

ZŁOTE  
MYSŁI

Andrzej Stefańczyk

# Sekrety języka C#

**Dlaczego tworzenie  
aplikacji w  
Visual Studio .NET  
2005 jest takie  
proste?**

© Copyright for Polish edition by [ZloteMysli.pl](http://ZloteMysli.pl)

Data: 29.06.2005

Tytuł: „Sekrety języka C#”

Autor: Andrzej Stefańczyk

Wydanie I

ISBN: 83-60237-90-5

Projekt okładki: Marzena Osuchowicz

Skład: Anna Grabka

Internetowe Wydawnictwo Złote Myśli

„Złote Myśli” s.c.

ul. Plebiscytowa 1

44-100 Gliwice

WWW: [www.ZloteMysli.pl](http://www.ZloteMysli.pl)

EMAIL: [kontakt@zlotemysli.pl](mailto:kontakt@zlotemysli.pl)

**Wszelkie prawa zastrzeżone.**

**All rights reserved.**

# SPIS TREŚCI

<b><u>OD AUTORA</u></b>	<b>9</b>
<b><u>CZEŚĆ I PODSTAWY C#</u></b>	<b>10</b>
<b><u>ROZDZIAŁ 1. PRZEGLĄD PLATFORMY MICROSOFT.NET</u></b>	<b>10</b>
<u>WPROWADZENIE DO PLATFORMY .NET</u>	10
<u>PRZEGLĄD FRAMEWORK .NET</u>	10
<u>WSPÓLNE ŚRODOWISKO URUCHOMIENIOWE</u>	12
<u>BIBLIOTEKA KLAS .NET FRAMEWORK</u>	13
<u>OBSŁUGA BAZ DANYCH (ADO.NET)</u>	15
<u>USŁUGI WEBOWE (XML WEB SERVICES)</u>	16
<u>APLIKACJE WEBOWE (WEB FORMS)</u>	16
<u>APLIKACJE OKIENKOWE (WINDOWS FORMS)</u>	16
<u>WSPÓLNA SPECYFIKACJA JĘZYKOWA (CLS)</u>	18
<u>JĘZYKI PROGRAMOWANIA W .NET FRAMEWORK</u>	18
<b><u>ROZDZIAŁ 2. PIERWSZY PROGRAM</u></b>	<b>20</b>
<u>HELLO WORLD</u>	20
<u>WEJŚCIE-WYJŚCIE</u>	21
<u>KOMPILACJA I URUCHOMIENIE</u>	23
<u>KOMENTOWANIE KODU</u>	24
<b><u>ROZDZIAŁ 3. WPROWADZENIE DO MICROSOFT VISUAL C#.NET 2005</u></b>	<b>26</b>
<u>ŚRODOWISKO PROGRAMISTY</u>	26
<u>OKNA NARZĘDZIOWE</u>	27
<u>GENEROWANIE SZABLONÓW PROJEKTÓW</u>	32
<u>Generowanie szablonu aplikacji konsolowej</u>	33
<u>KOMPILACJA I URUCHOMIENIE</u>	34
<b><u>ROZDZIAŁ 4. TYPY</u></b>	<b>36</b>
<u>DEKLARACJA ZMIENNEJ</u>	36
<u>INICJACJA ZMIENNEJ</u>	37
<u>SŁOWA KLUCZOWE</u>	37
<u>TYPY WARTOŚCI</u>	38
<u>Typy proste</u>	38
<u>Typ wyliczeniowy</u>	39
<u>Struktura</u>	41
<u>TYPY REFERENCYJNE</u>	41
<u>Typ object</u>	42
<u>Typ string</u>	42
<u>Tablica</u>	42
<u>Klasa</u>	42
<u>Interfejs</u>	43
<u>Delegacja</u>	43
<u>STAŁE</u>	43
<u>LITERAŁY</u>	44
<u>KONWERSJE</u>	46
<u>OPAKOWYWANIE I ROZPAKOWYWANIE</u>	48
<b><u>ROZDZIAŁ 5. OPERATORY I WYRAŻENIA</u></b>	<b>50</b>
<u>WYRAŻENIA</u>	50
<u>OPERATORY</u>	51
<u>Operatory arytmetyczne</u>	51
<u>Operatory logiczne bitowe</u>	52
<u>Operatory logiczne warunkowe</u>	54
<u>Operator konkatencji</u>	54
<u>Operatory jednostkowego zmniejszania i zwiększania</u>	54
<u>Operatory przesunięcia</u>	55
<u>Operatory relacji</u>	56
<u>Operatory przypisania</u>	56
<u>Operator dostępu do składnika klasy</u>	57
<u>Operator wyrażenia warunkowego ?:</u>	57
<u>Operator is</u>	58
<u>Operator as</u>	59
<u>ZNAKI IGNOROWANE W WYRAŻENIACH</u>	60
<b><u>ROZDZIAŁ 6. INSTRUKCJE STERUJĄCE</u></b>	<b>61</b>

<a href="#"><u>WPROWADZENIE</u></a>	61
<a href="#"><u>INSTRUKCJA PUSTA</u></a>	61
<a href="#"><u>BLOK INSTRUKCJI</u></a>	61
<a href="#"><u>INSTRUKCJE WYBORU</u></a>	62
<b><a href="#"><u>Instrukcja if</u></a></b>	62
<a href="#"><u>Klauzula else</u></a>	63
<a href="#"><u>Układanie warunków</u></a>	63
<a href="#"><u>Kaskadowe łączenie warunków</u></a>	64
<a href="#"><u>Zagnieżdżanie warunków</u></a>	65
<a href="#"><u>Zastępowanie wyrażeniami warunkowymi</u></a>	66
<b><a href="#"><u>Instrukcja switch</u></a></b>	66
<a href="#"><u>Zasady stosowania</u></a>	67
<a href="#"><u>Zastępowanie instrukcji warunkowych</u></a>	71
<a href="#"><u>INSTRUKCJE ITERACYJNE</u></a>	73
<b><a href="#"><u>Instrukcja while</u></a></b>	73
<a href="#"><u>Warunki zakończenia</u></a>	74
<a href="#"><u>Zagnieżdżanie pętli</u></a>	75
<b><a href="#"><u>Instrukcja do</u></a></b>	76
<a href="#"><u>Warunki zakończenia</u></a>	76
<a href="#"><u>Zagnieżdżanie pętli</u></a>	77
<b><a href="#"><u>Instrukcja for</u></a></b>	78
<a href="#"><u>Warunki zakończenia</u></a>	78
<a href="#"><u>Deklaracja liczników</u></a>	79
<a href="#"><u>Zagnieżdżanie pętli</u></a>	80
<a href="#"><u>Zastępowanie innymi pętlami</u></a>	80
<b><a href="#"><u>Instrukcja foreach</u></a></b>	81
<a href="#"><u>INSTRUKCJE SKOKU</u></a>	81
<b><a href="#"><u>Instrukcja goto</u></a></b>	82
<b><a href="#"><u>Instrukcja break</u></a></b>	83
<b><a href="#"><u>Instrukcja continue</u></a></b>	83
<b><a href="#"><u>ROZDZIAŁ 7. KLASY I OBIEKTY</u></a></b>	85
<a href="#"><u>PODSTAWOWE POJĘCIA</u></a>	85
<b><a href="#"><u>Klasa i obiekt</u></a></b>	85
<b><a href="#"><u>Relacje</u></a></b>	85
<b><a href="#"><u>Hermetyzacja</u></a></b>	85
<b><a href="#"><u>Abstrakcja</u></a></b>	85
<b><a href="#"><u>Kompozycja i dekompozycja</u></a></b>	86
<b><a href="#"><u>Składnik klasy</u></a></b>	86
<b><a href="#"><u>Składnica</u></a></b>	86
<a href="#"><u>DEFINIOWANIE KLAS</u></a>	86
<a href="#"><u>MODYFIKATORY</u></a>	87
<b><a href="#"><u>Modifikatory dostępu</u></a></b>	87
<a href="#"><u>TWORZENIE OBIEKTU KLASY</u></a>	88
<a href="#"><u>POLA</u></a>	89
<a href="#"><u>KONSTRUKTOR</u></a>	90
<b><a href="#"><u>Konstruktor domyślny</u></a></b>	90
<b><a href="#"><u>Inicjacja pól</u></a></b>	91
<b><a href="#"><u>Lista inicjacyjna</u></a></b>	95
<b><a href="#"><u>Konstruktor kopiujący</u></a></b>	95
<a href="#"><u>NISZCZENIE OBIEKTU KLASY</u></a>	96
<a href="#"><u>DESTRUKTOR</u></a>	96
<a href="#"><u>SŁOWO KLUCZOWE THIS</u></a>	97
<a href="#"><u>METODY KLASY</u></a>	97
<b><a href="#"><u>Definiowanie</u></a></b>	98
<b><a href="#"><u>Zwracanie wartości</u></a></b>	99
<b><a href="#"><u>Argumenty</u></a></b>	101
<a href="#"><u>Argumenty przekazywane przez wartość</u></a>	102
<a href="#"><u>Argumenty przekazywane przez referencje</u></a>	102
<a href="#"><u>Argumenty przekazywane przez wyjście</u></a>	103
<a href="#"><u>Lista argumentów o zmiennej długości</u></a>	103
<b><a href="#"><u>Wywoływanie</u></a></b>	104
<a href="#"><u>Rekurencja</u></a>	106

<a href="#"><u>Przeciążanie</u></a>	106
<a href="#"><u>STATYCZNE SKŁADNIKI KLASY</u></a>	108
<a href="#"><u>PRZECIĄŻANIE OPERATORÓW</u></a>	110
<a href="#"><u>Przeciążanie operatorów relacji</u></a>	111
<a href="#"><u>Przeciążanie operatorów logicznych</u></a>	113
<a href="#"><u>Przeciążanie operatorów konwersji</u></a>	114
<a href="#"><u>Przeciążanie operatorów arytmetycznych</u></a>	115
<a href="#"><u>WŁAŚCIWOŚCI</u></a>	116
<a href="#"><u>INDEKSATORY</u></a>	119
<a href="#"><u>DELEGACJE</u></a>	123
<a href="#"><u>ZDARZENIA</u></a>	124
<a href="#"><u>DZIEDZICZENIE</u></a>	126
<a href="#"><u>Dostęp do składowych klasy bazowej</u></a>	127
<a href="#"><u>Wywoływanie bazowych konstruktorów</u></a>	129
<a href="#"><u>Przesłanie metod</u></a>	131
<a href="#"><u>Ukrywanie metod</u></a>	135
<a href="#"><u>Klasy ostateczne</u></a>	138
<a href="#"><u>Klasy abstrakcyjne</u></a>	139
<a href="#"><u>Bazowa klasa System.Object</u></a>	142
<b><a href="#"><u>ROZDZIAŁ 8. STRUKTURY</u></a></b>	<b>143</b>
<a href="#"><u>DEFINIOWANIE STRUKTUR</u></a>	143
<a href="#"><u>PORÓWNANIE Z KLASAMI</u></a>	143
<a href="#"><u>GRUPOWANIE PÓŁ</u></a>	145
<b><a href="#"><u>ROZDZIAŁ 9. INTERFEJSY</u></a></b>	<b>146</b>
<a href="#"><u>DEFINIOWANIE INTERFEJSÓW</u></a>	146
<a href="#"><u>IMPLEMENTACJA INTERFEJSÓW</u></a>	147
<a href="#"><u>Implementacja metod interfejsów</u></a>	148
<a href="#"><u>Jawna implementacja metod interfejsów</u></a>	150
<a href="#"><u>INTERFEJS IDISPOSABLE</u></a>	153
<b><a href="#"><u>ROZDZIAŁ 10. WYJĄTKI</u></a></b>	<b>155</b>
<a href="#"><u>MECHANIZM WYJĄTKÓW</u></a>	155
<a href="#"><u>BLOKI TRY I CATCH</u></a>	155
<a href="#"><u>KLASY WYJĄTKÓW</u></a>	157
<a href="#"><u>RZUCANIE WYJĄTKÓW</u></a>	159
<a href="#"><u>BLOK FINALLY</u></a>	161
<a href="#"><u>PRZEPŁNIENIA ARYTMETYCZNE</u></a>	162
<b><a href="#"><u>ROZDZIAŁ 11. PRZESTRZENIE NAZW</u></a></b>	<b>164</b>
<a href="#"><u>DEKLAROWANIE PRZESTRZENI NAZW</u></a>	164
<a href="#"><u>NAZWY KWALIFIKOWANE</u></a>	166
<a href="#"><u>DYREKTYWA USING</u></a>	167
<a href="#"><u>TWORZENIE ALIASÓW</u></a>	169
<b><a href="#"><u>ROZDZIAŁ 12. TABLICE</u></a></b>	<b>172</b>
<a href="#"><u>DEKLARACJA TABLIC</u></a>	172
<a href="#"><u>WYMIARY TABLIC</u></a>	172
<a href="#"><u>TWORZENIE INSTANCIJ TABLIC</u></a>	173
<a href="#"><u>DOSTĘP DO ELEMENTÓW</u></a>	173
<a href="#"><u>INICJACJA ELEMENTÓW TABLIC</u></a>	174
<a href="#"><u>WŁAŚCIWOŚCI TABLIC</u></a>	175
<a href="#"><u>METODY OPERUJĄCE NA TABLICACH</u></a>	175
<a href="#"><u>ZWRACANIE TABLIC Z METOD</u></a>	177
<a href="#"><u>PRZEKAZYWANIE TABLIC DO METOD</u></a>	178
<a href="#"><u>TABLICA ARGUMENTÓW MAIN</u></a>	179
<b><a href="#"><u>ROZDZIAŁ 13. ŁAŃCUCHY</u></a></b>	<b>180</b>
<a href="#"><u>KLASA STRING</u></a>	180
<a href="#"><u>POLA, WŁAŚCIWOŚCI I METODY KLASY STRING</u></a>	180
<a href="#"><u>BUDOWANIE ŁAŃCUCHÓW – KLASA STRINGBUILDER</u></a>	185
<b><a href="#"><u>ROZDZIAŁ 14. KOLEKCJE</u></a></b>	<b>187</b>
<a href="#"><u>WPROWADZENIE</u></a>	187
<a href="#"><u>KLASA ARRAYLIST</u></a>	187
<a href="#"><u>KLASA BITARRAY</u></a>	191
<a href="#"><u>KLASA HASHTABLE</u></a>	193
<a href="#"><u>KLASA QUEUE</u></a>	195

<a href="#">KLASA SORTEDLIST</a>	197
<a href="#">KLASA STACK</a>	200
<b><a href="#">ROZDZIAŁ 15. DATA I CZAS</a></b>	<b>203</b>
<b><a href="#">ROZDZIAŁ 16. FOLDERY I PLIKI</a></b>	<b>210</b>
<a href="#">WPROWADZENIE</a>	210
<a href="#">KLASA DIRECTORY</a>	210
<a href="#">KLASA DIRECTORYINFO</a>	213
<a href="#">KLASA FILE</a>	216
<a href="#">KLASA FILEINFO</a>	221
<a href="#">KLASA FILESTREAM</a>	225
<a href="#">KLASA STREAMREADER</a>	227
<a href="#">KLASA STREAMWRITER</a>	228
<a href="#">KLASA BINARYREADER</a>	230
<a href="#">KLASA BINARYWRITER</a>	232
<a href="#">KLASA PATH</a>	233
<b><a href="#">ROZDZIAŁ 17. DEBUGOWANIE</a></b>	<b>235</b>
<a href="#">WPROWADZENIE</a>	235
<a href="#">PUŁAPKI I ŚLEDZENIE KROKOWE</a>	235
<a href="#">OKNA ŚLEDZENIA</a>	236
<b><a href="#">CZĘŚĆ II. TWORZENIE APLIKACJI OKIENKOWYCH</a></b>	<b>237</b>
<b><a href="#">ROZDZIAŁ 1. PODSTAWY WINDOWS FORMS</a></b>	<b>237</b>
<a href="#">WPROWADZENIE</a>	237
<a href="#">GENEROWANIE APLIKACJI WINDOWS FORMS</a>	237
<b><a href="#">ROZDZIAŁ 2. PRACA Z FORMĄ</a></b>	<b>239</b>
<a href="#">TWORZENIE FORMY</a>	239
<a href="#">WŁAŚCIWOŚCI FORMY</a>	240
<a href="#">OBSŁUGA ZDARZEŃ</a>	242
<a href="#">METODY FORMY</a>	243
<a href="#">PRZYKŁADOWA APLIKACJA</a>	243
<b><a href="#">ROZDZIAŁ 3. KORZYSTANIE Z PROSTYCH KONTROLEK</a></b>	<b>245</b>
<a href="#">DODAWANIE KONTROLEK DO FORMY</a>	245
<a href="#">ORGANIZOWANIE KONTROLEK NA FORMIE</a>	245
<a href="#">WSPÓLNE CECHY KONTROLEK</a>	246
<a href="#">Właściwości</a>	246
<a href="#">Zdarzenia</a>	247
<a href="#">Metody</a>	248
<a href="#">ETYKIETA TEKSTOWA</a>	248
<a href="#">Właściwości</a>	249
<a href="#">ETYKIETA ŁĄCZA</a>	249
<a href="#">Właściwości</a>	249
<a href="#">Zdarzenia</a>	250
<a href="#">PRZYCISK</a>	250
<a href="#">Właściwości</a>	251
<a href="#">Zdarzenia</a>	252
<a href="#">Przykładowa aplikacja</a>	252
<a href="#">PRZYCISK RADIOWY</a>	252
<a href="#">Właściwości</a>	253
<a href="#">Przykładowa aplikacja</a>	253
<a href="#">PRZYCISK SELEKCJI</a>	254
<a href="#">Właściwości</a>	254
<a href="#">Przykładowa aplikacja</a>	255
<a href="#">POLE TEKSTOWE</a>	257
<a href="#">Właściwości</a>	257
<a href="#">Przykładowa aplikacja</a>	258
<a href="#">POLE TEKSTOWE Z WZORCEM</a>	259
<a href="#">Właściwości</a>	259
<a href="#">Zdarzenia</a>	261
<a href="#">Przykładowa aplikacja</a>	261
<a href="#">LISTA PROSTA</a>	261
<a href="#">Właściwości</a>	261
<a href="#">Zdarzenia</a>	262
<a href="#">Kolekcja elementów</a>	262

<a href="#">Przykładowa aplikacja</a>	263
<a href="#">LISTA SELEKCJI</a>	264
<a href="#">Właściwości</a>	264
<a href="#">Zdarzenia</a>	265
<a href="#">Przykładowa aplikacja</a>	265
<a href="#">LISTA ROZWIJANA</a>	266
<a href="#">Właściwości</a>	266
<a href="#">Zdarzenia</a>	267
<a href="#">Przykładowa aplikacja</a>	267
<a href="#">POLE GRUPUJĄCE</a>	268
<a href="#">Właściwości</a>	268
<a href="#">Przykładowa aplikacja</a>	269
<a href="#">POLE OBRAZKOWE</a>	271
<a href="#">Właściwości</a>	271
<a href="#">Zdarzenia</a>	272
<a href="#">Przykładowa aplikacja</a>	272
<a href="#">PANEL</a>	274
<a href="#">Właściwości</a>	274
<a href="#">Przykładowa aplikacja</a>	274
<a href="#">PASEK POSTĘPU</a>	275
<a href="#">Właściwości</a>	275
<a href="#">Przykładowa aplikacja</a>	275
<a href="#">SUWAK</a>	276
<a href="#">Właściwości</a>	276
<a href="#">Zdarzenia</a>	277
<a href="#">Przykładowa aplikacja</a>	277
<a href="#">KALENDARZ</a>	278
<a href="#">Właściwości</a>	278
<a href="#">Zdarzenia</a>	278
<a href="#">Przykładowa aplikacja</a>	278
<a href="#">POLE NUMERYCZNE</a>	279
<a href="#">Właściwości</a>	279
<a href="#">Zdarzenia</a>	280
<a href="#">Przykładowa aplikacja</a>	280
<a href="#">LISTA OBRAZÓW</a>	282
<a href="#">Właściwości</a>	282
<a href="#">Kolekcja obrazów</a>	282
<b><a href="#">ROZDZIAŁ 4. KORZYSTANIE Z ZAAWANSOWANYCH KONTROLEK</a></b>	<b>284</b>
<a href="#">ZAKŁADKI</a>	284
<a href="#">Właściwości</a>	284
<a href="#">Zdarzenia</a>	285
<a href="#">Kolekcja stron</a>	285
<a href="#">Przykładowa aplikacja</a>	285
<a href="#">DRZEWO</a>	287
<a href="#">Właściwości</a>	287
<a href="#">Zdarzenia</a>	288
<a href="#">Kolekcja elementów drzewa</a>	289
<a href="#">Przykładowa aplikacja</a>	289
<a href="#">LISTA ZŁOŻONA</a>	291
<a href="#">Właściwości</a>	291
<a href="#">Zdarzenia</a>	293
<a href="#">Kolumny w widoku szczegółowym</a>	294
<a href="#">Kolekcja elementów listy</a>	294
<a href="#">Przykładowa aplikacja</a>	295
<a href="#">KONTENER PODZIELNIKA OBSZARÓW</a>	296
<a href="#">Właściwości</a>	296
<a href="#">Zdarzenia</a>	297
<a href="#">Przykładowa aplikacja</a>	297
<a href="#">CZASOMIERZ</a>	300
<a href="#">Właściwości</a>	300
<a href="#">Zdarzenia</a>	300
<a href="#">Przykładowa aplikacja</a>	300

<b><u>ROZDZIAŁ 5. INTERAKCJA Z UŻYTKOWNIKIEM</u></b>	<b>301</b>
<u>WSPÓŁPRACA Z MYSZĄ</u>	301
<u>Parametry zdarzenia</u>	301
<u>Przykładowa aplikacja</u>	301
<u>WSPÓŁPRACA Z KŁAWIATURĄ</u>	302
<u>Parametry zdarzenia</u>	302
<u>Przykładowa aplikacja</u>	303
<u>KORZYSTANIE Z MENU, PASKA NARZĘDZI I PASKA STANU</u>	304
<u>Właściwości</u>	304
<u>Zdarzenia</u>	305
<u>Kolekcja elementów</u>	305
<u>Zarządzanie kolekcją elementów</u>	306
<u>Przykładowa aplikacja</u>	307
<b><u>ROZDZIAŁ 6. KORZYSTANIE Z OKIEN DIALOGOWYCH</u></b>	<b>309</b>
<u>TWORZENIE OKIEN DIALOGOWYCH</u>	309
<u>Przykładowa aplikacja</u>	309
<u>WSPÓLNE OKNA DIALOGOWE</u>	310
<u>Okno wyboru pliku</u>	310
<u>Właściwości</u>	310
<u>Zdarzenia</u>	311
<u>Okno wyboru folderu</u>	311
<u>Właściwości</u>	312
<u>Okno wyboru koloru</u>	312
<u>Właściwości</u>	312
<u>Zdarzenia</u>	312
<u>Okno wyboru czcionki</u>	312
<u>Właściwości</u>	312
<u>Zdarzenia</u>	313
<u>Przykładowa aplikacja</u>	313
<b><u>ROZDZIAŁ 7. TWORZENIE APLIKACJI MDI</u></b>	<b>317</b>
<u>TWORZENIE APLIKACJI MDI</u>	317
<u>PRZYKŁADOWA APLIKACJA</u>	318
<b><u>ŹRÓDŁA</u></b>	<b>322</b>



## Od autora

Pisząc tą książkę starałem się przekazywać wiedzę stopniowo krok po kroku od prostych do bardziej złożonych zagadnień.

Opierając się na tej zasadzie postanowiłem podzielić książkę na dwie odrębne i różniące się nieco konwencją części.

Pierwsza część przedstawia składnię języka C# prostym i zrozumiałym dla każdego językiem z dużą ilością przykładów, wykorzystujących omówione w danym rozdziale elementy składni języka.

Druga część książki pokazuje w jaki sposób tworzy się aplikacje okienkowe *Windows Forms*, opisując najważniejsze komponenty *.NET Framework 2.0*. Zdecydowałem się opisać elementy najnowszej wersji *.NET Framework* ze względu na duże zmiany, jakie wprowadzono w stosunku do wersji poprzednich. To samo dotyczy nowej wersji środowiska *Microsoft Visual Studio .NET 2005* (w trakcie pisania książki zarówno *.NET Framework 2.0* jak nowe środowisko *Microsoft Visual Studio .NET 2005* dostępne były w wersji Beta).

Drogi czytelniku, mam nadzieję, że książka, którą napisałem pomoże Ci w poszerzaniu Twojej wiedzy z dziedziny programowania. Życzę Ci zatem wielu sukcesów i miłej lektury.

Pozdrawiam,

Andrzej Stefańczyk



# Część I

## Podstawy C#

### Rozdział 1.

## Przegląd platformy Microsoft.NET

### Wprowadzenie do platformy .NET

Zanim zaczniemy naszą przygodę z językiem C# powinniśmy przyjrzeć się platformie, w jakiej uruchamiane są aplikacje stworzone w tym języku. Przed pojawieniem się platformy .NET, programista korzystający ze środowisk programistycznych *Microsoft* zmuszony był do korzystania z funkcji *Windows API* lub też nie do końca dobrze przemyślanych klas przysłaniających te funkcje. Ze względu na dużą złożoność i coraz liczniej pojawiające się błędy spowodowane seriami poprawek i rozszerzeń, *Microsoft* zdecydował się na całkowitą zmianę koncepcji tworzenia aplikacji.

Nowe podejście polega przede wszystkim na zmniejszeniu liczby problemów, z jakimi musi zmagać się programista w czasie żmudnego procesu tworzenia. Do tej pory wiele problemów nastroczało poprawne zarządzanie pamięcią, zapewnianie przenośności kodu między różnymi językami programowania, poprawna orientacja w ogromnej liczbie funkcji *API*, obsługa błędów oraz brak mechanizmów kontroli. Wraz z pojawieniem się *platformy .NET* powyższe problemy przestały istnieć, dzięki czemu programista może skupić się nad tym, co ważne, czyli logiką aplikacji.

Platforma .NET to coś więcej niż środowisko do tworzenia aplikacji, to ogromny zbiór języków programowania funkcjonujących we wspólnym środowisku (*Visual Basic*, *Visual C++*, *Visual C#*, *Visual J#*), usług oferowanych przez serwery *.NET Enterprise* (*Microsoft Exchange Server*, *Microsoft SQL Server*, *Microsoft BizTalk Server*, *Microsoft Commerce Server*, itd.), usług dystrybuowanych (usługi dostępne przez Internet, wykorzystujące *XML* oraz *SOAP*) oraz usług dla urządzeń przenośnych (palmtopów, telefonów komórkowych, konsol, itp.).

### Przegląd Framework .NET

Framework .NET jest pakietem komponentów do budowy i uruchamiania aplikacji opartych na technologii .NET. Został on zaprojektowany w taki sposób, aby:

- Zapewniał zgodność z istniejącymi standardami.

Wsparcie dla istniejących technologii takich, jak: HTML, XML, SOAP, XSLT, XPath

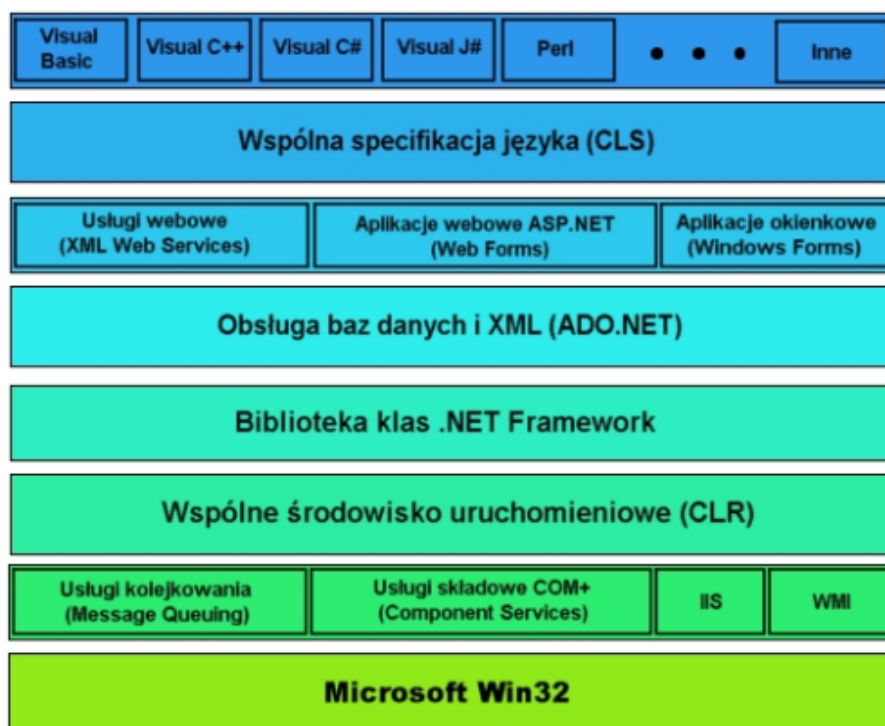
- Był prosty w użyciu.

Kod zorganizowany jest hierarchicznie poprzez przestrzenie nazw oraz klasy. Istnieje wspólna baza typów dla wszystkich języków. Każdy element języka jest obiektem.

- Pozwalał na rozszerzanie istniejącego zbioru klas.

Hierarchia przestrzeni nazw i klas nie jest ukryta. Programista może rozszerzać funkcjonalność każdej klasy poprzez mechanizm dziedziczenia (wyjątek stanowią jedynie klasy typu sealed).

- Zapewniał taką samą funkcjonalność niezależnie od języka programowania.



Rysunek 1. Architektura .NET Framework

Przyjrzyjmy się bliżej powyższemu rysunkowi. Na samym dole znajduje się system operacyjny, na którym działa framework.

Dalej mamy usługi aplikacyjne (dostępne poprzez klasy biblioteki klas): **Message Queuing** – kolejkowanie wiadomości, **Component Services** – usługi składowe (**COM+**), **IIS** (Internet Information Server) oraz **WMI** (Windows Management Instrumentation).

Kolejna warstwa to wspólne środowisko uruchomieniowe **CLR** (Common Language Runtime), które upraszcza proces tworzenia aplikacji, poprzez zapewnienie odseparowanego i zabezpieczonego środowiska uruchomieniowego, obsługę wielu języków oraz mechanizmów dystrybucji i zarządzania aplikacjami.

Następna warstwa to biblioteka klas, która zapewnia funkcjonalność (w postaci licznych zbiorów klas) niezbędną do implementacji każdej aplikacji.

Kolejna warstwa to **ADO.NET**, zapewniająca szeroki dostęp do baz danych oraz wsparcie dla standardu **XML**.

Następna warstwa zawiera:

- **Web services** (usługi webowe, czyli komponenty, które mogą być współdzielone poprzez Internet),
- **Web forms** (aplikacje webowe, czyli oparte o **ASP.NET** aplikacje dostępne poprzez dynamicznie zmieniające się webowe interfejsy użytkownika),
- **Windows Forms** (aplikacje okienkowe, czyli aplikacje klienckie systemu Windows).

Kolejna warstwa to wspólna specyfikacja językowa **CLS** (Common Language Specification), czyli ujednolicony zbiór reguł dla każdego z dostępnych języków platformy .NET.

Ostatnia warstwa zawiera zbiór aktualnie dostępnych języków programowania platformy .NET.

## Wspólne środowisko uruchomieniowe

Wspólne środowisko uruchomieniowe **CLR** (Common Language Runtime) znacznie ułatwia proces tworzenia aplikacji poprzez zapewnienie usług nadzorujących proces wykonywania kodu.

Aby możliwe było wykorzystanie usług wspólnego środowiska uruchomieniowego, kod należy skompilować przy pomocy współpracującego ze środowiskiem kompilatora. Kod wykonywalny przygotowany pod **CLR** nazywa się „*kodem nadzorowanym*”.

Wspólne środowisko uruchomieniowe zostało zaprojektowane z myślą o zintegrowaniu wielu języków programowania. Dzięki temu można przykładowo odziedziczyć w jednym języku klasę, która została napisana w innym języku. Poza tym, środowisko zapewnia mechanizm zarządzania pamięcią, więc programista nie musi dłużej przejmować się tym, że zapomni zwolnić pamięć. Inną zaletą środowiska jest mechanizm kontroli wersji, który dba o to, aby do aplikacji dołączono wszystkie potrzebne komponenty. Środowisko posiada również jednolity mechanizm obsługi wyjątków (współpracujący z różnymi językami) oraz mechanizm kontroli typów danych.

Aby wykorzystać wszystkie mechanizmy środowiska, kompilator z nim współpracujący oprócz kodu nadzorowanego, musi przygotować tzw. *metadane*. Metadane służą do opisu typów danych używanych w kodzie programu i są przechowywane wraz z kodem w przenośnym pliku wykonywalnym (w skrócie **PE** – Portable Executable). Kod nadzorowany, o którym wspominałem, nie jest jednak kodem maszynowym, tylko kodem pośrednim zapisanym przy pomocy *języka pośredniego* (**IL** – Intermediate Language).

Język IL jest całkowicie niezależny od procesora, na którym zostanie wykonany kod, więc aby możliwe było jego wykonanie na środowisku docelowym musi istnieć kompilator, który przekształci kod IL w kod maszynowy. Kompilator przekształcający kod IL w kod maszynowy nazywany jest kompilatorem **JIT** (Just In Time).

Przyjrzyjmy się teraz elementom środowiska CLR:

Element środowiska	Opis
<i>Nadzorca klas</i>	Zarządza metadanymi, wczytuje interfejsy klas.
<i>Kompilator JIT</i>	Konwertuje kod pośredni (IL) na maszynowy.
<i>Nadzorca kodu</i>	Nadzoruje uruchomieniem kodu.
<i>Garbage collector (GC)</i> <i>“Kolekcjoner nieużytków”</i>	Automatycznie zwalnia nieużywaną pamięć dla wszystkich obiektów.
<i>Interfejs bezpieczeństwa</i>	Zapewnia mechanizmy bezpieczeństwa oparte na sprawdzaniu oryginalności kodu.
<i>Interfejs debugowania</i>	Pozwala na wyszukiwanie błędów w aplikacji i śledzenie stanu uruchamianego kodu.
<i>Nadzorca typów</i>	Nadzoruje proces konwersji typów i zabezpiecza przed wykonywaniem niebezpiecznych z punktu widzenia wykonania kodu rzutowań.
<i>Nadzorca wyjątków</i>	Zarządza obsługą wyjątków, informuje o wystąpieniu błędów.

<i>Interfejs wątków</i>	Dostarcza interfejs dla programowania wielowątkowego.
<i>Marszaler COM</i>	Dostarcza mechanizmy marszalingu do i z COM.
<i>Bazowa biblioteka klas</i>	Integruje kod z biblioteką klas .NET Framework.

## Biblioteka klas .NET Framework

Biblioteka klas .NET Framework zawiera bardzo bogatą funkcjonalność w postaci dużej ilości klas. Klasy zorganizowane są hierarchicznie poprzez przestrzenie nazw. Podstawową przestrzenią nazw jest **System**, która zawiera bazowe klasy definiujące typy danych, zdarzenia i uchwytów zdarzeń, interfejsy, atrybuty oraz obsługę wyjątków. Pozostałe klasy zapewniają funkcjonalność: konwersji danych, manipulacji parametrami, operacji arytmetycznych i logicznych, zarządzania środowiskiem programu, nadzorowania zarządzanego i nie zarządzanego programu.

Znajomość podstawowych przestrzeni nazw .NET Framework jest bardzo istotna, gdyż dzięki tej znajomości można w łatwy sposób odnaleźć zestaw klas realizujących wymaganą przez nas w danej chwili funkcjonalność.

