansion1 = new House { Name="McMansion", Mortgage=250000 };
House mansion2 = (House) mansion1.Clone();
```

## Abstrakcyjne klasy i składowe

Jeśli klasa jest **abstrakcyjna**, nie można tworzyć jej obiektów. Możliwe jest tylko tworzenie obiektów jej konkretnych **podklas**.

Klasy abstrakcyjne mogą zawierać definicje **abstrakcyjnych składowych**. Są one podobne do składowych wirtualnych, tylko nie mają domyślnej implementacji. Musi ona zostać podana w podklasie, chyba że podklasa również jest abstrakcyjna:

```
public abstract class Asset
{
    // zwróć uwagę na pustą implementację
    public abstract decimal NetValue { get; }
}
public class Stock : Asset
{
    public long SharesOwned;
    public decimal CurrentPrice;
    // przesłonięcie, jak metody wirtualnej
    public override decimal NetValue => CurrentPrice * SharesOwned;
}
```

## Ukrywanie odziedziczonych składowych

Klasa bazowa i podklasa mogą definiować identyczne składowe. Na przykład:

```
public class A { public int Counter = 1; }
public class B : A { public int Counter = 2; }
```

Pole Counter w klasie B *chowa* pole Counter z klasy A. Często dzieje się to przypadkowo, gdy ktoś doda składową do typu bazowego *po* tym, jak dodano identyczną składową do podtypu. Dlatego kompilator generuje ostrzeżenie i rozwiązuje problem niejednoznaczności w następujący sposób:

- Referencje do A (w czasie kompilacji) wiążą się z A.Counter.
- Referencje do B (w czasie kompilacji) wiążą się z B.Counter.

Czasami jednak trzeba celowo schować składową. W takim przypadku można zastosować modyfikator `new` do tej składowej w podklasie. *Jedyną funkcją tego modyfikatora jest wyłączenie ostrzeżenia kompilatora, które w innym przypadku zostałyby zgłoszone:*

```
public class A { public int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

Modyfikator `new` przekazuje kompilatorowi — i innym programistom — informację, że duplikat składowej nie znalazł się tu przypadkowo.



W języku C# słowo kluczowe `new` ma różne znaczenia w zależności od kontekstu. W szczególności nie należy mylić operatora `new` z modyfikatorem składowych `new`.

## Słowa kluczowe new i override

Spójrz na poniższą hierarchię klas:

```
public class BaseClass
{
    public virtual void Foo() { Console.WriteLine ("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
    public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
}

public class Hider : BaseClass
{
    public new void Foo() { Console.WriteLine ("Hider.Foo"); }
}
```

Poniższy kod ilustruje różnice w działaniu między klasami Overrider i Hider:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();           // Overrider.Foo
b1.Foo();             // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();              // Hider.Foo
b2.Foo();             // BaseClass.Foo
```

## Pieczętowanie funkcji i klas

Przesłonięta składowa funkcyjna może **zapięczętować** swoją implementację za pomocą słowa kluczowego sealed, aby uniemożliwić jej dalsze przesłanianie w kolejnych podklasach. W poprzednim przykładzie wirtualnej funkcji składowej mogliśmy zapięczętować implementację składowej Liability w klasie House, przez co klasa pochodna klasy House nie mogłaby już jej przesłonić:

```
public sealed override decimal Liability { get { return Mortgage; } }
```

Można też zapięczętować całą klasę, aby uniemożliwić tworzenie jej egzemplarzy. Częściej pieczętuje się właśnie całe klasy niż pojedyncze składowe funkcyjne.

Choć można zapięczętować funkcję składową klasy, aby uniemożliwić jej przesłonięcie, nie chroni to jej przed *chowaniem*.

## Słowo kluczowe base

Słowo kluczowe base jest podobne do słowa kluczowego this. Ma dwa zastosowania:

- umożliwia dostęp do przesłoniętej składowej funkcyjnej w nadklasie;
- umożliwia wywoływanie konstruktora klasy bazowej (zob. następną sekcję).

W poniższym przykładzie w klasie House użyto słowa kluczowego base w celu uzyskania dostępu do implementacji składowej Liability z klasy Asset:

```
public class House : Asset
{
    ...
    public override decimal Liability => base.Liability + Mortgage;
}
```

Za pomocą słowa kluczowego base uzyskujemy dostęp do składowej Liability klasy Asset **w sposób niewirtualny**, tzn. zawsze będziemy otrzymywać wersję tej własności z klasy Asset, niezależnie od rzeczywistego typu egzemplarza w systemie wykonawczym.

Metoda ta zadziałałaby także wtedy, gdy składowa Liability byłaby *schowana*, a nie *przesłonięta*. (Dostęp do schowanych składowych można także uzyskać przez wykonanie rzutowania na klasę bazową przed wywołaniem funkcji).

## Konstruktory i dziedziczenie

Podklasa musi zawierać własne deklaracje konstruktorów. Konstruktory klasy bazowej są *dostępne* w klasie pochodnej, ale nie są automatycznie *dziedziczone*. Na przykład przy takich definicjach klasy BaseClass i podklasy Subclass:

```
public class Baseclass
{
    public int X;
    public Baseclass () { }
    public Baseclass (int x) { this.X = x; }
}

public class Subclass : Baseclass { }
```

poniższy kod jest nieprawidłowy:

```
Subclass s = new Subclass (123);
```

W podklasie Subclass programista musi ponownie zdefiniować wszystkie konstruktory, które mają być przez nią udostępniane. Przy okazji można korzystać z konstruktorów klasy bazowej za pomocą słowa kluczowego base:

```
public class Subclass : Baseclass
{
    public Subclass (int x) : base (x) { }
```

Słowo kluczowe base działa prawie jak this, tylko wywołuje konstruktor z klasy bazowej.

Konstruktory klasy bazowej są zawsze wywoływane na początku. Dzięki temu wiadomo, że obiekt klasy *bazowej* zostanie zainicjalizowany, zanim nastąpi inicjalizacja typu *specjalnego*.

## Niejawne wywoływanie bezparametrowego konstruktora klasy bazowej

Jeżeli konstruktor podklasy nie zawiera słowa kluczowego base, to następuje niejawne wywołanie *bezparametrowego* konstruktora typu bazowego:



```

public class BaseClass
{
    public int X;
    public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
    public Subclass() { Console.WriteLine (X); } //1
}

```

Jeżeli bezparametrowy konstruktor klasy bazowej jest nieosiągalny, konstruktory podklas muszą zawierać słowo kluczowe `base`. To oznacza, że klasa bazowa z (tylko) konstruktorem wieloparametrowym obciąża podklasę obowiązkiem jego wywołania:

```

class Baseclass
{
    public Baseclass (int x, int y, int z, string s, DateTime d) { ... }
}

public class Subclass : Baseclass
{
    public Subclass (int x, int y, int z, string s, DateTime d)
        : base (x, y, z, s, d) { ... }
}

```

## Składowe wymagane (C# 11)

Wymóg wywołania przez podklasę konstruktora klasy bazowej może być problematyczny w przypadku rozbudowanych hierarchii klas, w których znajduje się wiele konstruktorów z wieloma parametrami. Czasami najlepszym wyjściem jest całkowite unikanie konstruktorów i poleganie wyłącznie na inicjalizatorach obiektów w zakresie ustawiania pól lub własności podczas tworzenia obiektu. Pomóc w tym może dodanie do pola lub własności słowa kluczowego `required` (od C# 11):

```

public class Asset
{
    public required string Name;
}

```

Wymagana składowa *musi* otrzymać wartość od inicjalizatora obiektu:

```

Asset a1 = new Asset { Name="House" }; // OK
Asset a2 = new Asset(); // Błąd: nie przejdzie kompilacji!

```

Jeśli dodatkowo chcesz napisać konstruktor, to możesz zastosować atrybut `[SetsRequiredMembers]`, aby obejść w nim wymóg nadania składowej wartości:

```

public class Asset
{
    public required string Name;

    public Asset() { }

    [System.Diagnostics.CodeAnalysis.SetsRequiredMembers]
    public Asset (string n) => Name = n;
}

```

Teraz użytkownicy mogą wygodnie korzystać z tego konstruktora bez jakichkolwiek kompromisów:

```
Asset a1 = new Asset { Name = "House" }; // OK
Asset a2 = new Asset ("House");         // OK
Asset a3 = new Asset();                 // Błąd!
```

Zwróć uwagę, że zdefiniowaliśmy też konstruktor bezparametrowy (do użytku z inicjalizatorem obiektów). Jego obecność sprawia także, że podklasy są wolne od obowiązku odtwarzania któregośkolwiek konstruktora. W poniższym przykładzie klasa House nie implementuje konstruktora pomocniczego:

```
public class House : Asset { }           // Brak konstruktora, brak smartwień!
House h1 = new House { Name = "House" }; // OK
House h2 = new House();                 // Błąd
```

## Kolejność inicjalizacji pól

Podczas inicjalizacji obiektu kolejność wykonywania czynności jest następująca:

1. Od podklasy do klasy bazowej:
  - a. inicjalizacja pól,
  - b. obliczenie wartości argumentów konstruktora klasy bazowej.
2. Od klasy bazowej do podklasy:
  - a. wykonanie kodu konstruktorów.

Ilustruje to poniższy przykład kodu:

```
public class B
{
    int x = 1;           // 3. miejsce
    public B (int x)
    {
        ...             // 4. miejsce
    }
}
public class D : B
{
    int y = 1;          // 1. miejsce
    public D (int x)
        : base (x + 1) // 2. miejsce
    {
        ...             // 5. miejsce
    }
}
```

## Dziedziczenie z konstruktorami podstawowymi

Klasy z konstruktorami podstawowymi mogą tworzyć podklasy za pomocą następującej składni:

```
public class Baseclass (int x) { ... }
public class Subclass (int x, int y) : Baseclass (x) { ... }
```

W poniższym przykładzie wywołanie `Baseclass(x)` jest równoznaczne z wywołaniem `base(x)`:

```
public class Subclass : Baseclass
{
```

```
    public Subclass (int x, int y) : base (x) { ... }  
}
```

## Przeciążanie i rozpoznawanie

Dziedziczenie pociąga za sobą ciekawe skutki dla przeciążania metod. Spójrz na dwie poniższe przeciążone metody:

```
static void Foo (Asset a) { }  
static void Foo (House h) { }
```

W przypadku wywołania metody przeciążonej pierwszeństwo ma najbardziej konkretny typ:

```
House h = new House (...);  
Foo(h); // wywoła Foo(House)
```

Wybór metody do wywołania jest dokonywany statycznie (w czasie kompilacji). W poniższym przykładzie zostanie wywołana metoda `Foo(Asset)`, mimo że typem w czasie wykonywania programu jest `House`:

```
Asset a = new House (...);  
Foo(a); // wywoła Foo(Asset)
```



Jeżeli dokonamy rzutowania `Asset` na `dynamic` (rozdział 4.), to wybór przeciążonej wersji metody do wywołania zostanie odłożony do czasu wykonywania programu i zostanie oparty na rzeczywistym typie obiektu:

```
Asset a = new House (...);  
Foo ((dynamic)a); // wywoła Foo(House)
```

## Typ object

Typ `object` (`System.Object`) reprezentuje najwyższą klasę stanowiącą podstawę wszystkich typów. Do typu `object` można rzutować w górę każdy inny typ.

Jak bardzo jest to przydatne, pokażemy na przykładzie *stosu*. Jest to struktura danych zgodna z zasadą LIFO (ang. *last in, first out* — ostatni na wejściu, pierwszy na wyjściu). Każdy stos obsługuje dwie operacje: *push* do wstawiania obiektów na stos i *pop* do usuwania obiektów ze stosu. Oto prosta implementacja takiej struktury, w której można przechowywać maksymalnie 10 obiektów:

```
public class Stack  
{  
    int position;  
    object[] data = new object[10];  
    public void Push (object obj) { data[position++] = obj; }  
    public object Pop() { return data[--position]; }  
}
```

Dzięki temu, że klasa `Stack` pracuje z typem `object`, na stosie możemy umieszczać egzemplarze *dowolnego typu*:

```
Stack stack = new Stack();  
stack.Push ("kiełbasa");  
string s = (string) stack.Pop(); // rzutowanie w dół, więc musi być wykonane jawnie  
Console.WriteLine (s); // kielbasa
```

Typ `object` to typ referencyjny, ponieważ jest typem klasowym. Mimo to można na niego rzutować także typy wartościowe, np. `int`. Jest to cecha języka C# zwana **unifikacją** (ang. *unification*), a poniżej przedstawiono przykład jej wykorzystania:

```
stack.Push(3);
int three = (int) stack.Pop();
```

Gdy programista rzutuje typ wartościowy na obiektowy, CLR musi wykonać specjalne czynności związane z różnicami semantycznymi między typami wartościowymi a referencyjnymi. Proces ten nazywa się **pakowaniem** (ang. *boxing*) i **rozpakowywaniem** (ang. *unboxing*).



W podrozdziale „Typy generyczne” opisujemy, jak ulepszyć naszą klasę `Stack` tak, aby lepiej obsługiwała elementy o takim samym typie.

## Pakowanie i rozpakowywanie

Pakowanie (ang. *boxing*) to proces konwersji egzemplarza typu wartościowego na egzemplarz typu referencyjnego. Typ referencyjny może być klasą `object` lub interfejsem (o interfejsach piszemy w dalszej części rozdziału)<sup>1</sup>. W poniższym przykładzie opakowujemy egzemplarz typu `int` w `object`:

```
int x = 9;
object obj = x; // pakuje int
```

Rozpakowywanie to operacja odwrotna, tzn. polegająca na rzutowaniu obiektu na pierwotny typ wartościowy:

```
int y = (int) obj; // rozpakowanie int
```

Rozpakowywanie można przeprowadzić tylko jako rzutowanie jawne. System wykonawczy sprawdza, czy podany typ wartościowy odpowiada rzeczywistemu typowi obiektowemu, i w razie problemu zgłasza wyjątek `InvalidCastException`. Na przykład poniższy kod spowoduje wyjątek, ponieważ typ `long` nie odpowiada dokładnie typowi `int`:

```
object obj = 9; // typem 9 drogą dedukcji zostanie int
long x = (long) obj; // InvalidCastException
```

Natomiast w tym kodzie nie ma błędu:

```
object obj = 9;
long x = (int) obj;
```

I w tym też nie:

```
object obj = 3.5; // typem 3.5 drogą dedukcji zostanie double
int x = (int) (double) obj; // x ma teraz wartość 3
```

W ostatnim przykładzie operacja `(double)` *rozpakowuje* wartość, a następnie `(int)` wykonuje *konwersję liczbową*.

<sup>1</sup> Typem referencyjnym może też być `System.ValueType` lub `System.Enum` (rozdział 6.).



Operacje pakowania są niezbędne do zapewnienia jednolitego systemu typów. Ale ten system nie jest idealny: w podrozdziale „Typy generyczne” wyjaśniamy, że wariacja w przypadku tablic i typów generycznych pozwala tylko na *konwersje obejmujące typy referencyjne, ale niewymagające opakowywania*:

```
object[] a1 = new string[3]; // poprawne
object[] a2 = new int[3];    // błąd
```

## Semantyka kopiowania w pakowaniu i rozpakowywaniu

Operacja pakowania *kopiuje* egzemplarz typu wartościowego do nowego obiektu, natomiast operacja rozpakowywania *kopiuje* zawartość obiektu z powrotem do egzemplarza typu wartościowego. W poniższym przykładzie zmiana wartości zmiennej i nie powoduje modyfikacji jej wcześniej spakowanej kopii:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed); // 3
```

## Statyczna i dynamiczna kontrola typów

W programach C# kontrola typów jest sprawowana zarówno statycznie (w czasie kompilacji), jak i dynamicznie (przez CLR).

Statyczna kontrola typów umożliwia kompilatorowi sprawdzenie poprawności programu bez uruchamiania tegoż programu. Poniższy wiersz kodu np. nie przejdzie kompilacji, ponieważ kompilator sprawdza typy statycznie:

```
int x = "5";
```

Dynamiczna kontrola typów jest wykonywana przez CLR w przypadku rzutowania w dół przez konwersję referencji lub rozpakowywanie:

```
object y = "5";
int z = (int) y; // błąd wykonawczy, błąd rzutowania w dół
```

Sprawdzanie typów w czasie wykonywania programu jest możliwe dzięki temu, że każdy obiekt na stercie zawiera token określający jego typ. Token ten można pobrać za pomocą metody GetType typu object.

## Metoda GetType i operator typeof

Wszystkie typy w C# w czasie działania programu są reprezentowane przez egzemplarz typu System.Type. Obiekt System.Type można otrzymać na dwa sposoby:

- wywołując metodę GetType na egzemplarzu;
- stosując operator typeof do nazwy typu.

Metoda GetType jest wykonywana w czasie działania programu. Natomiast operator typeof jest wykonywany w czasie kompilacji (jeśli użyte są generyczne parametry typów, rozpoznaje go kompilator JIT).

System.Type zawiera własności do przechowywania takich informacji, jak: nazwa typu, zestaw, typ bazowy itd. Na przykład:

```
Point p = new Point();
Console.WriteLine (p.GetType().Name);           // Point
Console.WriteLine (typeof (Point).Name);       // Point
Console.WriteLine (p.GetType() == typeof(Point)); // True
Console.WriteLine (p.X.GetType().Name);        // Int32
Console.WriteLine (p.Y.GetType().FullName);    // System.Int32
```

```
public class Point { public int X, Y; }
```

Ponadto klasa System.Type zawiera metody stanowiące wejście do modelu refleksji systemu wykonawczego opisanego w rozdziale 18.

## Metoda ToString

Metoda ToString zwraca domyślną tekstową reprezentację egzemplarza typu. Wszystkie typy wbudowane przesłaniają ją własnymi wersjami. Poniżej znajduje się przykład użycia metody ToString typu int:

```
int x = 1;
string s = x.ToString(); // s ma wartość "1"
```

We własnym typie metodę ToString można przesłonić tak:

```
Panda p = new Panda { Name = "Piotrek" };
Console.WriteLine (p); // Piotrek
```

```
public class Panda
{
    public string Name;
    public override string ToString() => Name;
}
```

Jeżeli metoda ToString nie zostanie przesłonięta, to zwraca nazwę typu.



Jeśli wywoła się *przesłoniętą* składową klasy object, np. metodę ToString, bezpośrednio na typie wartościowym, nie powoduje to zapakowania wartości. Pakowanie następuje tylko w przypadku rzutowania:

```
int x = 1;
string s1 = x.ToString(); // wywołanie na niespakowanej wartości
object box = x;
string s2 = box.ToString(); // wywołanie na spakowanej wartości
```

## Lista składowych klasy Object

Oto lista wszystkich składowych klasy Object:

```
public class Object
{
    public Object();

    public extern Type GetType();

    public virtual bool Equals (object obj);
```

```

public static bool Equals (object objA, object objB);
public static bool ReferenceEquals (object objA, object objB);

public virtual int GetHashCode();

public virtual string ToString();

protected virtual void Finalize();
protected extern object MemberwiseClone();
}

```

Metody `Equals`, `ReferenceEquals` i `GetHashCode` opisujemy w podrozdziale „Sprawdzanie równości” w rozdziale 6.

## Struktury

**Struktury** są podobne do klas, ale różnią się od nich następującymi cechami:

- Struktura jest typem wartościowym, podczas gdy klasa jest typem referencyjnym.
- Struktury nie mogą dziedziczyć po innych strukturach (nie licząc faktu, że niejawnie pochodzą od klasy `object`, a dokładniej `System.ValueType`).

Struktura może mieć te same składowe co klasa, z wyjątkiem finalizatora. Dodatkowo, ponieważ nie można tworzyć podstruktur, składowe nie mogą być wirtualne, abstrakcyjne ani chronione.



Przed C# 10 struktury nie mogły definiować inicjalizatorów pól i konstruktorów bezparametrowych. Choć obecnie te warunki poluzowano — głównie ze względu na struktury rekordowe (zob. podrozdział „Rekordy” w rozdziale 4.) — warto dobrze się zastanowić przed zdefiniowaniem tych konstrukcji, ponieważ mogą powodować mylące zachowania, które opisaliśmy w podrozdziale „Semantyka tworzenia struktur”.

Struktury należy użyć, gdy potrzebny jest typ o semantyce wartościowej. Dobrymi przykładami struktur są typy liczbowe, w przypadku których skopiowanie wartości przy przypisaniu jest bardziej naturalne niż referencji. Jako że struktura jest typem wartościowym, nie każdy egzemplarz wymaga utworzenia obiektu na stercie. To pozwala poczynić duże oszczędności, gdy trzeba utworzyć wiele egzemplarzy jednego typu. Na przykład utworzenie tablicy elementów typu wartościowego wymaga tylko jednej alokacji na stercie.

Jako że struktury są typami wartościowymi, egzemplarz nie może być `null`. Domyślna wartość struktury jest pustym egzemplarzem, w którym wszystkie pola są puste (ustawione na wartości domyślne).

## Semantyka tworzenia struktur



Przed C# 11 każdemu polu w strukturze trzeba było jawnie nadać wartość w konstruktorze (lub inicjalizatorze pól). Obecnie ten warunek jest poluzowany.

## Konstruktor domyślny

Oprócz konstruktorów zdefiniowanych przez programistę każda struktura ma dodatkowo bezparametrowy konstruktor domyślny, który wykonuje bitowe zerowanie swoich pól (ustawia je na wartości domyślne):

```
Point p = new Point(); // p.x i p.y będą 0
struct Point { int x, y; }
```

Nawet jeśli programista zdefiniuje własny konstruktor bezparametrowy, ten niejawnie utworzony konstruktor domyślny nadal będzie istniał i będzie dostępny za pośrednictwem słowa kluczowego `default`:

```
Point p1 = new Point(); // p1.x i p1.y będą 1
Point p2 = default;    // p2.x i p2.y będą 0

struct Point
{
    int x = 1;
    int y;
    public Point() => y = 1;
}
```

W tym przykładzie zainicjalizowaliśmy `x` wartością 1 przez inicjalizator pól oraz zainicjalizowaliśmy `y` wartością 1 przez konstruktor bezparametrowy. Jednak mimo użycia słowa kluczowego `default` nadal mogliśmy utworzyć obiekt typu `Point` bez wykonywania obu inicjalizacji. Dostęp do konstruktora domyślnego można uzyskać także na inne sposoby, jak pokazano w poniższym przykładzie:

```
var points = new Point[10]; // Każdy punkt w tablicy będzie (0,0).
var test = new Test(); // test.p będzie (0,0)

class Test { Point p; }
```



Obecność dwóch konstruktorów bezparametrowych może być źródłem pomyłek, co stanowi dobry powód, aby unikać definiowania inicjalizatorów pól i jawnych konstruktorów bezparametrowych w strukturach.

Dobłą strategią podczas pracy ze strukturami jest projektowanie ich w taki sposób, aby ich domyślna wartość reprezentowała prawidłowy stan, co czyni inicjalizację zbędną. Zamiast np. inicjalizować własność w następujący sposób:

```
public string Protocol { get; set; } = "https";
```

można wybrać takie rozwiązanie:

```
struct WebOptions
{
    string protocol;
    public string Protocol { get => protocol ?? "https";
                          set => protocol = value; }
}
```



## Struktury tylko do odczytu i funkcje

Do struktur można dodawać modyfikator `readonly`, powodujący, że wszystkie pola są tylko do odczytu. Pomaga to programiście w wyrażeniu intencji oraz ułatwia kompilatorowi stosowanie optymalizacji:

```
readonly struct Point
{
    public readonly int X, Y; // X i Y muszą być tylko do odczytu
}
```

Jeśli potrzebna jest precyzyjniejsza kontrola pól tylko do odczytu, można skorzystać z wprowadzonej w C# 8 możliwości dodawania modyfikatora `readonly` do *funkcji* struktur. Jeśli opatrzona nim funkcja spróbuje zmodyfikować którekolwiek pole, zostanie wygenerowany błąd kompilacji:

```
struct Point
{
    public int X, Y;
    public readonly void ResetX() => X = 0; // Błąd!
}
```

Jeśli funkcja tylko do odczytu wywoła funkcję bez tego ograniczenia, kompilator wygeneruje ostrzeżenie (i defensywnie skopiuje strukturę, aby uniknąć możliwości zmiany).

## Struktury referencyjne



Struktury referencyjne wprowadzono w C# 7.2 jako niszową funkcję przeznaczoną głównie dla struktur `Span<T>` i `ReadOnlySpan<T>`, które opisujemy w rozdziale 23. (oraz wysoce zoptymalizowanej `Utf8JsonReader`, którą opisujemy w rozdziale 11.). Struktury te wspierają technikę mikrooptymalizacji, mającej na celu zmniejszenie liczby alokacji pamięci.

Typy wartościowe, w odróżnieniu od typów referencyjnych, których egzemplarze zawsze są zapisywane na stercie, są przechowywane *na miejscu* (tam, gdzie zadeklarowano zmienną). Jeśli typ wartościowy jest parametrem lub zmienną lokalną, zostaje zapisany na stosie:

```
void SomeMethod()
{
    Point p; // p będzie przechowywany na stosie
}
struct Point { public int X, Y; }
```

Jeśli jednak typ wartościowy zostanie użyty jako pole klasy, miejscem jego przechowywania będzie sterta:

```
class MyClass
{
    Point p; // Przechowywany na stercie, ponieważ egzemplarze klasy MyClass są przechowywane na stercie
}
```

Podobnie na stercie są przechowywane tablice struktur i struktury poddane pakowaniu.

Dodanie modyfikatora `ref` do deklaracji struktury powoduje, że jest ona zawsze przechowywana na stosie. Próba użycia *struktury referencyjnej* w sposób mogący spowodować jej zapisanie na stercie powoduje wygenerowanie błędu kompilacji:

```
var points = new Point [100]; // Błąd: nie przejdzie kompilacji!  
ref struct Point { public int X, Y; }  
class MyClass { Point P; } // Błąd: nie przejdzie kompilacji!  
...
```

Struktury referencyjne wprowadzono głównie ze względu na struktury `Span<T>` i `ReadOnlySpan<T>`. Ich egzemplarze mogą być przechowywane tylko na stosie, więc można ich bezpiecznie używać do opakowywania pamięci alokowanej na stosie.

Struktury referencyjne mogą nie być częścią żadnych konstrukcji C#, które pośrednio lub bezpośrednio wiążą się z możliwością przechowywania na sterwie. Dotyczy to pewnych zaawansowanych konstrukcji C#, które opisaliśmy w rozdziale 4., a konkretnie — wyrażeń lambda, iteratorów i funkcji asynchronicznych (ponieważ wszystkie one tworzą ukryte klasy z polami). Ponadto struktury referencyjne nie mogą występować w strukturach innego typu i nie mogą implementować interfejsów (ponieważ to mogłoby spowodować opakowanie).

## Modyfikatory dostępu

W celu zapewnienia hermetyczności można ograniczyć *dostępność* typu lub składowej typu dla innych typów lub zestawów za pomocą jednego z *modyfikatorów dostępu*, które dodaje się do deklaracji:

`public`

Pełny dostęp do składowej. Jest to domyślny poziom dostępności składowych wyliczeń i interfejsów.

`internal`

Dostępność ograniczona do zestawu lub zestawów zaprzyjaźnionych. Jest to domyślny poziom dostępności typów niezagnieżdżonych.

`private`

Dostępność ograniczona do typu. Jest to domyślny poziom dostępności składowych klas i struktur.

`protected`

Dostępność ograniczona do typu i jego podklas.

`protected internal`

*Suma* modyfikatorów `protected` i `internal`. Składowa zdefiniowana jako `protected internal` jest dostępna na dwa sposoby.

`private protected` (od C# 7.2)

*Przecięcie* funkcjonalności `protected` i `internal`. Składowa zdefiniowana jako `private protected` jest dostępna tylko w zawierającym ją typie lub podklasach *znajdujących się w tym samym zestawie* (co sprawia, że jest mniej dostępna niż składowa mająca tylko modyfikator `protected` lub `internal`).

`file` (od C# 11)

Dostępność tylko w tym samym pliku. Przeznaczony do użytku przez generatory  **kodu źródłowego**  (zobacz podrozdział „Rozszerzone metody częściowe”). Ten modyfikator można stosować tylko do deklaracji typów.

## Przykłady

Klasa `Class2` jest dostępna poza swoim zestawem, a `Class1` nie:

```
class Class1 { } // Class1 to klasa wewnętrzna (ustawienie domyślne)
public class Class2 { }
```

Klasa `ClassB` udostępniła pole `x` innym typom w tym samym zestawie, a `ClassA` nie:

```
class ClassA { int x; } // składowa x jest prywatna (ustawienie domyślne)
class ClassB { internal int x; }
```

Funkcje w klasie `Subclass` mogą wywoływać `Bar`, ale nie `Foo`:

```
class BaseClass
{
    void Foo() {} // Foo jest prywatna (ustawienie domyślne)
    protected void Bar() {}
}

class Subclass : BaseClass
{
    void Test1() { Foo(); } // błąd braku dostępu do Foo
    void Test2() { Bar(); } // OK
}
```

## Zestawy zaprzyjaźnione

Wewnętrzne (`internal`) składowe klasy można udostępniać innym **zaprzyjaźnionym** zestawom przez dodanie atrybutu `System.Runtime.CompilerServices.InternalsVisibleTo` z nazwą zestawu zaprzyjaźnionego:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Jeżeli zestaw zaprzyjaźniony ma silną nazwę (zob. rozdział 17.), należy podać *pełny* 160-bajtowy klucz publiczny:

```
[assembly: InternalsVisibleTo ("StrongFriend, PublicKey=0024f000048c...")]
```

Kompletny klucz publiczny można pobrać z zestawu o silnej nazwie, używając zapytania LINQ (szczegółowy opis technologii LINQ znajduje się w rozdziale 8.):

```
string key = string.Join ("",
    Assembly.GetExecutingAssembly().GetName().GetPublicKey()
    .Select (b => b.ToString ("x2")));
```



Załączony przykład LINQPad proponuje znalezienie i wskazanie zestawu, a następnie kopiuje jego pełny klucz publiczny do schowka.

## Ograniczanie dostępności

Typy ograniczają zakres dostępności zadeklarowanych w nich składowych. Najbardziej powszechnym tego przykładem jest typ wewnętrzny (`internal`) zawierający składowe publiczne (`public`). Na przykład:

```
class C { public void Foo() {} }
```

Domyślny zakres dostępności typu C (`internal`) ogranicza zakres dostępności składowej `Foo` do tego samego poziomu. Powodem zadeklarowania metody `Foo` jako publicznej w takim przypadku może być chęć ułatwienia refaktoryzacji, gdyby w przyszłości typ C miał zostać zmieniony na publiczny.

## Ograniczenia dotyczące modyfikatorów dostępu

Funkcja przesłaniająca funkcję z klasy bazowej musi mieć taki sam zakres dostępności jak funkcja przesłaniana. Na przykład:

```
class BaseClass { protected virtual void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} } // OK
class Subclass2 : BaseClass { public override void Foo() {} } // błąd
```

(Wyjątkiem są metody `protected internal` znajdujące się w innych zestawach — w tym przypadku metoda przesłaniająca musi być tylko chroniona).

Kompilator nie przepuści niezgodnych modyfikatorów dostępu. Na przykład podklasa może mieć węższy zakres dostępności niż klasa bazowa, ale nie szerszy:

```
internal class A {}
public class B : A {} // błąd
```

## Interfejsy

Interfejsy są podobne do klas. Różnią się od nich tym, że nie zawierają stanu (danych), lecz jedynie *określają zachowania*. W związku z tym:

- Interfejs może definiować tylko funkcje. Nie może zawierać pól.
- Składowe interfejsu są **domyślnie abstrakcyjne**. (Od tej reguły są wyjątki, które opisaliśmy w punktach „Domyślne składowe interfejsu” i „Statyczne składowe interfejsu”).
- Klasa (lub struktura) może implementować *wiele* interfejsów. Z drugiej strony klasa może dziedziczyć tylko po *jednej* innej klasie, a struktura nie może dziedziczyć w ogóle (nie licząc tego, że wszystkie struktury pochodzą od `System.ValueType`).

Deklaracja interfejsu wygląda jak deklaracja klasy, tylko zazwyczaj nie zawiera implementacji składowych, ponieważ wszystkie one są domyślnie abstrakcyjne. Ich implementacje zostaną dodane w klasach i strukturach implementujących dany interfejs. Interfejs może zawierać tylko funkcje, własności, zdarzenia i indeksatory, czyli te same składowe co klasy abstrakcyjne.

Oto definicja interfejsu `IEnumerator` z przestrzeni nazw `System.Collections`:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Składowe interfejsu są zawsze publiczne i nie można tego zmienić za pomocą żadnego modyfikatora dostępu. Implementacja interfejsu polega na zdefiniowaniu publicznych implementacji wszystkich jego składowych:

```
internal class Countdown : IEnumerator
{
    int count = 11;
    public bool MoveNext() => count-- > 0;
    public object Current => count;
    public void Reset() { throw new NotSupportedException(); }
}
```

Obiekt można niejawnie rzutować na typ dowolnego interfejsu, który implementuje. Na przykład:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
    Console.Write (e.Current); // 109876543210
```



Mimo że Countdown jest klasą wewnętrzną, jej składowe stanowiące implementację interfejsu IEnumerator można wywoływać publicznie przez dokonanie rzutowania egzemplarza tej klasy na typ IEnumerator. Na przykład gdyby typ publiczny znajdujący się w tym samym zestawie zawierał następującą definicję metody:

```
public static class Util
{
    public static object GetCountDown() => new Countdown();
}
```

to w innym zestawie można by było napisać taki kod:

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

Gdyby sam interfejs IEnumerator także był zdefiniowany jako wewnętrzny, nie byłoby to możliwe.

## Rozszerzanie interfejsu

Interfejsy mogą być tworzone na podstawie innych interfejsów. Na przykład:

```
public interface IUndoable { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

Interfejs IRedoable „dziedziczy” wszystkie składowe z interfejsu IUndoable. Innymi słowy: typy implementujące IRedoable muszą implementować też składowe interfejsu IUndoable.