Przestrzeń nazw	Funkcjonalność
<i>Microsoft.CSharp</i>	Zawiera klasy wspierające proces kompilacji i generacji kodu dla języka C#.
<i>Microsoft.Win32</i>	Zawiera dwa rodzaje klas: obsługujące zdarzenia generowane przez system i obsługujące rejestr systemowy.
<i>System</i>	Zawiera bazowe klasy definiujące typy danych, zdarzenia i uchwytów zdarzeń, interfejsy, atrybuty oraz obsługę wyjątków.
<i>System.CodeDom</i>	Zawiera klasy, które można użyć do reprezentacji elementów i struktur kodu źródłowego dokumentu.
<i>System.Collections</i>	Zawiera interfejsy i klasy definiujące kolekcje obiektów takich jak: listy, kolejki, tablice bitowe, słowniki oraz hash-tablice.
<i>System.ComponentModel</i>	Zawiera klasy do manipulacji zachowaniem komponentów i kontrolerek.
<i>System.Configuration</i>	Zawiera klasy oraz interfejsy pozwalające na programowalny dostęp do ustawień konfiguracyjnych .NET Framework oraz przechwytywania błędów związanych z obsługą plików konfiguracyjnych (plików z rozszerzeniem .config).
<i>System.Data</i>	Zawiera klasy wspierające architekturę <b>ADO.NET</b> (które pozwalają na łatwą manipulację danymi oraz źródłami danych).
<i>System.Diagnostics</i>	Zawiera klasy pozwalające na interakcję z procesami systemowymi, logami zdarzeń oraz licznikami wydajności.
<i>System.DirectoryServices</i>	Zapewnia łatwy dostęp do <b>Active Directory</b> .
<i>System.Drawing</i>	Zapewnia dostęp do podstawowej funkcjonalności obsługi grafiki <b>GDI+</b> .
<i>System.EnterpriseServices</i>	Zapewnia infrastrukturę dla aplikacji enterprise.
<i>System.Globalization</i>	Zawiera klasy definiujące informacje specyficzne dla danego kraju (język, strefa czasowa, format wyświetlania: daty, pieniędzy, liczb, etc.). Klasy są wykorzystywane do tworzenia aplikacji wielojęzycznych.
<i>System.IO</i>	Zawiera klasy pozwalające na manipulację plikami i strumieniami oraz zarządzanie plikami i folderami.

<b><i>System.Management</i></b>	Zapewnia dostęp do zbioru informacji administracyjnych oraz zdarzeń pochodzących z: systemu, urządzeń oraz aplikacji. Przykładowo można uzyskać informacje dotyczące ilości dostępnego miejsca na dysku, aktualnego obciążenia procesora i wielu innych).
<b><i>System.Messaging</i></b>	Zawiera klasy pozwalające na łączenie się, monitorowanie oraz zarządzanie kolejkami wiadomości w sieci (wysyłanie, odbieranie i przetwarzanie wiadomości)
<b><i>System.Net</i></b>	Zapewnia prosty interfejs dla wielu protokołów sieciowych.
<b><i>System.Reflection</i></b>	Zawiera klasy i interfejsy pozwalające na dynamiczne tworzenie i wybieranie typów.
<b><i>System.Resources</i></b>	Zawiera klasy i interfejsy pozwalające na tworzenie, przechowywanie i zarządzanie zasobami specyficznymi dla danego języka.
<b><i>System.Runtime.CompilerServices</i></b>	Zapewnia funkcjonalność pozwalającą na zarządzanie atrybutami i metadanymi w czasie działania programu w środowisku <b>CLR</b> .
<b><i>System.Runtime.InteropServices</i></b>	Zapewnia wsparcie dla wywołań <b>COM</b> .
<b><i>System.Runtime.Remoting</i></b>	Zawiera klasy i interfejsy pozwalające na tworzenie i konfigurację dystrybuowanych aplikacji.
<b><i>System.Runtime.Serialization</i></b>	Zapewnia wsparcie dla mechanizmów serializacji obiektów.
<b><i>System.Security</i></b>	Zapewnia dostęp do systemu bezpieczeństwa środowiska <b>CLR</b> wraz z podstawowymi klasami do zarządzania prawami dostępu.
<b><i>System.ServiceProcess</i></b>	Zawiera klasy pozwalające na implementację, instalację oraz kontrolę usług systemu Windows.
<b><i>System.Text</i></b>	Zawiera klasy reprezentujące standardy kodowania znaków takie jak: <b>ASCII</b> , <b>Unicode</b> , <b>UTF-7</b> oraz <b>UTF-8</b> . Dodatkowo można tu znaleźć bardzo użyteczną klasę wspomagającą proces manipulacji łańcuchami znaków ( <b>StringBuilder</b> ).
<b><i>System.Threading</i></b>	Zawiera klasy i interfejsy wspierające programowanie wielowątkowe.
<b><i>System.Web</i></b>	Zawiera klasy i interfejsy zapewniające komunikację z serwerem Webowym.
<b><i>System.Windows.Forms</i></b>	Zawiera klasy i interfejsy do tworzenia aplikacji okienkowych. Zbiór zawiera pełną gamę komponentów niezbędnych do tworzenia interfejsu użytkownika.
<b><i>System.Xml</i></b>	Zawiera standardy obsługi i przetwarzania <b>XML</b> . Wspierane są następujące standardy: <b>XML</b> wersja 1.0: <a href="http://www.w3.org/TR/1998/REC-xml-19980210">http://www.w3.org/TR/1998/REC-xml-19980210</a> (z uwzględnieniem wsparcia <b>DTD</b> ) Przestrzenie nazw <b>XML</b> : <a href="http://www.w3.org/TR/REC-xml-names/">http://www.w3.org/TR/REC-xml-names/</a> (zarówno poziom strumienia jak i <b>DOM</b> ). Schematy <b>XSD</b> : <a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a> Wyrażenia <b>XPath</b> : <a href="http://www.w3.org/TR/xpath">http://www.w3.org/TR/xpath</a> Transformacje <b>XSLT</b> : <a href="http://www.w3.org/TR/xslt">http://www.w3.org/TR/xslt</a> <b>DOM</b> Poziom 1:

	<a href="http://www.w3.org/TR/REC-DOM-Level-1/">http://www.w3.org/TR/REC-DOM-Level-1/</a> <b>DOM</b> Poziom 2: <a href="http://www.w3.org/TR/DOM-Level-2/">http://www.w3.org/TR/DOM-Level-2/</a>
--	--

## Obsługa baz danych (ADO.NET)

**ADO.NET** to następca technologii **ADO** (*Microsoft ActiveX Data Objects*) zawierająca wsparcie dla obsługi baz danych oraz formatu **XML**. Technologię tą można wykorzystać zarówno do tworzenia zwykłych aplikacji klienckich Windows, jak i aplikacji przeznaczonych dla Internetu.

**ADO.NET** wspiera następujące rodzaje typów przechowywania danych:

- bez określonej struktury (dane nie posiadają logicznego uporządkowania np. proste notatki),
- o niehierarchicznej strukturze (dane podzielone są na odseparowane od siebie i uporządkowane jednostki np.: pliki tekstowe z danymi separowanymi przez znaki tabulacji, arkusze *Microsoft Excel*, pliki *Microsoft Active Directory*, itp.),
- hierarchiczne (dane przechowywane są w postaci struktur drzewiastych np. dokumenty **XML**),
- relacyjne (dane przechowywane są w tabelach zawierających kolumny o określonym typie danych i wiersze z danymi, tablice mogą być ze sobą logicznie połączone poprzez kolumny z identycznymi danymi czyli tzw. relacje, np.: baza *Microsoft SQL Server*, baza *Oracle*, itp.),
- obiektowe (dane przechowywane są w postaci obiektów np. obiektowe bazy danych)

Poniższe przestrzenie nazw zawierają pełną funkcjonalność **ADO.NET** znajdującą się w .NET Framework:

Przestrzeń nazw	Funkcjonalność
<b>System.Data</b>	Zawiera bazowe klasy do obsługi baz danych (przykładowo klasy zbioru danych <b>DataSet</b> ).
<b>System.Data.Common</b>	Zawiera klasy i interfejsy z których dziedziczą „dostawcy danych” .NET (.NET data providers).
<b>System.Data.SqlClient</b>	Dostawca danych .NET <b>SQL Server</b>
<b>System.Data.OleDb</b>	Dostawca danych .NET <b>OLE DB</b>
<b>System.Data.Odbc</b>	Dostawca danych .NET <b>ODBC</b>
<b>System.Data.OracleClient</b>	Dostawca danych .NET <b>Oracle</b>
<b>System.Data.SqlTypes</b>	Zawiera klasy i struktury typów danych bazy <b>SQL Server</b> . Stanowi szybszą i bezpieczniejszą alternatywę dla pozostałych typów danych.
<b>System.Data.SqlServerCe</b>	Dostawca danych .NET <b>SQL Server CE</b> .
<b>System.Xml</b>	Zawiera klasy, interfejsy i typy wyliczeniowe zapewniające wsparcie dla przetwarzania dokumentów <b>XML</b> (np. klasa <b>XmlDataDocument</b> ).
<b>System.Xml.Schema</b>	<p>Zawiera wsparcie dla schematów <b>XML</b> (Schemas definition language - <b>XSD</b>).</p> <p>Wspierane są:</p> <p>Schematy dla struktur <b>XML</b>:  <a href="http://www.w3.org/TR/xmlschema-1/">http://www.w3.org/TR/xmlschema-1/</a>  (wsparcie dla mapowania i walidacji schematu)</p> <p>Schematy dla typów danych <b>XML</b>:  <a href="http://www.w3.org/TR/xmlschema-2/">http://www.w3.org/TR/xmlschema-2/</a></p>



	(wparcie dla typów danych XML)
<b><i>System.Xml.Serialization</i></b>	Zawiera wsparcie dla serializacji obiektów w postaci dokumentów <b><i>XML</i></b> .
<b><i>System.Xml.XPath</i></b>	Zawiera parser <b><i>XPath</i></b> . Wspiera: Język ścieżek <b><i>W3C XML (XPath)</i></b> w wersji 1.0 ( <a href="http://www.w3.org/TR/xpath">www.w3.org/TR/xpath</a> )
<b><i>System.Xml.Xsl</i></b>	Zawiera wsparcie dla <b><i>XSLT</i></b> (Extensible Stylesheet Transformation). Wspiera: Transformacje <b><i>W3C XSL (XSLT)</i></b> wersja 1.0 ( <a href="http://www.w3.org/TR/xslt">www.w3.org/TR/xslt</a> ).

## Usługi webowe (XML Web Services)

Usługi webowe (***XML Web Services***) to dostępne przez sieć usługi (realizujące określoną funkcjonalność), które można wykorzystać jako komponenty do budowy aplikacji rozproszonych. Usługi webowe oparte są na otwartych standardach Internetowych takich jak ***HTTP***, ***SOAP*** oraz ***XML***. Usługi webowe mogą dostarczać różną funkcjonalność począwszy od prostych komponentów informujących o cenach akcji publikowanych przez jakiś dom maklerski do złożonych komponentów pełniących funkcję aplikacji finansowych. Praktycznie nie ma ograniczeń, jeżeli chodzi o rozproszenie komponentów w sieci. Poza tym każdy komponent może wykorzystywać funkcjonalność innych komponentów rozproszonych w celu dostarczenia bardziej złożonej funkcjonalności.

Jedną z podstawowych cech usług webowych jest wysoki stopień abstrakcji istniejący między implementacją a użytkowaniem. Dzięki wykorzystaniu mechanizmu wymiany danych przez standard ***XML*** klient usługi jak i jej dostawca są zwolnieni z potrzeby informowania się nawzajem o formacie wejścia/wyjścia czy też położeniu.

Poniższe przestrzenie nazw są wykorzystywane przy tworzeniu usług webowych w .NET Framework:

Przestrzeń nazw	Funkcjonalność
<b><i>System.Web</i></b>	Dostarcza bazową funkcjonalność dla usług webowych.
<b><i>System.Web.Caching</i></b>	Dostarcza funkcjonalność związaną z przechowywaniem kopii często używanych danych (kaszowaniem) z serwera.
<b><i>System.Web.Configuration</i></b>	Dostarcza funkcjonalność związaną z konfiguracją.
<b><i>System.Web.Security</i></b>	Dostarcza funkcjonalność związaną z bezpieczeństwem.
<b><i>System.Web.Services</i></b>	Dostarcza funkcjonalność niezbędną do tworzenia usług webowych.
<b><i>System.Web.SessionState</i></b>	Przechowuje dane dotyczące sesji użytkownika.

## Aplikacje webowe (Web Forms)

Aplikacje webowe (***Web Forms***) są aplikacjami opartymi o ***ASP.NET***, które dostępne są poprzez Internet. Tworzenie aplikacji webowej przypomina tworzenie zwykłej aplikacji okienkowej. Podobnie jak to miało miejsce przy ***ASP (Active Server Pages)***, ***ASP.NET*** działa na serwerze webowym, dzięki czemu pozwala na rozwijanie spersonalizowanych oraz dynamicznie zmieniających się stron webowych. Aplikacje webowe oparte na ***ASP.NET*** są całkowicie niezależne od typu przeglądarki po stronie klienta oraz używanego przez niego systemu operacyjnego.

Aplikacja webowa składa się z różnych współpracujących ze sobą elementów:



- stron webowych .aspx (dostarczają dynamiczny interfejs dla aplikacji webowej);
- kodu ukrytego za stroną webową (kod niewidoczny dla klienta, który jest skojarzony ze stroną webową i zawiera funkcjonalność aplikacji znajdującej się po stronie serwera);
- plików konfiguracyjnych (pliki te są plikami XML i zawierają domyślne ustawienia dla aplikacji webowej oraz serwera webowego, każda aplikacja webowa zawiera jeden plik konfiguracyjny Web.config, dodatkowo serwer webowy zawiera swój plik konfiguracyjny machine.config);
- pliku global.aspx (zawiera kod niezbędny do obsługi zdarzeń aplikacji zgłaszanych przez ASP.NET);
- odsyłaczy do usług webowych (odsyłacze pozwalają na wysyłanie i odbieranie danych z i do usługi webowej);
- połączenia z bazą (pozwalające na wymianę danych ze źródłem danych);
- keshowania (pozwalające aplikacji webowej na szybszą odpowiedź po pierwszym żądaniu).

Poniższe przestrzenie nazw są wykorzystywane przy tworzeniu aplikacji webowych w .NET Framework:

Przestrzeń nazw	Funkcjonalność
<i>System.Web</i>	Dostarcza bazową funkcjonalność dla usług webowych.
<i>System.Web.Caching</i>	Dostarcza funkcjonalność związaną z przechowywaniem kopii często używanych danych (keshowaniem) z serwera.
<i>System.Web.Configuration</i>	Dostarcza funkcjonalność związaną z konfiguracją.
<i>System.Web.Security</i>	Dostarcza funkcjonalność związaną z bezpieczeństwem.
<i>System.Web.Services</i>	Dostarcza funkcjonalność niezbędną do tworzenia usług webowych.
<i>System.Web.SessionState</i>	Przechowuje dane dotyczące sesji użytkownika.
<i>System.Web.UI</i>	Dostarcza funkcjonalność pozwalającą na kontrolę zachowania kontrolek i stron, które pojawiają się w aplikacji webowej w postaci interfejsu strony webowej.

## Aplikacje okienkowe (Windows Forms)

Aplikacje okienkowe (*Windows Forms*) to klienckie aplikacje systemu Windows oparte o standardowy graficzny interfejs użytkownika. Z poziomu aplikacji klienckiej mamy do dyspozycji pełną gamę komponentów zawartych w .NET Framework

Poniższe przestrzenie nazw są wykorzystywane przy tworzeniu interfejsu użytkownika w aplikacjach okienkowych .NET Framework:

Przestrzeń nazw	Funkcjonalność
<i>System.Windows.Forms</i>	Dostarcza komponenty do budowy okienkowych aplikacji w standardzie Windows (okna, wspólne dialogi, paski narzędzi, menu, paski statusu, przyciski, listy rozwijane, napisy, pola edycyjne, itd.).
<i>System.Drawing</i>	Pozwala na dostęp do podstawowej funkcjonalności <b>GDI+</b> .
<i>System.Drawing.Drawing2D</i>	Zapewnia obsługę grafiki wektorowej i 2D.
<i>System.Drawing.Imaging</i>	Zapewnia zaawansowaną funkcjonalność <b>GDI+</b> wsparcia procesu przetwarzania obrazów.
<i>System.Drawing.Printing</i>	Dostarcza usługi związane z drukowaniem grafiki.
<i>System.Drawing.Text</i>	Zapewnia zaawansowaną funkcjonalność <b>GDI+</b> wsparcia rysowania tekstu (pozwala na korzystanie z kolekcji czcionek).

## Wspólna specyfikacja językowa (CLS)

Wspólna specyfikacja językowa **CLS** definiuje zespół typów oraz zasady posługiwania się nimi dla różnych języków w celu zapewnienia kompatybilności między nimi. Przestrzegając zasad dotyczących zgodności typów, autor biblioteki klas ma gwarancję, że jego biblioteka będzie mogła zostać użyta w dowolnym języku zgodnym z **CLS**. Dzięki takiemu podejściu, możemy przykładowo rozwinąć bazową funkcjonalność komponentu w C#, odziedziczyć po nim w Visual Basic i rozszerzyć jego możliwości, a następnie jeszcze raz odziedziczyć w C# i znowu rozszerzyć jego funkcjonalność.

Poniżej znajdują się niektóre zasady dotyczące typów we wspólnej specyfikacji językowej:

- typy proste należące do specyfikacji CLS: bool, char, short, int, long, float, double, decimal, string, object;
- numeracja tablicy rozpoczyna się od 0;
- rozmiar tablicy musi być znany i większy lub równy 1;
- typy mogą być abstrakcyjne lub ostateczne;
- typ może być pochodnym tylko dla jednego typu;
- typ nie musi mieć składowych;
- wszystkie typy wartości muszą dziedziczyć z obiektu System.ValueType (wyjątek stanowi typ wyliczeniowy – musi dziedziczyć z System.Enum);
- typy parametrów przekazywanych i zwracanych muszą być typami zgodnymi z CLS;
- można przeciążać konstruktory, metody i właściwości;
- metody statyczne muszą zawierać implementację a metody składowe i wirtualne mogą być metodami abstrakcyjnymi;
- metoda może być statyczna, wirtualna lub składowa;
- globalne statyczne metody i pola nie są dozwolone;
- pola statyczne mogą być stałymi lub polami do zainicjowania;
- typ wyniku zwracanego przez metody get i set danej właściwości musi być zgodny;
- właściwości mogą być indeksowane;
- dla typów wyliczeniowych dozwolone są jedynie następujące typy całkowite: byte, short, int lub long;
- wyjątki zgłaszane przez program muszą dziedziczyć z System.Exception;
- identyfikatory muszą się różnić między sobą czymś więcej niż tylko wielkością liter w nazwie (niektóre języki nie rozróżniają identyfikatorów na podstawie wielkości liter).

## Języki programowania w .NET Framework

Microsoft.NET Framework dostarcza pełne wsparcie dla kilku różnych języków programowania:

- Język C# – został zaprojektowany na platformę .NET i jest pierwszym nowoczesnym językiem komponentowym w linii języków C/C++. W skład języka wchodzi: klasy, interfejsy, delegacje, mechanizmy opakowywania i odpakowywania, przestrzenie nazw, właściwości, indeksatory, zdarzenia, operatory przeciążania, mechanizmy wersjonowania, atrybuty, wsparcie dla wywołań kodu niezabezpieczonego i mechanizmy generacji dokumentacji XML.

- Rozszerzenia dla zarządzanego języka C++ – zarządzany C++ stanowi rozszerzenie dla języka C++. To rozszerzenie zapewnia programiście dostęp do .NET Framework.
- Visual Basic .NET – stanowi innowację w stosunku do poprzedniej wersji Visual Basic. Wspiera mechanizmy dziedziczenia i polimorfizmu, przeciążania konstruktorów, obsługi wyjątków, sprawdzania typów i wiele innych.
- Visual J# .NET – przeznaczony dla programistów języka Java, którzy chcą korzystać ze wsparcia technologii .NET.
- JScript .NET.
- Pozostałe (APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, Python, Scheme i Smalltalk).

## Rozdział 2.

# Pierwszy program

### HelloWorld

Zazwyczaj pierwszym programem, jaki piszą ludzie ucząc się jakiegoś języka programowania, jest program, który wypisuje jakiś tekst na ekranie monitora. W większości książek przyjęło się używanie napisu z cyklu „Hello world”, czy polskiego odpowiednika „Witaj świecie”. Spróbujmy zatem napisać taką prostą mini-aplikację w języku C#:

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World");
    }
}
```

Język C# jest w pełni obiektowy, więc każdy program w tym języku jest klasą lub kolekcją klas, struktur oraz typów. Jak widać na powyższym przykładzie, nawet prosta aplikacja wymaga stworzenia klasy.

Klasę definiuje się poprzez słowo kluczowe **class**, po którym umieszcza się nazwę klasy (u nas HelloWorld). W języku C# każdy blok kodu umieszcza się między nawiasami: otwierającym **{** i zamykającym **}**. Jak widać w powyższym przykładzie, klasa HelloWorld zawiera tylko jedną metodę statyczną o nazwie **Main**. Metoda ta zawiera instrukcję, której zadaniem jest wypisanie określonego napisu.

Metoda **Main** jest statyczną metodą publiczną, która pełni w języku C# rolę szczególną – punktu startowego programu. Oznacza to, że uruchomienie programu możliwe jest jedynie wtedy, gdy co najmniej jedna z klas zawiera metodę **Main**. Ze względu na to, że tylko jedna metoda **Main** może być punktem startowym aplikacji, w przypadku, gdy inne klasy zawierają taką metodę, należy określić w czasie kompilacji, która z nich będzie używana.

Microsoft .NET Framework dostarcza ogromną liczbę użytecznych klas, które zorganizowane są w przestrzeniach nazw, (o czym wspominałem w poprzednim rozdziale).

Spójrzmy na nasz przykład. Metoda **Main** w naszym przypadku zawiera wywołanie metody **WriteLine**, będącej statyczną metodą składową klasy **Console**, która znajduje się w przestrzeni nazw **System**.

Aby za każdym razem przy odwoływaniu się do obiektów czy metod nie pisać pełnej ścieżki z uwzględnieniem przestrzeni nazw, można użyć dyrektywy **using** wskazując, iż będziemy używać klas z danej przestrzeni.

Przepiszmy nasz prosty przykład wykorzystując tą dyrektywę, a przykład stanie się bardziej przejrzysty:

```
using System;

class HelloWorld
```

```
{  
    public static void Main()  
    {  
        Console.WriteLine("Hello World");  
    }  
}
```

## Wejście-wyjście

Przejdźmy do dalszej analizy mini-aplikacji HelloWorld i przyjrzymy się bliżej klasie **Console**, która zapewnia prostą obsługę standardowych strumieni: wejścia, wyjścia oraz błędów.

Standardowy strumień wejścia skojarzony jest zazwyczaj z klawiaturą (cokolwiek, co wprowadza użytkownik może zostać wczytane poprzez standardowy strumień wejścia). Standardowy strumień wyjścia oraz strumień błędów skojarzony jest zazwyczaj z ekranem.

Statyczne metody **WriteLine** oraz **Write** (metoda ta różni się od **WriteLine** tym, że do napisu nie jest dołączany znak przejścia do nowego wiersza), należące do klasy **Console**, można wykorzystać do wyświetlania nie tylko tekstu, ale i innych typów danych. Przykładowo, aby wypisać liczbę 123 wystarczy napisać:

```
Console.WriteLine(123);
```

W przypadku, gdy chcemy wypisać bardziej złożony ciąg stanowiący kombinację łańcuchów, liczb i wyrażeń powinniśmy użyć mechanizmu formatowania tekstu. Mechanizm ten opiera się na prostej zasadzie. W pierwszej kolejności tworzymy łańcuch formatujący zawierający odnośniki do parametrów, a następnie uporządkowaną listę tych parametrów.

Przykładowo chcąc wypisać zdanie: „Dodając liczbę 1 do 2 uzyskasz liczbę 3” wystarczy napisać:

```
Console.WriteLine("Dodając liczbę do uzyskasz liczbę ", 1, 2, 1+2);
```

Jak łatwo zauważyć odnośnik do parametru podaje się wewnątrz łańcucha formatującego w nawiasach **{}** numerując je od 0 (0 oznacza pierwszy parametr), a następnie za tym łańcuchem (po przecinku) listę parametrów w takiej kolejności, w jakiej mają być wstawione w miejsce odnośników. W łańcuchu formatującym, oprócz odnośnika do parametru, można umieścić informacje o szerokości zajmowanego przez niego miejsca oraz czy parametr ma być wyrównany do lewej czy prawej.

```
Console.WriteLine("\Wyrównaj do lewej na polu o długości 10 znaków: {0,  
-10}\",123);
```

```
Console.WriteLine("\Wyrównaj do prawej na polu o długości 10 znaków: {0,  
10}\",123);
```

Powyższe dwie linie spowodują wyświetlenie na ekranie konsoli następujących napisów:

```
"Wyrównaj do lewej na polu o długości 10 znaków:123  
"Wyrównaj do prawej na polu o długości 10 znaków: 123"
```

Znak **\** wyłącza w łańcuchu formatującym specjalne znaczenie znaku, który znajduje się za nim (w naszym wypadku pozwoli na wyświetlenie znaku **"**). Przykładowo chcąc wyświetlić znak nawiasu **{**, który służy do umieszczania odnośników należy użyć ciągu **"\{"**, a chcąc wyświetlić znak **|** ciągu **"\|"**.

Warto tu również wspomnieć o znaczeniu symbolu **@**. Jego zadaniem jest wyłączenie specjalnego

znaczenia znaków w całym ciągu. Jest on bardzo użyteczny, gdy chcemy wyświetlić ścieżkę np.: `@\"C:\Windows\System32\"` da w rezultacie ciąg: `C:\Windows\System32` (jak łatwo się domyśleć bez znaku `@` mielibyśmy niepoprawny zapis ścieżki: `C:WindowsSystem32`).

Ciąg formatujący może zostać użyty również do określenia, w jaki sposób mają być wyświetlane dane numeryczne. Format zapisu wygląda wtedy tak: `{X,Y:Z}` gdzie X to numer parametru, Y to informacja o szerokości pola oraz sposobie jego wyrównania, a Z to informacja o sposobie wyświetlania danych numerycznych.

Poniższa tabela określa znaczenie poszczególnych symboli formatujących dane numeryczne:

Symbol	Znaczenie						
<b>C</b>	Wyświetla dane numeryczne w formacie pieniężnym używając lokalnej konwencji zapisu walutowego. Przykłady: <table border="1"> <thead> <tr> <th>Kod</th><th>Rezultat</th></tr> </thead> <tbody> <tr> <td><code>Console.WriteLine("{0:C}", 99.9);</code></td><td>99.9 zł</td></tr> <tr> <td><code>Console.WriteLine("{0:C2}", 99.9);</code></td><td>99,90 zł</td></tr> </tbody> </table>	Kod	Rezultat	<code>Console.WriteLine("{0:C}", 99.9);</code>	99.9 zł	<code>Console.WriteLine("{0:C2}", 99.9);</code>	99,90 zł
Kod	Rezultat						
<code>Console.WriteLine("{0:C}", 99.9);</code>	99.9 zł						
<code>Console.WriteLine("{0:C2}", 99.9);</code>	99,90 zł						
<b>D</b>	Wyświetla dane numeryczne w formacie dziesiętnym. Przykład: <table border="1"> <thead> <tr> <th>Kod</th><th>Rezultat</th></tr> </thead> <tbody> <tr> <td><code>Console.WriteLine("{0:D4}", 99);</code></td><td>0099</td></tr> </tbody> </table>	Kod	Rezultat	<code>Console.WriteLine("{0:D4}", 99);</code>	0099		
Kod	Rezultat						
<code>Console.WriteLine("{0:D4}", 99);</code>	0099						
<b>E</b>	Wyświetla dane numeryczne w postaci liczb zmiennoprzecinkowych z użyciem ekspozycji (notacja naukowa). Przykład: <table border="1"> <thead> <tr> <th>Kod</th><th>Rezultat</th></tr> </thead> <tbody> <tr> <td><code>Console.WriteLine("{0:E}", 99.9);</code></td><td>9,990000E+001</td></tr> </tbody> </table>	Kod	Rezultat	<code>Console.WriteLine("{0:E}", 99.9);</code>	9,990000E+001		
Kod	Rezultat						
<code>Console.WriteLine("{0:E}", 99.9);</code>	9,990000E+001						
<b>F</b>	Wyświetla dane numeryczne w postaci liczb zmiennoprzecinkowych o określonej precyzji. Przykład: <table border="1"> <thead> <tr> <th>Kod</th><th>Rezultat</th></tr> </thead> <tbody> <tr> <td><code>Console.WriteLine("{0:F3}", 99.9);</code></td><td>99.900</td></tr> </tbody> </table>	Kod	Rezultat	<code>Console.WriteLine("{0:F3}", 99.9);</code>	99.900		
Kod	Rezultat						
<code>Console.WriteLine("{0:F3}", 99.9);</code>	99.900						
<b>G</b>	Wyświetla dane numeryczne w postaci liczb zmiennoprzecinkowych o określonej precyzji lub liczb całkowitych (w zależności od tego, który format bardziej pasuje do sytuacji). Przykład: <table border="1"> <thead> <tr> <th>Kod</th><th>Rezultat</th></tr> </thead> <tbody> <tr> <td><code>Console.WriteLine("{0:G}", 99.99);</code></td><td>99.99</td></tr> </tbody> </table>	Kod	Rezultat	<code>Console.WriteLine("{0:G}", 99.99);</code>	99.99		
Kod	Rezultat						
<code>Console.WriteLine("{0:G}", 99.99);</code>	99.99						
<b>N</b>	Wyświetla dane numeryczne w postaci oddzielonych paczek.						

	Przykład:	
	<b>Kod</b>	<b>Rezultat</b>
	<code>Console.WriteLine("{0:N}", 1000000);</code>	1 000 000,00
X	Wyświetla dane numeryczne w formacie szesnastkowym.	
	Przykład:	
	<b>Kod</b>	<b>Rezultat</b>
	<code>Console.WriteLine("{0:X4}", 1234);</code>	04D2

Skoro wiemy już jak wyświetlać dane, przydałoby się poznać metody do ich wczytywania ze standardowego strumienia wejściowego. Do tego celu służą statyczne metody **Read** oraz **ReadLine**.

Metoda **Read** wczytuje pojedynczy znak wprowadzony przez użytkownika i zwraca wartość: -1, jeżeli nie ma już znaków do odczytania lub też wartość liczbową odpowiadającą wczytanemu znakowi. Z kolei metoda **ReadLine** wczytuje wszystkie znaki znajdujące się w wierszu i zwraca wynik w postaci łańcucha znaków.

Spróbujmy zmienić nasz przykład tak, aby zamiast tekstu "Hello World" wyświetlał napis, który wprowadzimy mu z klawiatury:

```
using System;
class HelloWorld
{
    public static void Main()
    {
        Console.Write("Jak masz na imię: ");
        string imie = Console.ReadLine();
        Console.WriteLine(" to ładne imię", imie);
    }
}
```

Program w pierwszej kolejności wypisze zdanie "Jak masz na imię: ", następnie w tej samej linii (**Write**) pozwoli nam na wpisanie naszego imienia (**ReadLine**), a na końcu wypisze je na ekranie w nowej linii (**WriteLine**).

## Kompilacja i uruchomienie

Framework .NET zawiera wszystkie potrzebne kompilatory (C#, C++, Visual, Basic), dlatego nie trzeba kupować oddzielnego narzędzia (środowiska programistycznego), by móc skompilować kod.

W pierwszej kolejności musimy zapisać nasz program w pliku pod jakąś określoną przez nas nazwą np.: helloworld.cs (pliki z kodem w C# mają rozszerzenie cs). Następnie wywołujemy wiersz poleceń, przechodzimy do katalogu, w którym znajduje się Framework i korzystamy z polecenia **csc**:

```
csc helloworld.cs
```

W wyniku kompilacji otrzymamy plik helloworld.exe, który możemy uruchomić. Warto wspomnieć w tym miejscu, że możemy określić nazwę pliku wynikowego za pomocą parametru out:

```
csc /out:hello.exe helloworld.cs
```

Kompilator C# posiada wiele parametrów. Najważniejsze z nich przedstawia poniższa tabela:

Parametr	Znaczenie
<code>/?, /help</code>	Wyświetla opcje kompilatora na standardowym wyjściu.
<code>/out</code>	Pozwala określić nazwę pliku wykonywalnego.
<code>/main</code>	Pozwala wskazać, która metoda Main (jeżeli jest ich wiele) ma być punktem startowym programu.
<code>/optimize</code>	Włącza lub wyłącza optymalizację kodu.
<code>/warn</code>	Pozwala ustawić poziom ostrzeżeń dla kompilatora.
<code>/warnaserror</code>	Pozwala traktować wszystkie ostrzeżenia jak błędy, które przerywają proces kompilacji.
<code>/target</code>	Pozwala określić rodzaj generowanej aplikacji.
<code>/checked</code>	Włącza lub wyłącza mechanizm sprawdzania przepełnienia arytmetycznego.
<code>/doc</code>	Pozwala na wygenerowanie dokumentacji XML z komentarzy.
<code>/debug</code>	Generuje informację dla debugera.

Tworzenie bardziej złożonych projektów bez użycia środowiska programistycznego jest bardzo uciążliwe, więc warto pomyśleć o jakimś już na początku. Na rynku jest dostępnych wiele dobrych narzędzi. Najbardziej zaawansowanym środowiskiem programistycznym jest niewątpliwie Microsoft Visual C#.NET. W książce będę posługiwał się środowiskiem Visual C#.NET z pakietu Microsoft Visual Studio.NET 2005. Środowisko to zostało omówione w dalszej części książki.

## Komentowanie kodu

Komentowanie kodu to bardzo istotna rzecz. Wielu programistów nie komentuje swojego kodu i kiedy wraca do niego po dłuższym czasie musi analizować całą funkcjonalność od początku. A co, jeżeli tworzy się kod wspólnie z innymi? Czy ich też musimy zmuszać do niepotrzebnego tracenia czasu na zrozumienie tego, co się dzieje w kodzie, który napisaliśmy? Nie. Właśnie po to wprowadzono możliwość komentowania kodu, aby łatwiej można było go zrozumieć nie tylko nam, ale i innym.

W języku C# istnieją trzy rodzaje komentarzy:

- jednowierszowe (wszystko, co pojawi się za znakami //):

```
// To jest komentarz do kodu
Console.WriteLine("To jest część kodu");
```

- wielowierszowe (wszystko między znacznikiem początku komentarza /\* a znacznikiem jego końca \*/)

```
/*
    To jest komentarz do kodu,
    który może występować w wielu liniach
*/
Console.WriteLine("To jest część kodu");
```



- dokumentacja kodu w ***XML*** (wszystko, co pojawi się za znakami `///` w postaci serii znaczników)