## Jawna implementacja interfejsu

Implementacja wielu interfejsów może skutkować kolizją sygnatur składowych. W takim przypadku rozwiązaniem problemu jest **jawna implementacja** składowej. Spójrz na poniższy przykład:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
    public void Foo()
    {
        Console.WriteLine ("Implementacja składowej I1.Foo w klasie Widget.");
    }
    int I2.Foo()
    {
```

```

        Console.WriteLine ("Implementacja składowej I2.Foo w klasie Widget.");
        return 42;
    }
}

```

Ze względu na konflikt sygnatur metod Foo z interfejsów I1 i I2 w klasie Widget w sposób bezpośredni zdefiniowano implementację tej metody z interfejsu I2. Dzięki temu metody te mogą współistnieć w jednej klasie. Jedynym sposobem na wywołanie jawnie zaimplementowanej składowej jest dokonanie rzutowania na typ jej interfejsu:

```

Widget w = new Widget();
w.Foo(); // Implementacja metody I1.Foo z klasy Widget.
((I1)w).Foo(); // Implementacja metody I1.Foo z klasy Widget.
((I2)w).Foo(); // Implementacja metody I2.Foo z klasy Widget.

```

Innym powodem, dla którego może być konieczne jawne implementowanie składowych interfejsu, jest chęć ukrycia tych z nich, które są wysoce specjalistyczne, aby nie przeszkadzały w normalnym korzystaniu z typu. Na przykład typy implementujące interfejs `ISerializable` z reguły nie powinny afiszować się ze swoimi składowymi z tego interfejsu, chyba że programista dokona rzutowania na jego typ.

## Wirtualna implementacja składowych interfejsu

Niejawnie zaimplementowana składowa interfejsu jest domyślnie zabezpieczona. W klasie bazowej musi więc zostać oznaczona modyfikatorem dostępu `virtual` lub `abstract`, aby można ją było przesłonić. Na przykład:

```

public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    public virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox
{
    public override void Undo() => Console.WriteLine ("RichTextBox.Undo");
}

```

Dla wywołania składowej interfejsu przez klasę bazową lub interfejs wybierana jest implementacja z podklasy:

```

RichTextBox r = new RichTextBox();
r.Undo(); // RichTextBox.Undo
((IUndoable)r).Undo(); // RichTextBox.Undo
((TextBox)r).Undo(); // RichTextBox.Undo

```

Jawnie zaimplementowana składowa interfejsu nie może być oznaczona modyfikatorem `virtual` ani nie można jej przesłonić w normalny sposób. Istnieje jednak możliwość jej **reimplementowania**.

## Reimplementacja interfejsu w podklasie

W podklasie można reimplementować każdą składową interfejsu, która została uprzednio zaimplementowana w klasie bazowej. Reimplementowana składowa przechwytuje implementację składowej (przy wywołaniu przez interfejs) i działa niezależnie od tego, czy składowa ta w klasie

bazowej jest wirtualna, czy nie. Ta technika funkcjonuje także bez względu na to, czy istniejąca implementacja jest jawna, czy niejawną — choć najlepiej działa w tym drugim przypadku, co pokazujemy poniżej.

W poniższym przykładzie klasa `TextBox` jawnie implementuje metodę `IUndoable.Undo`, więc nie można jej oznaczyć jako wirtualnej. Aby „przesłonić” tę metodę w klasie `RichTextBox`, należy dokonać jej reimplementacji:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
    void IUndoable.Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox, IUndoable
{
    public void Undo() => Console.WriteLine ("RichTextBox.Undo");
}
```

Wywołanie reimplementowanej składowej przez interfejs powoduje wywołanie implementacji z podklasy:

```
RichTextBox r = new RichTextBox();
r.Undo(); // RichTextBox.Undo Przypadek 1
((IUndoable)r).Undo(); // RichTextBox.Undo Przypadek 2
```

Teraz załóżmy, że definicja klasy `RichTextBox` pozostaje bez zmian, a klasa `TextBox` zawiera *niejawną* implementację metody `Undo`:

```
public class TextBox : IUndoable
{
    public void Undo() => Console.WriteLine ("TextBox.Undo");
}
```

W ten sposób zyskalibyśmy jeszcze jeden sposób wywoływania metody `Undo`, który „załamałby” system, jak pokazano w przypadku 3.:

```
RichTextBox r = new RichTextBox();
r.Undo(); // RichTextBox.Undo Przypadek 1
((IUndoable)r).Undo(); // RichTextBox.Undo Przypadek 2
((TextBox)r).Undo(); // TextBox.Undo Przypadek 3.
```

Przypadek 3. pokazuje, że przechwytywanie za pomocą reimplementacji jest skuteczne tylko, gdy składowa jest wywoływana przez interfejs, a nie przez klasę bazową. W większości przypadków jest to niepożądane, ponieważ wprowadza niespójność semantyczną. Dlatego reimplementacja jest techniką odpowiednią do przesłaniania *jawnie* zaimplementowanych składowych interfejsów.

## Alternatywy dla reimplementacji

Nawet w przypadku jawnych implementacji składowych reimplementacja interfejsów rodzi problemy z dwóch powodów:

- w podklasie nie ma możliwości wywołania metody z klasy bazowej;
- twórca klasy bazowej może nie przewidzieć, że metoda zostanie reimplementowana, i nie przewidzi ewentualnych skutków tego.

Reimplementacja jest zatem dobrym rozwiązaniem, gdy nie ma innego wyjścia. Jednak lepszym rozwiązaniem jest takie zaprojektowanie klasy bazowej, aby nigdy nie trzeba było stosować reimplementacji. Można to zrobić na dwa sposoby:

- Jeśli składowa jest implementowana niejawnie, można ją oznaczyć modyfikatorem dostępu `virtual`.
- Jeśli składowa jest implementowana jawnie i istnieje możliwość, że w podklasach trzeba będzie przesłonić jakąś logikę, można skorzystać z poniższego wzoru:

```
public class TextBox : IUndoable
{
    void IUndoable.Undo() => Undo(); // wywołuje poniższą metodę
    protected virtual void Undo() => Console.WriteLine ("TextBox.Undo");
}

public class RichTextBox : TextBox
{
    protected override void Undo() => Console.WriteLine("RichTextBox.Undo");
}
```

Jeśli nie są przewidywane żadne podklasy, klasę można oznaczyć modyfikatorem `sealed`, aby całkowicie wykluczyć reimplementację interfejsów.

## Interfejsy i pakowanie

Konwersja struktury na interfejs odbywa się w procesie pakowania (ang. *boxing*). Wywołanie niejawnie zaimplementowanej składowej na strukturze nie powoduje pakowania:

```
interface I { void Foo(); }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo(); // bez pakowania

I i = s; // nastąpi pakowanie przy rzutowaniu na typ interfejsu
i.Foo();
```

## Domyślne składowe interfejsu

W C# 8 wprowadzono możliwość definiowania domyślnej implementacji składowej interfejsu, dzięki czemu programista może wybrać, czy chce ją zaimplementować we własnym zakresie:

```
interface ILogger
{
    void Log (string text) => Console.WriteLine (text);
}
```

Jest to przydatne, kiedy chcemy dodać składową do interfejsu zdefiniowanego w popularnej bibliotece, nie uszkadzając (potencjalnie tysięcy) implementacji.

Domyślne implementacje zawsze są jawne. Jeśli więc klasa implementująca interfejs `ILogger` nie będzie miała definicji metody `Log`, jedynym sposobem na jej wywołanie będzie skorzystanie z pośrednictwa interfejsu:



```
class Logger : ILogger { }
...
((ILogger)new Logger()).Log ("message");
```

To rozwiązuje problem wielokrotnego dziedziczenia implementacji: jeśli ta sama domyślna składowa zostanie dodana do dwóch interfejsów implementowanych przez klasę, nigdy nie będzie wątpliwości, o którą z nich chodzi w danym przypadku.

## Statyczne składowe interfejsu

Interfejs może także zawierać deklaracje składowych statycznych. Istnieją ich dwa rodzaje:

- statyczne niewirtualne składowe interfejsu,
- statyczne wirtualne/abstrakcyjne składowe interfejsu.



W odróżnieniu od składowych **egzemplarza**, statyczne składowe interfejsów są niewirtualne domyślnie. Aby uczynić statyczną składową interfejsu wirtualną, należy dodać do jej deklaracji modyfikatory `static abstract` lub `static virtual`.

### Statyczne niewirtualne składowe interfejsu

Statyczne niewirtualne składowe interfejsu istnieją głównie po to, aby ułatwiać pisanie domyślnych składowych interfejsów. Nie są implementowane przez klasy ani struktury, tylko wykorzystywane bezpośrednio. Statyczne niewirtualne składowe interfejsu, razem z metodami, właściwościami, zdarzeniami i indeksatorami, umożliwiają tworzenie pól, które są zazwyczaj używane w kodzie wewnątrz domyślnej implementacji składowych:

```
interface ILogger
{
    void Log (string text) =>
        Console.WriteLine (Prefix + text);

    static string Prefix = "";
}
```

Statyczne niewirtualne składowe interfejsu domyślnie są publiczne, dzięki czemu dostęp do nich można uzyskać z zewnątrz:

```
ILogger.Prefix = "Dziennik: ";
```

Dostępność tę można ograniczyć, dodając modyfikator dostępności do definicji statycznej składowej interfejsu (np. `private`, `protected` lub `internal`).

Pola egzemplarzy są (wciąż) zabronione. Jest to zgodne z podstawową funkcją interfejsów, które mają definiować *zachowania*, nie *stan*.

### Statyczne wirtualne/abstrakcyjne składowe interfejsów

Statyczne wirtualne/abstrakcyjne składowe interfejsów (od C# 11) pozwalają na **polimorfizm statyczny**, czyli zaawansowaną funkcjonalność, której opis znajduje się w rozdziale 4. Statyczne wirtualne składowe interfejsów oznacza się modyfikatorami `static abstract` lub `static virtual`:

```
interface ITypeDescribable
{
    static abstract string Description { get; }
    static virtual string Category => null;
}
```

Klasa lub struktura implementująca musi implementować statyczne składowe abstrakcyjne oraz może implementować statyczne składowe wirtualne:

```
class CustomerTest : ITypeDescribable
{
    public static string Description => "Testy klienta"; // Obowiązkowe
    public static string Category => "Testy jednostkowe"; // Opcjonalne
}
```

Oprócz metod, własności i zdarzeń wirtualnymi składowymi interfejsów mogą być także operatory i konwersje (zob. podrozdział „Przeciążanie operatorów” w rozdziale 4.). Statyczne wirtualne składowe interfejsów wywołuje się przez ograniczony parametr typu. Demonstrujemy to w rozdziale 4., w podrozdziałach „Polimorfizm statyczny” i „Matematyka generyczna”, po omówieniu typów generycznych w dalszej części tego rozdziału.

## Klasa czy interfejs

Kiedy powinno się napisać klasę, a kiedy interfejs:

- Klas i podklas używaj do reprezentacji typów, które w sposób naturalny mają wspólną część implementacji.
- Interfejsów używaj do reprezentacji typów o niezależnych implementacjach.

Spójrz na poniższe klasy:

```
abstract class Animal {}
abstract class Bird : Animal {}
abstract class Insect : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore : Animal {}
```

*// klasy konkretne:*

```
class Ostrich : Bird {}
class Eagle : Bird, FlyingCreature, Carnivore {} // nieprawidłowe
class Bee : Insect, FlyingCreature {} // nieprawidłowe
class Flea : Insect, Carnivore {} // nieprawidłowe
```

Klasy Eagle, Bee i Flea nie przejdą kompilacji, ponieważ dziedziczenie po kilku klasach jest zabronione. Rozwiązaniem tego problemu jest zamiana niektórych typów na interfejsy. Pozostaje tylko pytanie, których. Zgodnie z przedstawioną ogólną zasadą, możemy powiedzieć, że owady (Insect) wykorzystują wspólną implementację i ptaki (Bird) wykorzystują wspólną implementację, więc te dwie klasy pozostawiamy. Natomiast stworzenia latające (FlyingCreature) mają odrębny mechanizm latania, a mięsożercy (Carnivore) mają własne strategie pożerania innych zwierząt, więc te dwie klasy zamienimy na interfejsy:

```
interface IFlyingCreature {}
interface ICarnivore {}
```

W typowym programie Bird i Insect mogłyby być formantem Windows i kontrolką sieciową, a FlyingCreature i Carnivore mogłyby odpowiadać interfejsom IPrintable i IUndoable.

# Wyliczenia

**Wyliczenie** to specjalny typ wartościowy służący do definiowania grup nazwanych stałych liczbowych. Na przykład:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Tego typu możemy używać następująco:

```
BorderSide topSide = BorderSide.Top;  
bool isTop = (topSide == BorderSide.Top);           // prawda
```

Każda składowa wyliczenia ma przypisaną wartość liczbową. Domyślnie:

- wartości te są typu `int`.
- automatycznie przypisywane są stałe 0, 1, 2... kolejnym deklarowanym składowym wyliczenia.

W razie potrzeby można zmienić `int` na inny typ całkowitoliczbowy:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

Istnieje też możliwość podania własnej wartości dla każdej składowej wyliczenia:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```



Kompilator także umożliwia jawne przypisanie wartości *niektórym* składowym wyliczenia. Składowym niemającym przypisanej wartości przydzielane są kolejne liczby od ostatniej jawnie przypisanej. Zatem powyższy przykład jest równoznaczny z poniższym:

```
public enum BorderSide : byte  
{ Left=1, Right, Top=10, Bottom }
```

## Konwersje wyliczeń

Egzemplarz wyliczenia można przekonwertować na używany w nim typ całkowitoliczbowy i odwrotnie za pomocą jawnego rzutowania:

```
int i = (int) BorderSide.Left;  
BorderSide side = (BorderSide) i;  
bool leftOrRight = (int) side <= 2;
```

Istnieje też możliwość jawnego rzutowania wyliczeń na inne wyliczenia. Powiedzmy, że mamy w programie następujące wyliczenie o nazwie `HorizontalAlignment`:

```
public enum HorizontalAlignment  
{  
    Left = BorderSide.Left,  
    Right = BorderSide.Right,  
    Center  
}
```

Do translacji między typami wyliczeniowymi wykorzystywane są ich wartości całkowitoliczbowe:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;  
// to samo co:  
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

Literał liczbowy 0 w wyrażeniach wyliczeniowych jest przez kompilator traktowany specjalnie i nie wymaga jawnego rzutowania:

```
BorderSide b = 0; // rzutowanie niepotrzebne
if (b == 0) ...
```

Literał 0 jest traktowany specjalnie z dwóch powodów:

- pierwsza składowa wyliczenia często jest używana jako wartość domyślna;
- w **wyliczeniach kombinacyjnych** 0 oznacza „brak flag”.

## Atrybut Flags

Można tworzyć kombinacje składowych wyliczeń. Aby uniknąć nieporozumień, składowe wyliczeń kombinacyjnych muszą mieć jawnie przypisane wartości, najczęściej kolejne potęgi dwójki. Na przykład:

```
[Flags]
enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

lub

```
enum BorderSides { None=0, Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3 }
```

Do pracy z wyliczeniami kombinacyjnymi używa się operatorów bitowych, np. | i &, które operują na wartościach odpowiadających składowym:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;

if ((leftRight & BorderSides.Left) != 0)
    Console.WriteLine ("Zawiera Left"); // Zawiera Left

string formatted = leftRight.ToString(); // "Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight); // prawda

s ^= BorderSides.Right; // Przełącza BorderSides.Right
Console.WriteLine (s); // Left
```

Atrybut `Flags` z zasady dodaje się do wszystkich wyliczeń, których składowe są kombinowalne. Gdyby w deklaracji takiego wyliczenia nie dodano atrybutu `Flags`, to wprawdzie nadal można by było postugiwać się kombinacjami składowych, ale metoda `ToString` dla egzemplarza takiego wyliczenia zwracałaby liczbę zamiast serii nazw.

Z zasady kombinacyjnym typom wyliczeniowym nadaje się nazwy w liczbie mnogiej.

Dla wygody składowe kombinacyjne wyliczenia można dodać w samej deklaracji wyliczenia:

```
[Flags]
enum BorderSides
{
    None=0,
    Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3,
    LeftRight = Left | Right,
    TopBottom = Top | Bottom,
    All      = LeftRight | TopBottom
}
```

## Operatory wyliczeń

Operatory, których można używać z wyliczeniami, to:

```
= == != < > <= >= + - ^ & | ~
+= -= ++ -- sizeof
```

Operatory bitowe, arytmetyczne i porównywania zwracają wynik przetwarzania wartości liczbowych odpowiadających składowym. Dodawać można wyliczenia do typów całkowitych, ale nie dwa wyliczenia.