```
/// <summary>
/// Krótki opis klasy
/// </summary>
```

Na szczególną uwagę zasługuje ostatni rodzaj komentarzy, który służy do generowania dokumentacji do kodu w ***XML***. Istnieje wiele różnych znaczników, które można używać. Można również tworzyć własne znaczniki. Znacznik początkowy `<nazwa>` określa początek, natomiast znacznik `</nazwa>` koniec. Wszystko, co znajduje się między znacznikami to opis, o określonym przez znacznik przeznaczeniu.

Poniższa tabela pokazuje wybrane znaczniki wraz z opisem ich przeznaczenia:

Znacznik	Przeznaczenie
<code>&lt;c&gt;...&lt;/c&gt;</code>	Kod bez opisu
<code>&lt;code&gt;...&lt;/code&gt;</code>	Kod z opisem
<code>&lt;example&gt;...&lt;/example&gt;</code>	Przykład użycia (np. metody)
<code>&lt;exception&gt;...&lt;/exception&gt;</code>	Opis wyjątków klasy
<code>&lt;list type="..."&gt;...&lt;/list&gt;</code>	Lista dla opisu szczegółowego
<code>&lt;para&gt;...&lt;/para&gt;</code>	Paragraf dla opisu szczegółowego
<code>&lt;param name="nazwa"&gt;...&lt;/param&gt;</code>	Opis parametru metody
<code>&lt;permission&gt;...&lt;/permission&gt;</code>	Dokumentacja dostępności składnika klasy
<code>&lt;remarks&gt;...&lt;/remarks&gt;</code>	Opis szczegółowy
<code>&lt;returns&gt;...&lt;/returns&gt;</code>	Opis zwracanych wartości
<code>&lt;see cref="odnośnik"&gt;...&lt;/see&gt;</code>	Wskazanie na inny składnik klasy
<code>&lt;seealso cref="odnośnik"&gt;...&lt;/seealso&gt;</code>	Dodatkowe wskazanie na inny składnik klasy
<code>&lt;summary&gt;...&lt;/summary&gt;</code>	Opis ogólny opis (krótki)
<code>&lt;value&gt;...&lt;/value&gt;</code>	Opis właściwości

W czasie kompilacji kodu można wygenerować dokumentację ***XML*** na podstawie znaczników w następujący sposób:

```
csc nazwa_programu.cs /doc:nazwa_dokumentacji.xml
```

Wygenerowaną dokumentację można następnie obejrzeć w przeglądarce ***Internet Explorer***.

## Rozdział 3.

# Wprowadzenie do Microsoft Visual C#.NET 2005

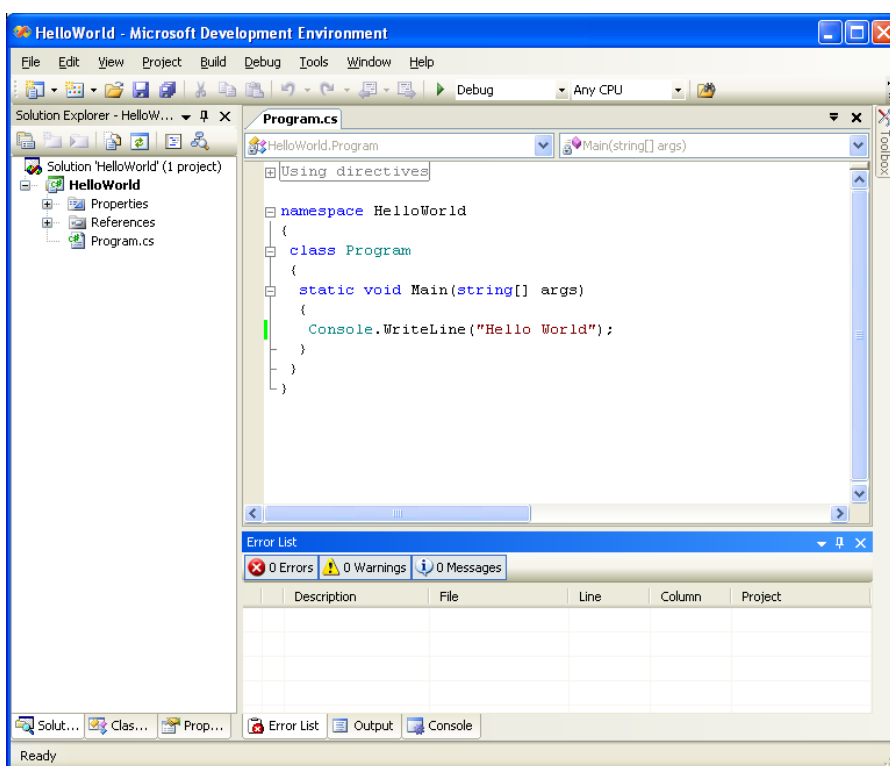
## Środowisko programisty

Microsoft Visual C#.NET 2005 jest jednym ze środowisk wchodzących w skład pakietu Microsoft Visual Studio .NET 2005. Oferuje ono potężne wsparcie przy tworzeniu aplikacji w języku C#. Środowisko dostarcza nam przede wszystkim bardzo rozbudowany edytor z podświetlaniem składni języka oraz systemem inteligentnych podpowiedzi. Ponadto zawiera również całą masę użytecznych okien, pozwalających na łatwiejszą orientację w strukturze złożonych projektów. Środowisko można konfigurować w dowolny sposób, dostosowując jego wygląd do własnych potrzeb.

Podstawowe elementy okna głównego:

- Menu
- Paski narzędzi
- Okna narzędziowe
- Okna edytora
- Pasek statusu

Oto przykładowy wygląd okna głównego, na którym oprócz samego edytora z kodem przykładowej aplikacji, widać wiele różnych zakotwiczonych okien narzędziowych:



Rysunek 2. Główne okno środowiska Microsoft Visual C#.NET 2005

## Okna narzędziowe

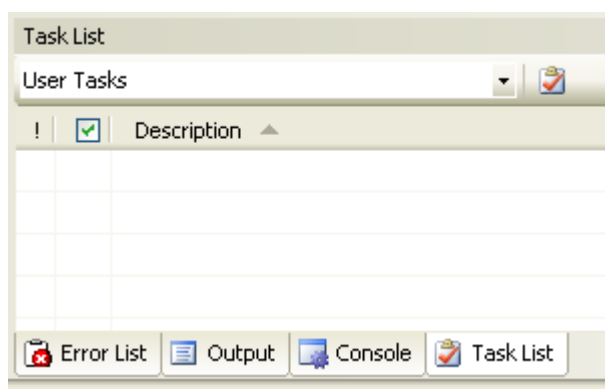
Okna narzędziowe można zakotwiczać i wpinać w dowolnym miejscu wewnątrz okna głównego. Jeżeli jakieś okno nie jest dostępne, wystarczy włączyć opcję pokazaniażądanego okna, wybierając je z menu *View*.

W momencie, w którym zakotwiczymy jakieś okno narzędziowe na ekranie pokazuje się specjalny zestaw ikon wskazujący kierunek zakotwiczenia. W tym momencie wystarczy najechać na ikonę odpowiadającą kierunkowi zakotwiczenia i upuścić trzymane okno:



Rysunek 3. Ikony zakotwiczenia

Jeżeli chcemy, aby okna narzędziowe nie zajmowały wiele miejsca możemy je połączyć w zakładki. Wystarczy wtedy na pierwsze zakotwiczone okno nałożyć kolejne, wtedy otrzymamy widok zakładkowy:



Rysunek 4. Widok zakładkowy okna

Jeżeli widok zakładkowy nam się nie podoba, możemy użyć opcji automatycznego ukrywania okna. W takim wypadku, po zakotwiczeniu okna, wystarczy nacisnąć przycisk pinezki na belce okna:



Rysunek 5. Fragment okna z widocznym przyciskiem pinezki.

I okno automatycznie ukryje się:

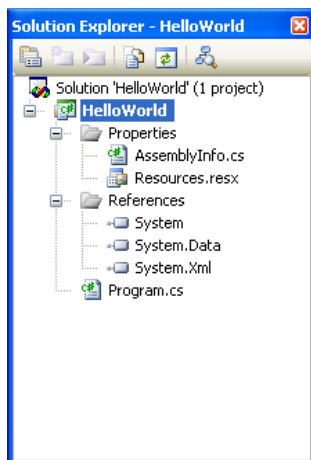


Rysunek 6. Wygląd ukrytych okien narzędziowych

Każde z zakotwiczonych okien narzędziowych ma swoje przeznaczenie. Umiejętność korzystania z tych okien jest bardzo istotna w czasie korzystania ze środowiska.

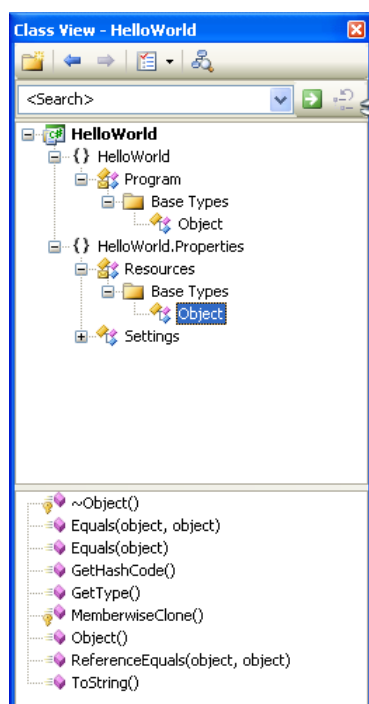
Do najważniejszych okien narzędziowych należą przede wszystkim:

- **Solution Explorer** – wyświetla grupę projektów aktualnej solucji w postaci drzewiastej i pozwala wybierać projekty oraz pliki należące do danego projektu. Podwójne kliknięcie na plik w drzewie powoduje otwarcie pliku w trybie edycji. Naciśnięcie prawego klawisza myszy na drzewie wyświetla menu podręczne, pozwalające na wykonanie różnych funkcji (np. dodanie nowego elementu do projektu).



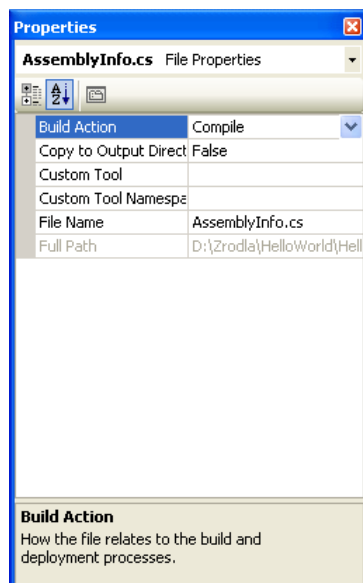
Rysunek 7. Okno Solution Explorer

- **Class View** – wyświetla hierarchiczny układ klas dla aktualnego projektu. Podwójne kliknięcie na nazwie klasy, pozwala na przejście w tryb edycji pliku, w którym klasa ta została zdefiniowana. Naciśnięcie prawego klawisza myszy na drzewie, wyświetla menu podręczne pozwalające na wykonanie różnych funkcji (np. dodanie nowej klasy do projektu).



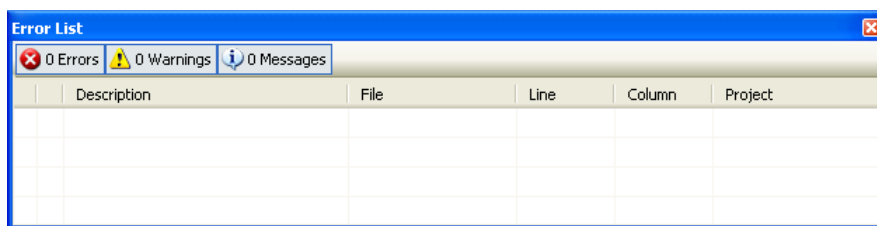
Rysunek 8. Okno Class View

- **Properties** – wyświetla właściwości elementów rozwiązania. Zawartość tego okna uzależniona jest od aktualnie edytowanego elementu (w przypadku elementów nie zawierających żadnych właściwości okno nie wyświetla niczego). Okno to jest jednym z najczęściej używanych w czasie edytowania właściwości kontrolki interfejsowych (np.: okien, przycisków, etc.).



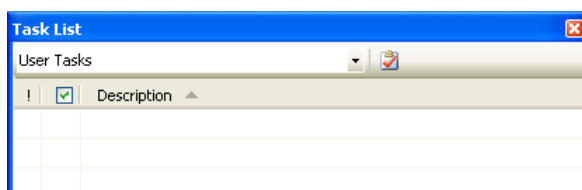
Rysunek 9. Okno Properties

- **Error List** – przedstawia listę błędów, ostrzeżeń oraz wiadomości, jakie zwraca kompilator. Jeżeli na liście znajdują się jakieś błędy, podwójne kliknięcie na dany element, spowoduje przejście do linii kodu, w której go znaleziono.



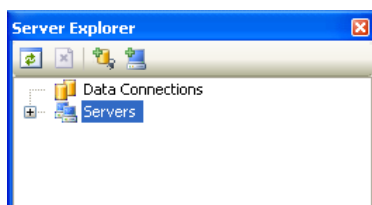
Rysunek 10. Okno Error List

• **Task List** – wyświetla listę komentarzy i zadań do wykonania, które programista może umieścić w kodzie (np.: zadanie: „wstawić tu funkcję kopiowania”). Jeżeli na liście znajdują się jakieś komentarze lub zadania, podwójne kliknięcie na dany element spowoduje przejście do linii kodu, w której się znajduje.



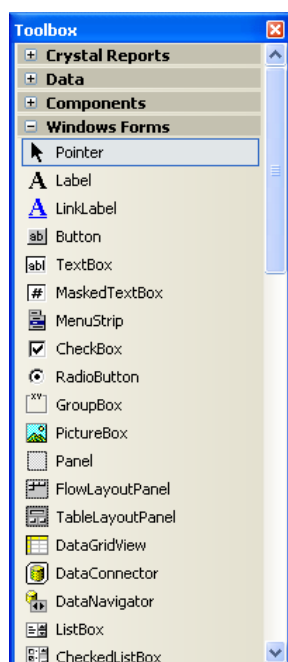
Rysunek 11. Okno Task List

• **Server Explorer** – wyświetla listę aktualnie dostępnych usług na maszynie programisty jak również na innych maszynach w sieci. Okno to zapewnia łatwy dostęp do serwerów baz danych, logów systemowych, liczników wydajności oraz wielu innych usług. Za pomocą techniki „przeciągnij i upuść” można dodać do aplikacji funkcjonalność związaną z obsługą określonej usługi (generowany jest wtedy kod, pozwalający na użycie obiektów obsługujących usługi np.: dostępu do bazy danych).



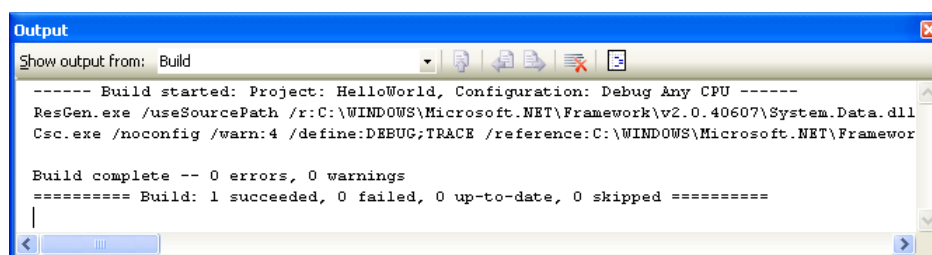
Rysunek 12. Okno Server Explorer

• **Toolbox** – zawiera listę komponentów interfejsowych, które mogą być łatwo dodawane do projektu. Komponenty zorganizowane są w grupy funkcjonalne, co pozwala na ich łatwiejszą lokalizację (kliknięcie na krzyżyk pozwala na rozwinięcie listy komponentów z określonej grupy).



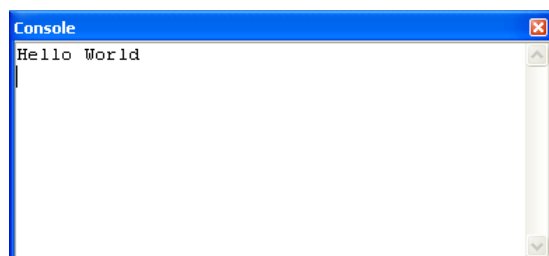
Rysunek 13. Okno Toolbox

- **Output** – wyświetla informacje dotyczące statusu kompilacji aktualnego projektu. Okno to wyświetla również dodatkowe informacje w czasie debugowania projektu (o debugowaniu będzie mowa w dalszej części).



Rysunek 14. Okno Output

- **Console** – wyświetla zawartość konsoli. Okno to wykorzystuje się do testowania aplikacji konsolowych.

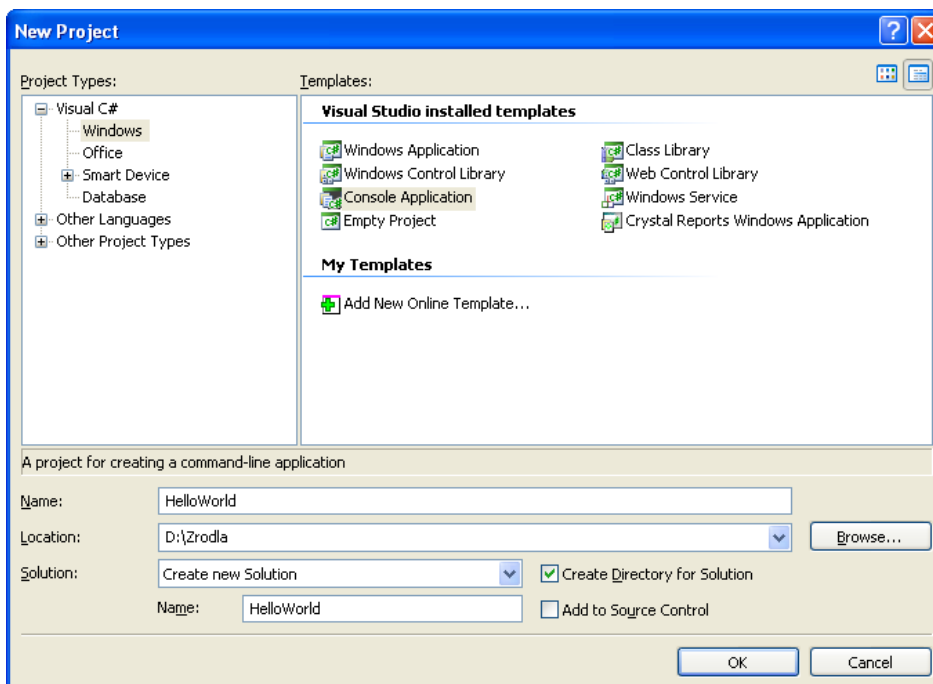


Rysunek 15. Okno Console

## Generowanie szablonów projektów

Środowisko Microsoft Visual C#.NET 2005 wspiera programistę od samego początku. Pierwszym etapem tworzenia aplikacji jest utworzenie projektu (grupy współpracujących klas oraz warunków ich kompilacji). Każdy projekt z kolei jest częścią jakiejś solucji (solucja to grupa projektów będąca rozwiązaniem problemu).

Aby wygenerować nowy projekt, należy z menu **File** wybrać pozycję **New->Project**. Po wybraniu tej opcji na ekranie pojawi się okno generatora szablonów projektów:



Rysunek 16. Okno generatora szablonów projektów

Okno podzielone jest na obszary, które mają określone znaczenie:

- **Project Types** – zawiera listę możliwych typów projektów;
- **Templates** – zawiera listę szablonów projektów dla danego typu projektu;
- **Name** – okno pozwalające określić nazwę nowego projektu;
- **Location** – okno pozwalające określić folder, w którym będzie założony nowy projekt;
- **Solution** – lista rozwijana zawierająca opcje pozwalające utworzyć nową solucję dla projektu lub dodać projekt do istniejącej solucji.

Interesująca nas z punktu widzenia tematyki książki grupa szablonów projektów, znajduje się w typie projektów Windows.

Do najczęściej używanych szablonów należą:

- **Windows Application** – szablon projektu aplikacji okienkowej;
- **Windows Control Library** – szablon projektu komponentu interfejsowego;
- **Console Application** – szablon projektu aplikacji konsolowej;
- **Class Library** – szablon projektu biblioteki klas;
- **Windows Service** – szablon projektu usługi systemowej;



- *Empty Project* – szablon pustego projektu.

## Generowanie szablonu aplikacji konsolowej

Spróbujmy wygenerować na początek szablon dla aplikacji konsolowej. Napisaliśmy wcześniej program typu HelloWorld, możemy go teraz utworzyć i skompilować w środowisku:

1. W obszarze *Project Types* wybieramy *Windows*.
2. W obszarze *Templates* wybieramy *Console Application*.
3. W obszarze *Name* wpisujemy nazwę naszego projektu *HelloWorld*.
4. W obszarze *Location* określamy ścieżkę katalogu głównego, w którym będzie umieszczony katalog projektu.
5. Upewniamy się, że w obszarze *Solution* wybrano z listy *Create new Solution*.
6. Naciskamy przycisk *Ok*.

Po tym zabiegu zostanie wygenerowany szablon projektu:

```
⊞ Using directives
⊞ namespace HelloWorld
{
  ⊞ class Program
  {
    ⊞ static void Main(string[] args)
    {
    }
  }
}
```

Rysunek 17. Wygenerowany szablon aplikacji konsolowej z ukrytą sekcją dyrektyw *using*

Jak widać, mamy standardową strukturę programu poza pewnym wyjątkiem. Wygenerowany kod zawiera sekcję *Using directives*, która ukrywa domyślnie listę dyrektyw *using*. Aby ją zobaczyć, wystarczy kliknąć na krzyżyk przy danej sekcji:

```

#region Using directives
using System;
using System.Collections.Generic;
using System.Text;
#endregion

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            I
        }
    }
}

```

Rysunek 18. Wygenerowany szablon aplikacji konsolowej z rozwiniętą sekcją dyrektyw using

Teraz wystarczy wpisać instrukcje i mamy gotowy kod:

```

+ Using directives

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Jak masz na imię: ");
            string imie = Console.ReadLine();
            Console.WriteLine("{0} to ładne imię", imie);
        }
    }
}

```

Rysunek 19. Treść programu w wygenerowanym szablonie

## Kompilacja i uruchomienie

Kompilacja programu w środowisku Microsoft Visual C#.NET 2005 odbywa się poprzez wybranie z menu **Build** opcji **Build solution** lub naciśnięcie klawisza **F6**. Możliwe jest również skorzystanie z przycisków szybkiego uruchomienia, które dostępne są na pasku narzędziowym **Build**. Pasek ten domyślnie nie jest widoczny, więc jeżeli chcemy z niego korzystać, należy go uaktywnić poprzez wybór z menu **View** opcji **Toolbars->Build**. Pasek ten można zakotwiczyć obok pozostałych pasków, aby był łatwo dostępny.



Rysunek 20. Pasek narzędziowy Build

Jeżeli kod został napisany poprawnie, w oknie narzędziowym **Error List** nie będzie błędów i możliwe będzie uruchomienie programu.

Uruchomienie programu w trybie bez debugowania (śledzenia błędów) odbywa się przez wybranie z menu **Debug** opcji **Start Without Debugging** lub naciśnięcie kombinacji klawiszy **Ctrl+F5**.

Uruchomienie programu w trybie debugowania z kolei zapewnia możliwość pełnej kontroli wykonywania programu (krokowego wykonywania i śledzenia stanów programu oraz jego poszczególnych elementów).

Aby uruchomić program w trybie z debugowaniem, należy wybrać z menu **Debug** opcję **Start** lub nacisnąć klawisz **F5**. Możliwe jest również skorzystanie z przycisku szybkiego uruchomienia, który dostępny jest bądź na głównym pasku narzędzi, bądź na pasku **Debug** (zielony trójkąt).



Rysunek 21. Pasek narzędziowy Debug

Zagadnieniu debugowania poświęcony jest osobny rozdział tej książki.

## Rozdział 4.

# Typy

### Deklaracja zmiennej

Każda zmienna używana w programie ma swój typ danych, który określa, jakie wartości mogą być w niej przechowywane. Aby móc skorzystać ze zmiennej danego typu, należy ją zadeklarować.

Składnia deklaracji wygląda następująco:

```
typ identyfikator [=wartość];
```

gdzie:

*typ*

określa typ zmiennej (wymagane),

*identyfikator*

nazwa zmiennej (wymagane),

*wartość*

wartość początkowa zmiennej (opcjonalne)

Kiedy deklarujemy zmienną musimy przede wszystkim pamiętać o jej przeznaczeniu. Przykładowo, jeżeli chcemy wykonywać proste operacje na liczbach całkowitych, nie ma sensu używać typu zmiennoprzecinkowego (liczby te bowiem zajmują więcej miejsca w pamięci a operacje na nich wykonują się wolniej).

Deklarując zmienną, należy również pamiętać o tym, że nazwa zmiennej (identyfikator) podlega pewnym regułom i zaleceniom, których należy przestrzegać:

Reguły:

- nazwy mogą zaczynać się od litery lub znaku podkreślenia;
- po pierwszym znaku można używać liter, cyfr i podkreślenia;
- nie wolno używać słów kluczowych języka;

Źle: `23liczba`, `%wynik`, `for`

Dobrze: `liczba23`, `_wynik`, `For`

Zalecenia:

- unikaj nazw pisanych w całości dużymi literami;
- unikaj nazw zaczynających się od znaku podkreślenia;
- unikaj nieczytelnych, skrótowych nazw;
- używaj czytelnej notacji nazewnicznej.

Nieładnie: `MAX_WARTOSC`, `_wynik`, `lcn`

Ładnie: `MaxWartosc`, `Wynik`, `Licznik`

Typy zmiennych ogólnie dzielą się na:

- typy wartości (typy proste, typy wyliczeniowe, struktury);
- typy referencyjne (tablice, klasy, interfejsy, delegacje, typ *string*, typ *object*).

## Inicjacja zmiennej

W języku C# wymaga się inicjacji zmiennej określoną wartością przed użyciem jej. Jeżeli tego nie zrobimy kompilator poinformuje nas, o próbie użycia zmiennej niezainicjowanej (*Use of unassigned local variable [nazwa\_zmiennej]*).

Przypisać wartość do zmiennej można na dwa sposoby:

- przypisać jej wartość w czasie deklarowania  
`int wartosc = 123;`
- przypisać wartość już zadeklarowanej zmiennej  
`int wartosc;`  
`wartosc = 123;`

Możemy więc zadeklarować zmienną nie inicjując jej do momentu, aż będzie użyta w wyrażeniu:

```
using System;

class HelloWorld
{
    static void Main()
    {
        int wartosc1 = 1, wartosc2, wartosc3;

        Console.WriteLine("wartosc1 = ", wartosc1);
        wartosc2 = 4;
        Console.WriteLine("wartosc2 = ", wartosc2);
        //BŁĄD: użycie niezainicjowanej zmiennej
        Console.WriteLine("wartosc3 = ", wartosc3);
    }
}
```

## Słowa kluczowe

Aby uniknąć błędów związanych z nadawaniem zmiennym nazw, warto sprawdzić, czy nazwa nie jest słowem kluczowym języka (w środowisku Visual Studio .NET wszystkie kluczowe słowa są domyślnie podświetlane).

Poniższa tabelka przedstawia zestawienie słów kluczowych języka C#:

<i>abstract</i>	<i>as</i>	<i>base</i>	<i>bool</i>	<i>break</i>
<i>byte</i>	<i>case</i>	<i>catch</i>	<i>char</i>	<i>checked</i>
<i>class</i>	<i>const</i>	<i>continue</i>	<i>decimal</i>	<i>default</i>
<i>delegate</i>	<i>do</i>	<i>double</i>	<i>else</i>	<i>enum</i>
<i>event</i>	<i>explicit</i>	<i>extern</i>	<i>false</i>	<i>finally</i>
<i>fixed</i>	<i>float</i>	<i>for</i>	<i>foreach</i>	<i>goto</i>
<i>if</i>	<i>implicit</i>	<i>in</i>	<i>int</i>	<i>interface</i>

<i>internal</i>	<i>is</i>	<i>lock</i>	<i>long</i>	<i>namespace</i>
<i>new</i>	<i>null</i>	<i>object</i>	<i>operator</i>	<i>out</i>
<i>override</i>	<i>params</i>	<i>private</i>	<i>protected</i>	<i>public</i>
<i>readonly</i>	<i>ref</i>	<i>return</i>	<i>sbyte</i>	<i>sealed</i>
<i>short</i>	<i>sizeof</i>	<i>stackalloc</i>	<i>static</i>	<i>string</i>
<i>struct</i>	<i>switch</i>	<i>this</i>	<i>throw</i>	<i>true</i>
<i>try</i>	<i>typeof</i>	<i>uint</i>	<i>ulong</i>	<i>unchecked</i>
<i>unsafe</i>	<i>ushort</i>	<i>using</i>	<i>virtual</i>	<i>volatile</i>
<i>void</i>	<i>while</i>			

## Typy wartości

Zmienne danego typu wartości zawierają wyłącznie wartości tego typu. Każda zmienna danego typu wartości posiada swoją kopię danych, więc nie jest możliwe, aby operacje wykonywane na jednej zmiennej wpływały na wartość innej. Poza tym w języku C# istnieje obowiązek inicjowania zmiennych przed ich użyciem (dzięki takiemu podejściu, unika się błędów związanych z ich nie zainicjowaniem).

Typy wartości dzieli się na:

- typy proste (wbudowane)
- typy definiowane przez użytkownika: typy wyliczeniowe oraz struktury

Różnice między tymi dwoma grupami są minimalne, ponieważ typy definiowane przez użytkownika mogą być używane w taki sam sposób jak typy proste. Jedyna różnica polega na tym, że dla typów prostych można zdefiniować wartości literalne (np.: **true**, **false**). Wartość **null** nie jest dozwolona dla żadnego z typów wartości.

## Typy proste

Wszystkie typy proste (wbudowane) są słowami kluczowymi języka C#. Słowa te jednak są jedynie aliasami do zdefiniowanych w języku struktur. Oznacza to, że deklarując zmienną typu prostego, możemy użyć zamiennie słowa kluczowego lub nazwy struktury danego typu.

Poniższa tabela opisuje przyporządkowanie tych struktur słowom kluczowym, ich znaczenie oraz zakres dozwolonych wartości:

Słowo kluczowe	Alias do struktury	Znaczenie
<i>sbyte</i>	<i>System.SByte</i>	liczba całkowita ze znakiem z przedziału od -128 do 127
<i>byte</i>	<i>System.Byte</i>	liczba całkowita bez znaku z przedziału od 0 do 255
<i>short</i>	<i>System.Int16</i>	liczba całkowita ze znakiem z przedziału od -32 768 do 32 767
<i>ushort</i>	<i>System.UInt16</i>	liczba całkowita bez znaku z przedziału od 0 do 65 535
<i>int</i>	<i>System.Int32</i>	liczba całkowita ze znakiem z przedziału

		od -2 147 483 648 do 2 147 483 647
<i>uint</i>	<i>System.UInt32</i>	liczba całkowita bez znaku z przedziału od 0 do 4 294 967 295
<i>long</i>	<i>System.Int64</i>	liczba całkowita ze znakiem z przedziału od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
<i>ulong</i>	<i>System.UInt64</i>	liczba całkowita bez znaku z przedziału od 0 do 18 446 744 073 709 551 615
<i>char</i>	<i>System.Char</i>	znak, wartości można wprowadzać w postaci znaku (np. 'A'), sekwencji szesnastkowej (np. '\x0065') lub sekwencji Unicode (np. '\u0065')
<i>float</i>	<i>System.Single</i>	liczba zmiennoprzecinkowa z przedziału od $1.5 \cdot 10^{-45}$ do $3.4 \cdot 10^{38}$
<i>double</i>	<i>System.Double</i>	liczba zmiennoprzecinkowa z przedziału od $5.0 \cdot 10^{-324}$ do $1.7 \cdot 10^{308}$
<i>bool</i>	<i>System.Boolean</i>	wartość logiczna przyjmuje wartości literalne <i>true</i> (prawda) lub <i>false</i> (fałsz)
<i>decimal</i>	<i>System.Decimal</i>	liczba o dużej dokładności z przedziału od $1.0 \cdot 10^{-28}$ do $7.9 \cdot 10^{28}$

Jak już wcześniej pisałem, jeżeli chcemy skorzystać ze zmiennej danego typu, musimy ją zadeklarować:

Przykłady:

```
int Liczba = 0;
char Znak = 'X';
```

Jeżeli chcemy zadeklarować kilka zmiennych takiego samego typu, możemy podać nazwę typu, a następnie nazwy poszczególnych zmiennych oddzielonych przecinkami:

```
int Liczba1, Liczba2;
lub:
int Liczba1,
    Liczba2;
```

## Typ wyliczeniowy

Typ wyliczeniowy jest zbiorem stałych o określonych wartościach. Definiując nowy typ wyliczeniowy, programista określa zbiór dopuszczalnych wartości całkowitych, którym przyporządkowuje niepowtarzalne identyfikatory. Zmienna typu wyliczeniowego może zatem przechowywać jedynie wartości z tego zbioru.

Składnia definicji typu wyliczeniowego wygląda następująco:

```
[modyfikatory] enum identyfikator[:typ_bazowy]
{
    etykieta_stalej_1 [=wartość_1]
```

```
[,etykieta_stalej_2 [=wartosc_2]  
[,...[etykieta_stalej_N [=wartosc_N]]]  
];
```

gdzie:

*modyfikator*

modyfikator widoczności (opcjonalne),

*identyfikator*

nazwa typu wyliczeniowego (wymagane),

*typ\_bazowy*

typ bazowy całkowity do przechowywania danych wyliczeniowych (opcjonalne),

*etykieta\_stalej\_1, etykieta\_stalej\_2,..., etykieta\_stalej\_N*

składnik typu wyliczeniowego (opcjonalny),

*wartosc\_1, wartosc\_2, ..., wartosc\_N*

wartość, która nadpisuje wartość domyślną składnika (opcjonalne).

Przykład:

```
enum DniTygodnia  
{  
    Poniedzialek,  
    Wtorek,  
    Sroda,  
    Czwartek,  
    Piatek,  
    Sobota,  
    Niedziela  
};
```

Bazowym typem wartości dla typu wyliczeniowego są liczby całkowite. Jeżeli programista bezpośrednio nie wskaże konkretnego typu całkowitego, domyślnie zostanie przyjęty *int*. Jeżeli jednak chcemy wskazać jakiś inny typ całkowity, który ma być wykorzystany do zdefiniowania wyliczenia, wystarczy wskazać ten typ:

```
enum Opcje: uint  
{  
    Opcja1,  
    Opcja2,  
    Opcja3  
};
```

Jeżeli nie przypisze się identyfikatorom określonych wartości, zostaną nadane im kolejne wartości począwszy od 0 (Opcja1 będzie miał wartość 0, Opcja2 wartość 1, itd.). Chcąc nadać inne wartości identyfikatorom, wystarczy przypisać im tę wartość:

```
enum Temperatury  
{  
    Mroz    = -10,  
    Zimno   = 0,
```



```
Chlodno = 10,  
Cieplo  = 20,  
Upal    = 30  
};
```

Przypisanie wartości jest opcjonalne (można nadawać wartości wybiórczo, identyfikator bez wartości otrzyma wartość o jeden większą od poprzedniego, a jeżeli będzie pierwszym to otrzyma wartość 0):

```
enum Opcje  
{  
    Opcja1,          // wartość 0  
    Opcja2 = 10,     // wartość 10  
    Opcja3,          // wartość 11  
};
```

Aby zadeklarować zmienną typu wyliczeniowego wystarczy określić nazwę typu i nazwę zmiennej. Przykładowo, dla ostatniego typu Opcje:

```
Opcje Zmienna;
```

Przypisując wartość zmiennej typu wyliczeniowego, należy wskazać identyfikator ze zbioru dopuszczalnych wartości (wskazując typ i identyfikator):

```
Zmienna = Opcje.Opcja1;
```

## Struktura

Struktura jest typem wartości definiowanym przez użytkownika. Jest to typ, który może składać się z wielu innych typów. Struktura może zawierać: konstruktory, stałe, pola, metody, właściwości, indeksatory, operatory i typy zagnieżdżone. Typ ten powstał z myślą o wykorzystaniu go do definiowania małych obiektów (np.: adres *IP*). Strukturę definiuje się za pomocą słowa kluczowego *struct*.

Struktury zostały szczegółowo opisane w osobnym rozdziale.

## Typy referencyjne

Zmienne typu referencyjnego nie przechowują wartości tego typu, tylko odwołania do właściwych danych (obiektów przechowujących dane). Jest możliwe, aby dwie lub więcej zmiennych typu referencyjnego wskazywały na ten sam obiekt, co oznacza, że operowanie na jednej zmiennej może wpływać na pozostałe.

Do typów referencyjnych należą:

- typ *object*
- typ *string*
- tablica
- klasa
- interfejs
- delegacja

## Typ object

Typ *object* (alias do *System.Object*) jest typem, z którego wywodzą się pozostałe typy referencyjne. Oznacza to, że w zmiennej typu *object* można przechowywać dowolną referencję. Typ ten używany jest przy opakowywaniu typu wartości.

Przykład przypisania zmiennej typu *object* wartości zmiennoprzecinkowej:

```
object Obiekt=0.12345d;
```

## Typ string

Typ *string* (alias do *System.String*) przeznaczony jest do przechowywania łańcuchów znaków (napisów) oraz wykonywania na nich szeregu operacji. Klasa *System.String* jest klasą ostateczną, czyli taką, po której nie można już dziedziczyć (nie można tworzyć innych klas na jej podstawie). Korzystanie z tego typu jest bardzo łatwe i wygodne.

Przykład deklaracji:

```
string napis = "tekst";
```

Podstawową operacją wykonywaną na łańcuchach znaków jest konkatencja (łączenie). Do konkatencji używa się operatora +:

```
string napis = "Ala" + " ma kota";
```

Operacje na łańcuchach zostały szczegółowo omówione w osobnym rozdziale.

## Tablica

Tablica to szereg zmiennych (elementów) jednakowego typu, (który określa typ danych tablicy). Dostęp do każdego elementu tablicy odbywa się za pomocą tzw. *indeksów* (pozycji w szeregu). W tablicach można przechowywać obiekty dowolnego typu (przykładowo: tablica łańcuchów, tablica liczb całkowitych, itd.). Każda tablica składa się z pewnej liczby elementów, którą określa się mianem wielkości tablicy.

Tablice mogą być jedno lub wielowymiarowe a indeks związany z danym wymiarem przyjmuje wartość od 0 do wielkości tablicy w danym wymiarze pomniejszoną o jeden. Wszystkie zmienne referencyjne typu tablicowego dziedziczą z klasy *System.Array*.

Przykłady deklaracji:

```
int[] tablica; // referencja do tablicy jednowymiarowej typu int
string[] napisy; // referencja do tablicy jednowymiarowej typu łańcuchowego
byte[,] tab_dwu; // referencja do tablicy dwuwymiarowej typu byte
```

Tablice zostały szczegółowo omówione w osobnym rozdziale.

## Klasa

Klasa jest typem, który może zawierać różne składowe: pola, stałe, zdarzenia, konstruktory, destruktory, metody, właściwości, indeksatory, operatory oraz inne typy zagnieżdżone. Klasy podobne są do struktur z tą jednak różnicą, że struktury są typami wartości, przeznaczonymi do tworzenia małych obiektów a

klasy typami referencyjnymi, przeznaczonymi do tworzenia większych. W języku C# dozwolone jest dziedziczenie jedynie z jednej klasy. Klasę definiuje się za pomocą słowa kluczowego **class**.

Klasy zostały szczegółowo omówione w osobnym rozdziale.

## Interfejs

Interfejs jest typem, który może zawierać jedynie składowe abstrakcyjne. Jest to więc jedynie rodzaj sygnatury opisujący interfejs, który zostanie zaimplementowany później (przez jakąś klasę, która będzie dziedziczyć po tym interfejsie). W interfejsie można deklarować jedynie sygnatury: metod, właściwości i indeksatorów. Interfejs definiuje się za pomocą słowa kluczowego **interface**.

Interfejsy zostały szczegółowo omówione w osobnym rozdziale.

## Delegacja

Delegacja jest typem, który **"kapsułkuje"** (obudowuje) metodę za pomocą określonej sygnatury. Delegacje służą do obsługi zdarzeń związanych z daną klasą (można je traktować jako wskaźnik na funkcję, który w odróżnieniu od innych języków, jest typem chronionym).

Delegacje zostały szczegółowo omówione w osobnym rozdziale.

## Stałe

Stała jest rodzajem danej, która nie może zostać zmodyfikowana. Stosuje się ją wtedy, gdy wymagamy gwarancji, że dana, którą używamy zachowa taką samą wartość w każdym punkcie wykonania programu. Jakakolwiek próba zmiany wartości stałej, spowoduje pojawienie się błędu: ***"The left-hand side of an assignment must be a variable, property or indexer"*** (lewa strona przypisania musi być zmienną, właściwością lub indeksatorem).

Do definiowania stałych używa się słowa kluczowego **const**. W odróżnieniu do typowej zmiennej, stała musi być zainicjowana w trakcie deklaracji. Składnia deklaracji wygląda następująco:

```
const typ Identyfikator = wartość;
```

Przykład:

```
const double Wspolczynnik = 0.345234d;  
Wspolczynnik = 0.0d; // BŁĄD
```

W przypadku, gdy chcemy zdefiniować kilka stałych, należących logicznie do jednej grupy lepiej skorzystać z typu wyliczeniowego.

Zamiast definiować stałe:

```
const int Mroz    = -10;  
const int Zimno   = 0;  
const int Chlodno = 10;  
const int Cieplo  = 20;  
const int Upal    = 30;
```

lepiej zdefiniować typ wyliczeniowy Temperatury:

```
enum Temperatuty
{
    Mroz      = -10,
    Zimno     = 0,
    Chlodno   = 10,
    Cieplo    = 20,
    Upal      = 30
};
```

## Literały

Literał to tekstowa reprezentacja wartości danego typu. Wprowadzono je w celu ułatwienia przypisywania wartości zmiennej danego typu.

Literały dzielą się na:

- literały logiczne – przeznaczone są dla typu logicznego **bool**, dopuszczalne są dwie możliwe wartości: **true** (prawda) i **false** (fałsz)

Przykład:

```
bool zmienna = true;
```

- literały dla liczb całkowitych – przeznaczone dla typów reprezentujących liczby całkowite, wartości podaje się w notacji dziesiętnej (składające się z cyfr 0-9) lub szesnastkowej (liczby z prefixem 0x lub 0X, składające się z cyfr 0-9 i liter a-f, A-F). W zależności od typu całkowitego stosuje się literały:

- bez suffixu - przeznaczone dla typów: **int**, **uint**, **long**, **ulong**.

Przykład:

```
int zmienna = 16;
```

- z suffixem **u** lub **U** (unsigned, czyli bez znaku) – przeznaczone dla typów: **uint**, **ulong**.

Przykład:

```
uint zmienna = 16u;
```

- z suffixem **l** lub **L** (long, czyli liczby całkowite długie) – przeznaczone dla typów: **long**, **ulong**.

Przykład:

```
long zmienna = -1000000000L;
```

- z suffixem **UL**, **Ul**, **uL**, **ul**, **LU**, **Lu**, **lU**, lub **lu**, dla typu **ulong**.

Przykład:

```
ulong zmienna = 1000000000UL;
```

- literały dla liczb zmiennoprzecinkowych – przeznaczone dla typów reprezentujących liczby rzeczywiste (zmiennoprzecinkowe), wartości podaje się w notacji dziesiętnej z kropką oddzielającą część dziesiętną od całkowitej. W zależności od typu stosuje się literały:

- z suffixem **f** lub **F** dla typu **float**.

Przykład:

```
float zmienna = 123.456F;
```

- z suffixem **d** lub **D** dla typu **double**.

Przykład:

```
double zmienna = 123.456D;
```

• z suffixem *m* lub *M* dla typu *decimal*.

Przykład:

```
decimal zmienna = 123.456M;
```

- literały znakowe – przeznaczone dla typu **char**, wartości podaje się w pojedynczych cudzysłowach np.: 'A'

Poniższa tabelka przedstawia znaki szczególne:

Sekwencja uproszczona	Kodowanie Unicode	Nazwa
'	0x0027	Pojedynczy cudzysłów
"	0x0022	Cudzysłów
	0x005C	Backslash
0	0x0000	Null
a	0x0007	Alarm (sygnał dźwiękowy)
b	0x0008	Backspace
n	0x000A	Nowa linia
r	0x000C	Powrót karetki
t	0x0009	Tabulacja

Przykład:

```
char cudzyslow = '\\';
```

- literały łańcuchowe – przeznaczone dla typu **string**, wartości podaje się w cudzysłowie np.: "napis", podstawowym literałem jest symbol @, który wyłącza specjalne znaczenie sekwencji sterujących w łańcuchu formatującym, poza tym można w łańcuchu używać sekwencji uproszczonych (patrz literały znakowe).

Przykład:

```
string sciezka = @"c:\windows\"," // -> c:\windows\
zsekwencjami = "\\tala\\"; // -> ' ala'
```

- literał **null** – literał oznaczający "pusty", jest niedozwolony dla typów wartości.

Przykład:

```
string pustynapis = null;
```

## Konwersje

Konwersja jest operacją zmiany wartości danego typu na wartość innego typu. Konwersje mogą mieć rodzaj poszerzający i przybliżający.

Konwersja o charakterze poszerzającym, jest rodzajem konwersji z jednego typu do innego, który jest w stanie przechować wartość bez utraty informacji. Konwersje tego typu zawsze się udają. Inaczej dzieje się w przypadku konwersji o charakterze przybliżającym. Konwersja z jednego typu na drugi w tym przypadku nie gwarantuje, że typ zmiennej, który ma przechować konwertowaną wartość, będzie w stanie ją pomieścić.

Istnieją trzy typy konwersji:

- niejawna (implicit) – może wystąpić w czasie operacji przypisania do zmiennej innego typu lub w czasie wywołania metody. Konwersja niejawna musi mieć charakter poszerzający.

Przykład:

```
int iWartosc = 123;
long lWartosc = iWartosc;
short sWartosc = lWartosc; // BŁĄD
```

- jawna (explicite) – występuje wyłącznie wtedy, gdy użytkownik wskaże, że ma nastąpić konwersja do zadanego typu (użyje operatora konwersji). Konwersja jawna może mieć charakter poszerzający lub przybliżający.

Przykład:

```
// maksymalna wartość dla typu
long lWartosc = Int64.MaxValue;
/* konwersja przybliżająca, zmiennej typu long do zmiennej typu short
z utratą informacji, w tym przypadku zmiana short nie pomieści
wartości zmiennej lWartosc równej jej maksymalnej wartości dla
typu long, więc wartość zostanie ona obcięta (dopasowana do
pojemności nowej zmiennej)*/
short sWartosc = (short)lWartosc;
// konwersja poszerzająca, może być niejawna lub jawna
int iWartosc = (int)sWartosc;
```

Przykład programu wykorzystującego konwersję o charakterze przybliżającym:

```
using System;

namespace Konwersje
{
    class Konwersje
    {
        static void Main()
        {
            ulong ulLiczba;
            uint uiLiczba;
            ushort usLiczba;
```

```
byte bLiczba;

    ulLiczba=UInt64.MaxValue;
    uiLiczba=(uint)ulLiczba;
    Console.WriteLine(" ",ulLiczba, uiLiczba);
    uiLiczba=UInt32.MaxValue;
    usLiczba=(ushort)uiLiczba;
    Console.WriteLine(" ",uiLiczba, usLiczba);
    usLiczba=UInt16.MaxValue;
    bLiczba=(byte)usLiczba;
    Console.WriteLine(" ",usLiczba, bLiczba);
}
}
}
```

W wyniku działania tego programu otrzymamy:

```
18446744073709551615 4294967295
4294967295 65535
65535 255
```

- poprzez użycie metod do konwersji – metody do konwersji można znaleźć w klasie

#### ***System.Convert.***

Większość metod do konwersji posiada nazwę składającą się z dwóch członów: ***To*** (czyli do) oraz nazwy struktury reprezentującej dany typ (np. ***Int16***, ***String***). Dzięki temu łatwo odnaleźć metodę realizującą określony typ konwersji (np. konwersja do ***int*** reprezentowanego przez strukturę ***Int32*** realizowana jest przez metodę ***ToInt32***).

W przypadku konwersji między typami, dla których konwersja nie ma sensu lub w przypadku, gdy może nastąpić utrata znaczącej wartości, zostanie zwrócony wyjątek. Oznacza to, że pisząc kod można określić dowolny rodzaj konwersji (np. z napisu na liczbę), jednak w przypadku braku możliwości takiej konwersji, pojawi się określony wyjątek (np.: napis „1” można przekonwertować na liczbę, ale „jeden” już nie ma sensu).

Przykład:

```
long lWartosc = Int64.MaxValue;
// Wyjątek System.OverflowException - przepełnienie
int iWartosc = System.Convert.ToInt32(lWartosc);
string strLiczba = "1";
// poprawna konwersja typu napisowego na liczbę
short sWartosc = System.Convert.ToInt16(strLiczba);
strLiczba = "jeden";
// Wyjątek System.FormatException - błędny format
sWartosc = System.Convert.ToInt16(strLiczba);
```

Nie wszystkie konwersje typu jawnego i niejawnego między typami są dozwolone:

Typ konwertowany	Dozwolona konwersja niejawna do typu	Dozwolona konwersja jawna do typu
<i>sbyte</i>	<i>short, int, long, float, double, decimal</i>	<i>byte, ushort, uint, ulong, char</i>
<i>byte</i>	<i>short, ushort, int, uint, long, ulong, float, double, decimal</i>	<i>sbyte, char</i>
<i>short</i>	<i>int, long, float, double, decimal</i>	<i>sbyte, byte, ushort, uint, ulong, char</i>
<i>ushort</i>	<i>int, uint, long, ulong, float, double, decimal</i>	<i>sbyte, byte, short, char</i>
<i>int</i>	<i>long, float, double, decimal</i>	<i>sbyte, byte, short, ushort, uint, ulong, char</i>
<i>uint</i>	<i>long, ulong, float, double, decimal</i>	<i>sbyte, byte, short, ushort, int, char</i>
<i>long</i>	<i>float, double, decimal</i>	<i>sbyte, byte, short, ushort, int, uint, ulong, char</i>
<i>ulong</i>	<i>float, double, decimal</i>	<i>sbyte, byte, short, ushort, int, uint, long, char</i>
<i>decimal</i>	<i>niedozwolone</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double</i>
<i>float</i>	<i>double</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal</i>
<i>double</i>	<i>niedozwolone</i>	<i>sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal</i>
<i>char</i>	<i>ushort, int, uint, long, ulong, float, double, decimal</i>	<i>sbyte, byte, short</i>
<i>bool</i>	<i>niedozwolone</i>	<i>niedozwolone</i>

## Opakowywanie i rozpakowywanie

Mechanizm opakowywania i rozpakowywania tworzy pomost między typami wartości a typami referencyjnymi. Pozwala on na przekształcenie dowolnego typu wartości w obiekt i vice versa. Dzięki temu mechanizmowi, korzystając z dowolnego typu wartości, korzystamy z możliwości jakie dają nam obiekty.

Opakowywanie zmiennej danego typu wartości jest operacją przekształcającą ją w typ **object**. Odbywa się to przez utworzenie nowej instancji obiektu typu **object** i skopiowania do niego wartości zmiennej, która jest opakowywana.

Przykład:

```
int iWartosc = 123;
object oWartosc = iWartosc;
```

W powyższym przykładzie wartość zmiennej *iWartosc* została skopiowana do obiektu *oWartosc* w sposób niejawny. Obie wartości są niezależne, więc wszelkie zmiany wartości jednej ze zmiennych nie wpłyną na wartość drugiej zmiennej.

Rozpakowywanie jest operacją odwrotną do opakowywania i wymaga jawnego podania typu wartości, jaki ma zostać wyciągnięty z obiektu.

Przykład:



```
int iWartosc = 123;  
object oWartosc = iWartosc;  
int iNowaWartosc = (int)oWartosc; // wskazanie typu int
```

Jeżeli podamy przez pomyłkę niewłaściwy typ wartości, próba rozpakowania zakończy się błędem rzutowania (wyjątek *InvalidCastException*).

Przykładowa błędna operacja odpakowywania dla naszego przypadku:

```
double dWartosc = (double)oWartosc; // wyjątek InvalidCastException
```

## Rozdział 5.

# Operatory i wyrażenia

## Wyrażenia

Po zadeklarowaniu i zainicjowaniu zmiennej można używać jej w wyrażeniach. Wyrażenie to sposób przetwarzania zmiennych za pomocą różnego rodzaju operatorów.

Wyrażenia dzielą się na:

- proste (operujące na jednej lub dwóch zmiennych)
- złożone (operujące na wielu zmiennych).

Wyrażenia złożone składają się z dwóch lub więcej wyrażen prostych. Wyliczanie wartości wyrażenia wykonuje się w kolejności uzależnionej od rodzaju wiązania (od lewej do prawej lub od prawej do lewej) z uwzględnieniem priorytetów operatorów oraz nawiasów grupujących, (jeżeli ich użyjemy).

Przyjrzyjmy się wyrażeniu:

```
int x;
x = 1+9/3*8-2; // x=23
```

Jak zostanie ono potraktowane?

Wyrażenie to zostanie pogrupowane w wyrażenia proste w następujący sposób:  $1+(((9/3)*8))-2$ . Oczywiście możemy zmienić kolejność wykonywania obliczeń, jeżeli chcemy wskazać, aby jakieś wyrażenie zostało policzone w innym porządku. Służą do tego nawiasy grupujące `()`.

Na przykład dla naszego wcześniejszego wyrażenia inne pogrupowanie wyrażen prostych:

```
int x;
x = 1+((9/3)*(8-2)); // x=19
```

Poniższa tabela określa priorytety i wiązania operatorów:

Kategoria	Operatory	Wiązanie
Nadrzędne	<code>., (), [], x++, x--, new, typeof</code>	od lewej do prawej ( <code>., (), []</code> ) od prawej do lewej ( <code>x++, x--, new, typeof</code> )
Unarne	<code>(typ), +, -, !, ~, ++x, --x</code>	od prawej do lewej
Dzielenie, mnożenie, reszta z dzielenia	<code>/, *, %</code>	od lewej do prawej
Dodawanie, odejmowanie	<code>+, -</code>	od lewej do prawej
Relacje	<code>&lt;, &gt;, &lt;=, &gt;=, is, as</code>	od lewej do prawej
Równość, różność	<code>=, !=</code>	od lewej do prawej
Iloczyn bitowy	<code>&amp;</code>	od lewej do prawej
Suma bitowa	<code> </code>	od lewej do prawej
Różnica symetryczna	<code>^</code>	od lewej do prawej
Iloczyn warunków	<code>&amp;&amp;</code>	od lewej do prawej
Suma warunków	<code>  </code>	od lewej do prawej

Wyrażenie warunkowe	?:	od prawej do lewej
---------------------	----	--------------------

## Operatory

Operatory można pogrupować ze względu na ich przeznaczenie. Poniższa tabela przedstawia podstawowe operatory używane w wyrażeniach:

Kategoria operatora	Operatory
Arytmetyczne	+, -, *, /, %
Logiczne bitowe	&,  , ^, ~
Logiczne warunkowe	&&,   , !
Konkatenacja łańcuchów	+
Jednostkowego zwiększania i zmniejszania	++, --
Przesunięcia	<<, >>
Relacji	==, !=, <, >, <=, >=
Przypisania	=, +=, -=, *=, /=, %=, &=,  =, ^=, <<=, >>=
Dostępu do składnika klasy	.
Indeksowania	[]
Rzutowania i grupowania	()
Informacji o typie	as, is, sizeof, typeof
Tworzenia obiektu	new
Wyrażenia warunkowego	?:

Operatory ogólnie dzielą się na dwie grupy:

- unarne (operują bezpośrednio na jednej zmiennej i dotyczą tylko jej np.: -5, +7, liczba++)
- binarne (operują bezpośrednio na dwóch zmiennych np.: 1\*9, liczba+=5)

Przyjrzyjmy się bliżej każdej grupie operatorów i wyjaśnijmy ich znaczenie w wyrażeniu:

### Operatory arytmetyczne

Służą do wykonywania prostych operacji arytmetycznych.

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażen
+	unarny/binarny	dodawanie	+5, 2+2, zmienna1+zmienna2
-	unarny/binarny	odejmowanie	-5, 2-2, zmienna1-zmienna2
*	binarny	mnożenie	1.5*5, zmienna*5, zmienna1*zmienna2
/	binarny	dzielenie	5/0.5, zmienna/1.1, zmienna2/zmienna1
%	binarny	reszta z dzielenia	10%2, zmienna%5, zmienna2%zmienna1

Przykład:

```
int wynik;
int = -4*(1+2)*3;
```

## Operatory logiczne bitowe

Służą do wykonywania operacji na bitach.

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażeń
&	binarny	logiczny iloczyn (AND)	0x07 & 0x08
	binarny	logiczna suma (OR)	0x07   0x08
~	unarny	logiczna negacja (NOT)	~0x08
^	binarny	logiczna różnica symetryczna (XOR)	0x08 ^ 0x01

Obliczanie wyrażenia z użyciem operatorów bitowych polega na wykonywaniu operacji bitowych na parach odpowiadających sobie bitów dwóch zmiennych w przypadku operatora: iloczynu (AND), sumy (OR) oraz różnicy symetrycznej (XOR) lub na bitach jednej zmiennej w przypadku operatora negacji (NOT).

Poniższa tabelka opisuje działania na bitach (b1 – bit zmiennej 1, b2 – bit zmiennej 2):

b1	b2	~b1 (NOT)	~b2 (NOT)	b1 & b2 (AND)	b1   b2 (OR)	b1 ^ b2 (XOR)
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1

Właściwości wyrażeń bitowych można wykorzystać do opisanego stanów jakiegoś elementu. Jeżeli przyjmiemy, że dany element może przyjąć różne stany i każdemu z nich przypiszemy określoną wartość bitową, możemy opisać aktualny stan elementu operując na poszczególnych bitach zmiennej opisującej stany.

Przykładowo powiedzmy, że mamy urządzenie, które posiada trzy przyciski kontrolujące czujniki: temperatury, ciśnienia i przepływu. Możemy zatem zdefiniować sobie typ wyliczeniowy nadając kolejnym stanom odpowiednie wartości bitowe:

```
enum Czujniki
{
    Temperatury = 0x01,    // 001, dziesiętnie 1
    Cisnienia = 0x02,      // 010, dziesiętnie 2
    Przepływu = 0x04,      // 100, dziesiętnie 3
}
```

Następnie definiujemy sobie zmienną opisującą aktualny stan:

```
Czujniki AktualnyStan;
```

Przyciski mogą być włączane w dowolnej konfiguracji i kolejności. Przyjmijmy, że początkowo wszystkie są włączone:

```
AktualnyStan = Czujniki.Temperatury | Czujniki.Cisnienia |
                Czujniki.Przeplywu;
```

Jeżeli chcemy włączyć lub wyłączyć dany przycisk i sprawić, aby określony czujnik zmienił stan swojej aktywności, wystarczy użyć operatora różnicy symetrycznej. Przykładowo wyłączmy czujnik temperatury:

```
AktualnyStan = AktualnyStan ^ Czujniki.Temperatury;
```

Teraz za pomocą operatora iloczynu bitowego możemy sprawdzić czy dany czujnik jest włączony:

```
AktualnyStan = AktualnyStan & Czujniki.Temperatury;
```

Zobaczmy jak wygląda to w całości:

```
using System;
class StanCzujnikow
{
    enum Czujniki
    {
        Temperatury = 0x01,    // 001
        Cisnienia = 0x02,      // 010
        Przeplywu = 0x04,      // 100
    }
    static void Main()
    {
        Czujniki AktualnyStan;

        // wszystkie włączone
        AktualnyStan = Czujniki.Temperatury | Czujniki.Cisnienia | Czujniki.Przeplywu;
        // wyłączamy czujnik temperatury (był włączony)
        AktualnyStan = AktualnyStan ^ Czujniki.Temperatury;
        // sprawdzamy czy włączony
        AktualnyStan = AktualnyStan & Czujniki.Temperatury;
        // Na ekranie pojawi się wartość 0 znaczy, że czujnik nie jest włączony
        Console.WriteLine("", AktualnyStan);
        // włączamy czujnik temperatury (był wyłączony)
        AktualnyStan = AktualnyStan ^ Czujniki.Temperatury;
        // sprawdzamy czy włączony
        AktualnyStan = AktualnyStan & Czujniki.Temperatury;
        // Na ekranie pojawi się wartość Temperatury znaczy, że czujnik jest włączony
        Console.WriteLine("", AktualnyStan);
    }
}
```

## Operatory logiczne warunkowe

Służą do wykonywania operacji logicznych (do łączenia wyrażeń relacji).

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażeń
<b>&amp;&amp;</b>	binarny	iloczyn warunków	(i==5) && (x<20)
<b>  </b>	binarny	suma warunków	(i<5)    (i=9)
<b>!</b>	unarny	negacja warunku	(!zmienna_logiczna)

Operatory logiczne warunkowe różnią się od bitowych tym, że przeznaczone są wyłącznie do operowania na wartościach typu **bool**. Wykorzystuje się je do łączenia wyrażeń, które w wyniku dają wartość typu **bool** (np.: operatory relacji).

Poniższa tabelka opisuje wyniki wyrażeń z operatorami warunkowymi

(zb1 – zmienna1 typu bool, zb2 – zmienna2 typu bool):

zb1	zb2	!zb1	!zb2	zb1 && zb2	zb1   zb2
<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>

Przykład:

```
bool wynik;

wynik = (1==1) && (1!=1); // wynik = false
wynik = (1==1) || (1!=1); // wynik = true
wynik = !((1==1) && (1!=1)); // wynik = true
```

## Operator konkatencji

Służy do łączenia napisów w całość.

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażeń
<b>+</b>	binarny	łączenie łańcuchów	"Ala ma " + "kota"

Przykład:

```
string str = "Ala ma " + "kota";
```

## Operatory jednostkowego zmniejszania i zwiększania

Służą do zmniejszania i zwiększania wartości liczby całkowitej o 1.

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażeń
<b>++</b>	unarny	zwiększanie o jeden	zmienna++, ++zmienna
<b>--</b>	unarny	zmniejszanie o jeden	zmienna--, --zmienna

Istnieją dwa sposoby zapisu operatora jednostkowego zmniejszania i zwiększania:

- przedrostkowy (++zmienna, --zmienna) – zwiększa/zmniejsza wartość zmiennej o jeden a następnie zwraca jej wartość do wyrażenia;
- przyrostkowy (zmienna++, zmienna--) – zwraca wartość zmiennej do wyrażenia a następnie zmniejsza/zwiększa jej wartość o jeden.

Przykładowy program:

```
using System;

class Wyniki
{
    static void Main()
    {
        int wynik1, wynik2, wynik3 , liczba = 2;

        wynik1 = liczba++;
        wynik2 = ++liczba;
        wynik3 = liczba++--liczba;
        Console.WriteLine("wynik1=\nwynik2=\nwynik3=", wynik1, wynik2, wynik3);
    }
}
```

Po wykonaniu powyższego programu na ekranie wyświetlą się następujące wyniki:

```
wynik1=2;
wynik2=4;
wynik3=8;
```

Skąd się wzięły te wartości? To proste. Początkowo zmiennej liczba nadano wartość 2. Następnie do zmiennej wynik1 została przypisana wartość zmiennej liczba, która po tej operacji zwiększyła swą wartość o jeden (operator przyrostkowy). Po tej operacji zmienna wynik1 otrzymała wartość 2 a zmienna liczba wartość 3. W kolejnym wyrażeniu w pierwszej kolejności zmienna liczba zwiększa swą wartość o jeden a następnie jej nowa wartość zostaje przypisana do zmiennej wynik2 (operator przedrostkowy). Po tej operacji zmienne liczba i wynik2 mają taką samą wartość równą 4. Kolejne wyrażenie jest bardziej złożone. Ze względu na priorytety operatorów zostanie ono potraktowane jak wyrażenie (liczba++) + (--liczba). Zatem w pierwszej kolejności zmienna liczba zwróci swą dotychczasową wartość, czyli 4 i zwiększy o jeden (liczba++), w dalszej jednak części zmniejszy swą wartość o jeden (--liczba) i zwróci nową wartość, czyli 4 (było 5 przez wcześniejsze działanie: liczba++), po czym obie wartości zostaną zsumowane i przypisane zmiennej wynik3 (4+4=8).

## Operatory przesunięcia

Służą do przesuwania wartości liczby całkowitej o n bitów w prawo lub w lewo.

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażen
<<	binarny	przesunięcie w lewo	1 << 2
>>	binarny	przesunięcie w prawo	2 >> 1

Przykład:

```
int wynik;
wynik = 1 << 2; // wynik = 4
wynik = 2 >> 1; // wynik = 1
```

Operatory przesunięcia, podobnie jak operatory binarne modyfikują wartość liczby poprzez działania na jej bitach. Po lewej stronie wyrażenia umieszczą się liczbę, na której operujemy a po prawej określa o ile bitów należy ją przesunąć.

## Operatory relacji

Wynik porównania daje zawsze jedną z wartości typu **bool**: **true** (jeżeli wyrażenie jest prawdziwe) lub **false** (jeżeli wyrażenie nie jest prawdziwe).

Operator	Rodzaj	Przeznaczenie	Przykłady wyrażień
==	binarny	równy	zmienna == 0, 1==1
!=	binarny	różny	zmienna != 0, 1!=2
<	binarny	mniejszy	zmienna < 0, 1 < 2
>	binarny	większy	zmienna > 0, 3 > 2
<=	binarny	mniejszy równy	zmienna <= 0
>=	binarny	większy	zmienna >= 0

Przykład:

```
bool wynik;
wynik = (1 == 1); // wynik = true
wynik = (1 != 1); // wynik = false
```

## Operatory przypisania

Wymagają po lewej stronie wyrażenia kontenera wartości (elementu przechowującego wartość wyrażenia: zmienna, własność, pole indeksatora).

Operator	Przeznaczenie	Przykłady wyrażień
=	przypisanie	zmienna = 1
+=	mnożenie z przypisaniem	zmienna += 1
-=	dzielenie z przypisaniem	zmienna -= 1
*=	dodawanie z przypisaniem	zmienna *= 1
/=	odejmowanie z przypisaniem	zmienna /= 1
%=	reszta z dzielenia z przypisaniem	zmienna %= 1
&=	iloczyn bitowy z przypisaniem	zmienna &= true
=	suma bitowa z przypisaniem	zmienna  = true
^=	różnica symetryczna z przypisaniem	zmienna ^= true
<<=	przesunięcie w lewo z przypisaniem	zmienna <<= 2
>>=	przesunięcie w prawo z przypisaniem	zmienna >>= 1



Operatory przypisania typu kontener\_wartości [operator]= wyrażenie są skrótowym zapisem kontener\_wartości = kontener\_wartości [operator] wyrażenie. Czyli pisząc wyrażenie zmienna += 2 tak naprawdę piszemy wyrażenie zmienna = zmienna + 2.

Przykład:

```
int wynik = 2;
wynik += 2; // wynik = 4
wynik -= 2; // wynik = 2
```

## Operator dostępu do składnika klasy

Operator ten służy do odwołania się do składnika klasy (np. metody).

Składnia wygląda następująco:

```
nazwa1.nazwa2
```

gdzie:

*nazwa1*

nazwa klasy lub przestrzeni nazw,

*nazwa2*

nazwa składnika klasy.

Przykład wywołania statycznej metody **WriteLine**, będącej składnikiem klasy **Console**:

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

Operator kropki może pojawiać się w pojedynczym odwołaniu wiele razy (np. w przypadku odwoływania się do składnika zagnieżdżonego). Przykład wywołania statycznej metody **WriteLine**, będącej składnikiem klasy **Console**, która należy do przestrzeni nazw **System**:

```
System.Console.WriteLine("Tekst");
```

## Operator wyrażenia warunkowego ?:

Służy do zwracania jednej z dwóch wartości w zależności od wyniku warunku zawartego w tym wyrażeniu. Składnia wyrażenia wygląda następująco:

```
warunek ? wyrażenie_1 : wyrażenie_2
```

gdzie:

*warunek*

wyrażenie dające wartość typu **bool** (np. porównanie),

*wartość\_1*

wartość zwracana w przypadku, gdy warunek jest spełniony,

*wartość\_2*

wartość zwracana w przypadku, gdy warunek nie jest spełniony.

Przykład:

```
double a=2, b=2, c;
c = (b!=0) ? a/b : 0;
```

Operator wyrażenia warunkowego jest operatorem wiążącym od prawej do lewej, więc każde wyrażenie

typu:

`a ? b : c ? d : e`

zostanie potraktowane jako:

`a ? b : (c ? d : e)`

Przykładowy program wykorzystujący zagnieżdżone wyrażenie warunkowe:

```
using System;

namespace Kalkulator
{
    class Kalkulator
    {
        static void Main()
        {
            double dLiczba1, dLiczba2, dWynik;
            string strNapis;
            char cZnakOperacji;

            Console.Write("Podaj wartość pierwszej zmiennej Liczba1 = ");
            strNapis = Console.ReadLine();
            dLiczba1 = Convert.ToDouble(strNapis);
            Console.Write("Podaj wartość pierwszej zmiennej Liczba2 = ");
            strNapis = Console.ReadLine();
            dLiczba2 = Convert.ToDouble(strNapis);
            Console.Write("Podaj rodzaj operacji (+, -, *, /):");
            cZnakOperacji = (char) Console.Read();
            dWynik = (cZnakOperacji == '+') ? dLiczba1 + dLiczba2 :
                ((cZnakOperacji == '-') ? dLiczba1 - dLiczba2 :
                ((cZnakOperacji == '*') ? dLiczba1 * dLiczba2 :
                ((cZnakOperacji == '/') ? dLiczba1 / dLiczba2 : 0.0d)));
            Console.WriteLine("Wynik = ", dWynik);
        }
    }
}
```

W powyższym przykładzie w pierwszej kolejności wprowadzane są liczby, na których zostaną wykonane działania dodawania, odejmowania, mnożenia i dzielenia. W zależności od znaku symbolizującego dane działanie, zostanie wykonane wyrażenie zgodne z operatorem.

## Operator is

Operator ten służy do sprawdzania czy konwersja między typami może być wykonana.

Składnia wygląda następująco:

`wyrażenie is typ`

gdzie:

*wyrażenie*

wyrażenie typu referencyjnego,

*typ*

typ referencyjny.

Przykład:

```
string Napis_1, Napis_2, Napis_3;
object obj;
int Liczba;

Napis_1 = "Test";
obj = (object)Napis_1;
// "Test" bo obj zawiera string
Napis_2 = (obj is string) ? (string)obj : null;
Liczba = 100;
obj = (object)Liczba;
// null bo obj zawiera int
Napis_3 = (obj is string) ? (string)obj : null;
```

## Operator as

Operator ten służy do konwersji między zgodnymi typami danych.

Składnia wygląda następująco:

wyrażenie *is* typ

gdzie:

*wyrażenie*

wyrażenie typu referencyjnego

*typ*

typ referencyjny

Przykład:

```
string Napis_1, Napis_2, Napis_3;
object obj;
int Liczba;

Napis_1 = "Test";
obj = (object)Napis_1;
Napis_2 = obj as string; // "Test" bo obj zawiera string
Liczba = 100;
obj = (object)Liczba;
Napis_3 = obj as string; // null bo obj zawiera int
```

Wyrażenie z użyciem operatora *as* jest równoważne wyrażeniu (patrz przykład w opisie operatora *is*):

```
(wyrażenie is typ) ? (typ)wyrażenie : (typ)null;
```

## Znaki ignorowane w wyrażeniach

Znaki spacji, tabulacji oraz przejścia do nowej linii należą do grupy znaków, które są ignorowane w wyrażeniach. Można więc napisać:

```
int iLiczba=10;
```

lub:

```
int      iLiczba  =          10;
```

Są jednak wyjątki. Przede wszystkim znak spacji jest wymagany zawsze między deklaracją typu zmiennej a jej nazwą. Jeżeli go pominiemy, kompilator nie będzie w stanie rozpoznać typu:

```
intiLiczba = 123; // BŁĄD
```

Z kolei w przypadku łańcucha znak spacji wewnątrz łańcucha jest traktowany jako jego część:

```
string napis="Ala ma kota"; // spacje są częścią napisu
```

Przypadek przejścia do nowej linii wewnątrz łańcucha (gdy początek łańcucha znajduje się w innej linii kodu niż jego koniec), jest traktowany jako błąd. Aby uniknąć tego błędu należy w takim wypadku zamknąć w jednej linii ciąg i otworzyć w następnej (za pomocą znaku "):

```
string napis="Ala ma "  
            "kota";
```

## Rozdział 6.

# Instrukcje sterujące

### Wprowadzenie

Instrukcje sterujące stanowią jeden z najważniejszych elementów w każdym języku programowania. Pozwalają one sterować przebiegiem programu. Instrukcje te w połączeniu z wyrażeniami, pozwalają na zapisanie dowolnego algorytmu działania programu.

Instrukcje sterujące w języku C# można podzielić na:

- instrukcje wyboru
- instrukcje iteracji
- instrukcje skoku

### Instrukcja pusta

W języku C# można wyróżnić tzw.: instrukcję pustą, która nie powoduje wykonania żadnych czynności, ale jest traktowana jak każda inna instrukcja. Instrukcję taką zapisuje się w postaci samego znaku średnika (nie poprzedzonego żadnym wyrażeniem czy słowem kluczowym). Dla poprawienia czytelności kodu zaleca się umieszczanie go w osobnej linii.

Przykład:

```
while (true) // pętla nieskończona
;           // która nic nie robi (powtarza pustą instrukcję)
```

### Blok instrukcji

Grupę instrukcji znajdującą się między nawiasami: otwierającym { i zamykającym } określa się mianem bloku instrukcji:

```
{
    //instrukcje
}
```

Każdy blok może zawierać dowolną liczbę instrukcji oraz innych bloków zagnieżdżonych wewnątrz. Każdy blok definiuje widoczność zmiennych lokalnych. Zmienna lokalna zadeklarowana w określonym bloku instrukcji jest dostępna od chwili jej zadeklarowania do końca bloku. Deklarując zmienne w blokach instrukcji należy pamiętać o tym, że nazwa zmiennej w bloku zagnieżdżonym (wewnętrznym) nie może się pokrywać z nazwą zmiennej zadeklarowanej w bloku nadrzędnym (zewnętrznym):

```
{
    int iLiczba;
    ...
}
```

```
// BŁĄD: zmienna o tej nazwie zadeklarowana jest w bloku zewnętrznym
int iLiczba;

...
}
}
```

Można jednak deklarować zmienne o tej samej nazwie w blokach niezagnieżdżonych:

```
{
    int iLiczba;
    ...
}

...
{
    int iLiczba;
    ...
}
```

Dla przejrzystości kodu zaleca się deklarowanie zmiennych na samym początku bloku (nie jest to jednak wymagane).

## Instrukcje wyboru

Niemal w każdym algorytmie zachodzi potrzeba sprawdzania czy zachodzą określone warunki i w zależności od tego wykonania pewnych instrukcji (lub nie). Przykładowo, zanim podzielimy jakąś zmienną przez inną, warto sprawdzić czy zmienna pełniąca rolę dzielnika nie ma wartości zero i operację dzielenia wykonać wyłącznie w przypadku, gdy ma wartość różną od zera.

Do grupy instrukcji warunkowych należą:

- instrukcja *if*
- instrukcja *switch*

### Instrukcja if

Instrukcja *if* („jeżeli”) jest podstawową instrukcją wyboru, pozwalającą na wykonanie grupy instrukcji w oparciu o wartość logiczną badanego wyrażenia.

Składnia instrukcji *if* wygląda następująco:

```
if (wyrażenie)
    instrukcja_1
[else
    instrukcja_2]
```

gdzie:

*wyrażenie*

wyrażenie sprawdzające, które daje w wyniku wartość typu *bool* (wymagane),

*instrukcja\_1*

instrukcja (zakończona średnikiem) lub grupa instrukcji umieszczona w bloku wykonywana w przypadku, gdy wartość wyrażenia jest prawdą (wymagane),

[instrukcja\\_2](#)

instrukcja (zakończona średnikiem) lub grupa instrukcji umieszczona w bloku wykonywana w przypadku, gdy wartość wyrażenia nie jest prawdą (opcjonalne).

### Klauzula else

Klauzula **else** („w innym wypadku”) stanowiąca część instrukcji **if** jest opcjonalna. W przypadku, gdy chcemy jedynie wykonać jakiś fragment kodu, gdy zachodzi określony warunek, nie ma potrzeby umieszczania tej klauzuli:

```
if (dzielnik != 0)
    wynik = dzielna / dzielnik;
```

Jeżeli jednak chcemy, aby w przypadku, gdy warunek nie jest prawdziwy wykonała się jakaś inna instrukcja, wtedy używamy klauzuli **else**:

```
if (dzielnik != 0)
    wynik = dzielna / dzielnik;
else
    Console.WriteLine("Dzielnik nie może być zerem!");
```

Należy zwrócić uwagę na to, że przed klauzulą **else** w przypadku pojedynczej instrukcji wymagany jest znak średnika, a w przypadku bloku - nie:

```
if (dzielnik != 0)
{
    wynik = dzielna / dzielnik;
    // inne instrukcje
}
else
{
    Console.WriteLine("Dzielnik nie może być zerem!");
    // inne instrukcje
}
```

Ponieważ wartość wyrażenia instrukcji warunkowej musi być typu **bool**, można stosować tylko wyrażenia, które w rezultacie zwracają wartość tego typu.

### Układanie warunków

Naczelną zasadą przy układaniu warunków jest analiza zdania definiującego warunek. Powiedzmy, że chcemy zdefiniować warunek dla takiego przypadku: zmienna *y* zwiększy swą wartość o jeden, jedynie, gdy zmienna *x* przyjmie jedną z wartości z przedziału: od 0 do 5 lub od 10 do 20.

Zdanie określające wymaganie można rozebrać logicznie na człony: jeżeli „przedział od 0 do 5” lub „przedział od 10 do 20”. Każdy z członów dotyczący przedziału również da się przekształcić na: „większe lub równe 0 i mniejsze lub równe 5” oraz „większe lub równe 10 i mniejsze lub równe 20”. Teraz zamieniamy zdania na wyrażenia. Rolę łącznika zdania „i” spełnia operator **&&** a rolę łącznika

„lub” operator `||`. Mamy więc zapisać warunek tak:

```
if (x>=0 && x<=5 || x>=10 && x<=20)
    y++;
```

Należy szczególną uwagę zwrócić na priorytety operatorów. Ponieważ użyte powyżej operatory relacji mają wyższy priorytet od operatorów łączenia warunków `&&` i `||`, w pierwszej kolejności sprawdzone zostaną poszczególne relacje, a następnie zostaną wykonane operacje iloczynów i sum logicznych na wynikach tych relacji. Inaczej jest w przypadku operatora negacji, ponieważ ma on wyższy priorytet od pozostałych operatorów, zatem, aby z niego skorzystać należy użyć nawiasów:

```
if (!(x>=0 && x<=100))
```

Nawiasy pozwalają również na łatwiejszą orientację i zrozumienie bardziej złożonych warunków, zatem warto o nich pamiętać:

```
if (((x>=0) && (x<=5)) || ((x>=10) && (x<=20)))
```

### Kaskadowe łączenie warunków

Instrukcje warunkowe mogą być łączone kaskadowo. Łączenie kaskadowe polega na dopisywaniu po klauzuli *else* kolejnej instrukcji warunkowej. W ten sposób można stworzyć rozgałęzienie, w którym będą sprawdzane kolejne warunki do momentu, aż jeden z nich zostanie spełniony, (w rezultacie czego zostanie wykonana instrukcja przypisana do tego warunku). Jeżeli rozgałęzienie posiada na końcu zamykającą klauzulę *else* (bez kolejnej instrukcji *if*), instrukcja umieszczona za tą klauzulą wykona się jedynie, gdy żaden z warunków w całym rozgałęzieniu nie będzie prawdziwy:

```
int x = 4, y = 100;
if (x == 1)
    y += 10;
else if (x == 2)
    y += 20;
else if (x == 3)
    y += 30;
else
    y += 40;
```

W powyższym przykładzie zmienna *y* uzyska wartość 140, ponieważ w wyniku nie spełnienia żadnego z warunków (*x* == 1, *x* == 2, *x* == 3) sterowanie zostanie przekazane do instrukcji umieszczonej za klauzulą *else*, czyli: *y* += 40. W tym przypadku wszystkie warunki zostały sprawdzone po kolei.

Jeżeli zmienna *x* miałaby przykładowo wartość 2, sprawdzone zostałyby jedynie dwa warunki: *x* == 1 oraz *x* == 2, a ponieważ jedynie *x* == 2 zostałby spełniony, wykonana zostałaby instrukcja: *y* += 20 (dając wartość *y* = 120). Reszta warunków nie zostałaby sprawdzona.

W przypadku, gdy instrukcje w rozgałęzieniu powtarzają się nie ma sensu tworzenia oddzielnych gałęzi:

```
if (x == 1)
    y++;
else if (x == 2)
    y++;
```



```
else if (x == 3)
    y += 10;
else if (x == 4)
    y += 10;
else
    y++;
```

Lepiej połączyć warunki operatorami logicznymi i skrócić zapis:

```
if (x != 3 && x != 4) // można też tak: if (x < 3 && x > 4)
    y++;
else
    y += 10;
```

### Zagnieżdżanie warunków

Instrukcje warunkowe można również zagnieżdżać:

```
if (ocena >= 1 && ocena <= 6)
{
    if (ocena > 1)
        Console.WriteLine("Zdałeś!");
    else
        Console.WriteLine("Nie zdałeś!");
}
else
    Console.WriteLine("Nie ma takiej oceny!");
```

W powyższym przykładzie w pierwszej kolejności sprawdzamy, czy ocena spełnia zadane kryteria (czy należy do zbioru liczb z przedziału od 1 do 6). Jeżeli ocena spełnia kryteria, przechodzimy do zagnieżdżonej instrukcji warunkowej w celu sprawdzenia czy ocena kwalifikuje nas do zdania egzaminu czy nie.

W przypadku instrukcji zagnieżdżonych, klauzula *else* znajdująca się na samym końcu dotyczy ostatniej instrukcji *if*. Przykładowo, jeżeli chcemy sprawdzić jedynie dwa przypadki: czy wartość zmiennych x i y są zerami oraz czy x jest różny od zera, poniższy przykład tego nie zrobi:

```
if (x == 0)
    if (y == 0)
        Console.WriteLine("Obie zmienne mają wartość zero.");
else
    Console.WriteLine("x jest różny od zera"); // dotyczy innego warunku
```

W rzeczywistości sprawdziliśmy czy x i y mają wartość zero i czy y jest różny od zera (nie x, jak zamierzaliśmy).

Jeżeli chcemy poprawnie sprawdzić drugi warunek musimy to zrobić tak:

```
if (x == 0)
{
    if (y == 0)
```

```
    Console.WriteLine("Obie zmienne mają wartość zero.");  
}  
else  
    Console.WriteLine("x jest różny od zera"); // teraz dobrze
```

### Zastępowanie wyrażeniami warunkowymi

Na koniec warto wspomnieć, iż istnieją liczne przypadki, w których można dowolną instrukcję *if* zamienić na wyrażenie warunkowe, jednak jego wielokrotne zagnieżdżenie może sprawić, że program stanie się mało czytelny.

Weźmy jeden z naszych wcześniejszych przykładów:

```
if (x == 1)  
    y += 10;  
else if (x == 2)  
    y += 20;  
else if (x == 3)  
    y += 30;  
else  
    y += 40;
```

Równoważne wyrażenie warunkowe wygląda tak:

```
y += (x == 1) ? 10 : ((x == 2) ? 20 : ((x == 3) ? 30 : 40));
```

W odróżnieniu od przejrzystej wersji z instrukcją warunkową, zrozumienie tego zapisu wymaga dłuższej chwili zastanowienia.

Spróbujmy napisać odpowiednik dla:

```
if (x == 0)  
{  
    if (y == 0)  
        Console.WriteLine("Obie zmienne mają wartość zero.");  
}  
else  
    Console.WriteLine("x jest różny od zera");
```

Z tym może być kłopot, bowiem nasz „odpowiednik” nie zadziała identycznie:

```
Console.WriteLine("", (x == 0) ? ((y == 0) ? "Obie zmienne mają wartość zero."  
: "") : "x jest różny od zera");
```

Mimo, że dla przypadku  $x = 0$  i  $y = 0$  oraz przypadku  $x \neq 0$  otrzymamy poprawne komunikaty (identyczne zachowanie), to w pozostałych przypadkach na ekranie pojawi się linia przejścia do nowego wiersza (za sprawą *WriteLine*, który wypisze pusty ciąg i przejdzie do nowej linii). Nie o to nam chodziło, gdyż ciąg instrukcji *if*, dla którego pisaliśmy ten warunek, dla tych przypadków nie wyświetli niczego na ekranie. Jak zatem napisać to wyrażenie warunkowe, aby działało identycznie jak ciąg instrukcji warunkowych? I tu mamy problem, bowiem mamy wykonać instrukcję *WriteLine* tylko w określonych warunkach, co za tym idzie, potrzebujemy instrukcji warunkowej *if*. To właśnie są przypadki, w których zastąpienie instrukcji warunkowej, wyrażeniem warunkowym nie jest dobrym

pomysłem.

## Instrukcja switch

Instrukcja warunkowa **switch** stanowi alternatywę dla instrukcji warunkowej **if**. W odróżnieniu jednak od niej, występuje tu jedynie jedno wyrażenie sterujące oraz podzielone na sekcje grupy instrukcji, które są wykonywane jedynie, gdy wartość wyrażenia sterującego przyjmie wartość odpowiadającą stałej przypisanej do danej sekcji.

Składnia instrukcji switch wygląda następująco:

```
switch (wyrażenie_sterujące)
{
    case stała_sekcji_1:
        [lista_instrukcji_1]
        [break;]
    [...]
    [case stała_sekcji_N:
        [lista_instrukcji_N]
        [break;]]]
    [default:
        [lista_instrukcji_domyślnych]]
}
```

gdzie:

*wyrażenie\_sterujące*

wyrażenie sterujące (wymagane),

*stała\_sekcji\_1, ..., stała\_sekcji\_N*

stała wartość będąca jedną z możliwych wartości zwracanych przez wyrażenie sterujące (wymagane)

*lista\_instrukcji\_1, ..., lista\_instrukcji\_N, lista\_instrukcji\_domyślnych*

jedna lub wiele instrukcji przypisanych do danej sekcji (opcjonalne)

Przykład:

```
switch(x)
{
    case 0:
        x++;
        break;
    case 1:
        x += 2;
        break;
    default:
        x += 3;
        break;
}
```

Instrukcja **switch** wykonuje działania w następującej kolejności:

1. Wyliczenie wartości wyrażenia sterującego.
2. Porównanie wartości wyrażenia ze stałymi poszczególnych sekcji.
3. Jeżeli którakolwiek ze stałych sekcji jest równa wartości wyrażenia, sterowanie przenoszone jest do tej sekcji i wykonywana jest lista instrukcji z tej sekcji, aż do momentu wystąpienia słowa kluczowego **break** (przerwij) lub końca instrukcji **switch**.
4. Jeżeli żadna ze stałych sekcji nie jest równa wartości wyrażenia, sterowanie przenoszone jest do sekcji domyślnej (**default**). Jeżeli ta sekcja nie została wcześniej utworzona, sterowanie przenoszone jest do końca instrukcji **switch**.

### Zasady stosowania

Istnieją zasady, których należy przestrzegać w czasie stosowania instrukcji **switch**:

- dozwolone są jedynie wyrażenia sterujące, które dają w wyniku wartość następujących typów: całkowitego (**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**), znakowego (**char**), wyliczeniowego (**enum**) lub łańcuchowego (**string**).

Przykład użycia dla typu **string**:

```
string strNazwa = "dom";

switch (strNazwa)
{
    case "dom":
        Console.WriteLine("Chodziło o dom.");
        break;
    case "rower":
        Console.WriteLine("Chodziło o rower.");
        break;
    default:
        Console.WriteLine("Chodziło o coś innego.");
        break;
}
```

Wynik:

Chodziło o dom.

Przykład użycia dla typu znakowego:

```
char cZnak = 'x';

switch (cZnak)
{
    case 'X':
        Console.WriteLine("To jest duże X.");
        break;
    case 'x':
        Console.WriteLine("To jest małe x.");
        break;
    default:

```

```
    Console.WriteLine("To jest inna litera.");  
    break;  
}
```

Wynik:

To jest małe x.

- w wyrażeniu warunkowym można stosować inne typy pod warunkiem, że istnieje możliwość konwersji tych typów do dozwolonych dla tego wyrażenia.

Przykładowo typ *double* nie jest dozwolony, ale jeżeli dokonamy konwersji:

```
double dWartosc = 9.0d;  
  
switch ((int)dWartosc)  
{  
    case 9:  
        Console.WriteLine("To jest 9.");  
        break;  
    default:  
        Console.WriteLine("To nie jest 9.");  
        break;  
}
```

Wynik:

To jest 9.

W powyższym przypadku nie było problemu, gdyż przy konwersji nie została utracona żadna informacja.

Co by się jednak stało, gdyby przy konwersji część informacji zostało utraconej:

```
double dWartosc = 9.99999999d;  
  
/*  
w trakcie konwersji dojdzie do utraty informacji, zostanie utracona  
część ułamkowa i wartość całkowita wyniesie 9 (nie 10)  
*/  
  
switch ((int)dWartosc)  
{  
    case 9:  
        Console.WriteLine("To jest 9.");  
        break;  
    default:  
        Console.WriteLine("To nie jest 9.");  
        break;  
}
```

Wynik:

To jest 9.

- wartość literalna *null* jest dozwolona jako wartość stałej dla sekcji.

Przykład użycia dla typu *string*:

```
string strNazwa = null;

switch (strNazwa)
{
    case null:
        Console.WriteLine("Jest null");
        break;
    default:
        Console.WriteLine("Nie null");
        break;
}
```

Wynik:

Jest null.

- wartości stałe dla sekcji muszą być niepowtarzalne (nie może być dwóch identycznych wartości).

Przykład:

```
int iWartosc = 2;

switch (iWartosc)
{
    case 1:
        Console.WriteLine("1");
        break;
    case 2:
        Console.WriteLine("2");
        break;
    case 2: // BŁĄD: stała sekcji 2 wystąpiła wcześniej
        Console.WriteLine("2");
        break;
}
```

- koniec sekcji musi zawierać słowo kluczowe **break** (wyjątek stanowi przypadek, gdy kilka sekcji wskazuje na tę samą listę instrukcji).

Przykład:

```
int iWartosc = 1;
bool bCos = true;

switch (iWartosc)
{
    case 1:
        Console.WriteLine("1");

        if (bCos)
        {
            Console.WriteLine("prawda");
        }
    // ...
}
```

```
        break;
    }
    else
        Console.WriteLine("fałsz");
    // BŁĄD: brakuje break, nie można przejść do innej sekcji
case 2:
    Console.WriteLine("2");
    break;
default:
    // BŁĄD: brakuje break, nie można przejść do innej sekcji
}
```

- stałe sekcji mogą wskazywać na jedną wspólną listę instrukcji.

Przykład:

```
int iWartosc = 2;

switch (iWartosc)
{
    case 1:
    case 2:
        Console.WriteLine("1 lub 2");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("3, 4 lub 5");
        break;
    default:
        Console.WriteLine("Inne");
        break;
}
```

Wynik:

1 lub 2.

### Zastępowanie instrukcji warunkowych

Instrukcja **switch** stanowi elegancką alternatywę dla niektórych konstrukcji złożonych warunków instrukcji **if** (kaskadowych lub zagnieżdżonych). W niektórych przypadkach lepiej zastosować instrukcję **switch** zamiast instrukcji **if**, (ale są również takie, dla których taka alternatywa nie ma sensu).

Przyjrzyjmy się konstrukcji z użyciem instrukcji **if**:

```
string strNazwa = "dom";

if (strNazwa == "dom")
{
    // ...
}
```

```
    strNazwa += "eczek";  
    Console.WriteLine(strNazwa); // strNazwa = "domeczek";  
}  
else if (strNazwa == "gramofon")  
{  
    strNazwa += "ik";  
    Console.WriteLine(strNazwa); // strNazwa = "gramofonik";  
}  
else if (strNazwa == "kot")  
{  
    strNazwa += "ek";  
    Console.WriteLine(strNazwa); // strNazwa = "kotek";  
}  
else  
    Console.WriteLine("coś innego");
```

Wynik:

domeczek

Alternatywna dla niej konstrukcja z użyciem instrukcji **switch** jest bardziej przejrzysta:

```
string strNazwa = "dom";  
  
switch (strNazwa)  
{  
    case "dom":  
        strNazwa += "eczek";  
        Console.WriteLine(strNazwa); // strNazwa = "domeczek";  
        break;  
    case "gramofon":  
        strNazwa += "ik";  
        Console.WriteLine(strNazwa); // strNazwa = "gramofonik";  
        break;  
    case "kot":  
        strNazwa += "ek";  
        Console.WriteLine(strNazwa); // strNazwa = "kotek";  
        break;  
    default:  
        Console.WriteLine("coś innego");  
        break;  
}
```

Wynik:

domeczek

Z kolei dla poniższego przypadku, próba stworzenia alternatywnej postaci instrukcji **switch** nie ma zbytniego sensu:



```
double dWartosc = 1.201d;

if (dWartosc < 1.20d)
    Console.WriteLine("poniżej 1.20");
else
{
    if (dWartosc <= 2.29d)
        Console.WriteLine("między 1.20 a 2.29");
    else
        Console.WriteLine("powyżej 2.29");
}
```

Wynik:

między 1.20 a 2.29

Jak wiadomo nie ma rzeczy niemożliwych, ale jaki sens jest w wymyślaniu niezbyt czytelnych zapisów tego typu:

```
double dWartosc = 1.201d;

switch ((dWartosc < 1.20d) ? 1 : ((dWartosc <= 2.29d) ? 2 : 3))
{
    case 1:
        Console.WriteLine("poniżej 1.20");
        break;
    case 2:
        Console.WriteLine("między 1.20 a 2.29");
        break;
    default:
        Console.WriteLine("powyżej 2.29");
        break;
}
```

Wynik:

między 1.20 a 2.29

## Instrukcje iteracyjne

Zawsze, gdy zachodzi potrzeba ponownego wykonania jakiejś instrukcji lub grupy instrukcji, można skorzystać z instrukcji iteracyjnych (tzw.: pętle).

Przykładowo, gdy chcemy dla każdej liczby z jakiegoś przedziału wyliczyć wartość na podstawie podanej funkcji, nie ma potrzeby pisania dla każdej z liczb osobnej instrukcji, skoro można wykorzystać do tego celu jedną z dostępnych pętli.

W języku C# mamy do dyspozycji cztery instrukcje iteracyjne:

- instrukcja ***while***

- instrukcja *do*
- instrukcja *for*
- instrukcja *foreach*

## Instrukcja *while*

Pętla *while* („w czasie gdy”) powtarza wykonywanie instrukcji lub grupy instrukcji do momentu, aż testowany warunek nie zwróci wartości *false* (np.: zmienna nie osiągnie pewnej wartości). Składnia instrukcji *while* wygląda następująco:

```
while (wyrażenie_sterujące)
    instrukcja
```

gdzie:

*wyrażenie\_sterujące*

wyrażenie, które daje w rezultacie wartość typu *bool*, używane do testowania warunków zakończenia pętli (wymagane),

*instrukcja*

instrukcja (zakończona średnikiem) lub grupa instrukcji w bloku, powtarzana w pętli (wymagane).

Przykład:

```
while (i < 10)
{
    // instrukcje
}
```

Instrukcja *while* wykonuje działania w następującej kolejności:

1. Wyliczana jest wartość wyrażenia sterującego.
2. Jeżeli wyrażenie sterujące przyjmie wartość *true* wykonywana jest instrukcja (lub grupa instrukcji w bloku).
3. Jeżeli wyrażenie sterujące przyjmie wartość *false*, sterowanie przekazywane jest na zewnątrz pętli (ze względu na to, że wartość wyrażenia sterującego sprawdzana jest przed wejściem do pętli, w przypadku, gdy wynikiem tego wyrażenia jest *false*, nie nastąpi wejście do pętli).

## Warunki zakończenia

Wpisując wartość literalną *true* jako wyrażenie sterujące możemy uzyskać „*pętlę nieskończoną*”:

```
while (true)
{
}
```

Jeżeli chcemy wykorzystać pętlę *while* do wykonania zadanej liczby powtórzeń, musimy zadeklarować sobie zmienną pełniącą rolę licznika i zainicjować ją wartością początkową. Następnie musimy napisać wyrażenie sterujące, które będzie sprawdzać czy określona wartość została osiągnięta. Najważniejszą rzeczą jest pamiętanie o zwiększaniu licznika po wykonaniu określonych instrukcji, (jeżeli tego nie zrobimy, licznik nigdy nie osiągnie zadanej wartości i pętla nigdy nie zakończy działania):

```
int iLicznik = 0;

while (iLicznik < 10)
```

```
{  
    Console.WriteLine("", iLicznik);  
    iLicznik++;  
}
```

Możemy również użyć skróconego zapisu (licznik jednak w takim wypadku będzie miał wartość o jeden większą zaraz po wejściu do bloku instrukcji w pętli):

```
int iLicznik = 0;  
  
while (iLicznik++ < 10)  
    Console.WriteLine("", (iLicznik - 1));
```

Możemy posługiwać się różnymi zmiennymi i określać na podstawie ich wartości warunki zakończenia wykonywania pętli. Przykładowo, jeżeli chcemy, aby pętla 10 razy poprosiła użytkownika o wprowadzenie z klawiatury jakiegoś wyrazu i w przypadku, gdy będzie to „koniec” zakończyła działanie (bez względu na aktualną wartość licznika) możemy to zrobić tak:

```
int iLicznik = 0;  
string strNapis;  
  
while (iLicznik++ < 10)  
{  
    Console.Write("Podaj napis:");  
    strNapis = Console.ReadLine();  
  
    if (strNapis == "koniec")  
    {  
        Console.WriteLine("To już koniec.");  
        break;  
    }  
  
    Console.WriteLine("To jeszcze nie koniec.");  
}
```

### Zagnieżdżanie pętli

Pętlę **while** można umieścić wewnątrz innej pętli (zagnieżdżyć):

```
while (i < 10)
```

Przy pisaniu pętli zagnieżdżonych należy zwrócić uwagę na to, która pętla dla której jest nadrzędną a która podrzędną:

```
while (i < 10)  
    while (j < 10)  
        j++;  
    while (k > 20)  
        k--;
```

Wcięcia wprowadzają w błąd, gdyż tak naprawdę pętla operująca na zmiennej **k** nie jest zagnieżdżona w

żadnej z powyższych pętli. Tylko pętla operująca na zmiennej *j* jest zagnieżdżona w pętli operującej na zmiennej *i*. Jeżeli naszym zamiarem było umieszczenie pętli operującej na zmiennej *k* w pętli operującej na zmiennej *j* należało to zrobić tak:

```
while (i < 10)
    while (j < 10)
    {
        j++;

        while (k > 20)
            k--;
    }
```

## Instrukcja **do**

Instrukcja **do** sama w sobie nie jest pętlą. Dopiero po połączeniu z instrukcją **while** tworzy pętlę **do...while** („powtarzaj...gdy”).

Pętla **do...while** powtarza wykonywanie instrukcji lub grupy instrukcji do momentu, aż testowany warunek nie zwróci wartości **false** (np.: zmienna nie osiągnie pewnej wartości). Różni się ona od pętli **while** tym, że instrukcje w pętli **do...while** zostaną wykonane co najmniej raz, bez względu na wartość wyrażenia sterującego (w przypadku pętli **while** przy warunku o wartości **false** nie nastąpi wejście do pętli).

Składnia instrukcji **do...while** wygląda następująco:

```
do
    instrukcja
while (wyrażenie_sterujące);
```

gdzie:

*wyrażenie\_sterujące*

wyrażenie, które daje w rezultacie wartość typu **bool**, używane do testowania warunków zakończenia pętli (wymagane),

*instrukcja*

instrukcja lub grupa instrukcji w bloku, powtarzana w pętli (wymagane).

Przykład:

```
do
{
    // instrukcje
}
while (i < 10);
```

Instrukcja **do...while** wykonuje działania w następującej kolejności:

1. Następuje wejście do pętli i wykonanie instrukcji.
2. Następuje wyliczenie wartości wyrażenia sterującego.
3. Jeżeli wyrażenie sterujące przyjmie wartość **true**, instrukcje są ponownie wykonywane.
4. Jeżeli wyrażenie sterujące przyjmie wartość **false**, sterowanie przekazywane jest na zewnątrz

pętli.

### Warunki zakończenia

Wpisując wartość literalną **true** jako wyrażenie sterujące możemy uzyskać „*pętlę nieskończoną*”:

```
do
{
}
while (true);
```

Pętla **do...while** zawsze wykonuje w pierwszej kolejności instrukcje a dopiero po tym wylicza wartość wyrażenia, zatem należy z niej korzystać, jeżeli zależy nam na tym, aby instrukcje wykonały się co najmniej raz.

Przykładowo, jeżeli chcemy wczytać jakąś wartość w pętli i użyć jej wartości w wyrażeniu sterującym, wykonanie tego przy pomocy pętli **while** może być kłopotliwe:

```
string strNapis;

Console.Write("Podaj wyraz : ");
strNapis = Console.ReadLine();

while (strNapis != "koniec")
{
    Console.Write("Podaj wyraz : ");
    strNapis = Console.ReadLine();
}
```

Jak widać niepotrzebnie powtórzyliśmy grupę instrukcji (najpierw wczytaliśmy wartość zmiennej, następnie użyliśmy jej w wyrażeniu warunkowym, by w zależności od wyniku powtórzyć te same instrukcje lub nie). W takim wypadku lepiej posłużyć się pętlą **do...while** i napisać to samo bardziej elegancko:

```
string strNapis;

do
{
    Console.Write("Podaj wyraz : ");
    strNapis = Console.ReadLine();
}
while (strNapis != "koniec");
```

### Zagnieżdżanie pętli

Tak samo jak ma to miejsce w przypadku pętli **while**, pętlę **do...while** można zagnieżdżać, w odróżnieniu jednak od pętli **while** nie ma możliwości pomylenia się co do przynależności danej pętli (ponieważ instrukcje zawsze wpisywane są między **do** i **while**):

```
do // zewnętrzna (nadrzędna)
{
```

```
do // wewnętrzna (zagnieżdżona)
{
}
while(j > 10)
}
while(i < 10);

do // inna niezagnieżdżona
{
}
while(k != 10)
```

## Instrukcja for

Pętla **for** („dla”) powtarza wykonywanie instrukcji lub grupy instrukcji do momentu, aż testowany warunek nie zwróci wartości **false** (np.: zmienna licznika nie osiągnie oczekiwanej wartości).

Składnia instrukcji **for** wygląda następująco:

```
for ([inicjatory]; [wyrażenie_sterujące]; [iteratory])
    instrukcja
```

gdzie:

*inicjatory*

oddzielona przecinkami lista instrukcji lub wyrażeń inicjujących liczniki pętli (opcjonalne),

*wyrażenie\_sterujące*

wyrażenie, które daje w rezultacie wartość typu **bool**, używane do testowania warunków zakończenia pętli (opcjonalne, ale w przypadku braku wyrażenia domyślnie przyjmowana jest wartość **true**),

*iteratory*

lista wyrażeń zwiększających lub zmniejszających wartości liczników pętli (opcjonalne),

*instrukcja*

instrukcja lub grupa instrukcji w bloku, powtarzana w pętli (wymagane).

Przykład:

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++)
{
    // instrukcje
}
```

Instrukcja **for** wykonuje działania w następującej kolejności:

1. Wykonywane są instrukcje lub wyrażenia związane z inicjacją liczników pętli (*inicjatory*).
2. Wyliczana jest wartość wyrażenia sterującego.
3. Jeżeli wyrażenie sterujące przyjmie wartość **true**, wykonywana jest instrukcja lub grupa instrukcji w bloku, po czym wykonywane są operacje zwiększające lub zmniejszające wartości liczników.
4. Jeżeli wyrażenie sterujące przyjmie wartość **false**, sterowanie przekazywane jest na zewnątrz pętli (ze względu na to, że wartość wyrażenia sterującego sprawdzana jest przed wejściem do pętli, w przypadku, gdy wynikiem tego wyrażenia jest **false**, nie nastąpi wejście do pętli).

### Warunki zakończenia

Ze względu na opcjonalność składników wewnątrz nawiasów, w skrajnym przypadku pętla **for** może przyjąć taką formę (tzw. „*pętli nieskończonej*”):

```
for ( ; ; )  
{  
    // instrukcje  
}
```

Ponieważ **brak** wyrażenia sterującego powoduje, że domyślnie przyjmowana jest wartość **true**, pętla ta jest równoważna w zapisie pętli:

```
for ( ; true; )  
{  
    // instrukcje  
}
```

### Deklaracja liczników

Wewnątrz instrukcji **for** można deklarować zmienne lokalne widoczne wyłącznie w obrębie pętli:

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++)  
{  
      
    iLicznik++; //BŁĄD: zmienna nie jest już widoczna
```

Można też deklarować kilka zmiennych (każdą oddzielając znakiem przecinka):

```
for (int iLicznik = 0, iZmienna = 2; iLicznik < 10; iLicznik++)
```

W przypadku deklaracji kilku zmiennych należy pamiętać o tym, aby wszystkie należały do tego samego typu całkowitego:

```
for (int iLicznik=0, long lZmienna = 0L; iLicznik < 10; iLicznik++) //BŁĄD
```

Zadeklarowane wewnątrz pętli zmienne mogą być użyte zarówno w wyrażeniu sterującym jak i iterującym:

```
for (int i = 0, j = 0; i < 10 || j < 20; i++, j += 2)
```

Deklarowane zmienne nie mogą jednak pokrywać się z wcześniej zadeklarowanymi:

```
int iLicznik;
```

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++) // BŁĄD  
{  
}
```

Powyższy błąd można jednak łatwo naprawić:

```
int iLicznik;
```

```
for (iLicznik = 0; iLicznik < 10; iLicznik++)  
{  
      
}
```

## Zagnieżdżanie pętli

Pętlę **for** można umieścić wewnątrz innej pętli (zagnieżdżać):

```
for(int y = 0; y < 100; y++)
    for (int x = 0; x < 100; x++)
        Console.WriteLine("y = , x = ", y, x);
```

Powyższa pętla spowoduje pojawienie się 10 000 komunikatów z cyklu „x = wartość, y = wartość”, bowiem pętla wewnętrzna (z licznikiem x), która ma za zadanie wypisać 100 komunikatów, zostanie wywołana 100 razy przez pętlę zewnętrzną (z licznikiem y). Skoro więc  $100 * 100 = 10\,000$ , więc tyle razy zostanie wykonana instrukcja **WriteLine**.

Przy pisaniu pętli zagnieżdżonych, należy zwrócić uwagę na to, która pętla dla której jest nadrzędną a która podrzędną:

```
for(int y = 0; y < 100; y++)
    for (int x = 0; x < 100; x++)
        Console.WriteLine("y = , x = ", y, x);

for (int v = 0; v < 100; v++)
    Console.WriteLine("v = ", v);
```

W powyższym przykładzie mamy do czynienia tylko z jedną zagnieżdżoną pętlą (z licznikiem x, dla której nadrzędną jest pętla z licznikiem y). Jeżeli naszym zamiarem było umieszczenie pętli z licznikiem v wewnątrz pętli z licznikiem x, należało umieścić obie pętle (z licznikiem x i v) w jednym bloku instrukcji:

```
for(int y = 0; y < 100; y++)
{
    for (int x = 0; x < 100; x++)
        Console.WriteLine("y = , x = ", y, x);

    for (int v = 0; v < 100; v++)
        Console.WriteLine("v = ", v);
}
```

## Zastępowanie innymi pętlami

Każdą pętlę **for** można zamienić na równoważną pętlę **while**. Należy jednak przy tym pamiętać o licznikach oraz o tym, że wyrażenie sterujące w pętli **while** jest zawsze wymagane.