## Kwestia bezpieczeństwa typów

Spójrz na poniższe wyliczenie:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Jako że wyliczenie można rzutować na wykorzystywany przez nie typ całkowitoliczbowy i odwrotnie, jego rzeczywista wartość może wypaść poza dozwolonymi granicami dla składowej. Na przykład:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);           // 12345
```

Także operatory bitowe i arytmetyczne mogą zwracać nieprawidłowe wartości:

```
BorderSide b = BorderSide.Bottom;
b++;           // nie ma błędu
```

Nieprawidłowa wartość BorderSide spowodowałaby wadliwe działanie poniższego kodu:

```
void Draw (BorderSide side)
{
    if (side == BorderSide.Left) {...}
    else if (side == BorderSide.Right) {...}
    else if (side == BorderSide.Top) {...}
    else {...}           // zakłada BorderSide.Bottom
}
```

Jednym z możliwych rozwiązań byłoby dodanie kolejnej klauzuli else:

```
...
else if (side == BorderSide.Bottom) ...
else throw new ArgumentException ("Nieprawidłowa wartość BorderSide: " + side);
```

Innym rozwiązaniem jest jawne sprawdzenie wartości wyliczenia. Można do tego wykorzystać statyczną metodę Enum.IsDefined:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side)); //fałsz
```

Niestety metoda Enum.IsDefined nie działa z wyliczeniami oflagowanymi. Ale za to poniższa metoda pomocnicza (sztuczka zależna od zachowania metody Enum.ToString()) zwraca true, jeśli podane oflagowane wyliczenie jest poprawne:

```
for (int i = 0; i <= 16; i++)
{
    BorderSides side = (BorderSides)i;
```

```

    Console.WriteLine (IsFlagDefined (side) + " " + side);
}

bool IsFlagDefined (Enum e)
{
    decimal d;
    return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }

```

## Typy zagnieżdżone

**Typ zagnieżdżony** to taki, który jest zadeklarowany w zakresie innego typu. Na przykład:

```

public class TopLevel
{
    public class Nested { } // klasa zagnieżdżona
    public enum Color { Red, Blue, Tan } // zagnieżdżone wyliczenie
}

```

Typy zagnieżdżone mają następujące właściwości:

- mają dostęp do prywatnych składowych typu nadrzędnego i wszystkiego tego, do czego ma dostęp typ nadrzędny;
- mogą być deklarowane przy użyciu wszystkich modyfikatorów dostępu, a nie tylko `public` i `internal`;
- domyślna dostępność typu zagnieżdżonego to `private`, nie `internal`;
- aby użyć typu zagnieżdżonego na zewnątrz zawierającego go typu, należy użyć kwalifikatora w postaci nazwy typu nadrzędnego (tak jak przy dostępie do składowych statycznych).

Aby np. użyć składowej wyliczenia `Color.Red` poza klasą `TopLevel`, należy napisać następującą instrukcję:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Wszystkie typy (klasy, struktury, interfejsy, delegaty i wyliczenia) można zagnieżdżyć w klasie lub strukturze.

Oto przykład użycia prywatnej składowej typu wewnątrz zagnieżdżonego w nim typu:

```

public class TopLevel
{
    static int x;
    class Nested
    {
        static void Foo() { Console.WriteLine (TopLevel.x); }
    }
}

```

Przykład zastosowania modyfikatora dostępu `protected` do typu zagnieżdżonego:

```

public class TopLevel
{
    protected class Nested { }
}

```

```

}

public class SubTopLevel : TopLevel
{
    static void Foo() { new TopLevel.Nested(); }
}

```

Przykład odwołania do zagnieżdżonego typu spoza typu nadrzędnego:

```

public class TopLevel
{
    public class Nested { }
}

class Test
{
    TopLevel.Nested n;
}

```

Typy zagnieżdżone są często wykorzystywane przez kompilator do generowania prywatnych klas przechowujących stan takich konstrukcji, jak iteratory czy metody anonimowe.



Jeżeli jedynym powodem użycia typu zagnieżdżonego jest unikanie namnożenia zbyt wielu typów w przestrzeni nazw, to warto rozważyć możliwość zdefiniowania w zamian zagnieżdżonej przestrzeni nazw. Typ zagnieżdżony powinien być używany, gdy potrzebne są ściślejsze restrykcje dotyczące dostępu lub gdy klasa zagnieżdżona musi mieć dostęp do prywatnych składowych klasy nadrzędnej.

## Typy generyczne

W języku C# występują dwa mechanizmy umożliwiające pisanie kodu działającego z różnymi typami: **dziedziczenie** i **typy generyczne**. Dziedziczenie umożliwia wielokrotne wykorzystanie kodu poprzez typ podstawowy, natomiast typy generyczne służą do tworzenia „szablonów” zawierających typy „zastępcze”. W porównaniu z dziedziczeniem typy generyczne mogą zwiększyć *bezpieczeństwo typowe* oraz *zredukować liczbę potrzebnych operacji rzutowania i pakowania*.



Typy generyczne C# i szablony C++ są do siebie podobne, ale ich działanie jest różne. Różnice te opisujemy niżej, w sekcji „Typy generyczne C# a szablony C++”.

## Typy generyczne

Typ generyczny deklaruje **parametry typów** — typy zastępcze, które powinny zostać zastąpione konkretnymi typami przez użytkownika dostarczającego **argumenty typów**. Poniżej przedstawiony jest generyczny typ `Stack<T>` służący do umieszczania na stosie egzemplarzy typu `T`. Typ ten zawiera deklarację jednego parametru typu o nazwie `T`:

```

public class Stack<T>
{
    int position;
    T[] data = new T[100];
}

```

```

    public void Push (T obj) => data[position++] = obj;
    public T Pop()          => data[--position];
}

```

Typu tego można użyć następująco:

```

var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop(); // x wynosi 10
int y = stack.Pop(); // y wynosi 5

```

W definicji Stack<int> w miejsce parametru typu T podano argument typu int, co spowodowało niejawnie utworzenie typu w locie (synteza następuje w czasie działania programu). Próba wstawienia łańcucha na stos Stack<int> zakończyłaby się jednak błędem kompilacji. Ostateczna definicja typu Stack<int> wyglądałaby następująco (miejsca podmiany zostały oznaczone pogrubieniem, a nazwę klasy usunęliśmy dla uproszczenia):

```

public class ###
{
    int position;
    int[] data = new int[100];
    public void Push (int obj) => data[position++] = obj;
    public int Pop()          => data[--position];
}

```

Technicznie Stack<T> to tzw. **typ otwarty**, a Stack<int> to **typ zamknięty**. W czasie działania programu wszystkie egzemplarze typu generycznego są zamknięte, tzn. typy zastępcze są zamienione na konkretne typy. Oznacza to, że poniższa instrukcja jest nieprawidłowa:

```

var stack = new Stack<T>(); // Niepoprawne: co to jest T?

```

Jedynym wyjątkiem jest sytuacja, gdy powyższy kod znalazłby się w klasie lub metodzie mającej zdefiniowany parametr typu T:

```

public class Stack<T>
{
    ...
    public Stack<T> Clone()
    {
        Stack<T> clone = new Stack<T>(); // poprawne
    }
    ...
}

```

## Po co są typy generyczne

Typy generyczne służą do pisania kodu działającego bez zmian z różnymi typami danych. Wyobraź sobie, że potrzebujesz stosu liczb całkowitych, ale nie masz możliwości definiowania typów generycznych. Jednym z rozwiązań w takim przypadku byłoby napisanie osobnej wersji klasy dla każdego potrzebnego typu elementów (np. IntStack, StringStack itd.). Nietrudno sobie wyobrazić, jak wiele byłoby wtedy duplikatów kodu. Innym wyjściem mogłoby być napisanie stosu generalizowanego przez użycie typu object jako typu elementów:



```

public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj) => data[position++] = obj;
    public object Pop()           => data[--position];
}

```

Ale taka klasa działałaby gorzej od zakodowanej na sztywno klasy `IntStack` przeznaczonej specjalnie do zapisywania na stosie liczb całkowitych. Klasa `ObjectStack` wymuszałaby stosowanie pakowania i rzutowania w dół, a tych działań nie można by było kontrolować pod kątem typów w czasie kompilacji:

```

// powiedzmy, że chcemy zapisywać tylko liczby całkowite:
ObjectStack stack = new ObjectStack();

stack.Push ("s");           // Nieprawidłowy typ, ale nie ma błędu!
int i = (int)stack.Pop();   // rzutowanie w dół — błąd wykonawczy

```

Potrzebna jest nam ogólna implementacja stosu, która będzie działać ze wszystkimi typami elementów, oraz możliwość łatwej specjalizacji stosu dla konkretnego typu elementów w celu zwiększenia bezpieczeństwa typowego i pozbycia się rzutowania oraz pakowania. Te wymagania spełniają typy generyczne, w których można przekazywać typ elementów jako parametr. Klasa `Stack<T>` ma zalety zarówno klasy `ObjectStack`, jak i `IntStack`. Podobnie jak `ObjectStack`, klasę `Stack<T>` wystarczy napisać raz, aby móc jej używać *uniwersalnie* z wszystkimi typami. Orz podobnie jak `IntStack`, klasa `Stack<T>` jest *wyspecjalizowana* pod kątem jednego konkretnego typu — piękne jest to, że ten typ to `T`, który można podmieniać w locie.



Klasa `ObjectStack` jest pod względem funkcjonalnym równoważna klasie `Stack<object>`.

## Metody generyczne

Metoda generyczna zawiera deklaracje parametrów typów w sygnaturze.

Za pomocą metod generycznych można zaimplementować w sposób uniwersalny wiele podstawowych algorytmów. Poniżej znajduje się metoda generyczna zamieniająca wartościami dwie zmienne dowolnego typu `T`:

```

static void Swap<T> (ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}

```

Oto przykład użycia metody `Swap<T>`:

```

int x = 5;
int y = 10;
Swap (ref x, ref y);

```

Zasadniczo nie ma obowiązku przekazywania argumentów typów do metody generycznej, ponieważ kompilator w razie potrzeby automatycznie je wydedukuje. W razie wątpliwości metodę generyczną można wywołać z argumentami typów w następujący sposób:

```
Swap<int> (ref x, ref y);
```

Metoda znajdująca się w *typie* generycznym nie jest automatycznie generyczna, chyba że *wprowadza nowe* parametry typów (umieszczone w nawiasie ostrym). Metoda Pop w naszym generycznym stosie wykorzystuje tylko istniejący parametr zawierającego ją typu, więc nie jest metodą generyczną.

Metody i typy to jedyne konstrukcje, które mogą wprowadzać parametry typów. Własności, indeksatory, zdarzenia, pola, konstruktory, operatory itd. nie mogą ich deklarować, ale mogą wykorzystywać już zadeklarowane przez typ nadrzędny. Na przykład w generycznym stosie, który zdefiniowaliśmy wcześniej, możemy napisać indeksator zwracający ogólny element:

```
public T this [int index] => data [index];
```

Podobnie jest z konstruktorami, które również mogą wykorzystywać istniejące parametry typów, ale nie mogą *wprowadzać nowych*:

```
public Stack<T>() { } // niepoprawne
```

## Deklarowanie parametrów typów

Parametry typów można wprowadzać w deklaracjach klas, struktur, interfejsów, delegatów (opisane w rozdziale 4.) i metod. Inne konstrukcje, takie jak własności, nie mogą *wprowadzać* parametrów typów, ale mogą ich *używać*. Na przykład własność Value wykorzystuje T:

```
public struct Nullable<T>
{
    public T Value { get; }
}
```

Typy i metody generyczne mogą mieć wiele parametrów. Na przykład:

```
class Dictionary<TKey, TValue> { ... }
```

Sposób utworzenia egzemplarza:

```
Dictionary<int,string> myDic = new Dictionary<int,string>();
```

Inna metoda:

```
var myDic = new Dictionary<int,string>();
```

Nazwy generycznych typów i metod można przeciążać, pod warunkiem że zmieni się liczbę parametrów typów. Na przykład poniższe trzy nazwy typów nie kolidują ze sobą:

```
class A {}
class A<T> {}
class A<T1,T2> {}
```



Przyjęło się, że jeśli metoda lub typ generyczny zawiera tylko *jeden* parametr typu, to nadaje się mu nazwę T, oczywiście pod warunkiem że nie ma wątpliwości co do jej znaczenia. Jeśli parametrów typów jest *więcej*, to na początku dodaje się literę T, ale cała nazwa jest bardziej opisowa.

## Operator typeof i niepowiązane typy generyczne

Otwarte typy generyczne nie istnieją w czasie wykonywania programu, ponieważ zostają zamknięte w procesie kompilacji. Ale w czasie wykonywania programu może istnieć *niepowiązany* typ generyczny — jako obiekt `Type`. Jedynym sposobem na określenie niepowiązanego typu generycznego w języku C# jest użycie operatora `typeof`:

```
class A<T> {}
class A<T1,T2> {}
...

Type a1 = typeof (A<>); // typ niepowiązany (zwróć uwagę na brak argumentów).
Type a2 = typeof (A<,>); // dodatkowe argumenty typów oznacza się przecinkami
```

Otwartych typów generycznych używa się w połączeniu z API refleksji (rozdział 18.).

Za pomocą operatora `typeof` można też określić typ zamknięty:

```
Type a3 = typeof (A<int,int>);
```

lub typ otwarty (który w czasie wykonywania programu będzie zamknięty):

```
class B<T> { void X() { Type t = typeof (T); } }
```

## Domyślna wartość generyczna

Za pomocą słowa kluczowego `default` można określić wartość domyślną dla generycznego parametru typu. Dla typów referencyjnych wartością domyślną jest `null`, natomiast dla typów wartościowych wartość domyślna jest wynikiem bitowego wyzerowania pól:

```
static void Zap<T> (T[] array)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = default(T);
}
```

Od C# 7.1 argument typu można ominąć w przypadkach, w których kompilator może go wydedukować. Ostatni wiersz powyższego kodu moglibyśmy zastąpić następującym:

```
array[i] = default;
```

## Ograniczenia typów generycznych

Domyślnie parametr typu można zastąpić dowolnym konkretnym typem. Jeśli jednak są określone szczegółowe wymagania co do argumentów typów, można zastosować specjalne **ograniczenia**. Możliwości w tym zakresie są następujące:

```
where T : klasa-bazowa // ograniczenie dotyczące klasy bazowej
where T : interfejs   // ograniczenie dotyczące interfejsu
where T : klasa       // ograniczenie dotyczące typu referencyjnego
where T : class?     // (zobacz „Typy referencyjne dopuszczające wartość null”)
where T : struktura   // ograniczenie dotyczące typu wartościowego
// (nie dotyczy typów dopuszczających wartość null)
where T : unmanaged  // ograniczenie niekontrolowane
```

```

where T : new()           // ograniczenie dotyczące konstruktora bezparametrowego
where U : T               // ograniczenie dotyczące typu nagiego
where T : notnull        // Typ wartościowy nie dopuszczający wartości null lub od C# 8
                        // typ referencyjny nie dopuszczający wartości null

```

Przedstawiona poniżej klasa `GenericClass<T,U>` zawiera wymóg, by typ `T` był pochodną typu `SomeClass` (lub był tą klasą) oraz implementował interfejs `Interface1`, a typ `U` zawierał konstruktor bezparametrowy:

```

class SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{ ... }

```

Warunki ograniczające można stosować wszędzie, gdzie są zdefiniowane parametry typów, zarówno w definicjach metod, jak i typów.



**Ograniczenie** kojarzy się z odebraniem jakiejś możliwości, ale podstawowym celem ograniczeń parametrów typu jest umożliwienie wykonywania czynności, które bez nich byłyby zabronione.

Na przykład ograniczenie `T:Foo` umożliwia traktowanie egzemplarzy typu `T` jako `Foo`, a ograniczenie `T:new()` pozwala tworzyć nowe egzemplarze typu `T`.

**Ograniczenie dotyczące klasy bazowej** oznacza, że parametr typu musi reprezentować podklasę określonej klasy lub samą tę klasę. **Ograniczenie dotyczące interfejsu** oznacza, że parametr typu musi implementować dany interfejs. W ten sposób można zapewnić, że egzemplarze parametru typu będzie można niejawnie przekonwertować na określoną klasę lub określony interfejs.

Powiedzmy np., że chcemy napisać generyczną metodę o nazwie `Max` zwracającą większą z dwóch wartości. Możemy wykorzystać generyczny interfejs z przestrzeni nazw `IComparable<T>`:

```

public interface IComparable<T> // uproszczona wersja interfejsu
{
    int CompareTo (T other);
}

```

Metoda `CompareTo` zwraca liczbę dodatnią, jeśli wartość `this` jest większa od `other`. Przy użyciu tego interfejsu jako ograniczenia możemy tak napisać metodę `Max` (dla uproszczenia pominięliśmy test wartości `null`):

```

static T Max <T> (T a, T b) where T : IComparable<T>
{
    return a.CompareTo (b) > 0 ? a : b;
}

```

Metoda `Max` przyjmuje argumenty każdego typu implementującego interfejs `IComparable<T>` (zalicza się do nich większość typów wbudowanych, np. `int` i `string`):

```

int z = Max (5, 10);           // 10
string last = Max ("mrówka", "zoo"); // zoo

```



Od C# 11 ograniczenie interfejsu umożliwia także wywoływanie statycznych składowych wirtualnych/abstrakcyjnych na tym interfejsie (zob. podrozdział „Statyczne wirtualne/abstrakcyjne składowe interfejsy” w rozdziale 1.). Jeśli np. interfejs `IFoo` definiuje statyczną abstrakcyjną metodę o nazwie `Bar`, to ograniczenie `T:IFoo` sprawia, że dozwolone są wywołania `T.Bar()`. Do tego tematu wracamy jeszcze w podrozdziale „Polimorfizm statyczny” w rozdziale 4.

**Ograniczenia dotyczące klasy i struktury** oznaczają, że `T` musi być typu referencyjnego lub (nie dopuszczającego wartości `null`) typu wartościowego. Świetnym przykładem ograniczenia dotyczącego struktury jest struktura `System.Nullable<T>` (jej szczegółowy opis znajduje się w podrozdziale „Typy wartościowe dopuszczające wartość `null`” w rozdziale 4.):

```
struct Nullable<T> where T : struct {...}
```

**Ograniczenie niekontrolowane** (wprowadzone w C# 7.3) to silniejsza wersja ograniczenia dotyczącego struktury: `T` musi być prostym typem wartościowym lub strukturą, która jest (rekurencyjnie) wolna od wszelkich typów referencyjnych.

**Ograniczenie dotyczące konstruktora bezparametrowego** to wymóg, aby typ reprezentowany przez parametr `T` miał konstruktor bezparametrowy. Jeśli warunek ten jest zdefiniowany, na `T` można wywoływać `new()`:

```
static void Initialize<T>(T[] array) where T : new()
{
    for (int i = 0; i < array.Length; i++)
        array[i] = new T();
}
```

**Ograniczenie dotyczące typu nagiego** (ang. *naked type*) to wymóg, aby jeden parametr typu był pochodną innego parametru typu lub był mu równoważny. W poniższym przykładzie metoda `FilteredStack` zwraca obiekt typu `Stack` zawierający tylko podzbiór elementów, przy czym parametr typu `U` reprezentuje ten sam typ co parametr typu `T`:

```
class Stack<T>
{
    Stack<U> FilteredStack<U>() where U : T {...}
}
```

## Tworzenie podklas typów generycznych

Klasy generyczne, podobnie jak zwykle, mogą mieć podklasy. W podklasie parametry typów można pozostawić otwarte, jak w poniższym przykładzie:

```
class Stack<T> {...}
class SpecialStack<T> : Stack<T> {...}
```

Ale też nic nie stoi na przeszkodzie, aby je zamknąć konkretnym typem:

```
class IntStack : Stack<int> {...}
```

Podtyp może też wprowadzać nowe argumenty typów:

```
class List<T> {...}
class KeyedList<T, TKey> : List<T> {...}
```



Z technicznego punktu widzenia *wszystkie* argumenty typów w podtypie są nowe, tzn. można powiedzieć, że podtyp zamyka, a następnie ponownie otwiera argumenty typu bazowego. Oznacza to, że w podklasie ponownie otwieranym argumentom można nadać nowe (i np. bardziej opisowe) nazwy:

```
class List<T> { ... }  
class KeyedList<TElement, TKey> : List<TElement> { ... }
```

## Odwolań do samego siebie w deklaracjach generycznych

W zamknięciu argumentu typu można wpisać nazwę *samego* deklarowanego typu:

```
public interface IEquatable<T> { bool Equals (T obj); }  
  
public class Balloon : IEquatable<Balloon>  
{  
    public string Color { get; set; }  
    public int CC { get; set; }  
  
    public bool Equals (Balloon b)  
    {  
        if (b == null) return false;  
        return b.Color == Color && b.CC == CC;  
    }  
}
```

Ten kod też jest poprawny:

```
class Foo<T> where T : IComparable<T> { ... }  
class Bar<T> where T : Bar<T> { ... }
```

## Dane statyczne

Dane statyczne są osobną własnością każdego typu zamkniętego:

```
Console.WriteLine (++Bob<int>.Count); //1  
Console.WriteLine (++Bob<int>.Count); //2  
Console.WriteLine (++Bob<string>.Count); //1  
Console.WriteLine (++Bob<object>.Count); //1  
  
class Bob<T> { public static int Count; }
```

## Parametry typów i konwersje

Operator rzutowania języka C# wykonuje kilka rodzajów konwersji:

- liczbowe,
- referencji,
- pakowania i rozpakowywania,
- własne programisty (poprzez przeciążanie operatorów, zob. rozdział 4.).

Rodzaj konwersji jest wybierany *w czasie kompilacji* na podstawie znanych typów argumentów. To stwarza interesującą sytuację w odniesieniu do generycznych parametrów typów, ponieważ typy argumentów stają się znane dopiero podczas kompilacji. Jeśli powstaną jakiegokolwiek wątpliwości, kompilator zgłasza błąd.

Najczęstszą sytuacją jest konwersja referencji:

```
StringBuilder Foo<T> (T arg)
{
    if (arg is StringBuilder)
        return (StringBuilder) arg; // to nie przejdzie kompilacji
    ...
}
```

Jeśli rzeczywisty typ T jest nieznanym, dla kompilatora oznacza to, że programista mógł chcieć wykonać *własną konwersję*. Najprostszym rozwiązaniem jest użycie operatora as, który jest jednoznaczny, ponieważ nie obsługuje własnych konwersji programisty:

```
StringBuilder Foo<T> (T arg)
{
    StringBuilder sb = arg as StringBuilder;
    if (sb != null) return sb;
    ...
}
```

Bardziej ogólnym rozwiązaniem byłoby dokonanie najpierw rzutowania na typ object. Konwersje z udziałem tego typu nie są uważane za niestandardowe, tylko za pakowanie lub rozpakowywanie. Ten przykład dotyczy referencyjnego typu StringBuilder, więc musi nastąpić konwersja referencji:

```
return (StringBuilder) (object) arg;
```

Konwersje rozpakowujące także mogą być niejednoznaczne. To, co widać poniżej, może być operacją rozpakowywania albo konwersją liczbową lub własną programisty:

```
int Foo<T> (T x) => (int) x; // błąd kompilacji
```

W tym przypadku rozwiązaniem również jest dokonanie najpierw rzutowania na typ object, a dopiero potem na int (jest to jednoznaczny sygnał, że chodzi o rozpakowywanie):

```
int Foo<T> (T x) => (int) (object) x;
```

## Kowariancja

Jeśli typ A można przekonwertować na typ B, to X ma kowariantny parametr typu, jeśli  $X < A >$  można przekonwertować na  $X < B >$ .



W odniesieniu do kowariancji (i kontrawariancji) w języku C# określenie „można przekonwertować” oznacza możliwość konwersji poprzez **niejawną konwersję referencji** — jak A *jest podklasą* B lub A *implementuje* B. Konwersje liczbowe, pakowanie oraz własne konwersje programisty się nie liczą.

Na przykład typ IFoo<T> ma kowariancję T, jeśli poniższy kod jest poprawny:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Interfejsy zezwalają na kowariantne parametry typów (tak jak delegaty — zob. rozdział 4.), natomiast klasy nie. Tablice także zezwalają na kowariancję (A[] można przekonwertować na B[], jeśli A umożliwia niejawną konwersję referencji na B). Opisujemy je poniżej dla porównania.



Kowariancja i kontrawariancja (lub prościej wariacja) to zaawansowane pojęcia. Motywem wprowadzenia i rozszerzenia wariacji w C# było umożliwienie interfejsom i typom generycznym (szczególnie tym zdefiniowanym w platformie .NET, np. `IEnumerable<T>`) działanie w sposób bliższy oczekiwaniom programisty. Korzyści z tego można odnosić, nawet nie znając szczegółowo pojęć kowariancji i kontrawariancji.

## Wariancja nie jest automatyczna

Ze względu na bezpieczeństwo typów statycznych parametry typów nie są automatycznie wariantne. Spójrz na poniższy przykład:

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T> // prosta implementacja stosu
{
    int position;
    T[] data = new T[100];
    public void Push (T obj) => data[position++] = obj;
    public T Pop()           => data[--position];
}
```

Poniższy kod nie przejdzie kompilacji:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears; // błąd kompilacji
```

To ograniczenie zapobiega ewentualnym błędom wykonawczym w takim kodzie:

```
animals.Push (new Camel()); // próba dodania wielbłąda do niedźwiedzi
```

Z drugiej strony brak kowariancji może utrudniać wielokrotne wykorzystanie kodu. Powiedzmy np., że chcemy napisać metodę do mycia (Wash) stosu zwierząt:

```
public class ZooCleaner
{
    public static void Wash (Stack<Animal> animals) {...}
}
```

Wywołanie metody Wash ze stosem niedźwiedzi spowoduje błąd kompilacji. Jednym z rozwiązań tego problemu jest dodanie do definicji metody Wash ograniczenia:

```
class ZooCleaner
{
    public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
}
```

Teraz metodę tę można wywoływać w następujący sposób:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Innym rozwiązaniem jest implementacja przez klasę Stack<T> interfejsu z kowariantnym parametrem typu, czego przykład pokazujemy nieco dalej.



## Tablice

Typy tablicowe obsługują kowariancję z powodów historycznych. Oznacza to, że tablicę `B[]` można rzutować na `A[]`, jeśli `B` jest podklasą `A` (oraz oba typy są referencyjne). Na przykład:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears; // OK
```

Wadą tego jest ryzyko niepowodzenia operacji przypisania wartości do elementów w czasie działania programu:

```
animals[0] = new Camel(); // błąd wykonania
```

## Deklarowanie kowariantnego parametru typu

Kowariantne typy parametrów w interfejsach i delegatach można deklorować za pomocą modyfikatora `out`. Dzięki temu modyfikatorowi kowariantne typy parametrów, inaczej niż jest w tablicach, są w pełni bezpieczne typowo.

Możemy to zilustrować na przykładzie implementacji przez naszą klasę `Stack<T>` poniższego interfejsu:

```
public interface IPoppable<out T> { T Pop(); }
```

Modyfikator `out` parametru `T` oznacza, że `T` może być używany tylko na **pozycjach wyjściowych** (np. w typach zwrotnych metod). Modyfikator ten oznacza parametr typu jako **kowariantny** i pozwala nam na robienie takich rzeczy:

```
var bears = new Stack<Bear>();
bears.Push(new Bear());
// Bears implementuje IPoppable<Bear>. Możemy dokonać konwersji na IPoppable<Animal>:
IPoppable<Animal> animals = bears; // poprawne
Animal a = animals.Pop();
```

Kompilator zezwoli na konwersję z `bears` na `animals` — dzięki temu, że parametr typu jest kowariantny. Operacja ta jest bezpieczna typowo, ponieważ sytuacja, której kompilator stara się uniknąć — wstawienie obiektu typu `Camel` na stos — nie może wystąpić: nie ma możliwości wprowadzenia takiego obiektu *do* interfejsu, w którym `T` może występować tylko na *pozycjach wyjściowych*.



Kowariancja (i kontrawariancja) w interfejsach jest czymś, czego się raczej tylko *używa*, tzn. programista rzadko musi sam *napisać* wariantny interfejs.

Co ciekawe, parametry metod z modyfikatorem `out` nie nadają się do kowariancji ze względu na ograniczenia CLR.

Możliwość kowariantnego rzutowania da się wykorzystać w celu rozwiązania opisanego wcześniej problemu z wielokrotnym użyciem kodu:

```
public class ZooCleaner
{
    public static void Wash (IPoppable<Animal> animals) { ... }
}
```



Interfejsy `IEnumerator<T>` i `IEnumerable<T>`, opisane w rozdziale 7., mają kowariancję `T`. Dzięki temu można np. rzutować `IEnumerable<string>` na `IEnumerable<object>`.

Użycie kowariantnego parametru typu na pozycji *wejściowej* (np. jako parametru metody lub własności z możliwością zapisywania) spowoduje błąd kompilacji.



Kowariancja (i kontrawariancja) działa tylko dla elementów obsługujących *konwersję referencji* — nie *konwersje pakowania*. (Dotyczy to zarówno wariacji parametrów typów, jak i tablic). Jeśli więc napiszesz metodę przyjmującą parametr typu `IPoppable<object>`, to możesz ją wywołać przy użyciu `IPoppable<string>`, ale nie `IPoppable<int>`.

## Kontrawariancja

Wcześniej wyjaśniliśmy, że jeśli referencję `A` można niejawnie przekonwertować na `B`, to typ `X` ma kowariantny parametr typu, jeśli referencję `X<A>` można przekonwertować na `X<B>`. **Kontrawariancja** zachodzi wtedy, gdy można dokonać konwersji w przeciwnym kierunku — z `X<B>` na `X<A>`. Takie coś jest możliwe tylko wtedy, gdy parametr typu występuje jedynie na pozycjach *wejściowych* i jest oznaczony modyfikatorem `in`. Wracając do poprzedniego przykładu, jeśli klasa `Stack<T>` implementuje poniższy interfejs:

```
public interface IPushable<in T> { void Push (T obj); }
```

to możemy legalnie napisać taki kod:

```
IPushable<Animal> animals = new Stack<Animal>();  
IPushable<Bear> bears = animals; // poprawne  
bears.Push (new Bear());
```

Żadna składowa interfejsu `IPushable` nie podaje *na wyjście* `T`, więc nie możemy wpaść w kłopoty z powodu rzutowania `animals` na `bears` (nie ma np. możliwości użycia metody `Pop` przez ten interfejs).



Nasza klasa `Stack<T>` może implementować zarówno interfejs `IPushable<T>`, jak i `IPoppable<T>`, mimo że w każdym z nich parametr `T` ma modyfikator o przeciwnym znaczeniu! Jest to możliwe, ponieważ wariancja działa przez interfejs, a nie klasę. Zatem należy wybrać, czy skupić się na jednym, czy drugim interfejsie przed dokonaniem konwersji wariacyjnej. Później ten wybór będzie rzutował na to, jakiego rodzaju operacje będą dopuszczalne zgodnie z odpowiednimi zasadami wariacji.

Stanowi to też ilustrację, dlaczego *klasy* nie dopuszczają wariantnych parametrów typów — konkretne implementacje zazwyczaj wymagają przepływu danych w dwóch kierunkach.

W ramach jeszcze jednego przykładu spójrz na poniższy interfejs zdefiniowany w przestrzeni nazw `System`:

```
public interface IComparer<in T>  
{  
    // zwraca wartość wskazującą kolejność a i b  
    int Compare (T a, T b);  
}
```

Dzięki temu, że interfejs ten ma kontrawariantny parametr `T`, przy użyciu `IComparer<object>` możemy porównać dwa łańcuchy:

```
var objectComparer = Comparer<object>.Default;
// objectComparer implementuje IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Bartek", "Justyna");
```

Dla porównania z kowariancją kompilator zgłosi błąd, jeśli spróbujemy użyć kontrawariantnego parametru typu na pozycji wyjściowej (np. jako typu zwrotnego lub we własności z możliwością odczytu).

## Typy generyczne C# a szablony C++

Typy generyczne języka C# mają podobne zastosowanie jak szablony w C++, ale znacznie różnią się od nich w sposobie działania. W obu przypadkach ważna jest relacja między producentem a konsumentem, w której typy zastępcze producenta są wypełniane przez konsumenta. Ale w C# typy producenta (tzn. otwarte, np. `List<T>`) mogą być kompilowane do postaci bibliotek (np. `mscorlib.dll`). Jest to możliwe, ponieważ synteza między producentem a konsumentem, w wyniku której powstają typy zamknięte, odbywa się dopiero w czasie wykonywania. W przypadku szablonów C++ do tego procesu dochodzi podczas kompilacji. Oznacza to, że w C++ nie wdraża się bibliotek szablonów jako plików `.dll` — istnieją one tylko w postaci kodu źródłowego. Ponadto utrudnione jest dynamiczne prowadzenie inspekcji — nie mówiąc już o tworzeniu — typów parametryzowanych.

Aby lepiej zrozumieć, dlaczego jest to problem, spójrz jeszcze raz na metodę C# `Max`:

```
static T Max <T> (T a, T b) where T : IComparable<T>
=> a.CompareTo (b) > 0 ? a : b;
```

Czemu nie moglibyśmy jej zaimplementować tak?