Przykład pętli for:

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++)
{
    // instrukcje
}
```

Równoważny mu zapis przy pomocy pętli **while** wygląda tak:

```
int iLicznik = 0;
```



```
while (iLicznik < 10)
{
    // instrukcje
    iLicznik++;
}
```

Z kolei dla „*pętli nieskończonej*”:

```
for ( ; ; )
{
    // instrukcje
}
```

Równoważny mu zapis przy pomocy pętli **while** wygląda tak:

```
while (true)
{
    // instrukcje
}
```

## Instrukcja foreach

Pętla **foreach** („dla każdego”) powtarza wykonywanie instrukcji lub grupy instrukcji dla każdego elementu w określonym zbiorze (np.: dla każdego znaku w łańcuchu znaków).

Składnia instrukcji **foreach** wygląda następująco:

```
foreach (typ identyfikator in wyrażenie)
    instrukcja
```

gdzie:

*typ*

typ danych elementu określonego zbioru (wymagane),

*identyfikator*

nazwa zmiennej dla elementu zbioru (wymagane),

*wyrażenie*

obiekt kolekcji lub wyrażenie tablicowe, musi istnieć możliwość konwersji typu elementu kolekcji

lub tablicy na typ danych identyfikatora (wymagane),

*instrukcja*

instrukcja lub grupa instrukcji w bloku, powtarzana w pętli (wymagane).

Przykład:

```
string strNapis = "Ala ma kota";

foreach (char cZnak in strNapis)
    Console.WriteLine("", cZnak);
```

Powyższy przykład dla każdego znaku w łańcuchu **strNapis**, wypisze na ekranie pojedynczą linię zawierającą ten znak.

Pętlę **foreach** stosuje się przede wszystkim do operowania na elementach kolekcji lub tablicy. Przy

używaniu pętli **foreach** należy jednak pamiętać, że zmienna elementu, na której operujemy nie może być modyfikowana (jest dostępna jedynie w trybie do odczytu).

## Instrukcje skoku

Instrukcje skoku używane są do przenoszenia sterowania z jednego miejsca programu do innego, w dowolnym momencie jego działania.

Jeżeli instrukcja skoku umieszczona jest w bloku programu, a miejsce, do którego ma nastąpić skok znajduje się poza tym blokiem, to blok ten zostanie opuszczony.

Do podstawowych instrukcji skoku należą:

- instrukcja **goto**
- instrukcja **break**
- instrukcja **continue**

### Instrukcja goto

Przenosi sterowanie programu do punktu wskazanego za pomocą etykiety. Każda etykieta składa się z identyfikatora oraz znaku dwukropka. Etykietami domyślnie są również stałe sekcji instrukcji **switch** (zarówno stałe zdefiniowane przez użytkownika jak i sekcja domyślna **default**).

Składnia skoku może zatem przyjąć jedną z możliwych form wywołania:

```
goto identyfikator  
goto case stała_sekcji  
goto default
```

gdzie:

*identyfikator*

nazwa etykiety (wymagane),

*stała\_sekcji*

stała wartość będąca jedną z możliwych wartości zwracanych przez wyrażenie sterujące w instrukcji **switch** (wymagane),

*default*

domyślna sekcja instrukcji **switch** oznaczająca pozostałe przypadki (wymagane).

Przykład:

```
if (bWyswietl == true)  
    goto Pokaz;  
  
goto Koniec;  
  
Pokaz:  
    Console.WriteLine("Pokaz");  
  
Koniec:  
;
```

Zazwyczaj nie powinno się używać tej instrukcji, gdyż powoduje ona, że program staje się nieczytelny. Jedyna sytuacja, w której używanie **goto** może być rozsądne, to przypadek, kiedy przekazuje się kontrolę wewnątrz instrukcji **switch** z jednej sekcji do innej:

```
int iNumer = 2;

switch (iNumer)
{
    case 0:
        iNumer += 10;
        break;
    case 1:
        iNumer++;
        break;
    case 2:
        iNumer++;
        goto case 0;
        break;
}
```

## Instrukcja break

Instrukcja **break** powoduje wyjście z bloku dowolnej pętli lub z dowolnej sekcji w bloku **switch**, w której występuje.

Przykład:

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++)
{
    if (iLicznik == 2)
        break;

    Console.WriteLine("Liczba ", iLicznik);
}
```

Wynik:

```
Liczba 0
Liczba 1
```

W powyższym przykładzie instrukcja wyświetlająca wartość licznika zostanie powtórzona tylko 2 razy. Przy trzecim wejściu do pętli wyrażenie warunkowe instrukcji **if** da wartość **true** (`iLicznik == 2`), co spowoduje wykonanie instrukcji skoku **break** (pętla zostanie opuszczona).

## Instrukcja continue

Instrukcja **continue** powoduje pominięcie dalszych instrukcji zawartych w pętli (od momentu wystąpienia do końca bloku) dla aktualnej wartości licznika.

Przykład:

```
for (int iLicznik = 0; iLicznik < 10; iLicznik++)  
{  
    if (iLicznik == 2)  
        continue;  
  
    Console.WriteLine("Liczba ", iLicznik);  
}
```

Wynik:

```
Liczba 0  
Liczba 1  
Liczba 3  
Liczba 4  
Liczba 5  
Liczba 6  
Liczba 7  
Liczba 8  
Liczba 9
```

W powyższym przykładzie instrukcja wyświetlająca wartość licznika zostanie powtórzona 9 razy. Dla wartości licznika równej 2 dalsze instrukcje w pętli zostaną pominięte.

## Rozdział 7.

# Klasy i obiekty

### Podstawowe pojęcia

Zanim przejdziemy do programowania obiektowego w języku C#, musimy poznać kilka podstawowych pojęć, które będą wykorzystywane w dalszej części książki.

#### Klasa i obiekt

Każdy obiektowy język programowania daje programiście możliwość tworzenia nowych typów danych. Nowy typ danych definiuje się poprzez zdefiniowanie klasy.

Czym zatem jest klasa? Z filozoficznego punktu widzenia klasa to rodzaj klasyfikacji, czyli próby zdefiniowania cech wspólnych określonego obiektu (opis czegoś, co istnieje). Z programistycznego punktu widzenia klasa stanowi typ danych, który odwzorowuje wspólne cechy jakiegoś obiektu.

Czym zatem jest obiekt? Obiekt jest instancją klasy, czyli rzeczywistym tworem o określonych cechach i zachowaniu (coś, co istnieje). Przykładowo weźmy pod uwagę kota. Kot jest obiektem (jest czymś, co istnieje), natomiast jego opis znajdujący się np.: w encyklopedii jest klasą, (czyli opisem istniejącego obiektu).

Każdy obiekt charakteryzuje jego:

- niepowtarzalność – obiekty różnią się od siebie, przez co są niepowtarzalne (np. dwa koty należące do tej samej rasy nie są identyczne);
- zachowanie – każdy obiekt może wykonywać zestaw określonych czynności (np. kot: miauczy, łasi się, etc.);
- stan – każdy obiekt przechowuje informacje o swoim stanie, informacja ta wpływa na jego zachowanie (np.: jeżeli kot jest głodny jego zachowanie się zmieni: będzie szukał jedzenia, będzie się upominał, etc., jeżeli zaspokoi głód, jego zachowanie znów się zmieni: będzie leżał, będzie mruczeć, etc.)

#### Relacje

Między obiektami zachodzą wszelkiego rodzaju relacje (powiązania). Na przykład obiekt Pracownik jest powiązany z obiektem Firma poprzez relację zatrudnienia. Pracownik jest zatrudniony w firmie. Firma zatrudnia pracownika. Inny przykład to obiekt Człowiek i obiekt Pies. Obiekty mogą być ze sobą powiązane wieloma różnymi relacjami: człowiek karmi psa, pies łasi się do człowieka, itp.

#### Hermetyzacja

Hermetyzacja (ang. encapsulation) to ukrywanie informacji w celu kontrolowania użycia obiektu oraz minimalizacji efektów jego modyfikacji. Umożliwia ona rozróżnienie pomiędzy interfejsem dostępu do obiektu (informacją co robi i co zawiera), a zawartością definiującą jak jest zbudowany i jak działa. Przykładowo, telewizor jest dla oglądającego przykładem czarnej skrzynki, która posiada interfejs

w postaci pilota, obudowy i ekranu oraz ukrytą wewnątrz elektronikę. Oglądający kontroluje zachowanie telewizora za pomocą interfejsu i nie interesuje się tym, co się dzieje wewnątrz (nie analizuje jak to działa i jakie elementy są w środku).

## Abstrakcja

Abstrakcja jest zwana również selektywną ignorancją i polega na eliminowaniu, ukrywaniu lub pomijaniu mniej istotnych informacji, a skupieniu się na wyodrębnieniu informacji wspólnych i niezmiennych dla zbioru pewnych obiektów. Abstrakcja wprowadza pewne pojęcia i symbole służące do opisu cech wspólnych grupy obiektów. Przykładowo słowo „przedmiot” jest pojęciem abstrakcyjnym, który wyodrębnia wspólne cechy grupy obiektów: element nieożywiony, coś czym można się posługiwać, coś co ma kształt, itd. nadając mu nazwę („przedmiot”). Szczegóły danego przedmiotu są pomijane. Weźmy piłkę, która jest przedmiotem. Piłka może mieć określony kolor, zastosowanie i łatwo ją odróżnić od innego przedmiotu np. czajnika. Ktoś powie, że piłka też jest pojęciem abstrakcyjnym, bo grupuje cechy wspólne i pomija szczegóły. Racja. Cechy wspólne piłki to okrągły kształt, możliwość używania w grach i zabawach, ale są przecież różne piłki (pingpongowe, tenisowe, plażowe, nożne, itd.), które mają specyficzne cechy. Gdzie zatem kończy się abstrakcja, skoro każde pojęcie abstrakcyjne tworzymy poprzez dobieranie pewnego zestawu innych pojęć abstrakcyjnych (stworzonych i nazwanych wcześniej)? Odpowiedź brzmi: „Nigdzie” (nawet sama odpowiedź jest abstrakcyjnym pojęciem).

## Kompozycja i dekompozycja

Każdy obiekt można rozłożyć na obiekty składowe (zdekomponować), a z obiektów składowych można zbudować obiekt złożony (skomponować). Przykładowo, komputer jest obiektem złożonym i składa się z innych obiektów: procesora, płyty głównej, pamięci, napędów, kart rozszerzeń, urządzeń peryferyjnych, itd. Każdy element komputera również składa się z innych obiektów np. dysk, który składa się z: obudowy, sterownika, talerzy, głowicy, itd. Te elementy również da się zdekomponować.

## Składnik klasy

Składnik klasy to element należący do danej klasy. Składnikami klasy mogą być: pola, konstruktory, destruktory, metody, właściwości, indeksatory, zdarzenia, delegacje, operatory jak również zagnieżdżone: klasy, struktury, listy wyliczeniowe czy interfejsy.

## Składnica

Składnice (ang. assembly) to bloki, z których zbudowane są aplikacje .NET Framework. Składnica jest kolekcją typów i zasobów, które współdziałają ze sobą jako logiczna jednostka pewnej funkcjonalności (np.: kod stanowiący część tej samej biblioteki czy aplikacji).

## Definiowanie klas

Klasy definiuje się za pomocą słowa kluczowego **class**.

Składnia definicji klasy jest następująca:

```
[modyfikatory] class Identyfikator [:lista_bazowa]
{
    składniki_klasy
}
```

gdzie:

*modyfikatory*

dozwolone są modyfikatory: **new**, **abstract**, **sealed** oraz cztery modyfikatory dostępu (opcjonalne), na temat modyfikatorów będzie mowa w dalszej części,

*Identyfikator*

nazwa klasy (wymagane),

*lista\_bazowa*

lista zawierająca nazwę jednej klasy bazowej oraz nazwy interfejsów oddzielone przecinkami (opcjonalne), na temat dziedziczenia klas oraz interfejsów będzie mowa w dalszej części,

*składniki klasy*

deklaracja składowych klasy.

Przykład:

```
class Kot
{
    // składowe klasy
}
```

Wewnątrz klas można zagnieżdżać inne klasy:

```
class Samochod
{
    class Kolo
    {
        class Opona
        {
        }
    }
}
```

## Modyfikatory

Modyfikatory służą do zmiany zachowania typów pól składowych klasy. W języku C# można wyróżnić następujące modyfikatory (o większości z nich będzie mowa przy zagadnieniach, których dotyczą bezpośrednio):

- **abstract** – określa, że klasa jest abstrakcyjna (nie można utworzyć instancji takiej klasy), klasa abstrakcyjna może być bazową dla innej klasy;
- **const** – określa, że wartość składowej nie może być zmieniana;
- **event** – służy do deklarowania zdarzeń;
- **override** – służy nadpisywaniu metod wirtualnych odziedziczonych z klasy bazowej;
- **readonly** – określa, że polu można przypisywać wartości jedynie w deklaracji oraz

w konstruktorze tej samej klasy;

- **sealed** – określa, że klasa nie może być bazową klasą dla żadnej innej (stanowi klasę ostateczną);
- **static** – określa, że składnik klasy należy do typu a nie do konkretnego obiektu;
- **virtual** – służy do deklaracji metod wirtualnych, czyli metod, które mogą być nadpisywane w klasach dziedziczących z klasy, w której się znajdują;

- **volatile** – określa, że pole może zostać zmodyfikowane przez system operacyjny, urządzenie lub inny wątek,

oraz modyfikatory dostępu opisane poniżej.

## Modyfikatory dostępu

Modyfikatory dostępu to słowa kluczowe służące do określania dostępności składnika danej klasy. Zastosowanie określonego modyfikatora dostępu niesie pewne ograniczenia dla składnika klasy, którego on dotyczy:

Modyfikator dostępu	Ograniczenia dla składnika
<b>public</b>	Nie ma ograniczeń. Składnik posiadający ten modyfikator jest dostępny dla dowolnej klasy.
<b>protected</b>	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje oraz klas dziedziczących.
<b>private</b>	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje.
<b>internal</b>	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje oraz klas w obrębie składnicy.
<b>internal protected</b>	Składnik posiadający ten modyfikator jest dostępny jedynie w obrębie klasy, w której się znajduje, klas w obrębie składnicy oraz klas dziedziczących.

Przykład:

```
class Komputer
{
    private class Procesor
    {
    }

    public class Monitor
    {
    }
}
```

W powyższym przykładzie klasa **Procesor**, będzie dostępna jedynie wewnątrz klasy **Komputer** (zarówno nadrzędna klasa **Komputer** jak i wewnętrzna względem klasy **Komputer** klasa **Monitor**, będą mogły z niej korzystać). W przypadku klasy **Monitor** jest inaczej, bowiem modyfikator **public** zezwala na swobodne korzystanie z tej klasy dowolnym klasom.



## Tworzenie obiektu klasy

Jak już wspomniałem wcześniej, obiekt jest instancją klasy. Aby utworzyć instancję zdefiniowanej wcześniej klasy, należy zaalokować pamięć dla obiektu tej klasy (służy do tego operator *new*), a następnie zainicjować go (wskazując określony konstruktor klasy). Dostęp do tego obiektu będzie możliwy jednak jedynie wtedy, gdy utworzymy odpowiednie wskazanie referencyjne (utworzenie odnośnika).

Weźmy za przykład zdefiniowaną wcześniej klasę **Komputer**. Utworzenie obiektu tej klasy wygląda tak:

```
Komputer komp = new Komputer();
```

Jak widać, zadeklarowana została zmienna typu referencyjnego klasy **Komputer** (za pośrednictwem której, będziemy mogli operować na obiekcie tej klasy). Następnie zaalokowana została pamięć dla obiektu za pomocą operatora *new*. Obiekt został zainicjowany poprzez wywołanie konstruktora klasy. Na końcu utworzono wskazanie do obiektu, przez nadanie zmiennej referencyjnej wartości określającej jego położenie w pamięci.

## Pola

Każda klasa może zawierać pola, czyli zmienne dowolnego typu, które są dostępne w obrębie tej klasy. W zależności od modyfikatora znajdującego się przy deklaracji pola, może ono być dostępne lub nie dostępne dla innej klasy.

Składnia deklaracji pola klasy wygląda tak:

```
[modyfikatory] typ Identyfikator [= wartość_początkowa];
```

gdzie:

*modyfikatory*

dozwolone są modyfikatory: *static*, *readonly*, *new*, *volatile* oraz modyfikatory dostępu (opcjonalne),

*typ*

typ danych pola (wymagane),

*Identyfikator*

unikalna w obrębie klasy nazwa pola (wymagane),

*wartość\_początkowa*

początkowa wartość pola (opcjonalne).

Przy deklarowaniu pól klasy zazwyczaj określa się również sposób dostępu do tego pola stosując jeden z modyfikatorów dostępu. Jeżeli pole nie będzie miało określonego modyfikatora dostępu, zostanie przyjęty domyślnie modyfikator *private*.

Przykład:

```
class Osoba
{
    public string Imie;
    private short Wiek;
```

```
short Waga;  
}
```

W powyższym przykładzie w klasie **Osoba** zadeklarowano trzy pola: publiczne pole **Imie** oraz prywatne pola **Wiek** i **Waga**.

## Konstruktor

Konstruktor jest specjalną metodą klasy, która jest wywoływana zawsze jako pierwsza w czasie powoływania obiektu tej klasy. Każdy konstruktor musi mieć taką samą nazwę jak nazwa klasy, w której został zdefiniowany i nie może zwracać żadnych wartości (w przeciwieństwie do standardowych metod). Klasa może zawierać wiele konstruktorów muszą się jednak różnić między sobą ilością argumentów.

### Konstruktor domyślny

Jeżeli nie zdefiniujemy żadnego konstruktora, kompilator zrobi to za nas. Konstruktor utworzony przez kompilator nazywa się konstruktorem domyślnym.

Konstruktor domyślny posiada następujące właściwości:

- jest widoczny na zewnątrz klasy (ma modyfikator dostępu **public**);
- ma taką samą nazwę jak nazwa klasy;
- nie zwraca żadnych wartości;
- nie posiada żadnych argumentów;
- inicjuje wartości pól klasy zerem (w przypadku pól typu wartości za wyjątkiem **bool**), **false** (w przypadku typu **bool**) oraz **null** w przypadku pól typu referencyjnego.

Przykład:

```
class Komputer  
{  
}  
  
class Test  
{  
    static void Main()  
    {  
        Komputer komp = new Komputer();  
        // ...  
    }  
}
```

W powyższym przykładzie zdefiniowaliśmy klasę **Komputer** nie umieszczając w niej żadnych składników. Następnie w metodzie startowej **Main** znajdującej się w klasie **Test** utworzyliśmy obiekt tej klasy. Mimo, że w klasie nie zdefiniowaliśmy żadnego konstruktora, kompilator wygenerował konstruktor domyślny, który posłużył do zainicjowania utworzonego obiektu.

Z kolei, jeżeli zdefiniujemy sami jakikolwiek konstruktor (z argumentami lub bez) domyślny

konstruktor nie zostanie utworzony. Ze względu na to, że domyślnie każdy konstruktor klasy jest prywatny (wyjątek stanowi jedynie klasa abstrakcyjna) musimy pamiętać o umieszczeniu modyfikatora **public** w jego definicji, aby późniejsze utworzenie obiektu było w ogóle możliwe:

```
class Komputer
{
    Komputer()
    {
    }
}
class Test
{
    static void Main()
    {
        Komputer komp = new Komputer(); // BŁĄD: konstruktor niedostępny
        // ...
    }
}
```

## Inicjacja pól

W odróżnieniu od zmiennych lokalnych, które wymagają inicjacji, pola składowe klasy nie muszą być inicjowane, gdyż zajmie się tym konstruktor. Wszystkie pola, które nie posiadają przypisanej wartości, zostaną domyślnie zainicjowane wartościami 0, *false* lub *null*. Zatem konstruktor domyślny wygenerowany dla takiej klasy:

```
class Data
{
    private short Rok, Miesiac, Dzień;
}
```

Będzie równoważny konstruktorowi zdefiniowanemu w taki sposób:

```
class Data
{
    public Data()
    {
        Rok = 0;
        Miesiac = 0;
        Dzień = 0;
    }

    private short Rok, Miesiac, Dzień;
}
```

Programista może chcieć sam zainicjować pola składowe klasy jakimiś innymi wartościami początkowymi. W takim wypadku musi sam zdefiniować konstruktor i przypisać wartości poszczególnym polom składowym wewnątrz tego konstruktora:

```
class Data
{
    public Data()
    {
        Rok = 2004;
        Miesiac = 11;
        // pole Dzień niezainicjowane, więc domyślnie przyjmie wartość 0
    }

    private short Rok, Miesiac, Dzień;
}
```

W powyższym przykładzie zdefiniowaliśmy konstruktor bezargumentowy, który inicjuje pola Rok i Miesiac zadanymi wartościami, oraz pole Dzień domyślną wartością. Oznacza to, że każdy nowo utworzony obiekt będzie zainicjowany tymi wartościami.

Wartości można inicjować również w taki sam sposób jak zmienne lokalne, czyli przypisanie wartości w deklaracji pola:

```
class Data
{
    public Data()
    {
    }

    private short Rok = 2004, Miesiac = 11, Dzień = 12;
}
```

Aby móc wpływać na wartości pól klasy w czasie tworzenia obiektów, musimy zdefiniować konstruktor zawierający argumenty, które posłużą do zainicjowania pól klasy:

```
class Data
{
    public Data(short sRok, short sMiesiac, short sDzien)
    {
        Rok = sRok;
        Miesiac = sMiesiac;
        Dzień = sDzien;
    }

    private short Rok, Miesiac, Dzień;
}
```

Dzięki tym argumentom tworząc obiekt klasy, możemy zainicjować go zadanymi parametrami używając właściwego konstruktora.

```
Data data = new Data(2004, 11, 12), innadata = new Data(2004, 12, 12);
```

Należy przy tym uważać na kolejność argumentów, (jeżeli typ będzie zgodny zainicjujemy nie to pole

klasy, o które nam chodzi, jeżeli niezgodny nastąpi próba konwersji niejawnej, która może się nie powieść):

```
Data data = new Data(12, 11, 2004); // BŁĄD logiczny Dzień = 2004
// BŁĄD: konwersja niemożliwa
Data innadata = new Data(2004.0d, 11.0d, 12.0d);
```

Dla jednej klasy można zdefiniować wiele różnych konstruktorów, lecz każdy z nich musi być niepowtarzalny. Technika definiowania konstruktorów o identycznej nazwie, które różnią się między sobą argumentami, nazywa się przeciążaniem konstruktora. Przeciążając konstruktor należy pamiętać, że jest on rozróżniany na podstawie typu danych argumentów oraz ich ilości (nazwy argumentów nie są brane pod uwagę):

```
class Data
{
    public Data(int iRok)
    {
        Rok = iRok;
    }

    // OK: inny typ danych niż powyżej
    public Data(short sRok)
    {
        Rok = sRok;
    }

    // BŁĄD: już zdefiniowano konstruktor z takim typem danych (short sRok)
    public Data(short sMiesiac)
    {
        Miesiac = sMiesiac;
    }

    // OK: inna liczba argumentów
    public Data(short sRok, short sMiesiac, short sDzien)
    {
        Rok = sRok;
        Miesiac = sMiesiac;
        Dzień = sDzien;
    }

    private short Rok, Miesiac, Dzień;
}
```

Jak widać z powyższego przykładu, nie możemy zdefiniować dwóch konstruktorów o takiej samej liczbie i typie argumentów.

Powiedzmy, że chcemy zdefiniować konstruktory inicjujące datę w konwencji *yyyy-mm-dd* oraz

konwencji *mm-dd-yyyy*. Poniższy przykład nie jest poprawnym rozwiązaniem, ze względu na identyczną liczbę i typ argumentów:

```
class Data
{
    public Data(short sRok, short sMiesiac, short sDzien)
    {
        ...
    }

    public Data(short sMiesiac, short sDzien, short sRok) // BŁĄD
    {
        ...
    }
}
```

Jak rozwiązać ten problem? Bardzo prosto. Musimy zdefiniować nowe typy danych: **Rok**, **Miesiac**, **Dzien** i wskazać je jako typy argumentów wejściowych. W przypadku małych typów danych, zamiast klas lepiej zastosować struktury:

```
struct Rok
{
    public short Wartosc;
}

struct Miesiac // tu można wykorzystać typ wyliczeniowy
{
    public short Wartosc;
}

struct Dzien
{
    public short Wartosc;
}

class Data
{
    public Data(Rok rok, Miesiac miesiac, Dzien dzien)
    {
        ...
    }

    public Data(Miesiac miesiac, Dzien dzien, Rok rok)
    {
        ...
    }
}
```

```
}  
}
```

## Lista inicjacyjna

Do inicjacji pól można również wykorzystać listę inicjacyjną, która w rzeczywistości jest odwołaniem do innego konstruktora zawierającego argumenty. Do tego celu służy słowo kluczowe *this* oznaczające wskazanie, iż chodzi o tą klasę. Aby jednak można było skorzystać z listy inicjacyjnej w klasie musi istnieć konstruktor, na który się powołujemy:

```
class Data  
{  
    public Data() : this(2004, 11, 12)  
    {  
    }  
  
    public Data(short sRok, short sMiesiac, short sDzien)  
    {  
        Rok = sRok;  
        Miesiac = sMiesiac;  
        Dzien = sDzien;  
    }  
  
    private short Rok, Miesiac, Dzien;  
}
```

W powyższym przykładzie, w przypadku skorzystania z konstruktora bezargumentowego wywołany zostanie drugi konstruktor z wartościami argumentów podanymi w nawiasie (istotna jest kolejność argumentów). Oznacza to, że powołanie takiego obiektu:

```
Data data=new Data();
```

Będzie równoznaczne powołaniu takiego obiektu:

```
Data data=new Data(2004, 11, 12);
```

Lista inicjacyjna zarezerwowana jest jednak jedynie dla konstruktorów. Inne metody nie mogą z niej korzystać.

## Konstruktor kopiujący

Wśród konstruktorów szczególną rolę pełni konstruktor kopiujący. Jego zadaniem jest kopiowanie wartości pól istniejącego obiektu, do innego obiektu tego samego typu, który jest powoływany. W języku C# kompilator domyślnie nie tworzy konstruktora kopiującego, dlatego musimy go zdefiniować sami (jeżeli będzie nam potrzebny). Konstruktor ten charakteryzuje się tym, że ma tylko jeden argument, którym jest nazwa klasy, w której go zdefiniowano.

```
public Data(Data wartosc)
```

Wewnątrz tego konstruktora, należy nadać każdemu z pól klasy, wartość zgodną z wartością pól klasy argumentu:

```
public Data(Data wartosc)
{
    Rok = wartosc.Rok;
    Miesiac = wartosc.Miesiac;
    Dzień = wartosc.Dzień;
}
```

Skopiowanie wartości jednego obiektu wykonuje się tak:

```
// data_2 zawierać będzie takie same wartości pól jak data_1
Data data_1 = new Data(2004, 11, 12), data_2 = new Data(data_1);
```

## Niszczenie obiektu klasy

W języku C# nie ma możliwości bezpośredniego zniszczenia utworzonego wcześniej obiektu (zajmie się tym „*kolekcjoner nieużytków*”). Dzięki takiemu podejściu unika się wielu błędów: niezwolnienia pamięci (tzw. problem „*wycieków pamięci*”), próby ponownego zwolnienia pamięci zajmowanej (próba zniszczenia nieistniejącego obiektu) przez obiekt, czy też zniszczenia aktywnego obiektu (obiekt może być jeszcze potrzebny, jeżeli zostanie zwolniony wcześniej, późniejsze odwołania do niego spowodują pojawienie się błędu). Oznacza to również, że nie ma możliwości kontrolowania kolejności, w jakiej obiekty będą niszczone (można jedynie kontrolować kolejność ich tworzenia).

„Kolekcjoner nieużytków” jest procesem automatycznym, który gwarantuje:

- zniszczenie obiektu (nie jest jednak określone kiedy dokładnie nastąpi jego zniszczenie);
- obiekt zostanie zniszczony tylko raz;
- jedynie obiekty dłużej nieużywane zostaną zniszczone (obiekt jest niszczone tylko, jeżeli żaden inny obiekt nie przechowuje referencji do niego, funkcja wyszukiwania obiektów, do których nie ma żadnych referencji jest czasochłonna, więc „*kolekcjoner nieużytków*” wykonuje tę czynność tylko, gdy wielkość dostępnej pamięci jest niska).

Niszczenie obiektu w języku C# przebiega w dwóch etapach:

- deinicjacja obiektu – czynności wykonywane przez destruktora, które polegają na tzw. „sprzątaniu” oraz zaznaczeniu, że pamięć zajmowana przez obiekt nie będzie już używana (proces ten można kontrolować poprzez dodanie do klasy własnego destruktora);
- zwolnienie nieużywanej pamięci – czynności wykonywane przez „kolekcjonera nieużytków”, które polegają na zwracaniu do puli nieużywanej już pamięci (proces ten nie może być kontrolowany).

W przypadku, gdy obiekt wykorzystuje jakiś zasób niezarządzalny (np.: uchwyt do pliku), który nie może być zniszczony przez „kolekcjonera nieużytków”, należy samemu zwolnić pamięć. Do kontrolowania tego typu zasobów służy destruktora (lub metoda **Finalize**), który jest wywoływany automatycznie przez „kolekcjonera nieużytków”, kiedy obiekt jest niszczone.

W języku C# nie ma możliwości bezpośredniego wywołania i nadpisania metody **Object.Finalize**, dlatego kod odpowiedzialny za deinicjację obiektu, należy umieścić w destruktora, który zostanie automatycznie skonwertowany przez kompilator do metody **Finalize**.



## Destruktor

Destruktor jest specjalną metodą klasy, która wykonywana jest jako ostatnia. Charakteryzuje się on tym, że podobnie jak konstruktor ma taką samą nazwę jak nazwa klasy, w której go zdefiniowano, z tą jednak różnicą, że przed nazwą umieszcza się znak tyldy (~). Poza tym destruktory nie posiadają modyfikatora dostępu, nie zwracają żadnej wartości i nie przyjmują żadnych argumentów:

```
class Data
{
    ~Data()
    {
        // deinicjacja
    }
}
```

Destruktora należy unikać, o ile nie jest niezbędny (głównie w przypadku zwalniania pamięci zajmowanej przez zasoby niezarządzalne). Jeżeli chodzi o zasoby zarządzalne (obiekty), nie ma potrzeby definiowania w klasie destruktora.

## Słowo kluczowe **this**

Słowo kluczowe **this** („to”) określa odwołanie się do aktywnej instancji klasy.

Stosuje się je głównie, gdy:

- nazwy argumentów pokrywają się z nazwami pól klasy (bez **this** nie możemy przypisać polu wartości przekazywanej przez argument, gdyż kompilator nie będzie wiedział, że chodzi o pole klasy).

Przykład:

```
class Data
{
    Data(short sRok, short sMiesiac, short sDzien)
    {
        this.sRok = sRok;
        this.sMiesiac = sMiesiac;
        this.sDzien = sDzien;
    }

    private short sRok, sMiesiac, sDzien;
}
```

- przekazujemy obiekt do metody.

Przykład:

```
PrzetworzDate(this);
```

- deklarujemy indeksatory (na temat indeksatorów będzie mowa w dalszej części książki).

## Metody klasy

Naczelną zasadą poprawnego pisania programów, jest dzielenie kodu na funkcjonalne części. Podział na funkcjonalne części pozwala na ich łatwiejsze rozumienie oraz daje możliwość ponownego użycia.

Te funkcjonalne bloki kodu nazywa się metodami. Przykładowo metoda `WriteLine` znajdująca się w klasie `Console`, ma zamkniętą funkcjonalność pozwalającą na wyświetlanie informacji na ekranie. Trudno sobie wyobrazić, gdybyśmy musieli za każdym razem pisać te same operacje, które zawiera ta metoda zawsze, gdy byłoby wymagane wyświetlenie czegoś na ekranie.

## Definiowanie

W języku C# każda metoda jest składnikiem jakiejś klasy.

Składnia definicji metody wygląda następująco:

```
[modyfikator] typ Identyfikator([lista_argumentów])
{
    [ciało_metody]
}
```

gdzie:

*modyfikator*

modyfikatory określające zachowanie i dostępność metody (opcjonalne),

*typ*

typ danych zwracany przez metodę (wymagane),

*Identyfikator*

nazwa metody musi być różna od nazwy klasy, w której ją zdefiniowano (wymagane),

*lista\_argumentów*

lista argumentów przekazywanych do metody (opcjonalne),

*ciało\_metody*

kod realizujący zadaną funkcjonalność (opcjonalne, gdy typ zwracany przez metodę jest **void**).

Przykład definiowania metody `WyświetlRok` w klasie `Data`:

```
class Data
{
    // konstruktory, inne metody, itp.
    // ...
    public void WyświetlRok()
    {
        Console.WriteLine("Rok  A.D.", sRok);
    }

    private short sRok, sMiesiac, sDzien;
}
```

Definiując metodę należy pamiętać (tak samo jak przy definiowaniu pól) o tym, że jeżeli nie

zdefiniujemy żadnego modyfikatora dostępu, domyślnie metoda będzie prywatna (będzie można z niej korzystać jedynie wewnątrz klasy, w której ją zdefiniowano).

Nie wszystkie metody powinny być publiczne. Czasami jedynie na potrzeby danej klasy (chcąc rozbić pewną funkcjonalność na mniejsze człony), tworzymy dodatkowe metody, których nie ma sensu udostępniać:

```
class Liczba
{
    private void Drukuj()
    {
        Console.WriteLine(Wartosc);
    }

    public void Wydruk()
    {
        Console.Write("Wartość = ");
        Drukuj();
    }

    private int Wartosc;
}
```

Wewnątrz metod można umieszczać: deklaracje zmiennych lokalnych, szeregi instrukcji i wyrażeń oraz wywołania innych metod:

```
public void Liczenie()
{
    int iLiczba1, iLiczba2 = 100, iWynik = 0;

    for (iLiczba1 = 0; iLiczba1 < iLiczba2; iLiczba1++)
    {
        if ((iLiczba1 % 10) == 0)
            iWynik += iLiczba1;
    }

    Console.WriteLine(iWynik);
}
```

## Zwracanie wartości

Każda metoda może zwracać wartość zgodną z typem umieszczonym w definicji. Do zwracania wartości służy słowo kluczowe **return**. Po zastosowaniu **return** następuje natychmiastowe zwrócenie wartości i wyjście z metody (wszystko co znajdzie się za słowem **return** nie zostanie wykonane).

Przykład:

```
public int Kwadrat(int a)
```

```
{  
    return a * a;  
}
```

Jedynie w przypadku metod nie zwracających wartości (*void*) nie wymaga się umieszczania słowa kluczowego *return*:

```
void Pokaz()  
{  
    Console.WriteLine("Test");  
}
```

Można je jednak wykorzystać do opuszczenia metody, gdy zaistnieje jakiś warunek:

```
void Pokaz(bool bPokaz)  
{  
    if (!bPokaz)  
        return;  
  
    Console.WriteLine("Test");  
}
```

Powyższy przykład wypisze słowo „Test” na ekranie tylko, jeżeli parametr bPokaz uzyska wartość *true*. Jeżeli będzie miał wartość *false* nastąpi wyjście z funkcji.