```
static T Max <T> (T a, T b)
=> (a > b ? a : b); // błąd kompilacji
```

Powodem jest to, że metoda `Max` musi zostać skompilowana raz i działać dla wszystkich możliwych wartości `T`. Kompilacja nie powiedzie się, ponieważ operator `>` może mieć różne znaczenie w zależności od wartości `T` — w istocie nawet nie każdy typ `T` w ogóle ma ten operator zdefiniowany. Dla porównania przedstawiamy tę samą metodę `Max` napisaną przy użyciu szablonów C++. Ten kod będzie kompilowany osobno dla każdej wartości `T`, przyjmując semantykę operatora `>` dla danego typu `T`. A jeśli dany typ `T` nie będzie obsługiwał operatora `>`, nastąpi błąd kompilacji:

```
template <class T> T Max (T a, T b)
{
    return a > b ? a : b;
}
```

# Skorowidz

.NET, 277, 280  
  data i godzina, 302  
  docelowe środowiska wykonawcze, 278  
  flagi parsowania, 320  
  formatowanie, 314  
  globalizacja, 330  
  klasy pomocnicze, 353  
  kolekcje, 281  
  kryptografia, 284  
  liczby, 331  
  łańcuchy i tekst, 289  
  mechanizmy konwersji, 326  
  określanie kolejności, 350  
  operacje wejścia i wyjścia, 283  
  parsowanie, 314  
  porównywanie, 340  
  sieć, 283  
  standardowe łańcuchy formatu, 320  
  strefy czasowe, 309  
  strumienie, 283  
  TFM, 278  
  wersje środowiska, 280  
  wyczerpanie, 336  
  zestawy referencyjne, 280  
.NET Framework, 23, 280  
.NET Standard 2.0, 279

## A

abstrakcyjne klasy i składowe, 142  
adaptery strumienia  
  binarne, 692  
  tekstowe, 687  
  zamykanie, 693  
adnotacje, 528  
adresy  
  IP, 718  
  URI, 719  
agregacje, 495

aktywacja typów, 778  
ALC, Assembly Load Context  
  bieżący kontekst, 765  
  domyślny kontekst, 763  
  stare metody ładujące, 770  
  usuwanie załadowanych kontekstów, 770  
algorytm RSA, 853  
algorytmy obliczania skrótów, 844  
aliasy, 111  
analiza  
  działających procesów, 595  
  wątków, 596  
anonimowe wywoływanie składowych, 834  
API  
  refleksji, 788  
  siecione, 285  
APM, Asynchronous Programming Model, 667  
architektura  
  sieci, 716  
  strumienia, 670  
  środowiska wykonawczego, 20  
argument NumberStyles, 322  
argumenty nazwane, 87  
asercja, 591  
  granicy słowa, 978  
  o zerowej wielkości, 975  
ASP.NET Core, 286  
asynchroniczne wyrażenia lambda, 650  
asynchroniczność, 282, 609  
atak słownikowy, 845  
atomowość, 860  
atrybut  
  [Conditional], 589  
  [SkipLocalsInit], 270  
  [ThreadStatic], 888  
  [UnmanagedCallersOnly], 952

AttributeUsage, 801  
CallerArgumentExpression, 249  
Flags, 164  
atrybuty, 246, 283  
  bitmapowe, 800  
  debuggera, 595  
  informacji wywołującego, 248  
  nazwane i pozycyjne, 246  
  niestandardowe, 800  
  przypisywanie, 248  
  pseudoniestandardowe, 801  
  stosowanie, 247  
  warunkowe, 272  
  własne, 802

## B

bariera, 885  
  wątku wykonawczego, 884  
BCL, Base Class Library, 19, 21, 277, 281, 358  
bezpieczeństwo  
  plików, 703  
  systemu operacyjnego, 709  
  typów, 18  
  wątkowe, 676, 865  
  przy odczycie, 866  
  w serwerach aplikacji, 867  
białe znaki, 73  
biblioteka  
  ASP.NET Core, 285  
  BCL, 277  
  Uncapsulator, 790  
biblioteki DLL, 946  
BitArray, 382  
blok instrukcji, 47  
blokady, 864  
  asynchroniczne, 871  
  odczytu i zapisu, 872  
  stosowanie, 859  
  z możliwością uaktualnienia, 875

- z podwójnym zatwierdzeniem, 886
- zagnieżdżanie, 861
- bloki try-catch-finally, 212
- blokowanie, 860
  - bez wykluczenia, 856, 869
- obiektów bezpiecznych
  - wątkowo, 865
- rekurencja, 876
- wykluczające, 856
- błędy
  - parsowania, 324
  - zaokrąglania liczb, 68
- bufory o stałym rozmiarze, 266

## C

### C#

- asynchroniczne strumienie, 37
- atributy, 246
- atributy informacji
  - wywołującego, 248
- BCL, 21
- bezpieczeństwo, 264
- bezpieczeństwo typów, 18
- CLR, 20
- dekonstrukcja krotek, 29
- dekonstruktory, 40
- delegaty, 180
- dokumentacja XML, 273
- domyślne parametry lambda, 25
- domyślne składowe interfejsów, 35
- dyrektywy preprocesora, 270
- dziedziczenie, 136
- funkcje asynchroniczne, 43, 639
- generyczna matematyka, 28
- globalna dyrektywa using, 29
- identyfikatory, 49
- indeksy i zakresy, 34
- inicjalizatory pól, 29
- instrukcje, 96
- instrukcje najwyższego poziomu, 31, 55
- interfejsy, 156
- klasy, 114
- komentarze, 50
- kompilacja, 48
- konstruktory bezparametrowe, 29
- konstruktory podstawowe, 25
- krotki, 41, 226
- literały, 50
- literały liczbowe, 62
- łańcuchy UTF-8, 26

- łańcuchy znaków, 71
- metody
  - anonimowe, 200
  - lokalne, 40
  - rozszerzające, 222
- modyfikatory dostępu, 154
- niedestrukcyjne
  - modyfikowanie rekordów, 29
- obiektowość, 17
- odrzućca, 39
- operator warunkowy null, 42
- operatory, 50, 64, 93–95
- parametry opcjonalne, 43
- pierwszy program, 46
- polimorfizm statyczny, 261
- przeciążanie operatorów, 258
- przekazywanie stanu, 841
- przestrzenie nazw, 106
- rekordy, 32, 230
- rozszerzenia wyrażeń lambda, 30
- separatory cyfr, 39
- settery tylko do inicjalizacji, 31
- składnia, 49
- składowe tylko do odczytu, 35
- słowa kluczowe, 49
- statyczne metody lokalne, 35
- struktury, 151
- struktury rekordowe, 30
- surowe literały łańcuchowe, 26, 72
- środowiska wykonawcze, 19
- tablice, 75
- typ logiczny, 69
- typ object, 147
- typizowane wyrażenia new, 33
- typy
  - anonimowe, 224
  - danych, 51
  - generyczne, 45, 167
  - referencyjne, 219
  - wartościowe, 214
- wartość null, 36
- wiązanie dynamiczne, 43, 250
- wskazniki, 264
- wyjątki, 201
- wyliczenia, 163, 209
- wyrażenia
  - kolekcyjne, 24
  - lambda, 194
  - operatory, 90
  - switch, 36
  - zapytań, 415

- wzorce, 241
  - list, 27
  - pozycyjne, 36
  - relacyjne, 32
  - typów, 39
  - własności, 36
- zarządzanie pamięcią, 19
- zdarzenia, 188
- zmienne
  - lokalne ref, 38
  - parametry, 80
  - wyjściowe, 39
  - wzorcowe, 39
- znaki interpunkcyjne, 50
- CCW, COM-Callable Wrapper, 967
- ciasteczko, cookie, 729
- CLR, Common Language Runtime, 19, 281
  - implementacja indeksatorów, 129
  - implementacja własności, 127
- COM, Component Object Model, 284, 946, 961
  - system typów, 962
  - wywołanie komponentu, 963
- cyfrowy certyfikat witryny, 852
- czas letni, 313

## D

- data i godzina, 302
- DbContext, 440
  - konfiguracja modelu, 441
  - konfiguracja połączenia, 441
  - śledzenie obiektów, 444
  - śledzenie zmian, 446
  - usuwanie obiektów, 445
  - używanie klasy, 444
- debugger, 594
- debugowanie, 591
- definiowanie
  - metod generycznych, 819
  - metody Main, 55
  - przestrzeni nazw, 54
  - rekordu, 231
  - typów generycznych, 820
- deklaracja klasy, 114
- deklaracje XML, 521
- deklarowanie wielu pól, 115
- dekonstruktory, 120
- dekoratory
  - kompozycja, 424
  - łączenie w łańcuchy, 423
  - sekwencja, 422
  - warstwy sekwencji, 423

- delegat
  - Action, 184
  - Func, 184
  - MatchEvaluator, 980
- delegaty, 18, 180, 451
  - a interfejsy, 185
  - multiemisji, 182
  - poprawa wydajności, 793
  - zgodność, 186
- DLR, dynamic language runtime, 829
- DNS, 718, 733
- docelowy framework, 278
- DOM, Document Object Model, 502
- dostawcy formatu, 315
- dostęp do składowych niepublicznych, 793
- drzewo
  - wyrażenia, 451–454
  - wywołań asynchronicznych, 648
  - X-DOM, 504, 506
- dynamiczne
  - obiekty, 837
  - wybieranie przeciążonych składowych, 831
  - wywoływanie składowej, 790
- dynamiczny system wykonawczy języka, DLR, 829
- dyrektywy preprocesora, 270–272, 587
  - global using, 108
  - using, 108, 111
  - using static, 109
- dziedziczenie, 136–147
- dziennik zdarzeń Windows, 598
  - monitorowanie, 599
  - odczyt, 599
  - zapis danych, 598

## E

- EAP, Event-based Asynchronous Pattern, 668
- EF Core, 440
  - klasa DbContext, 440
  - klasy jednostek, 440
  - migracja, 443
  - tworzenie bazy danych, 442
  - własności nawigacyjne, 447
  - wykonywanie opóźnione, 449
- egzemplarz
  - referencyjny, 58
  - typu wartościowego, 58

- egzemplarze nasłuchujące, 592, 593
- element główny, 567
- emitowanie
  - generycznych metod, 819
  - generycznych typów, 819
  - kłopotliwe, 821
  - konstruktorów, 817
  - metod, 814
  - pól i właściwości, 816
  - typów, 811
  - zestawów, 811
- enumeratory, 209, 942

## F

- FileStream
  - tworzenie, 677
  - wybór trybu pliku, 679
  - zaawansowane funkcje, 680
- filtrowanie, 459
- finalizatory, 133, 568
- format JSON, 547
- formatowanie, 314
  - złożone, 317
- formularz, 729
- framework, 21
- FTP, 718
- funkcje, 18
  - pieczętowanie, 143
  - asynchroniczne, 639
    - kontekst synchronizacji, 653
    - oczekiwanie, 640
    - optymalizacje, 655
  - równoległość, 649
  - tworzenie, 646
  - wyrażenia lambda, 650
- operatorowe, 258

## G

- generowanie
  - dynamicznego kodu, 804
    - DynamicMethod, 804
  - obsługiwanie wyjątków, 810
  - parsowanie argumentów, 806
  - rozgałęzianie, 808
  - stos ewaluacji, 806
  - tworzenie instancji
    - obiektów, 809
    - zmiennych lokalnych, 807
  - IL, 804
    - metod instancji, 815
- generyczne typy delegacyjne, 184
- globalizacja, 330

- głębokie klonowanie, 508
- grupowanie, 485
  - GroupJoin, 478
  - według wielu kluczy, 488

## H

- HashSet<T>, 383
- HTTP, 718
- HttpClient, 721
  - ciasteczka, 729
  - łańcuch zapytania, 728
  - nagłówki, 728
  - proxy, 726
  - przekazywanie danych formularza, 729
  - uwierzytelnienie, 726

## I

- IIS, 718
- IL, Intermediate Language, 777
- implementacja jawna interfejsu, 157
- implementowanie
  - indeksatora, 128
  - interfejsów przeliczeniowych, 362
  - interfejsu IComparable, 352
  - obiektów dynamicznych, 837
- indeksatory, 128, 965
- inicjalizatory
  - kolekcji, 210
  - obiektów, 122, 432
  - pól, 131
  - własności, 125, 131
- instrukcja
  - break, 105
  - continue, 105
  - fixed, 265
  - goto, 105
  - if, 98
  - lock, 857
  - return, 106
  - switch, 100, 101
  - throw, 106
  - try, 201
  - using, 204
  - yield break, 212
- instrukcje
  - deklaracji, 97
  - iteracyjne, 103
  - najwyższego poziomu, 48, 55
  - skoku, 105
  - wyboru, 98
  - wyrażeń, 97

interfejsy, 17, 156, 781  
  COM, 962  
  ICollection, 365, 366  
  ICollection<T>, 366  
  IComparable, 351, 352  
  IComparer, 403  
  ICustomFormatter, 318  
  IDictionary, 386  
  IDictionary<TKey,TValue>, 385  
  IDispatch, 963  
  IDisposable, 560  
  IEnumerable, 359  
  IEnumerable<T>, 360  
  IEnumerator, 359  
  IEnumerator<T>, 360  
  IEqualityComparer, 401  
  IEquatable<T>, 344, 349  
  IFormatProvider, 318  
  IList, 365, 367  
  IList<T>, 367  
  INumber<TSelf>, 263  
  IOrderedEnumerable, 484  
  IOrderedQueryable, 484  
  IProducerConsumerCollection<T>, 930  
  IProgress<T>, 661  
  IReadOnlyCollection<T>, 368  
  IReadOnlyList<T>, 368  
  IStructuralComparable, 405  
  IStructuralEquatable, 405  
  IUnknown, 963

interfejsy, 17, 156, 781  
  API, 285  
    LINQ to XML, 502  
    MAUI, 288  
    UWP, 287  
    Windows Forms, 287  
    WinUI3, 287  
    WPF, 286  
  jawna implementacja, 157  
  niegeneryczne, 361  
  pakowanie, 160  
  przeliczeniowe, 362  
  reimplementacja w podklasie, 158  
  rozszerzanie, 157  
  składowe, 156  
    domyślne, 160  
    statyczne, 161  
  wirtualna implementacja  
    składowych, 158  
interoperacyjność macierzysta, 284  
interpolacja łańcuchów, 73

IP, 718  
iteratory, 209–212

## J

język IL, 777  
języki dynamiczne, 840  
JIT, just-in-time, 20  
JSON, 282, 547

## K

katalogi  
  operacje, 703  
  specjalne, 706

klasa, 17, 114, 156  
  AggregateException, 927  
  AppContext, 357  
  Array, 368  
  ArrayList, 376  
  Assembly, 743  
  AssemblyDependencyResolver, 769  
  AssemblyName, 746  
  AsyncLocal<T>, 890  
  AutoResetEvent, 877  
  BackgroundWorker, 669  
  Barrier, 884  
  BitConverter, 329  
  BitOperations, 335  
  BlockingCollection<T>, 932  
  BufferedStream, 685  
  Collection<T>, 391  
  CollectionBase, 391, 393  
  Comparer, 403  
  ConcurrentBag<T>, 931  
  Console, 353  
  Convert, 326  
  CountdownEvent, 881  
  DbContext, 440, 444  
  Debugger, 594  
  DelegatingHandler, 725  
  Dictionary<TKey,TValue>, 387  
  DictionaryBase, 393  
  Directory, 703  
  DirectoryInfo, 704  
  Dns, 733  
  DynamicMethod, 804  
  DynamicObject, 837  
  Encoding, 300  
  Environment, 354  
  EqualityComparer, 401  
  ExpandableObject, 839  
  File, 699

FileInfo, 704  
FileStream, 677  
FileSystemWatcher, 708  
HashAlgorithm, 844  
Hashtable, 387  
HttpClient, 721  
HttpContent, 723  
HttpListener, 730  
HttpMessageHandler, 724  
HybridDictionary, 389  
JsonDocument, 551  
JsonNode, 554  
KeyedCollection<TKey,TItem>, 393  
Lazy<T>, 886  
LazyInitializer, 887  
LinkedList<T>, 379  
List<T>, 376  
ListDictionary, 389  
ManualResetEvent, 880  
Math, 332  
MemberInfo, 787  
MemoryStream, 680  
Mutex, 863  
NullabilityInfoContext, 790  
Object, 150  
OrderedDictionary, 388  
Parallel, 895, 911  
Path, 705  
PeriodicTimer, 891  
PipeStream, 681  
Process, 355  
Progress<T>, 662  
Queue, 381  
Queue<T>, 381  
Random, 334  
ReadOnlyCollection<T>, 395  
RSA, 852  
SmtClient, 733  
Socket, 735  
SortedSet<T>, 383  
Stack, 382  
Stack<T>, 382  
StackFrame, 596  
StackTrace, 596  
Stopwatch, 604  
Stream, 672  
StringBuilder, 298  
StringComparer, 404  
System.Exception, 207  
System.Tuple, 230  
Task<TResult>, 647  
TaskCompletionSource, 632

- klasa
    - TaskFactory, 926
    - TcpClient, 735, 737
    - TcpListener, 737
    - ThreadLocal<T>, 888, 903
    - TimeZoneInfo, 310
    - TraceListener, 592
    - Type, 779, 781
    - TypeInfo, 779
    - UdpClient, 733
    - XmlConvert, 328
    - XmlReader, 533, 545
    - XmlWriter, 541, 545
  - klasy, 17, 114, 156
    - abstrakcyjne, 142
    - atrybutów, 246
    - dekonstrukcyjne, 120
    - finalizatory, 133
    - indeksatory, 128
    - konstrukcyjne, 398
    - konstrukcyjne, 118
      - podstawowe, 129
      - statyczne, 132
    - metody, 116, 134
    - monitorowania, 591
    - operator nameof, 136
    - pieczętowanie, 143
    - pochodne, 137
    - pola, 114
    - pomocnicze, 353
    - referencja this, 123
    - słownikowe, 385
    - stałe, 115
    - statyczne, 133
    - własności, 124
  - klauzula
    - case, 101
    - catch, 202
    - else, 98
    - inicjalizacji, 104
    - iteracyjna, 104
    - warunkowa, 104
  - kod niebezpieczny, 265
  - kodowanie tekstu, 299
  - kolejka typu
    - producent-konsument, 933
  - kolejki, 376
  - kolejność
    - inicjalizacji pól, 133, 146
    - wykonywania działań, 92
  - kollekcje, 210, 358
    - interfejsy, 359
    - kolejki, 376
  - listy, 376
    - niezmiennie, 396
      - tworzenie, 397
      - wydajność, 398
  - porządkowanie, 400
  - pośredniki, 390
  - protokoły równości, 400
  - przeliczalność, 358
  - słowniki, 384
  - stosy, 376
  - tablice, 368
  - współbieżne, 929
  - zamrożone, 399
  - zbiory, 376
  - kombinatory wzorców, 242
  - komentarz, 50
    - dokumentacyjny, 273
  - komparatory, 403, 484
  - kompilacja
    - na żądanie, JIT, 20
    - warunkowa, 587
      - atrybut [Conditional], 589
      - opcje zmiennej statycznej, 588
  - kompilator, 48
    - Roslyn, 285
  - kompresja, 700
    - plików gzip, 696
    - strumienia
      - w pamięci, 695
  - komunikaty
    - odpowiedzi, 722
    - żądania, 723
  - konkatenacja łańcuchów, 73
  - konstruktor domyślny struktury, 152
  - konstrukcyjne
    - egzemplarzy, 118
    - klasy bazowej, 144
    - kolejność inicjalizowania pól, 120
    - niejawne bez parametrów, 120
    - niepubliczne, 120
    - podstawowe, 129, 131
      - maskowanie parametrów, 131
      - rekordów, 238
      - weryfikacja parametrów, 131
    - przeciążanie, 119
    - statyczne, 132
    - wywoływanie niejawne, 144
  - kontekst
    - adnotacji, 221
    - ostrzegania, 221
  - kontekstowe ALC, 765
  - kontrawariancja, 178
  - kontrola typów, 149
    - statyczna, 18
    - ściśła, 19
  - kontynuacje, 636, 882
  - konwencje wywoływania, calling conventions, 951
  - konwersje, 174, 326
    - dynamiczne, 254, 327
    - jawne, 57, 260
    - liczb całkowitych, 338
    - liczb rzeczywistych, 327
    - liczbowe, 63, 331
    - łańcuchowe, 338
    - na format base64, 328
    - niejawne, 56, 260
    - niejawne referencje, 175
    - referencje, 137, 175
    - typu logicznego, 69
    - wyliczeń, 163, 336
    - znaków, 71
  - konwertery typów, 329
  - kowariancja, 175
  - krotki, 226
    - dekonstruowanie, 229
    - klasa System.Tuple, 230
    - nazywanie elementów, 227
    - porównywanie, 229
    - tworzenie aliasów, 228
  - kryptografia, 284, 842
  - kultury, 756
  - kwantyfikatory, 974
- ## L
- LAN, 718
  - leniwa
    - ewaluacja, 236
    - inicjalizacja, 885
  - liczniki wydajności, 600
    - odczyt danych, 602
    - sprawdzenie, 601
    - tworzenie, 603
    - zapis danych, 603
  - LIFO, last in, first out, 147, 382
  - likwidowanie
    - obiektów szyfrowania, 850
    - uchwyty oczekiwania, 878
  - LINQ, Language Integrated Query, 44, 282, 407
    - filtrowanie, 459
    - grupowanie, 485
    - kwantyfikatory, 500
    - łączenie, 475



- łączenie operatorów zapytań, 409
- metoda progresywna, 428
- metody agregacyjne, 495
- metody generujące, 501
- metody konwersji, 490
- modyfikowanie drzewa X-DOM, 514
- opakowywanie zapytań, 431
- operatory, 414, 455
  - elementów, 493
  - zbiorów, 489
- podzapytania, 425, 428
- ponowne wykonanie zapytania, 420
- porządkowanie, 482
- porządkowanie naturalne, 414
- projekcja, 463
- projekcja do X-DOM, 529
- przechwytywanie zmiennych, 421
- schemat, 416
- składnia płynna, 409
- wykonywanie opóźnione, 419, 422, 428
- wykonywanie zapytań, 424
- zapytania interpretowane, 434
- zapytania złożone, 428
- zmiennie zakresowe, 417

## L

- ładowanie
  - jawne, explicit loading, 448
  - leniwe, lazy loading, 449
- łańcuchy, 291
  - dzielenie i łączenie, 293
  - formatowania
    - daty i godziny, 323
    - wyliczeń, 326
  - formatu, 308
    - niestandardowe, 321
    - numeryczne, 320
    - standardowe, 320
  - modyfikowanie, 293
  - null, 291

- pobieranie znaków, 292
- porównywanie, 295
  - kulturowe, 295
  - pod względem kolejności, 297
  - pod względem równości, 296
  - porządkowe, 295
- przeszukiwanie, 292
- puste, 291
- strumieni szyfrowania, 849
- zapytań, 728
- złożone, 294
- znaków
  - interpolacja, 73
  - konkatenacja, 73
  - porównywanie, 74
  - UTF-8, 75
- łącznik zadań, 663
- łączność operatorów, 92

## M

- magazyn danych strumieni, 676
- mapowanie
  - plików w pamięci, 711
  - struktury, 957
- MAUI, Multi-platform App UI, 23, 288
- mechanizm usuwania nieużytków, GC, 560, 573
  - dostrajanie, 578
  - pule tablic, 579
  - techniki optymalizacji, 574
  - wymuszenie działania, 577
- metadane, 20, 777
  - składowych, 787
  - typu, 778
- metoda, 18, 46, 116, 134
  - Aggregate, 497
  - All, 500
  - Any, 500
  - AsEnumerable, 493
  - AsParallel, 900
  - AsQueryable, 493
  - Assembly.Load, 765
  - Average, 496
  - Cast, 491
  - Chunk, 488
  - Close, 562
  - Concat, 489
  - Contains, 500
  - CopyTo, 939
  - Count, 495
  - DefaultIfEmpty, 495
  - Delay, 634
  - Dispose, 564, 569
  - Distinct, 463
  - DistinctBy, 463
  - ElementAt, 494
  - Empty, 501
  - EnterContextualReflection, 767
  - EqualityComparer<T>.Default, 402
  - Equals statyczna, 343
  - Equals wirtualna, 342
  - Equals, 345, 348
  - Except, 490
  - ExceptBy, 490
  - First, 493
  - FirstNode, 510
  - Flatten, 928
  - get, 124, 126
  - GetAsync, 722
  - GetData, 889
  - GetHashCode, 347
  - GetType, 149
  - GroupBy, 485
  - GroupJoin, 475, 478
  - Handle, 928
  - Intersect, 490
  - IntersectBy, 490
  - Join, 475
  - Last, 493
  - LastNode, 510
  - Load(byte[]), 771
  - LoadFile, 771
  - LoadFrom, 771
  - LoadFromAssemblyName, 760
  - LongCount, 495
  - Main, 55
  - Max, 496
  - MaxBy, 494
  - Min, 496
  - MinBy, 494
  - Monitor.Enter, 857
  - Monitor.Exit, 857
  - Nodes, 510
  - object.ReferenceEquals, 344
  - OfType, 491
  - OperationCompleted(), 655
  - OperationStarted(), 655
  - Parallel.For, 912
  - Parallel.ForEach, 912
  - Parallel.ForEachAsync, 872
  - Parallel.Invoke, 911
  - Parse, 315
  - Range, 501

- metoda
  - Regex.Split(), 981
  - Repeat, 501
  - ReRegisterForFinalize(), 573
  - Select, 468–74
  - SelectMany, 468–74
  - SendAsync, 723–725
  - SequenceEqual, 500
  - set, 124, 126
  - SetData, 889
  - SignalAndWait, 883
  - Single, 493
  - Skip, 462
  - SkipLast, 462
  - SkipWhile, 462
  - Stop(), 562
  - string.Format, 294
  - Sum, 496
  - Take, 462
  - TakeLast, 462
  - TakeWhile, 462
  - Task.WhenAll, 664
  - Task.WhenAny, 663
  - ToArray, 492
  - ToDictionary, 492
  - ToHashSet, 492
  - ToList, 492
  - ToLookup, 492
  - ToString, 150, 315
  - TryCopyTo, 939
  - TryEnter, 858
  - Union, 489
  - UnionBy, 489
  - ValueTuple.Create, 229
  - WaitAll, 883
  - WaitAny, 883
  - WaitHandle.WaitAny, 883
  - Where, 460
- metody, 18, 46, 116, 134
  - agregacyjne, 495
  - anonimowe, 200
  - asynchroniczne, 653, 662
  - częściowe, 135
  - częściowe rozszerzone, 135
  - docelowe delegatu, 182
  - dostępowe, 124, 126
  - dostępowe zdarzeń, 193
  - egzemplarzowe, 223
  - filtrowania, 459
  - generujące, 501
  - generyczne, 169, 792, 794
  - emitowanie, 819
  - klasy Assembly, 744

- klasy DynamicObject, 837
- lokalne, 117, 118
- lokalne statyczne, 118
- płynnego API, 443
- porządkowania, 482
- przeciążanie, 118, 147
- rozszerzające, 222, 223
  - degradowanie, 224
  - wywoływanie łańcuchowe, 222
- skrótów klasy File, 678
- synchroniczne, 660
- TryXXX, 208
- typu char, 290
- ustawiające tylko do
  - inicjalizacji, 126
- wtyczek, 181
- wyrażeńowe, 117
- z rodziny ReadXXX, 537
- miejsca wywołania, 830
- model
  - obiekty Reflection.Emit, 812
  - programowania
    - asynchronicznego, APM, 667
- modreq, 789
- modyfikator
  - in, 85
  - out, 84
  - params, 86
  - readonly, 115
  - ref, 83
- modyfikatory
  - dostępu, 154
  - ograniczanie dostępności, 155
  - ograniczenia, 156
  - zestawy zaprzyjaźnione, 155
  - zdarzeń, 194
- modyfikowanie drzewa
  - X-DOM, 514
- multiemisja, multicast, 182
- mutacja niedestrukcyjna, 230, 234

## N

- nagłówek uwierzytelnienia, 728
- narzędzie diagnostyczne, 605
  - dotnet-counters, 605
  - dotnet-dump, 608
  - dotnet-trace, 606
- nawiasy klamrowe, 99
- nawigacja
  - do rodzica, 512
  - na tym samym poziomie, 513

- po atrybutach, 513
- po węzłach potomnych, 509

## O

- obiektość, 17
- obiekty, 80
  - DateTime, 314
  - niezmienne, 868
  - szyfrowania, 850
  - typu
    - DateTime, 305, 309
    - DateTimeOffset, 306, 309
    - Span, 939
- obliczanie skrótów, 842–845
- obsługa wyjątków, 618
- odrzućenia, 85
- odwołania
  - do typów i składowych, 275
  - lokalne ref, 88
- ograniczenia typów
  - generycznych, 171
- operacje
  - asynchroniczne, 634
  - synchroniczne, 634
  - wejścia i wyjścia, 283
- operator
  - !=, 342
  - &, 217
  - |, 217
  - < i >, 352
  - ==, 342, 345
  - as, 139
  - AsEnumerable, 439
  - AsQueryable, 452
  - ignorowania null, 220
  - is, 139
  - Join, 476
  - nameof, 136
  - OrderBy, 483
  - OrderByDescending, 483
  - ThenBy, 483
  - ThenByDescending, 483
  - trójargumentowy, 70
  - typeof, 149, 171
  - wskaźnika do składowej, 266
  - Zip, 482
- operatory, 50, 90–95, *Patrz także*
  - metody
    - arytmetyczne, 64
    - bitowe, 66
    - elementów, 493
    - inkrementacji
      - i dekrementacji, 65

kontrolowane, 259  
LINQ, 455  
logiczne, 69  
null, 95, 217  
polimorficzne, 262  
porównywania, 69  
porządkowania, 482  
pożyczanie, 215  
przeciążanie, 258, 348  
relacyjne, 216  
równości, 69, 216  
warunkowe, 70  
wyliczeń, 165  
zbiorów, 489  
zmieniające kształt, 457  
osadzanie typów  
współpracujących, 966  
ostrzeżenia, 221  
pragma, 272

**P**

pakowanie, boxing, 148, 160  
pamięć, 60  
lokalna wątku, 887  
niezarządzana, 945, 957  
współdzielona, 713, 955  
współdzielona procesów, 713  
parametry, 47, 82  
opcjonalne, 86, 122  
ref i out, 792  
parsowanie, 314, 318  
IL, 824  
liczb, 327  
pętla  
do-while, 103  
for, 103  
foreach, 104  
while, 103  
PFX, Parallel Framework, 895, 896  
komponenty bibliotek, 897  
równoległe wykonywanie  
zadań, 917  
używanie biblioteki, 898  
zastosowania biblioteki, 898  
pieczętowanie funkcji i klas, 143  
plasterkowanie, slicing, 936, 937  
platforma .NET, 277, 280  
pliki  
.resources, 752  
.resx, 753  
.tar, 698  
.zip, 697  
losowe operacje  
wejścia-wyjścia, 712

mapowanie w pamięci, 711, 713  
operacje, 699  
PLINQ, 897, 899  
anulowanie zapytania, 905  
czystość funkcyjna, 904  
kolejność elementów, 901  
model wykonawczy, 899  
ograniczenia, 902  
optymalizacja, 906  
od strony wejściowej, 907  
własnych agregacji, 909  
stopień zrównoleglenia, 905  
używanie, 904  
wykonywanie równoległe, 901  
płynne API, 441  
pobieranie typów  
osadzonych, 779  
tablicowych, 779  
poczta elektroniczna, 733  
podklasy typów generycznych, 173  
podpisy cyfrowe, 853  
podzapytania, 425, 428  
kompozycja, 426  
pola, 114  
polimorfizm, 137  
statyczny, 261  
wielokierunkowy, 834  
POP, 718  
POP3, 738  
porty, 718  
potoki  
anonimowe, 681, 683  
nazwane, 681  
późne wiązanie, 790  
programowanie  
asynchroniczne, 635, 636  
dynamiczne, 283, 829  
funkcyjne, 18  
równoległe, 284, 895  
protokoły równości, 341  
protokół  
POP3, 738  
TCP, 734  
UDP, 734  
przeciążanie  
konstruktorów, 119  
metod, 118, 147  
operatorów, 258, 348  
konwersje jawne i  
niejawne, 260  
równości i porównywania,  
259  
true i false, 261

przekazywanie argumentów  
przez referencję, 85  
przez wartość, 83  
przesłanie metody  
Equals, 348  
GetHashCode, 347  
przestrzenie nazw, 54, 106  
aliasy, 111  
o zasięgu plikowym, 107  
reguły, 109  
w XML, 523  
właściwości zaawansowane,  
111  
przypisanie, 81  
przypisywanie atrybutów, 818  
przysłanie metod, 815  
pule  
wątków, 624  
tablic, 579

**R**

refaktoryzacja, 47  
referencja, 58  
this, 123  
refleksja, 20, 283, 777  
dla zestawów, 799  
dostęp do składowych  
niepublicznych, 793  
dynamiczne  
generowanie kodu, 804  
wywoływanie składowej, 790  
interfejsy, 781  
nazwy typów, 780  
generycznych, 780  
osadzonych, 780  
parametrów ref i out, 781  
tablic i wskaźników, 781  
parametry metod, 791  
pobieranie  
atrybutów w czasie działania,  
803  
typów, 778  
refleksja  
składowe, 785  
abstrakcyjne, 797  
interfejsu generycznego, 795  
typów generycznych, 790  
statyczne wirtualne, 797  
tworzenie instancji typów, 782  
typy bazowe, 781  
typy składowych, 787  
własności tylko do inicjalizacji,  
789

- rekordy, 18, 230
    - definiowanie, 231
    - konstruktory podstawowe, 238
    - leniwa ewaluacja, 236
    - listy parametrów, 233
    - mutacja niedestrukcyjna, 234
    - porównywanie, 240
    - walidacja własności, 236
  - rekurencja blokowania, 876
  - reprezentacja typu dynamic, 253
  - rozpakowywanie, unboxing, 148
  - równoległa biblioteka zadań, TPL, 895
  - równoległe wykonywanie zadań, 917
    - anulowanie zadań, 920
    - czekanie na kilka zadań, 920
    - kontynuacje, 921, 923
    - kontynuacje warunkowe, 923
    - planowanie zadań, 925
    - uruchamianie zadań, 918
  - równoległość, 649
  - równość, 400
    - referencyjna, 340, 342
    - wartościowa, 340
  - równoważność typów, 967
  - rzutowanie, 56, 137
- S**
- sekwencje, 213
    - specjalne, 71
  - semafory, 869, 871
  - serializacja, 285
  - serwer HTTP, 730
  - sieć, 283, 716–739
  - składnia
    - mieszana, 419
    - plynna, fluent syntax, 409, 418
    - SQL, 418
    - zapytaniowa, 416, 418
  - składowe
    - abstrakcyjne, 142
    - C#, 788
    - CLR, 788
    - domyślne interfejsu, 160
    - egzemplarza, 53
    - funkcyjne, 140
    - interfejsu, 156
    - klasy Object, 150
    - niepubliczne, 793
    - statyczne interfejsu, 161
    - typów generycznych, 790
    - wymagane, 145
  - słabe odwołania, 583
    - buforowanie, 584
    - zdarzenia, 584
  - słowo kluczowe, 49
    - abstract, 158
    - async, 639
    - await, 639
    - base, 143
    - class, 114
    - Component, 754
    - extern, 111
    - fixed, 266, 961
    - from, 417
    - into, 430
    - let, 433
    - namespace, 107
    - new, 90, 143
    - override, 143
    - public, 54
    - required, 145
    - stackalloc, 266
    - static, 198
    - switch, 102
    - struct, 57
    - throw, 206
    - var, 89, 225
    - virtual, 140, 158
  - słowa kluczowe kontekstowe, 50
  - słowniki, 384
    - sortowane, 389
  - SMTP, 718
  - sortowanie, 484
  - specjalne działania
    - całkowitoliczbowe, 65
  - stałe, 115
  - sterta, 80
  - stos, 80, 376, 944
    - ewaluacji, 806
  - strefy czasowe, 310
  - struktura, 151
    - BigInteger, 332
    - Complex, 334
    - DateOnly, 309
    - DateTime, 303, 309, 313
    - DateTimeOffset, 303, 310
    - DOM, 453
    - DOM XmlNode, 558
    - Guid, 339
    - Half, 333
    - Memory<T>, 936, 941
    - Nullable<T>, 214
    - Span, 937
    - Span<T>, 936
    - TimeOnly, 309
    - TimeSpan, 302
    - Utf8JsonReader, 548
    - Utf8JsonWriter, 550
  - struktury, 151
    - funkcje, 153
    - konstruktor domyślny, 152
    - referencyjne, 153
    - tylko do odczytu, 153
  - strumienie, 283
    - adaptery, 686
    - architektura, 670
    - archiwa TAR, 698
    - archiwa ZIP, 697
    - asynchroniczne, 651
    - bezpieczeństwo wątków, 676
    - dekoracyjne, 671
    - kompresja, 694
    - limit czasu, 676
    - magazyn danych, 676, 671
    - odczyt i zapis, 674
    - operacje na katalogach, 703
    - operacje na plikach, 699
    - użycie, 672
    - wyszukiwanie, 675
    - zamknięcie i opróżnienie, 675
  - strumień
    - BufferedStream, 685
    - FileStream, 677
    - MemoryStream, 680
    - PipeStream, 681
  - subkultury, 756
  - sygnalizacja, 877
    - dwustronna, 880
  - sygnalizowanie, 856
  - sygnały dwustronne, 878
  - symulowanie unii C, 954
  - synchronizacja, 856
    - wyбір obiektu, 858
  - system typów COM, 962
  - systemy wykonawcze, 22
    - szeregowanie
      - In i Out, 950
      - klas i struktur, 949
      - typów wspólnych, 947
    - szyfrowanie, 700, 842
      - kluczem publicznym, 851
    - symetryczne, 846
    - w pamięci, 848
- Ś**
- środowisko wykonawcze, 21
    - .NET Framework, 23
    - MAUI, 23

Windows Desktop, 22  
WinUI 3, 22

## T

tablice, 75, 368  
  długość, 373  
  domyślna inicjalizacja, 76  
  indeksowanie, 371  
  indeksy i zakresy, 77  
  inicjalizacja uproszczona, 79  
  konwertowanie, 376  
  kopiowanie, 375  
  odwracanie kolejności  
    elementów, 375  
  płytkie kopiowanie, 370  
  przeglądanie zawartości, 372  
  przeszukiwanie, 373  
  skrótów, hash tables, 347  
  sortowanie, 374  
  sprawdzanie granic, 80  
  tworzenie, 371  
  w pamięci, 369  
  wielowymiarowe, 77  
  wymiary, 373  
  zmienianie rozmiarów, 376  
TAP, Task-based Asynchronous  
  Pattern, 663  
TCP, 718, 734, 738  
technologia Authenticode, 748  
testowanie zestawów  
  satelickich, 756  
TFM, Target Framework  
  Moniker, 278  
TPL, Task Parallel Library, 895  
tworzenie  
  aliasów, 228  
  bazy danych, 442  
  drzewa X-DOM, 506  
  fabryk zadań, 926  
  funkcji asynchronicznych, 646  
  instancji typów, 782  
  kolekcji, 397  
  łańcuchów, 291  
  łańcuchów strumieni  
    zcyfrowania, 849  
  serwera HTTP, 730  
  strumienia FileStream, 677  
  systemu wtyczek, 772  
  tablic, 371  
  typów, 114  
  wątku, 610  
  wyrażeń lambda, 412  
  zasobu, 754

zestawu satelickiego, 755  
złączeń, 476  
typ, 51  
  bool, 217  
  char, 71, 289  
  CultureInfo, 317  
  DateTimeFormatInfo, 317  
  decimal, 68  
  delegacyjny, 180  
  double, 67, 68  
  dynamic, 253, 254  
  float, 67  
  logiczny, 69  
  NumberFormatInfo, 317  
  object, 147  
  string, 72, 291  
  var, 254  
  wyliczeniowy RegexOptions, 971  
  zwrotny ref, 88  
TypeInfo, 779, 785  
typizowane wyrażenia new, 90  
typy  
  anonimowe, 224, 432  
  bazowe, 781  
  całkowitoliczbowe, 67  
  częściowe, 134  
  delegacyjne generyczne, 184  
  dynamiczna kontrola, 149  
  emitowanie, 811  
  generyczne, 167, 168, 783  
    anonimowe wywoływanie  
      składowych, 834  
    dane statyczne, 174  
    deklarowanie parametrów, 170  
    domyślna wartość  
      generyczna, 171  
    kontrawariancja, 178  
    konwersje, 174  
    kowariancja, 175  
    odwołania do samego  
      siebie, 174  
    ograniczenia, 171  
    operator typeof, 171  
    parametry, 174  
    szablony C++, 179  
    tworzenie podklas, 173  
  konwersje, 56  
    rozmiar macierzysty, 268  
  liczbowe, 61  
  niezmienne, 18  
  predefiniowane, 51, 60  
  referencyjne, 58, 61, 114  
  wartość null, 219

składowych, 787  
statyczna kontrola, 149  
statyczne, 256  
systemowe, 281  
wartościowe, 57, 60  
  wartość null, 214, 217  
wyjątków, 207  
wyrażeń, 453  
X-DOM, 504  
zagnieżdżone, 166  
zdefiniowane przez  
  programistę, 52  
zwrotne kowariantne, 141

## U

uchwyt EventWaitHandle, 881  
uchwyty  
  oczekiwania, 882  
  zdarzeń oczekiwania, 877  
udostępnianie obiektów C#  
  COM, 967  
UDP, 718, 734  
ukrywanie odziedziczonych  
  składowych, 142  
UNC, 718  
unia, 954  
Unicode, 299  
unifikacja, unification, 148  
URI, 718, 719  
URL, 718  
usługi P/Invoke, 946  
usuwanie  
  anonimowe, anonymous  
    disposal, 564  
  automatyczne nieużytków, 566  
  nieużytków, 944  
UTF-16, 301  
UWP, Universal Windows  
  Platform, 287

## V

Visual Studio, 753–756, 963, 966

## W

wariancja, 176  
warstwa aplikacji, 277, 285  
  Windows Desktop, 286  
wartości domyślne, 82  
wartość, 57  
  null, 59, 214, 217  
  skrótów, hash code, 347

- wątki, 284
  - aktywne, 620
  - bezpieczeństwo, 616
  - blokowanie, 612, 616
  - dołączanie, 612
  - działające w tle, 620
  - interfejsu użytkownika, 623
  - kontekst synchronizacji, 623
  - obsługa, 618
  - pamięć lokalna, 887
  - priorytet, 620
  - przekazywanie danych, 617
  - pula, 624
  - stan lokalny, 614
  - stan współdzielony, 614
  - sygnalizowanie, 621
  - tworzenie, 610
  - uspianie, 612
  - w aplikacjach klientów, 621
  - wywłaszczenia, 611
- węzeł XML, 534
- węzły atrybutów, 539
- wiązanie
  - dynamiczne, 250, 965
  - językowe, 252
  - niestandardowe, 252
  - statyczne, 251
- widok wyszukiwania, lookup, 480
- widoki, 714
- wielowątkowość, 855
- Windows
  - Data Protection, 843
  - Desktop, 22, 286
  - Forms, 287
- WinUI 3, 22
- WinUI3, 287
- wirtualne składowe funkcyjne, 140
- Wizytator, 831
- własności, 18, 124
  - automatyczne, 125
  - obliczane, 124
  - tylko do inicjalizacji, 789
  - tylko do odczytu, 124
  - wyrażeńiowe, 125
- wolumin, 707
- WPF, 286
- wskaznik pusty, void\*, 267
- wskazniki, 264
  - do funkcji, 269
- wskrzeszenie, 571
- współbieżność, 282, 609
  - gruboziarnista, 645
- współpraca COM, 961
- wtyczki, 772
- wyciek pamięci, 579
  - diagnozowanie, 582
  - zegar, 581
- wydajność, 863
- wyjątek
  - DivideByZeroException, 927
  - RuntimeBinderException, 253
  - XmlException, 536
- wyjątki, 201, 629
  - blok finally, 203
  - filtry, 203
  - klasa System.Exception, 207
  - klauzula catch, 202
  - metody TryXXX, 208
  - ponawianie zgłoszenia, 206
  - typy, 207
  - wyrażenia throw, 205
  - zgłaszanie, 205
- wyliczenia, 163, 209, 336
  - atrybut Flags, 164
  - bezpieczeństwo typów, 165
  - działanie, 339
  - konwersje, 163, 336
  - operatory, 165
  - pobieranie wartości, 338
- wyliczenie
  - DateTimeStyles, 325
  - TaskCreationOptions, 919
- wyrażenia, 90
  - dynamiczne, 255
  - lambda, 18, 194
    - asynchroniczne, 650
    - metody lokalne, 199
    - parametry domyślne, 196
    - statyczne, 198
    - sygnatury Func, 413
    - tworzenie, 412
    - zmienne przechwycone, 196
  - podstawowe, 91
  - przypisania, 91
  - puste, 91
  - regularne, 969
    - alternatywy, 987
  - asercje o zerowej wielkości, 975, 986
  - dzielenie tekstu, 980
  - granica słowa, 978
  - grupy, 978
  - kategorie znaków, 985
  - kompilowane, 971
  - konstrukcje grupujące, 987
  - konstrukcje różne, 987
  - kotwice, 977
  - kwantyfikatory leniwy, 975
  - kwantyfikatory zachłanny, 975
  - kwantyfikatory, 974, 986
  - odwołania wsteczne, 987
  - opcje, 972, 987
  - przewidywanie, 976
  - przewidywanie wsteczne, 976
  - receptury, 981
  - zastąpienia, 986
  - zastępowanie tekstu, 980
  - zbiory znaków, 985
  - zestawy znaków, 973
  - znaki sterujące, 972, 985
- typy statyczne, 256
- zapytań, 18, 409, 415, 428, 450
- wyrażenie kolekcyjne, collection expression, 210, 939
- wywołania
  - dynamiczne, 255
  - zwrotne z delegatami, 953
  - zwrotne ze wskaźnikami do funkcji, 951
- wywołanie komponentu COM, 963
  - argumenty nazwane, 964
  - indeksatory, 965
  - niejawne parametry ref, 964
  - parametry opcjonalne, 964
  - wiązanie dynamiczne, 965
- wywoływanie
  - niejawne konstruktora, 144
  - składowych, 784
- wzorce, 241
  - asynchroniczności, 659
    - informacje o postępie, 661
    - łącznik zadań, 663
    - oparte na zadaniu, 663
    - przerwanie operacji, 659
  - kombinatory, 242
  - krotek, 243
  - list, 245
  - pozycyjne, 243
  - relacyjne, 242
  - własności, 244
- wzorzec
  - asynchroniczności opartej na zdarzeniach, EAP, 668
  - asynchroniczny oparty na zadaniu, TAP, 663

leniwa ewaluacja, 236  
metod TryXXX, 208  
Model-Widok-Kontroler, 286  
projektowy Wizytator, 831  
stałej, 241  
var, 243  
zdarzeń, 190

## X

Xamarin, *Patrz* MAUI  
X-DOM, 503  
adnotacje, 528  
definiowanie treści, 508  
dokumenty i deklaracje, 519  
domyślne przestrzenie nazw, 526  
głębokie klonowanie, 508  
konstrukcja funkcyjna, 507  
modyfikowanie, 514  
nawigowanie, 509  
praca z wartościami, 517  
przedrostki, 527  
przestrzenie nazw, 525  
tworzenie drzewa, 506  
wysyłanie zapytań, 509  
XML, 282  
XmlReader  
odczytywanie elementów, 535  
przedrostki, 540  
przestrzenie nazw, 540  
wczytywanie atrybutów, 539  
wczytywanie węzłów, 534  
zastosowania, 543  
XmlWriter  
przedrostki, 542  
przestrzenie nazw, 542  
wpisywanie atrybutów, 542  
wpisywanie typów węzłów, 542  
zastosowania, 543

## Z

zadania, 626  
autonomiczne, 629  
klasa TaskCompletionSource, 632  
kontynuacje, 630  
potomne, 919  
uruchomienie, 627  
wartość zwrotna, 628  
wyjątki, 629  
zakleszczenia, 861  
zależności cykliczne, 822

zapytania  
interpretowane, 434, 436  
LINQ, 407  
złożone, 428  
zarządzanie  
kluczami, 851  
pamięcią, 19  
zasoby, 750  
osadzanie w zestawach, 751  
pliki .resources, 752  
pliki .resx, 753  
tworzenie, 754  
zbiory, 376  
zdarzenia, 18, 188  
metody dostępne, 193  
modyfikatory, 194  
niestandardowe, 607  
standardowy wzorzec, 190  
systemu plików, 708  
zegary, 891  
jednowątkowe, 894  
wielowątkowe, 892  
zestaw, assembly, 20, 740  
znaków, 299  
zestawy, 283  
emitowanie, 811  
izolowanie, 757  
konteksty ładowania, 759  
ładowanie, 757, 759, 768  
manifest, 741  
manifest aplikacji, 742  
moduły, 743  
nazwy, 745  
osadzanie zasobów, 751  
podpisywanie, 744, 749  
referencyjne, 280  
satelickie, 750, 755  
silne nazwy, 744  
użycie Authenticode, 749  
wersja informacyjna, 747  
wersja pliku, 747  
zasoby, 750  
znajdowanie, 757, 761, 768  
złączenia, 475  
płaskie zewnętrzne, 479  
w składni płynnej, 478  
z widokami wyszukiwania, 480  
zewnętrzne, 473  
zmiennne, 51, 80  
iteracyjne, 199  
lokalne, 46, 89, 97  
przechwycone, 196  
typu referencyjnego, 58

typu wartościowego, 57  
wyjściowe, 85  
wzorcowe, 140  
zakresowe, 417  
zewnętrzne, 196  
znaczniki  
dokumentacyjne XML, 274  
niestandardowe, 275  
znak ?, 214, 969  
znaki interpunkcyjne, 50  
zrzut, dump, 608  
rdzenia, core dump, 608  
zwalnianie  
zasobów, 560–563





# PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

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

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

GRUPA  
**Helion**

## Książka zawiera opisy najnowszych technik programowania w języku C#, przydatnych i dla nowicjuszy, i dla profesjonalistów!

— Eric Lippert, Komisja Standaryzacyjna do spraw Języka C#

Eksperti uważają język C# za flagowy produkt firmy Microsoft udostępniający zarówno wysoko-poziomowe abstrakcje, jak i niskopoziomowe mechanizmy, które pozwalają uzyskać maksymalną wydajność aplikacji. Wersja oznaczona numerem 12 wprowadza kilka istotnych usprawnień i nowych funkcji, które mogą znacząco wpłynąć na sposób pisania kodu. Chociaż niektóre z nich mogłyby się wydawać drobnymi usprawnieniami, ich skumulowany efekt znacząco poprawia jakość kodu i produktywność programisty. Tych nowości warto się nauczyć, ponieważ nagrodą za poświęcony czas jest przyjemność płynąca z tworzenia znakomitych aplikacji.

Oto przejrzane i zaktualizowane wydanie doskonałego podręcznika dla programistów. Jak wszystkie pozycje z serii „...w pigułce”, stanowi najlepsze jednotomowe źródło praktycznej wiedzy. Znalazły się tu zwięzłe i dokładne informacje na temat języka C#, Common Language Runtime (CLR) i biblioteki klas .NET 8 Base Class Library (BCL). Nowe składniki języka C# 12 i związanej z nim platformy specjalnie wyróżniono, dzięki czemu książka może służyć także jako podręcznik do nauki C# 10 i C# 11 i pozwoli Ci błyskawicznie uzupełnić wiedzę o aktualne zagadnienia. Znalazły się tu precyzyjne opisy pojęć i przypadków użycia z naciskiem na praktyczność zastosowań. Dzięki temu jest to doskonała pomoc w codziennej pracy programisty C#.

### W książce między innymi:

- składnia C#, a także wskaźniki, rekordy, domknięcia i wzorce
- tajniki technologii LINQ
- programowanie współbieżne i asynchroniczne
- wątki i programowanie równoległe
- narzędzia .NET: wyrażenia regularne, struktury Span, kryptografia i reflection.emit

**Joseph Albahari** — architekt i programista, autor cenionych książek o C#, takich jak *C# 10 w pigułce* i *C# 12. Leksykon kieszonkowy*. Napisał też popularny program dla programistów do roboczego wypróbowywania zapytań LINQ — LINQPad.

**To jedna z nielicznych książek, które trzymam cały czas na biurku!**

— **Scott Guthrie**, Microsoft

	<b>KOD KORZYŚCI</b> Sięgnij po więcej! ▶	
 <a href="http://helion.pl">helion.pl</a>	ISBN 978-83-289-1483-4	
 <b>HELION S.A.</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788328 914834	
<b>Cena: 179,00 zł</b>		