Jedynie w przypadku funkcji nie zwracających żadnych wartości, nie wymaga się słowa *return*. W pozostałych przypadkach jest ono wymagane. Wymaganie to dotyczy wtedy każdej możliwej ścieżki zakończenia metody (przypadek gdy występują instrukcje warunkowe zawierające słowo *return*). Oznacza to, że gdy używamy *return* w instrukcji warunkowej, musimy pamiętać, że wartość zostanie zwrócona jedynie w przypadku zaistnienia określonego warunku. Jeżeli w pozostałych przypadkach nie umieścimy *return*, kompilator zwróci nam błąd („not all code paths return a value” – nie wszystkie ścieżki kodu zwracają wartość):

```
public int Abs(int x)  
{  
    if (x < 0)  
        return (-1) * x;  
    // BŁĄD: brakuje return dla przypadku x >= 0  
}
```

W powyższym przypadku ścieżka dla warunku ( $x < 0$ ) zwróci wartość  $(-1) * x$ , natomiast dla pozostałych przypadków nie ma odpowiedniej instrukcji *return*.

Poprawnie metoda powinna wyglądać tak:

```
public int Abs(int x)  
{  
    if (x < 0)  
        return (-1) * x;  
  
    return x; // wykona się dla x >= 0  
}
```

```
}
```

W przypadku korzystania z **return** w instrukcji warunkowej, nie ma potrzeby umieszczania klauzuli **else**, ze względu na specyfikę działania samej instrukcji **return**. Co nie oznacza, że taki zapis jest zły:

```
public int Abs(int x)
{
    if (x < 0)
        return (-1) * x;
    else
        return x;    // OK
}
```

## Argumenty

Argumenty stanowią informacje, które mogą być przekazywane do oraz z metody. W metodzie umieszcza się je w okrągłych nawiasach oddzielając przecinkami. Każdy argument przyjmuje następującą postać:

```
[modyfikator_argumentu] typ Identyfikator
```

gdzie:

*modyfikator\_argumentu*

dopuszczalne są modyfikatory: **params**, **out**, **ref** (opcjonalne, jeżeli nie zostanie podany domyślnie, przyjmowane jest, iż chodzi o argument wejściowy przekazywany przez wartość);

*typ*

typ danych argumentu (wymagane);

*Identyfikator*

nazwa argumentu (wymagane).

Przykład:

```
public void Dodaj(int Liczba1, int Liczba2, out int Wynik)
{
    Wynik = Liczba1 + Liczba2;
}
```

Istnieją trzy metody przekazywania argumentów:

- przez wartość – tzw. argumenty wejściowe, ponieważ dane mogą być przekazywane jedynie do metody (nie można ich zwracać z metody);
- przez referencję – tzw. argumenty wejściowo/wyjściowe, ponieważ dane mogą być przekazywane w obu kierunkach (do oraz z metody);
- przez wyjście – tzw. argumenty wyjściowe, ponieważ dane mogą być przekazywane jedynie z metody (nie można ich przekazywać do metody).

Każdy argument przekazywany do metody jest traktowany jak zmienna lokalna (argumenty dostępne są w obrębie całej metody, ale nie poza nią). Ze względu na ten fakt należy uważać, aby nie deklarować zmiennych o tej samej nazwie co nazwy argumentów:

```
public void Test(int Liczba)
{
```

```
Liczba = 100;      // OK  
  
int Liczba = 0; // BŁĄD: nie można zadeklarować zmiennej o tej samej nazwie  
}
```

### Argumenty przekazywane przez wartość

W języku C# przekazywanie argumentów do metody przez wartość jest domyślnym sposobem, więc nie stosuje się w tym przypadku żadnych modyfikatorów argumentu. Typ wartości lub zmiennej przekazywanej do metody jako argument, musi być zgodny z zadeklarowanym typem argumentu (lub typem, który może zostać skonwertowany w sposób niejawny).

Weźmy przykładową metodę:

```
public void Licz(int Liczba)  
{  
    ...  
}
```

Argumentem może być liczba zgodnego typu:

```
Licz(10);
```

Zmienna zgodnego typu:

```
int Liczba = 100;  
Licz(Liczba);
```

Zmienna typu, który konwertuje się niejawnie do typu argumentu:

```
short Liczba = 100;  
Licz(Liczba); // short konwertuje się niejawnie do int
```

Oczywiście można również pod tym samym warunkiem przekazać wartość zwracaną z innej metody:

```
Licz(Abs(-4)); // metoda Abs zwracała wartość typu int, więc jest zgodna
```

Argumenty przekazywane przez wartość są jedynie kopią danych, które są przekazywane do metody, więc wszelkie zmiany wykonywane na tej kopii nie wpływają na zawartość danych, które przekazano.

Przykład metody:

```
public void Licz(int Liczba)  
{  
    Liczba++; // inkrementacja dotyczy kopii  
}
```

Wywołanie:

```
int Liczba = 10; // oryginalne dane  
Licz(Liczba); // liczba ma nadal wartość 10 i nie została zmieniona
```

### Argumenty przekazywane przez referencję

Argumenty przekazywane przez referencję nie przechowują wartości, tylko takie samo wskazanie do danych, jakie zostanie przekazane do metody. Oznacza to, że operując na argumencie referencyjnym, operujemy tak naprawdę na danych, które zostały przekazane do metody. W przypadku argumentów przekazywanych przez wartość mieliśmy do czynienia jedynie z kopią danych, tu pracujemy na

oryginalnych danych. Argument referencyjny deklaruje się poprzez dodanie słowa *ref* zarówno w deklaracji jak i wywołaniu metody. Typ argumentu musi być zgodny z typem referencyjnym danych przekazywanych do metody.

Przykładowo dla metody:

```
public void Licz(ref int Liczba)
{
    Liczba++;
}
```

Wwołanie wygląda tak:

```
int Liczba = 10; // oryginalne dane
Licz(ref Liczba); // liczba będzie miała wartość 11
```

### Argumenty przekazywane przez wyjście

Argumenty przekazywane przez wyjście służą tylko do przekazywania danych z metody. Operują one na kopii danych, które muszą zostać zainicjowane jakąś wartością (podobnie jak niezainicjowane zmienne). Następnie wartość ta zostanie przypisana danym oryginalnym, przekazanym do metody. Oznacza to, że nie ma możliwości odczytania danych przekazanych do metody, bowiem w pierwszej kolejności musi nastąpić operacja przypisania wartości. Argument przekazywany przez wyjście, deklaruje się przez dodanie słowa *out* zarówno w deklaracji jak i wywołaniu metody. Typ argumentu musi być zgodny z typem danych przekazywanych do metody.

Przykładowo dla metody:

```
public void Licz(out int Liczba)
{
    // nie ma możliwości odczytu kopii danych
    Liczba = 0;
    // można odczytać wartość kopii
}
```

Wwołanie wygląda tak:

```
int Liczba = 10; // oryginalne dane
Licz(out Liczba); // zmiennej Liczba przypisana zostanie wartość kopii
```

### Lista argumentów o zmiennej długości

W niektórych sytuacjach istnieje potrzeba zdefiniowania metody pozwalającej przekazywać zmienną liczbę argumentów wejściowych. Lista argumentów pozwala na określenie wspólnego typu danych, więc wszystkie argumenty przekazywane do metody muszą być zgodne z typem listy lub typem, który może zostać niejawnie skonwertowany do typu listy (podobnie jak w przypadku argumentów przekazywanych przez wartość). Do deklarowania listy argumentów o zmiennej długości służy słowo *params*, (po którym umieszcza się typ tablicy jednowymiarowej oraz nazwę argumentu).

Przykładowo dla metody:

```
public int Sumuj(params int[] Lista)
{

```

```
int iSuma = 0;

for (int i = 0; i < Lista.Length; i++)
    iSuma += Lista[i];

return iSuma;
}
```

Wywołanie wyglądać może tak:

```
int Suma1 = Sumuj(1,2,3);
int Suma2 = Sumuj(2,3,4,4,5,6,6);
int Suma3 = Sumuj(new int[]{1,2,3,4,5}); // przekazane w postaci tablicy
```

## Wywoływanie

Zdefiniowaną metodę woła się poprzez podanie jej nazwy i przekazanie listy argumentów w okrągłych nawiasach. Jeżeli metoda nie ma żadnych argumentów nawiasy są i tak wymagane:

```
NazwaMetody();
```

Jeżeli metoda zwraca jakąś wartość, możemy użyć tej wartości w jakimś wyrażeniu np. przypisać ją do zmiennej lub użyć w czasie sprawdzania prawdziwości warunku.

Przykładowo dla metody:

```
public bool Sprawdz()
{
    if (1 == 0)
        return false;

    return true;
}
```

Użycie jej w wyrażeniu wygląda tak:

```
if (Sprawdz())
    Console.WriteLine("Ok");
```

Przypisanie do zmiennej:

```
bool bSprawdzanie = Sprawdz();
```

Jeżeli metoda posiada listę argumentów, przy jej wywołaniu należy te argumenty podać w odpowiedniej kolejności, zwracając uwagę na typ.

Przykładowo dla metody:

```
public int Abs(int x)
{
    if (x < 0)
        return (-1) * x;

    return x;
}
```



```
}
```

Przykłady wywołania:

```
int Wynik = Abs(-5), innyWynik = Abs(Wynik);
```

Wołając metodę w tej samej klasie, w której została zdefiniowana, nie stosujemy operatora dostępu do składowej (operator '.'):

```
class Liczba
{
    private void Drukuj()
    {
        Console.WriteLine(Wartosc);
    }

    public void DrukujWiele(int Ile)
    {
        for (int i = 0; i < Ile; i++)
            Drukuj();
    }

    private int Wartosc;
}
```

Jeżeli jednak powołujemy obiekt i chcemy skorzystać z jego metody, musimy użyć operatora dostępu do składowej '.':

```
Liczba liczba = new Liczba(); // tworzymy obiekt
// wołamy metodę w formie NazwaKlasy.NazwaMetody
liczba.DrukujWiele(100);
```

Szczególnym przypadkiem wywołania metod są metody statyczne, które nie wymagają tworzenia obiektów (gdyż należą do klasy a nie jej instancji). Metoda WriteLine z klasy Console jest taką metodą, dlatego kiedy ją wołamy nie tworzymy obiektu klasy Console (metody statyczne zostały omówione w dalszej części książki):

```
class Druk
{
    public static void Drukuj(string s)
    {
        Console.WriteLine(s);
    }
}

class Wyświetlanie
{
    public void Pokaz()
    {
        Druk.Drukuj("Test");
    }
}
```

```
}  
}
```

## Rekurencja

Metody można wywołać z wnętrza innych metod. Można również wewnątrz metody wywołać tą samą metodę. Wywołanie siebie przez metodę nazywa się rekurencją:

```
public void Test()  
{  
    Test(); // rekurencyjne wołanie  
}
```

W powyższym przykładzie po wywołaniu metody Test nastąpi jej rekurencyjne wywoływanie do momentu przepełnienia stosu. Tego typu rekurencji należy unikać.

Z rekurencji można korzystać tylko, jeżeli zapewni się obsługę wyjścia z metody (po spełnieniu wymaganych warunków). Np.:

```
class TestWartosc  
{  
    public void Test()  
    {  
        if ((Wartosc++) == 10)  
            return;  
  
        Test();  
    }  
  
    private int Wartosc = 0;  
}
```

W powyższym przykładzie wywołanie metody Test spowoduje rekurencyjne wywołania do momentu, gdy pole Wartość nie osiągnie wartości 10 (wtedy nastąpi wyjście z metody).

## Przeciążanie

Wewnątrz klasy może być zdefiniowanych kilka różnych metod mających taką samą nazwę, ale muszą się różnić między sobą listą argumentów. Definiowanie kilku metod o takiej samej nazwie i różnej liście argumentów, nazywa się przeciążaniem.

Przykład:

```
class Test  
{  
    public void Pokaz()  
    {  
        ...  
    }  
}
```

```
public void Pokaz(int Liczba)
{
    ...
}

public void Pokaz(string Napis)
{
    ...
}

public void Pokaz(int Liczba, string Napis)
{
    ...
}
}
```

Przy przeciążaniu metody, typ zwracanej wartości nie jest brany pod uwagę, więc zdefiniowanie dwóch metod, różniących się jedynie typem zwracanej wartości jest błędem:

```
class Test
{
    public void Pokaz()
    {
        ...
    }

    // BŁĄD: już zdefiniowano Pokaz z taką listą
    public string Pokaz()
    {
        ...
    }

    public int Pokaz(int Liczba)
    {
        ...
    }
}
```

Metod przeciążonych używa się, kiedy istnieje potrzeba zdefiniowania podobnie działających metod dla różnych typów i ilości argumentów oraz rozszerzania funkcjonalności w istniejącym kodzie:

```
class Dodawanie
{
    public int Dodaj(int a, int b)
    {
        return a + b;
    }
}
```

```
    }

    public int Dodaj(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

## Statyczne składniki klasy

Statyczne składniki klasy związane są z klasą, a nie z konkretną instancją obiektu tej klasy. Oznacza to, że składnik taki jest częścią wspólną dla każdej instancji obiektu tej klasy i istnieje niezależnie od tego, czy utworzymy jakąkolwiek instancję obiektu czy nie. Składową statyczną tworzy się umieszczając modyfikator **static** przed typem zwracanej wartości składnika.

Przykład:

```
class Test
{
    public static int Liczba;    // pole statyczne

    public static void Zwiększ() // metoda statyczna
    {
        Liczba++;
    }
}
```

Odwołanie do składnika klasy możliwe jest przez użycie operatora dostępu do składowej ('.').

```
Test.Liczba = 100; // odwołanie do pola statycznego
Test.Zwiększ();   // Wywołanie metody statycznej
```

Pola statyczne tworzy się, gdy istnieje potrzeba przechowywania wspólnej informacji dla każdej instancji obiektu danej klasy.

Przykładowo, pole dotyczące informacji o oprocentowaniu lokat bankowych:

```
class Lokata
{
    public static float Procent = 6.5f;
}
```

Pole statyczne może zostać również wykorzystane jako kontener informacji dostępnej dla każdej klasy:

```
class Ustawienia
{
    public static bool bWidoczność;
    public static Kolor KolorTła;
}
```

Metody statyczne tworzy się, gdy istnieje potrzeba wykonywania jakiejś grupy instrukcji na prywatnych

polach statycznych klasy:

```
class Lokata
{
    private static float Procent = 6.5f;

    public static void Zmien(float NowyProcent)
    {
        Procent=NowyProcent;
    }
}
```

W przypadku składowych statycznych należy pamiętać, że skoro nie są one związane z instancją klasy, nie możemy korzystać ze słowa kluczowego *this*.

Jeżeli chcemy odwołać się do składowej, musimy skorzystać z operatora odwołania do składowej:

```
class Lokata
{
    private static float Procent = 6.5f;

    public static void Zmien(float Procent)
    {
        Lokata.Procent = Procent; // nie możemy korzystać z this
    }
}
```

Metody statyczne można również zgrupować w klasie oferującej jakąś funkcjonalność (np.: klasa Console zawiera kilka znanych nam metod statycznych oferujących podstawową funkcjonalność obsługi strumieni wejścia/wyjścia). Metody zdefiniowane w takiej funkcjonalnej klasie, mogą być dostępne w dowolnym momencie dla dowolnej klasy:

```
class Matematyka
{
    public static int Abs(int x)
    {
        ...
    }

    public static double Sin(double x)
    {
        ...
    }

    public static double Cos(double x)
    {
        ...
    }
}
```

```
}
```

## Przeciążanie operatorów

W języku C# istnieje możliwość zdefiniowania funkcjonalności dużej części operatorów dla typów stworzonych przez użytkownika. Dzięki takiemu zabiegowi, działając na obiektach możemy używać czytelnego zapisu operatorowego.

Przykładowo, zamiast definiować dla klasy `Liczba` metodę dodawania o nazwie `Dodaj`, możemy przeciążyć operator `+`. Przeciążanie operatora polega na zdefiniowaniu statycznej metody z użyciem słowa kluczowego *operator*.

Składnia definicji przeciążonego operatora może przyjąć jedną z dostępnych składni:

```
public static zwr_typ operator operator_unarny(typ_argumentu argument)
public static zwr_typ operator operator_binarny(typ_argumentu1 argument1,
                                                typ_argumentu2 argument2)

public static implicit operator typ_konw_wyj(typ_konw_wej argument)
public static explicit operator typ_know_wyj(typ_know_wyj argument)
```

gdzie:

*zwr\_typ*

typ wyniku zwracany przez operator (wymagane);

*operator\_unarny*

jeden z dozwolonych operatorów unarnych: `+`, `-`, `!`, `~`, `++`, `--`, *true*, *false* (wymagane);

*operator\_binarny*

jeden z dozwolonych operatorów binarnych: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `<`, `>`, `<=`, `>=` (wymagane);

*typ\_argumentu*

typ argumentu wejściowego (wymagane);

*typ\_argumentu1*

typ pierwszego argumentu wejściowego (wymagane);

*typ\_argumentu2*

typ drugiego argumentu wejściowego (wymagane);

*argument, argument1, argument2*

nazwa argumentu (wymagane);

*typ\_konw\_wej*

typ wejściowy dla operatora konwersji (wymagane);

*typ\_konw\_wyj*

typ wyjściowy dla operatora konwersji (wymagane).

Przykład:

```
class Liczba
{
    private int Wartosc;

    public Liczba(int Wartosc)
```

```
{
    this.Wartosc = Wartosc;
}

public static Liczba operator+(Liczba l1, Liczba l2)
{
    return new Liczba(l1.Wartosc + l2.Wartosc); // zwracamy obiekt więc new
}
}
```

Wywołanie:

```
Liczba l1 = new Liczba(100), // inicjacja
        l2 = new Liczba(200),
        l3;                  // zmienna na wynik
l3 = l1 + l2;                // dodanie wartosci obu typów
```

Nie wolno przeciążać operatorów:

- logicznych warunkowych: **&&**, **||** (są rozwijane przy pomocy: **&**, **|**, **true** oraz **false**)
- konwersji: **()** (służą do tego celu składnie ze słowami kluczowymi: **implicit** i **explicit**)
- indeksacji: **[][]** (służą do tego celu indeksatory)
- przypisań: **+=**, **-=**, **\*=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=** (są rozwijane automatycznie)
- oraz: **=**, **.,**, **?:**, **new**, **is**, **sizeof**, **typeof**

## Przeciążanie operatorów relacji

Operatory relacji muszą być przeciążane parami. Jeżeli przeciążymy operator **==** musimy również przeciążyć operator **!=**. To samo dotyczy pary **<=** oraz **>=** jak i pary **<** oraz **>**. Definiując operatory równości, powinniśmy nadpisać wirtualną metodę **Equals**, która jest dziedziczona przez klasę z klasy **Object** (na temat dziedziczenia i metod wirtualnych będziemy mówić w dalszej części książki, należy jednak pamiętać o nadpisaniu metody **Equals**, by zachowana została spójność w zachowaniu obiektów klasy). Metoda **Equals** zwraca wartość **true** dla równości a **false** dla różności.

Najlepszą techniką definiowania operatorów, jest stworzenie metody, która zbada relacje między obiektami i zwróci odpowiednią wartość. Taką metodę możemy następnie użyć w wyrażeniu w czasie definiowania par operatorów:

```
class Liczba
{
    private int Wartosc;

    public Liczba(int Wartosc)
    {
        this.Wartosc = Wartosc;
    }

    // metoda pomocnicza (dzięki niej unikamy redundancji)
    private int Porownaj(Liczba Inna)
```

```
{
    if (Wartosc < Inna.Wartosc)
        return -1;                // mniejsza

    if (Wartosc > Inna.Wartosc)
        return 1;                  // większa

    return 0;                       // równe
}

public static bool operator==(Liczba l1, Liczba l2)
{
    // gdy będą równe, metoda Porównaj zwróci wartość 0, warunek 0 == 0 da true
    return l1.Porównaj(l2) == 0;
}

public static bool operator!=(Liczba l1, Liczba l2)
{
    // gdy będą różne, metoda Porównaj zwróci wartość różną od 0 (1 lub -1),
    // warunek (1 lub -1) != 0 da true
    return l1.Porównaj(l2) != 0;
}

public static bool operator<(Liczba l1, Liczba l2)
{
    return l1.Porównaj(l2) < 0;
}

public static bool operator>(Liczba l1, Liczba l2)
{
    return l1.Porównaj(l2) > 0;
}

public static bool operator<=(Liczba l1, Liczba l2)
{
    return l1.Porównaj(l2) <= 0;
}

public static bool operator>=(Liczba l1, Liczba l2)
{
    return l1.Porównaj(l2) >= 0;
}
```



```
// nadpisanie metody wirtualnej Equals
public override bool Equals(object ob)
{
    // upewniamy się, że obiekt przechowuje Liczba, rozpakowujemy go i używamy
    // jak poprzednio funkcji Porownaj (nie jest to statyczna metoda, więc
    // nie odwołujemy się do składowej)
    return ob is Liczba && Porownaj((Liczba)ob) == 0;
}
}
```

## Przeciążanie operatorów logicznych

Nie ma możliwości zdefiniowania operatorów logicznych wprost, ponieważ są one rozwijane za pomocą: **&**, **|**, **true** oraz **false**.

Jeżeli zmienne *x* oraz *y* są typu *T*, rozwinięcie wygląda następująco:

- dla operatora **&&** jest to *T.false(x) ? x : T.&(x, y)*
- dla operatora **||** jest to *T.true(x) ? x : T.|(x, y)*

Oznacza to, że aby rozwinąć wyrażenie z operatorem **&&** i **||** musimy przeciążyć operatory *false*, *true*, **&&**, oraz **||**:

```
class Liczba
{
    private int Wartosc;

    public Liczba(int Wartosc)
    {
        this.Wartosc = Wartosc;
    }

    public static bool operator false(Liczba l)
    {
        return l.Wartosc == 0; // przykładowo niech wartość zero będzie fałszem
    }

    public static bool operator true(Liczba l)
    {
        return l.Wartosc != 0; // a wartość różna od zera niech będzie prawda
    }

    public static Liczba operator&(Liczba l1, Liczba l2)
    {
        return new Liczba(l1.Wartosc & l2.Wartosc);
    }
}
```

```
public static Liczba operator|(Liczba l1, Liczba l2)
{
    return new Liczba(l1.Wartosc | l2.Wartosc);
}
}
```

Przykład użycia:

```
Liczba l1 = new Liczba(0),
      l2 = new Liczba(100);

if (l2)
    Console.WriteLine("l2 jest różna od zera"); // prawda

if (l1 && l2)
    Console.WriteLine("l1 i l2 są różne od zera"); // fałsz

if (l1 || l2)
    Console.WriteLine("l1 lub l2 jest różne od zera"); // prawda
```

## Przeciążanie operatorów konwersji

Możliwe jest przeciążenie operatora konwersji, zarówno jego jawnej jak i niejawnej postaci. Służą do tego słowa kluczowe *implicit* (niejawny) oraz *explicit* (jawny). W przypadku definiowania operatorów konwersji do typu *string* zaleca się, aby dla zachowania konsekwencji, zachowania obiektu nadpisana została wirtualna metoda *ToString*, która jest dziedziczona z klasy *Object*:

```
class Liczba
{
    private int Wartosc;

    public Liczba(int Wartosc)
    {
        this.Wartosc = Wartosc;
    }

    // niejawna konwersja do typu int
    public static implicit operator int(Liczba l)
    {
        return l.Wartosc;
    }

    // jawna konwersja do typu string
    public static explicit operator string(Liczba l)
    {
        return l.ToString();
    }
}
```

```
}

// nadpisanie metody wirtualnej ToString
public override string ToString()
{
    // konwersja do string
    return Convert.ToString(Wartosc);
}
}
```

Przykład użycia:

```
Liczba l1 = new Liczba(100);
int iLiczba = l1;
string strLiczba = (string)l1;
string strInnaLiczba = l1; // BŁĄD: niejawna konwersja nie była zdefiniowana
```

## Przeciążanie operatorów arytmetycznych

Operatory arytmetyczne mogą być przeciążane w różnych kombinacjach z udziałem obiektu, którego dotyczą. Jest tylko jedno ograniczenie. Co najmniej jeden argument definiowanego operatora, musi być obiektem klasy, dla której został przeciążony:

```
class Liczba
{
    private int Wartosc;

    public Liczba(int Wartosc)
    {
        this.Wartosc = Wartosc;
    }

    public static Liczba operator+(Liczba l1, Liczba l2)
    {
        return new Liczba(l1.Wartosc + l2.Wartosc);
    }

    public static Liczba operator/(Liczba l1, short sLiczba)
    {
        return new Liczba(l1.Wartosc / (short)sLiczba);
    }

    public static Liczba operator-(Liczba l1, int InnaLiczba)
    {
        return new Liczba(l1.Wartosc - InnaLiczba);
    }
}
```

```
public static Liczba operator+(int l1, int l2) // BŁĄD: co najmniej jeden
{
    // argument musi być typu
    return new Liczba(l1 + l2); // Liczba
}

public static int operator*(Liczba l1, Liczba l2)
{
    return l1.Wartosc * l2.Wartosc;
}
}
```

Przykład użycia:

```
Liczba l1 = new Liczba(100),
    l2 = new Liczba(10),
    l3;
int Wynik;
short Dzielna = 20;

/* wykorzystujemy dwa przeciążone operatory + dla zwracanego typu Liczba i
argumentów typu Liczba oraz operator / dla zwracanego typu Liczba oraz
argumentów typu Liczba i short */
l3 = (l1 + l2 + l2 + l1) / Dzielna;

/* wykorzystujemy przeciążony operator* dla zwracanego typu int oraz
argumentów typu Liczba */
Wynik = l1 * l2;
```

## Właściwości

Właściwości są mechanizmem wspierania hermetyzacji danych wewnątrz klasy. Mechanizm ten pośredniczy w zmianie wartości pól klasy. Definiując właściwość możemy zdefiniować rodzaj dostępu do właściwości jako: tylko do odczytu, tylko do zapisu lub do odczytu i zapisu. Służą do tego słowa **get** (odczyt) i **set** (zapis), które umieszczają się na początku bloku instrukcji wewnątrz definicji właściwości.

Jeżeli właściwość zawiera jedynie blok **get**, traktowana jest jako tylko do odczytu. Jeżeli zawiera tylko blok **set** traktowana jest jako tylko do zapisu. Jeżeli zawiera oba bloki traktuje się ją jako do zapisu i odczytu.

W przypadku bloku **get** należy użyć słowa **return** (tak jak w przypadku metod), aby zwrócić wartość właściwości. Natomiast w przypadku bloku **set**, polu do którego odwołuje się właściwość, przypisuje się wartość **value** (inaczej niż w przypadku metod, gdzie korzystać można z listy argumentów).

Składnia wygląda tak:

```
[modyfikator] typ Identyfikator
{
    deklaracja_dostępu
}
```

gdzie:

*modyfikatory*

dopuszczalne są modyfikatory: *new*, ***static***, ***virtual***, ***abstract***, ***override*** oraz modyfikatory dostępu (opcjonalne);

*typ*

typ danych właściwości (wymagane);

*Identyfikator*

nazwa właściwości (wymagane);

*deklaracja\_dostępu*

blok ***get*** lub/i ***set***, dla bloku ***get*** wymaga się zwrócenia wartości za pomocą ***return***, dla bloku ***set***, wartość przypisania przechowywana jest w ***value*** (wymagany co najmniej jeden z bloków).

Przykład:

```
class Liczba
{
    private int Wartosc;

    public int WartoscWl    // właściwość
    {
        // odczyt danych
        get
        {
            return Wartosc;
        }

        // zapis danych
        set
        {
            Wartosc=value;
        }
    }
}
```

Do właściwości można odwoływać się w taki sam sposób, jak odwołuje się do pól (odwołanie zostanie przetłumaczone przez kompilator na odpowiednie wywołanie ***get*** lub ***set***):

```
Liczba l = new Liczba();
l.WartoscWl = 100;           // zapisanie wartości, wywołanie l.WartoscWl.set
int Wartosc = l.WartoscWl;   // odczytanie wartości, wywołanie l.WartoscWl.get
```

Właściwości wykazują podobieństwo do pól klasy, ze względu na składnię odwołania. Z tego względu określa się je mianem „logicznych pól klasy”. W odróżnieniu jednak od pól, nie mogą być one przekazywane do metod jako argumenty ani przez referencje, ani przez wyjście. Mogą natomiast być przekazywane przez wartość:

```
Przelicz(l.WartoscWl); // ok. można przekazać przez wartość
Przelicz(ref l.WartoscWl); //BŁĄD: nie można przekazać przez referencję
```

*Przelicz(out l.WartoscWl); //BŁĄD: nie można przekazać przez wyjście*

Właściwości można definiować jako statyczne, ale muszą podobnie jak metody statyczne, odwoływać się do statycznych pól klasy:

```
class Liczba
{
    private static int Wartosc;

    public static int WartoscWl
    {
        get
        {
            return Wartosc;
        }

        set
        {
            Wartosc=value;
        }
    }
}
```

Przykład użycia:

```
Liczba.WartoscWl++;
int Ile = Liczba.WartoscWl;
```

Właściwości wykazują też podobieństwo do metod, ponieważ: zawierają instrukcje, służą do hermetyzacji danych i mogą, podobnie jak metody być: abstrakcyjne, wirtualne i nadpisywane. Różnią się jednak od nich tym, że nie używają listy argumentów (przekazywana jest jedynie wartość zgodna z typem właściwości poprzez domyślny argument *value*) i muszą zwracać wartość (nie mogą być *void*).

Właściwości mogą być stosowane do powoływania obiektów i inicjowania w momencie pierwszego odwołania się do nich. Mechanizm ten nazywa się „tworzeniem w samą porę” (ang. „Just-in-Time Creation”) lub „leniwym tworzeniem” (ang. „lazy creation”):

```
class Liczba
{
}

class Dzialania
{
    private static Liczba liczba = null; // puste odwołanie do obiektu

    public static Liczba Wartosc
    {
        get
        {

```

```
if (liczba == null)        // jeżeli odwołanie jest puste, nastąpi
    liczba = new Liczba(); // utworzenie obiektu i utworzenie odwołania

return liczba;
}
}
```

Przykład użycia:

```
// stworzenie obiektu (pierwsze odwołanie) i zwrócenie wartości liczba
Liczba l1 = Dzialania.Wartosc;
// zwrócenie wartości liczba (kolejne odwołanie, obiekt już istnieje)
Liczba l2 = Dzialania.Wartosc;
```

## Indeksatory

Indeksatory są mechanizmem pozwalającym na indeksowanie obiektu w taki sam sposób, w jaki indeksuje się tablice. W składni indeksatory przypominają właściwości z tą jednak różnicą, że zamiast nazwy, umieszcza się tu słowo kluczowe **this** oraz argument określający indeks w kwadratowych nawiasach.

Składnia wygląda następująco:

```
[modyfikator] typ this[argumenty_indeksów]
{
    deklaracja_dostępu
}
```

gdzie:

*modyfikator*

dopuszczalne są modyfikatory: **new**, **virtual**, **sealed**, **override**, **abstract**, **extern** oraz modyfikatory dostępu (opcjonalne);

*typ*

typ danych elementu zwracanego przez indeksator (wymagane);

*argumenty\_indeksu*

lista argumentów indeksów, w formacie typ\_indeksu nazwa\_indeksu oddzielone przecinkami, nie dopuszcza się argumentów przekazywanych przez referencję i wyjście (wymagany co najmniej jeden argument indeksu);

*deklaracja\_dostępu*

blok **get** lub/i **set**, dla bloku **get** wymaga się zwrócenia wartości elementu za pomocą **return**, dla bloku **set** wartość przypisania dla elementu przechowywana jest w **value** (wymagany co najmniej jeden z bloków).

Przykład:

```
class CiagLiczb
{
    private int[] lista;
```

```
public CiagLiczb(int Dlugosc)
{
    lista = new int[Dlugosc];
}

public int this[int index] // indeksator
{
    get
    {
        return lista[index]; // zwrócenie elementu tablicy
    }

    set
    {
        lista[index]=value; // przypisanie elementowi wartości
    }
}
```

Przykład użycia:

```
CiagLiczb ciag = new CiagLiczb(100);
ciag[0] = 1;
```

Indeksatory używają takiej samej notacji jak tablice, ale w odróżnieniu od tablic mogą używać indeksów różnych typów (nie tylko typu całkowitego jak tablice):

```
class Imiona
{
    /*
    Korzystamy z klasy dynamicznej tablicy typu „klucz-wartość” wchodzącej w
    skład przestrzeni nazw System.Collections (o kolekcjach będziemy mówić w
    dalszej części książki)
    */
    private Hashtable Nazwiska = new Hashtable();

    public string this[string Nazwisko]
    {
        get
        {
            return (string)Nazwiska[Nazwisko];
        }

        set
        {

```



```
        Nazwiska[Nazwisko]=value;
    }
}
}
```

Przykład użycia:

```
Imiona imiona = new Imiona();
imiona["Stefańczyk"] = "Andrzej";
string MojeImie = imiona["Stefańczyk"];
```

Indeksatory mogą być przeciążane w podobny sposób, w jaki przeciąża się metody (tzn., że można zdefiniować kilka różnych indeksatorów o różnych typach indeksów):

```
class CiagLiczb
{
    private int[] lista;

    public CiagLiczb(int Dlugosc)
    {
        lista = new int[Dlugosc];
    }

    public int this[int index]
    {
        get
        {
            return lista[index];
        }

        set
        {
            lista[index]=value;
        }
    }

    public int this[string strIndex]
    {
        get
        {
            return lista[Convert.ToInt32(strIndex)];
        }

        set
        {
            lista[Convert.ToInt32(strIndex)] = value;
        }
    }
}
```

```
    }  
    }  
}
```

Przykład użycia:

```
CiagLiczba ciag = new CiagLiczba(100);  
ciag[1] = 1;           // indeksacja liczbą całkowitą  
ciag["10"] = 0;       // indeksacja łańcuchem
```

Indeksatory mogą używać różnej ilości argumentów indeksujących. Każdy indeksator musi mieć jednak co najmniej jeden argument indeksujący. Argumenty te przekazywane są przez wartość:

```
class Matryca  
{  
    private int[,] matryca;  
  
    public Matryca(int M, int N)  
    {  
        matryca = new int[M, N];  
    }  
  
    public int this[int MIndex, int NIndex]  
    {  
        get  
        {  
            return matryca[MIndex, NIndex];  
        }  
  
        set  
        {  
            matryca[MIndex, NIndex] = value;  
        }  
    }  
}
```

Przykład użycia:

```
Matryca mat = new Matryca(10,10);  
mat[1,1] = 100;
```

Podobnie jak w przypadku właściwości, indeksatory nie mogą być przekazywane do metod jako argumenty ani przez referencje, ani przez wyjście. Mogą być za to przekazywane przez wartość:

```
Przelicz(ciag[0]); // ok. można przekazać przez wartość  
Przelicz(ref ciag[0]); //BŁĄD: nie można przekazać przez referencję  
Przelicz(out ciag[0]); //BŁĄD: nie można przekazać przez wyjście
```

Ze względu na to, że indeksatory związane są z instancją klasy, nie dopuszcza się możliwości tworzenia statycznych indeksatorów (inaczej niż w przypadku właściwości):

```
public static int this[int index] // BŁĄD
{
    ...
}
```

## Delegacje

Delegacja jest typem referencyjnym, który pełni rolę bezpiecznego „wskaźnika do funkcji”, dzięki któremu można stworzyć kod pozwalający na dynamiczną zmianę wołanych metod. Dzięki delegatom metody mogą być przekazywane jako argumenty. Do definiowania delegacji stosuje się słowo kluczowe *delegate*.

Składnia delegacji wygląda następująco:

```
[modyfikator] delegate typ Identyfikator([lista_argumentów]);
```

gdzie:

*modyfikator*

dozwolone są modyfikatory: *new* oraz modyfikatory dostępu (opcjonalne);

*typ*

typ danych zwracany przez metodę, na którą wskazuje delegacja (wymagane);

*Identyfikator*

nazwa delegacji musi być różna od nazwy klasy, w której ją zdefiniowano (wymagane);

*lista\_argumentów*

lista argumentów przekazywanych do metody, na którą wskazuje delegacja (opcjonalne).

Przykład:

```
class Metody
{
    // deklaracja delegacji dla metody typu void Nazwa()

    private delegate void Metoda();

    private void DrukujText()
    {
        Console.WriteLine("Text");
    }

    private void DrukujLiczbe()
    {
        Console.WriteLine("Liczba");
    }

    private void Drukuj(Metoda metoda)
    {
        metoda(); // w czasie definicji nie wiadomo jaka metoda będzie wywołana
    }
}
```

```
}

public void Wywołaj()
{
    Drukuj(new Metoda(DrukujLiczbe));
    Drukuj(new Metoda(DrukujText));
}
}
```

Przykład użycia:

```
Metody met = new Metody();
met.Wywołaj();
```

## Zdarzenia

Zdarzenia pozwalają obiektom na monitorowanie zmian zachodzących w innych obiektach. Wykorzystują one model wydawcy i czytelnika. Wydawcą jest obiekt, w którym zachodzą pewne zmiany. Czytelnikiem jest obiekt, który jest zainteresowany zmianami zachodzącymi u wydawcy. W momencie, w którym u wydawcy zajdzie jakaś zmiana, zainteresowani czytelnicy (ci, którzy zgłoszą zainteresowanie zmianami) zostaną powiadomieni o jej wystąpieniu. Innymi słowy zdarzenie jest wiadomością wysyłaną przez obiekt wydawcy do obiektu czytelnika, na którą czytelnik może zareagować wykonując jakąś czynność (analogia do rzeczywistości: kiedy wydawca ogłasza ukazanie się nowej książki zainteresowani jej wydaniem czytelnicy, mogą zareagować na tą wiadomość chęcią dokonania zakupu).

Zdarzenia wykorzystują delegacje do wywoływania metod w obiektach czytelników. Informacja o zmianie występującej u wydawcy, może spowodować wywołanie wielu delegacji. Nie ma jednak możliwości kontrolowania kolejności ich wywołania. Jeżeli wywołanie jednej z delegacji nie powiedzie się, pozostałe delegacje nie zostaną wywołane.

Składnia definicji wygląda następująco:

```
[modyfikatory] event typ nazwa_zdarzenia;
```

gdzie:

*modyfikatory*

dozwolone są modyfikatory: *abstract*, *new*, *override*, *static*, *virtual*, *extern* oraz modyfikatory dostępu (opcjonalne);

*typ*

delegacja, z którą ma być skojarzone zdarzenie (wymagane);

*nazwa\_zdarzenia*

nazwa zdarzenia (wymagane).

Przykład:

```
public delegate void Metoda();
private event Metoda zdarzenie;
```

Zdarzenia są dość złożonym elementem języka C#. Aby łatwiej móc zrozumieć zasadę ich działania, spróbujemy przeanalizować prosty program, który najlepiej odwzoruje model wydawcy i czytelnika:

```
using System;

class Delegacje
{
    // delegacja potrzebna do deklaracji zdarzenia
    public delegate void MetodaObslugi(int Parametr);

    public class Wydawca
    {
        // zdarzenie ukazania się książki "Piotruś pan"
        public event MetodaObslugi wydanie_ksiazki_Piotrus_Pan;
        // zdarzenie ukazania się książki "Czerwony kapturek"
        public event MetodaObslugi wydanie_ksiazki_Czerwony_Kapturek;

        // metoda wysłania informacji o ukazaniu się książek i ich cenie
        public void WyslijInformacje()
        {
            // jeżeli ktoś jest zainteresowany zdarzeniem ukazania się książki "Piotruś pan"
            // dostanie informację na ten temat
            if (wydanie_ksiazki_Piotrus_Pan != null)
                wydanie_ksiazki_Piotrus_Pan(120);

            // jeżeli ktoś jest zainteresowany zdarzeniem ukazania się książki
            // "Czerwony kapturek" dostanie informację na ten temat
            if (wydanie_ksiazki_Czerwony_Kapturek != null)
                wydanie_ksiazki_Czerwony_Kapturek(12);
        }
    }

    public class Czytelnik
    {
        // metoda określająca reakcję czytelnika na ukazanie się książki
        public void Kupuje(int Cena)
        {
            if (Cena <= 100)
                Console.WriteLine("Kupuję");
            else
                Console.WriteLine("Proszę o rabat");
        }
    }

    static void Main(string[] args)
    {
        Wydawca wyd = new Wydawca();
        Czytelnik czyt = new Czytelnik();
        // czytelnik zgłasza chęć kupna książki "Piotruś pan" kiedy tylko się ukaże
        // pod warunkiem, że jej cena nie będzie większa od 100, jeżeli będzie,
```

```
// poprosi o rabat
wyd.wydanie_ksiazki_Piotrus_Pan += new MetodaObslugi(czyt.Kupuje);
// wydawca wydaje obie książki i wysyła informacje o ich ukazaniu się, co spowoduje
// reakcję czytelnika (czytelnik zareaguje jedynie na informacje o ukazaniu się
// książki "Piotruś pan", gdyż nie zgłaszał chęci kupna innej książki)
wyd.WyslijInformacje();
}
}
```

W przykładowym programie rozgrywa się scenariusz, w którym wydawca przygotowuje się do wydania dwóch książek „Piotruś pan” i „Czerwony kapturek” (mamy dwa zdarzenia odpowiadające wydaniu każdej z książek i powiązaną z nimi delegację). Mamy również czytelnika, który zadeklarował się, że kupi książkę „Piotruś pan” zaraz po jej wydaniu, pod warunkiem, że jej cena nie przekroczy 100, a jeżeli przekroczy poprosi o rabat (czytelnik ma metodę Kupuje z parametrem cena, która ma składnię zgodną ze składnią delegacji, metoda ta podpisana jest pod zdarzenie odpowiadające wydaniu książki „Piotruś pan”). W momencie, w którym następuje wydanie książek, wydawca wysyła informacje do zainteresowanego czytelnika podając ceny książek (wywołanie metody WyslijInformacje, spowoduje wywołanie odpowiedniej metody zgłoszonej przez czytelnika, jako reakcję na zdarzenie). Czytelnik podejmuje decyzję: prosi o rabat, gdyż cena jest wyższa od 100 (wywołanie metody Kupuje przez metodę WyslijInformacje).

## Dziedziczenie

Dziedziczenie jest jednym z podstawowych mechanizmów obiektowości, który pozwala obiektowi przejąć (odziedziczyć) cechy (dane oraz funkcjonalność) od innego obiektu. Obiekt dziedziczący nazywa się obiektem potomnym lub obiektem dziedziczącym, natomiast obiekt, z którego się dziedziczy nazywa się obiektem rodzica lub obiektem bazowym.

Dzięki mechanizmowi dziedziczenia można tworzyć nowe klasy w oparciu o istniejące, bez potrzeby ponownego pisania tej samej funkcjonalności. Klasa dziedzicząca stanowi w takim wypadku rozszerzenie klasy bazowej.

W języku C# istnieje możliwość dziedziczenia tylko z jednej klasy bazowej (oraz wielu interfejsów, o których będziemy mówić później). Aby wskazać, że nowa klasa dziedziczy z jakiejś klasy bazowej, w definicji nowej klasy należy podać nazwę klasy bazowej po znaku dwukropka (patrz definiowanie klas):

```
class NowaKlasa : KlasaBazowa
{
    ...
}
```

Klasa potomna nie może być bardziej dostępna od klasy rodzica (nie może mieć mniej restrykcyjnego modyfikatora dostępu):

```
class Przykład
{
    public class Bazowa1
```

```
{
    ...
}

private class Bazowa2
{
    ...
}

protected class Dziedziczaca1 : Bazowa1
{
    ...
}

private class Dziedziczaca2 : Bazowa1
{
    ...
}

// BŁĄD: może być tylko private
public class Dziedziczaca3 : Bazowa2
{
    ...
}

// BŁĄD: może być tylko private
protected class Dziedziczaca4 : Bazowa2
{
    ...
}
}
```

## Dostęp do składowych klasy bazowej

Klasa potomna dziedziczy z klasy rodzica wszystkie elementy poza konstruktorami i destruktorami.

Oznacza to, że:

- składowe publiczne (**public**) klasy rodzica stają się składowymi publicznymi klasy potomnej:

```
class Bazowa
{
    public int liczba;
}

class Dziedziczaca : Bazowa
{

```

```
public int Liczba()
{
    return liczba;    // OK: dostęp do składowej publicznej
}
}
```

```
class InnaNieDziedziczaca
{
    public int Liczba(Bazowa baz)
    {
        return baz.liczba; // OK: dostęp do składowej publicznej
    }
}
```

- składowe chronione (*protected*) klasy rodzica stają się dostępne dla klasy potomnej:

```
class Bazowa
```

```
{
    protected int liczba;
}
```

```
class Dziedziczaca : Bazowa
```

```
{
    public int Liczba()
    {
        return liczba;    // OK: dostęp do składowej chronionej
    }
}
```

```
class InnaNieDziedziczaca
```

```
{
    public int Liczba(Bazowa baz)
    {
        return baz.liczba; // BŁĄD: klasa nie dziedziczająca nie ma prawa
    }
}
```

- składowe prywatne (*private*) klasy rodzica są dostępne tylko w klasie rodzica (można uzyskać do nich dostęp pośrednio przez publiczne lub chronione składniki klasy rodzica, operujące na jego prywatnych składowych np.: metody czy właściwości):

```
class Bazowa
```

```
{
    private int liczba;

    public int WartoscLiczby()
    {
```



```
        return liczba;
    }
}

class Dziedziczaca : Bazowa
{
    public int Liczba()
    {
        return liczba;    // BŁĄD: nie ma dostępu do prywatnej składowej
                           // można użyć: return WartoscLiczby();
    }
}

class InnaNieDziedziczaca
{
    public int Liczba(Bazowa baz)
    {
        return baz.liczba; // BŁĄD: nie ma dostępu do prywatnej składowej
                           // można użyć: return baz.WartoscLiczby();
    }
}
```

## Wywoływanie bazowych konstruktorów

Domyślnie w momencie wywołania konstruktora klasy potomnej następuje odwołanie do bezargumentowego konstruktora klasy bazowej. Można to jednak zmienić wskazując konkretny konstruktor klasy bazowej za pomocą słowa kluczowego **base**, umieszczonego w liście inicjującej konstruktora klasy potomnej:

```
class Bazowa
{
    public Bazowa()
    {
        ...
    }

    protected Bazowa(int Arg)
    {
        ...
    }
}

class Dziedziczaca : Bazowa
{
    public Dziedziczaca() // domyślnie :base()
    {
        ...
    }
}
```

```
{  
    ...  
}  
  
public Dziedziczaca(int Arg) : base(Arg)  
{  
    ...  
}
```

Odwołując się do konstruktora klasy bazowej, należy mieć na względzie jego dostępność. Jeżeli konstruktor w klasie bazowej będzie konstruktorem prywatnym, jego wywołanie nie powiedzie się. To samo dotyczy domyślnego odwołania, jeżeli sami definiujemy konstruktor bezargumentowy:

```
class Bazowa  
{  
    private Bazowa()  
    {  
        ...  
    }  
  
    private Bazowa(int Arg)  
    {  
        ...  
    }  
}  
  
class Dziedziczaca : Bazowa  
{  
    // BŁĄD: base() nie dostępna  
    public Dziedziczaca()  
    {  
        ...  
    }  
  
    // BŁĄD: base(Arg) nie dostępna  
    public Dziedziczaca(int Arg) : base(Arg)  
    {  
        ...  
    }  
}
```

Należy pamiętać również o tym, że w przypadku, gdy dodamy jakiegokolwiek konstruktor, domyślny bezargumentowy konstruktor publiczny nie zostanie utworzony:

```
class Bazowa
```

```
{
    public Bazowa(int Arg)
    {
        ...
    }
}

class Dziedziczaca : Bazowa
{
    // BŁĄD: nie ma publicznego konstruktora bezargumentowego
    public Dziedziczaca()
    {
        ...
    }
}
```

Słowo kluczowe **base** można również wykorzystać do odwołania się do składowych klasy bazowej (np.: gdy nazwy argumentów pokrywają się z nazwami pól klasy bazowej):

```
class Bazowa
{
    protected int liczba;
}

class Dziedziczaca : Bazowa
{
    public void Przelicz(int liczba)
    {
        base.liczba = liczba;
    }
}
```

## Przesłanianie metod

Przesłanianie metod zwane również polimorfizmem, jest mechanizmem, który pozwala na zmianę definicji metody z klasy bazowej w klasie pochodnej w taki sposób, iż w czasie wywołania nastąpi odwołanie do metody z właściwej klasy. Oznacza to, że w przypadku, w którym nastąpi rzutowanie z klasy pochodnej na klasę rodzica, nastąpi wywołanie metody przesłaniającej (znajdującej się w klasie pochodnej), a nie metody przesłanianej (znajdującej się w klasie rodzica).

Do definiowania metod, które mogą być przesłaniane polimorficznie w klasach dziedziczących, służy modyfikator **virtual**. Metody te określa się mianem metod wirtualnych.

Przykład:

```
class Bazowa
{
    public virtual void MetodaPrzeslaniana()
```

```
{  
    ...  
}  
}
```

W czasie definiowania metod wirtualnych należy pamiętać o tym, że:

- muszą zawierać ciało (nawet gdy niczego nie implementujemy, należy umieścić pusty blok instrukcji):

```
public virtual MetodaPrzeslaniana();    // BŁĄD  
public virtual MetodaPrzeslaniana()  
{  
}  
}
```

- nie mogą być definiowane jako statyczne (polimorfizm dotyczy obiektu nie klasy):

```
public static virtual MetodaPrzeslaniana()    // BŁĄD  
{  
    ...  
}  
}
```

- nie mogą być definiowane jako prywatne (nie będzie możliwości ich przesłonięcia w klasie potomnej):

```
private virtual MetodaPrzeslaniana()    // BŁĄD  
{  
    ...  
}  
}
```

W klasie dziedziczącej można przesłonić metodę wirtualną używając modyfikatora **override**. Metody te określa się mianem metody przesłaniającej lub kolejnej metody wirtualnej. Pojęcie kolejnej metody wirtualnej oznacza, że metoda przesłaniająca automatycznie staje się metodą wirtualną w klasie potomnej i może być przesłonięta w innej klasie, która będzie z niej dziedziczyć.

Przykład:

```
class Bazowa  
{  
    // metoda wirtualna  
    public virtual void MetodaPrzeslaniana()  
    {  
        ...  
    }  
}  
  
class Dziedziczaca : Bazowa  
{  
    // metoda przesłaniająca metodę MetodaPrzeslaniana z klasy Bazowa  
    public override void MetodaPrzeslaniana()  
    {  
        ...  
    }  
}
```

```
    }  
}  
  
class KolejnaDziedziczaca : Dziedziczaca  
{  
    // metoda przesłaniająca metodę MetodaPrzeslaniana z klasy Dziedziczaca  
    public override void MetodaPrzeslaniana()  
    {  
        ...  
    }  
}
```

W czasie definiowania metod przesłaniających, należy pamiętać o tym, że:

- muszą zawierać ciało (nawet gdy niczego nie implementujemy, należy umieścić pusty blok instrukcji):

```
public override MetodaPrzeslaniana();    // BŁĄD  
public override MetodaPrzeslaniana()  
{  
}  
}
```

- muszą mieć składnię identyczną ze składnią metod wirtualnych z klasy bazowej:

```
class Bazowa  
{  
    protected virtual int Ile()  
    {  
        ...  
    }  
}  
  
class Dziedziczaca : Bazowa  
{  
    public override int Ile(int ile)    // BŁĄD  
    {  
        ...  
    }  
}
```

- nie można używać modyfikatora **virtual**, ponieważ modyfikator **override** oznacza, że metoda jest kolejną metodą wirtualną:

```
class Bazowa  
{  
    protected virtual int Ile()  
    {  
        ...  
    }  
}
```

```
class Dziedziczaca : Bazowa
{
    protected virtual override int Ile() // BŁĄD
    {
        ...
    }
}
```

- nie mogą być definiowane jako statyczne (polimorfizm dotyczy obiektu nie klasy):

```
public static override MetodaPrzeslaniana() // BŁĄD
{
    ...
}
```

- nie mogą być definiowane jako prywatne (nie będzie możliwości ich przesłonięcia w klasie potomnej):

```
private override MetodaPrzeslaniana() // BŁĄD
{
    ...
}
```

Jak już wspomniałem polimorfizm daje nam pewność, że wywołana metoda pochodzi z właściwej klasy. Jeżeli dokonamy rzutowania z klasy pochodnej na klasę rodzica i wywołamy metodę wirtualną, zostanie wywołana metoda z klasy pochodnej a nie z klasy rodzica:

```
((KlasaBazowa) InstancjaKlasyPochodnej).MetodaPrzelaniana();
```

Przyjrzyjmy się przykładowemu programowi:

```
using System;

class Polimorfizm
{
    class Bazowa
    {
        public virtual void MetodaPrzeslaniana()
        {
            Console.WriteLine("Bazowa");
        }
    }

    class Dziedziczaca : Bazowa
    {
        public override void MetodaPrzeslaniana()
        {
            Console.WriteLine("Dziedziczaca");
        }
    }
}
```

```
static void Main()
{
    Dziedziczaca dzie = new Dziedziczaca();
    Bazowa baz = new Bazowa();

    baz.MetodaPrzeslaniana(); // wywołanie metody z klasy bazowej
    dzie.MetodaPrzeslaniana(); // wywołanie metody z klasy dziedziczacej
    baz = (Bazowa)dzie;        // rzutowanie klasy dziedziczacej na bazowa
    baz.MetodaPrzeslaniana(); // wywołanie metody z klasy dziedziczacej
}
}
```

Wynik:

```
Bazowa
Dziedziczaca
Dziedziczaca
```

W powyższym programie wykorzystaliśmy mechanizm polimorfizmu. W momencie, w którym wywołaliśmy metodę wirtualną `MetodaPrzeslaniana` dla instancji klasy bazowej nastąpiło odwołanie do metody z tej klasy. Kiedy wywołaliśmy metodę wirtualną `MetodaPrzeslaniana` dla instancji klasy dziedziczacej, nastąpiło odwołanie do metody z tej klasy. Natomiast kiedy wykonaliśmy operację rzutowania instancji klasy dziedziczacej na klasę bazową i wywołaliśmy metodę wirtualną `MetodaPrzeslaniana` dla instancji klasy bazowej nastąpiło odwołanie do metody z klasy dziedziczacej (nie bazowej).

## Ukrywanie metod

Metody dziedziczone oprócz tego, że mogą być przysłaniane, mogą również być ukrywane. Ukrycie metody oznacza zastąpienie metody odziedziczonej z klasy bazowej zupełnie inną metodą w klasie dziedziczacej. Do ukrywania metod służy słowo kluczowe ***new***.

Przykład:

```
class Bazowa
{
    public void Metoda()
    {
    }
}

class Dziedziczaca : Bazowa
{
    new public void Metoda()
    {
    }
}
```

Jeżeli w klasie potomnej definiujemy metodę o takiej samej nazwie co w klasie bazowej, powinniśmy

użyć słowa kluczowego **new**, aby uniknąć ostrzeżenia o pokrywaniu się nazw:

```
class Bazowa
{
    public void Metoda()
    {
    }
}

class Dziedziczaca : Bazowa
{
    // brakuje new, otrzymamy ostrzeżenie podczas kompilacji
    public void Metoda()
    {
    }
}
```

Ukrycie metody pozwala również na zmianę tych elementów sygnatury metody, które nie są brane pod uwagę w czasie sprawdzania ich identyczności (jak pamiętamy przy porównaniu bierze się pod uwagę nazwę metody i typ argumentów):

```
class Bazowa
{
    public void Metoda()
    {
    }
}

class Dziedziczaca : Bazowa
{
    // typ zwracanej wartości nie jest brany pod uwagę przy porównaniu więc
    // metoda Metoda jest uznawana za identyczną, new jest wymagane
    new public string Metoda()
    {
    }

    new private void Metoda()
    {
    }

    // ostrzeżenie: new nie jest potrzebne, gdyż w klasie bazowej nie ma
    // identycznej metody
    new public void Metoda(int i)
    {
    }
}
```



```
}
```

Mechanizm ukrywania może być używany zarówno w przypadku wirtualnych jak i niewirtualnych metod. Stosowanie mechanizmu ukrywania w przypadku metod wirtualnych niesie za sobą pewne komplikacje w stosunku do mechanizmu polimorfizmu. Ukrycie metody wirtualnej z klasy bazowej w klasie dziedziczącej spowoduje, że nie będzie ona traktowana jako kolejna metoda wirtualna tylko jako pierwsza metoda wirtualna w łańcuchu dziedziczenia:

```
class Bazowa
{
    public virtual void Metoda()
    {
    }
}

class Dziedziczaca : Bazowa
{
    // ukrywamy metodę z klasy Bazowa (w łańcuchu dziedziczenia jest teraz
    // pierwszą funkcją wirtualną
    new public virtual void Metoda()
    {
    }
}

class KolejnaDziedziczaca : Dziedziczaca
{
    // przesłaniamy metodę z klasy Dziedziczaca
    public override void Metoda()
    {
    }
}
```

Jeżeli teraz dokonamy rzutowania ostatniej klasy KolejnaDziedziczaca na klasę Bazowa i wywołamy wirtualną metodę Metoda, wywołana zostanie metoda z klasy Bazowa, gdyż w klasie Dziedziczaca została zdefiniowana na nowo metoda wirtualna i to ona jest pierwszą metodą wirtualną w całym łańcuchu dziedziczenia.

Przyjrzyjmy się przykładowemu programowi:

```
using System;

class Polimorfizm_I_Ukrywanie
{
    class Bazowa
    {
        public virtual void Metoda()
        {
            Console.WriteLine("Bazowa");
        }
    }
}
```

```
    }  
}  
  
class Dziedziczaca : Bazowa  
{  
    new public virtual void Metoda()  
    {  
        Console.WriteLine("Dziedziczaca");  
    }  
}  
  
class KolejnaDziedziczaca : Dziedziczaca  
{  
    public override void Metoda()  
    {  
        Console.WriteLine("KolejnaDziedziczaca");  
    }  
}  
  
static void Main()  
{  
    KolejnaDziedziczaca kdzie = new KolejnaDziedziczaca();  
    Dziedziczaca dzie = new Dziedziczaca();  
    Bazowa baz = new Bazowa();  
  
    baz.Metoda();           // wywołanie metody z klasy bazowej  
    dzie.Metoda();          // wywołanie metody z klasy dziedziczącej  
    kdzie.Metoda();         // wywołanie metody z kolejnej klasy dziedziczącej  
    baz = (Bazowa)kdzie;  
    baz.Metoda();           // wywołanie metody z klasy bazowej (polimorfizm nie działa)  
    dzie = (Dziedziczaca)kdzie;  
    dzie.Metoda();          // wywołanie metody z kolejnej klasy dziedziczącej (polimorfizm)  
}  
}
```

Wynik:

```
Bazowa  
Dziedziczaca  
KolejnaDziedziczaca  
Bazowa  
KolejnaDziedziczaca
```

## Klasy ostateczne

Czasami istnieje potrzeba zdefiniowania klasy ostatecznej (nierozszerzalnej), czyli takiej, z której nie można już dziedziczyć. Służy do tego modyfikator *sealed*, który umieszcza się przed słowem *class*:

```
public sealed class Ostateczna
```

```
{  
}  
  
// BŁĄD: nie można odziedziczyć  
public class JakasPotomna : Ostateczna  
{  
}
```

W .NET Framework mamy do czynienia z wieloma klasami ostatecznymi np. *System.String*.

Modyfikator *sealed* może również być wykorzystany w mechanizmie przysłaniania do określenia, iż kolejna metoda wirtualna jest ostateczną metodą, której nie można już przysłaniać:

```
class Bazowa  
{  
    // metoda wirtualna  
    public virtual void MetodaPrzeslaniana()  
    {  
    }  
}  
  
class Dziedziczaca : Bazowa  
{  
    // metoda przesłaniająca metodę MetodaPrzeslaniana z klasy Bazowa  
    public sealed override void MetodaPrzeslaniana()  
    {  
    }  
}  
  
class KolejnaDziedziczaca : Dziedziczaca  
{  
    // BŁĄD: nie można przesłonić ostatecznej metody wirtualnej  
    public override void MetodaPrzeslaniana()  
    {  
    }  
}
```

## Klasy abstrakcyjne

Klasy abstrakcyjne używane są do definiowania elementów, które będą implementowane w klasach potomnych. Do zdefiniowania klasy abstrakcyjnej służy modyfikator *abstract*, który umieszcza się przed słowem kluczowym *class*:

```
abstract class KlasaAbstrakcyjna  
{  
}
```

Klasy abstrakcyjne różnią się od innych klas tym, że:

- nie można utworzyć instancji klasy abstrakcyjnej:

```
KlasaAbstrakcyjna kabs = new KlasaAbstrakcyjna(); // BŁĄD
```

- można utworzyć metodę abstrakcyjną w klasie abstrakcyjnej:

```
abstract class KlasaAbstrakcyjna
{
    public abstract void Metoda();
}

class KlasaNieAbstrakcyjna
{
    // BŁĄD: metoda nie może być abstrakcyjna w klasie nie abstrakcyjnej
    public abstract void Metoda();
}
```

Klasy abstrakcyjne stosuje się do definiowania grupy cech wspólnych dla klas potomnych.

Przykład:

```
abstract class Przedmiot
{
    protected float waga; // waga jest cechą wspólną wszystkich przedmiotów
}

class Kubek : Przedmiot
{
    public Kubek()
    {
        base.waga = 0.25f;
    }
}
```

W każdej klasie abstrakcyjnej można zdefiniować metodę abstrakcyjną. Metoda abstrakcyjna jest metodą, która musi być zaimplementowana w klasie potomnej. Definiuje się je umieszczając modyfikator **abstract** przed typem zwracanej wartości:

```
public abstract void Metoda();
```

Implementując metody abstrakcyjne należy pamiętać, że:

- metody abstrakcyjne są metodami wirtualnymi, z tym jednak wyjątkiem, że w klasie abstrakcyjnej nie mogą być implementowane (implementuje się je w klasie potomnej):

```
abstract class KlasaAbstrakcyjna
{
    // BŁĄD: nie może mieć ciała
    public abstract void Metoda()
    {
        ...
    }
}
```

- do implementacji metod abstrakcyjnych w klasie potomnej wykorzystuje się modyfikator

***override***:

```
class KlasaDziedziczaca : KlasaAbstrakcyjna
{
    public override void Metoda()
    {
        ...
    }
}
```

- metody abstrakcyjne mogą przysłaniać metody wirtualne znajdujące się w klasie bazowej (zastosowanie tego mechanizmu spowoduje, że oryginalna implementacja metody wirtualnej z klasy bazowej stanie się niedostępna i będzie musiała być zaimplementowana w kolejnej klasie potomnej):

```
class Bazowa
{
    public virtual void Metoda()
    {
        ...
    }
}

abstract class DziedziczacaAbstrakcyjna : Bazowa
{
    public abstract override void Metoda();
}
```

- metody abstrakcyjne mogą przysłaniać kolejne metody wirtualne znajdujące się w klasach potomnych:

```
class Bazowa
{
    public virtual void Metoda()
    {
        ...
    }
}

class Dziedziczaca
{
    public override void Metoda()
    {
        ...
    }
}
```

```
abstract class DziedziczacaAbstrakcyjna : Dziedziczaca
{
    public abstract override void Metoda();
}
```

## Bazowa klasa System.Object

Jeżeli w czasie definiowania nowej klasy, nie określimy żadnej klasy bazowej, domyślnie zostanie przyjęte, że nowa klasa dziedziczy z **System.Object** (która jest klasą bazową dla wszystkich innych klas).

Klasa **System.Object** składa się z:

- publicznego bezparametrowego konstruktora **Object** tworzącego nową instancję klasy **Object**:

```
public Object();
```

- metod publicznych:

- **Equals** – określającej czy dwa obiekty są równe:

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

```
public static bool Equals(object, object);
```

- **GetHashCode** – dostarczającej mechanizm haszowania charakterystyczny dla danego obiektu, który używany jest w algorytmach haszujących lub strukturach danych takich jak tablice haszujące:

```
public virtual int GetHashCode();
```

- **GetType** – zwracającej typ aktualnej instancji klasy:

```
public Type GetType();
```

- **ToString** – zwracającej napis reprezentujący aktualną instancję klasy:

```
public virtual string ToString();
```

- **ReferenceEquals** – określającej czy instancje klasy są sobie równe:

```
public static bool ReferenceEquals(object, object);
```

- metod chronionych:

- **Finalize** – pozwalającej obiektowi na zwalnianie zasobów i wykonywanie czynności

deinicjujących przed zwolnieniem pamięci zajmowanej przez obiekt, mechanizm dostępny za pomocą destruktora:

```
~Object();
```

- **MemberwiseClone** – tworzącej kopię aktualnego obiektu:

```
protected object MemberwiseClone();
```

## Rozdział 8.

# Struktury

### Definiowanie struktur

Struktura pozwala na zdefiniowanie typu danych, który nie charakteryzuje się zbyt złożoną funkcjonalnością (np. punkt, kolor, etc.). Do definiowania struktury służy słowo kluczowe **struct**.

Składnia definicji struktury wygląda następująco:

```
[modyfikator] struct Identyfikator [: lista_interfejsów]
{
    składowe_struktury
}
```

gdzie:

*modyfikator*

Dozwolony jest modyfikator **new** oraz modyfikatory dostępu (opcjonalne);

*Identyfikator*

Nazwa struktury (wymagane);

*lista\_interfejsów*

Lista zawierająca implementowane interfejsy oddzielone przecinkami (opcjonalne);

*składowe\_struktury*

Składowe struktury.

Przykład:

```
struct Punkt
{
    public int x, y;

    public Punkt(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

### Porównanie z klasami

Struktura wykazuje duże podobieństwo do klasy, poza pewnymi elementami:

- struktura jest typem wartości, a nie jak klasa typem referencyjnym:

Strukturę można deklarować jak zmienną typu wartości:

```
Punkt pkt;

pkt.x = 10;
```

```
pkt.y = 100;
```

W takim wypadku jednak pól składowych nie można używać do momentu, aż nie zostaną poprawnie zainicjowane:

```
Punkt pkt;
```

```
if (pkt.x == 0) ... // BŁĄD: użycie niezainicjowanego pola
```

Można również utworzyć instancję struktury:

```
Punkt pkt = new Punkt(10, 20);
```

Jeżeli do inicjacji wykorzystamy konstruktor domyślny, (który jest automatycznie tworzony i nie może być nadpisany) pola zostaną zainicjowane wartościami domyślnymi:

```
Punkt pkt = new Punkt();
```

```
if (pkt.x == 0) ... // OK: zmienne zainicjowane wartościami domyślnymi
```

- struktura nie wspiera mechanizmu dziedziczenia (wspiera mechanizm implementowania interfejsów):

```
class BazowaK
```

```
{  
}
```

```
struct BazowaS
```

```
{  
}
```

```
struct DziedziczacaK : BazowaK // BŁĄD
```

```
{  
}
```

```
struct DziedziczacaS : BazowaS // BŁĄD
```

```
{  
}
```

- operacje na strukturze charakteryzują się większą wydajnością niż operacje na klasach (dane przechowywane są bezpośrednio);

- w strukturze nie można zdefiniować destruktora:

```
struct Punkt
```

```
{  
    ~Punkt() // BŁĄD
```

```
    {  
    }  
}
```

- w strukturze nie można zdefiniować konstruktora bezparametrowego:

```
struct Punkt
```

```
{  
    public Punkt() // BŁĄD
```



```
{  
}  
}
```

- wewnątrz struktury nie można inicjować pól składowych:

```
struct Punkt  
{  
    public int x = 10, y = 200; // BŁĄD  
}
```

## Grupowanie pól

Strukturę często wykorzystuje się również jako typ grupujący inne pola (np. dane dotyczące osoby):

```
struct Pracownik  
{  
    public string Imie;  
    public string Nazwisko;  
    public int Wiek;  
}
```

Mimo, że struktury mogą zawierać metody, nie zaleca się ich używania (struktury powinny przechowywać dane). Inaczej ma się sprawa definiowania operatorów, gdyż nie wprowadzają one nowych wzorców zachowania, a jedynie dostarczają funkcjonalność dla istniejących (np.: dodawanie).

## Rozdział 9.

# Interfejsy

### Definiowanie interfejsów

Interfejs stanowi rodzaj kontraktu lub też specyfikacji funkcjonalności, jaka ma zostać zaimplementowana przez klasę (lub strukturę) go implementującą. W interfejsie mogą znaleźć się jedynie sygnatury metod. Do definiowania interfejsów używa się słowa kluczowego *interface*.

Składnia definicji interfejsu wygląda następująco:

```
[modyfikatory] interface Identyfikator [: lista_bazowa]
{
    ciało_interfejsu
}
```

gdzie:

*modyfikatory*

Dozwolony jest modyfikator *new* oraz modyfikatory dostępu (opcjonalne),

*Identyfikator*

Nazwa interfejsu, zaleca się poprzedzać nazwę interfejsu dużą literą 'I' (wymagane),

*lista\_bazowa*

Lista zawierająca interfejsy bazowe oddzielone przecinkami (opcjonalne),

*ciało\_interfejsu*

Sygnatury metod (opcjonalne).

Przykład:

```
interface ISlownik
{
    string Tlumacz(string Wyrac);
}
```

Interfejsy cechują dwie ważne cechy:

- domyślnie wszystkie sygnatury metod są publiczne, a jawne podanie modyfikatora dostępu nie jest dozwolone:

```
interface ISlownik
{
    public string Tlumacz(string Wyrac); // BŁĄD
}
```

- w interfejsie można umieścić jedynie sygnatury metod (nie można ich implementować):

```
interface ISlownik
{
    string Tlumacz(string Wyrac) // BŁĄD
    {
        ...
    }
}
```

```
}  
}
```

## Implementacja interfejsów

Mimo, że w C# dozwolone jest dziedziczenie tylko z jednej klasy, możliwe jest implementowanie wielu interfejsów w pojedynczej klasie:

```
interface IBazowy_1  
{  
    ...  
}  
  
interface IBazowy_2  
{  
    ...  
}  
  
class Implementujaca : IBazowy_1, IBazowy_2  
{  
    ...  
}
```

Każdy interfejs może być interfejsem rozszerzającym dla wielu innych interfejsów. Taki interfejs łączy kontrakty interfejsów bazowych i pozwala na ich rozszerzenie:

```
interface IBazowy_1  
{  
    void Metoda1();  
}  
  
interface IBazowy_2  
{  
    void Metoda2();  
}  
  
interface IRozszerzajacy: IBazowy_1, IBazowy_2 // wspólny kontrakt  
{  
}  
}
```

Żaden interfejs rozszerzający nie może być bardziej dostępny od interfejsu bazowego:

```
private interface IBazowy  
{  
    ...  
}
```

```
public interface IRozszerzajacy : IBazowy // BŁĄD
{
    ...
}
```

Inaczej jest jednak w przypadku klasy implementującej interfejs, która może być bardziej dostępna od bazowego interfejsu:

```
private interface IBazowy
{
    ...
}

public class Implementacja : IBazowy
{
    ...
}
```

## Implementacja metod interfejsów

Każda klasa implementująca dany interfejs musi zachować zgodność z kontraktem zawartym w interfejsie. Oznacza to, że wszystkie metody z implementowanych interfejsów muszą mieć zgodną sygnaturę z określoną w kontrakcie.

Sygnatura obejmuje:

- modyfikator dostępu (który ze względu na charakterystykę interfejsu musi być **public**):

```
interface IBazowy
{
    void Metoda();
}

class Implementacja : IBazowy
{
    // BŁĄD: musi być public
    protected void Metoda()
    {
        ...
    }
}
```

- typ zwracanej wartości:

```
interface IBazowy
{
    int Metoda();
}

class Implementacja : IBazowy // BŁĄD: brak implementacji metody
```

```
{
    string Metoda()
    {
        ...
    }
}
```

- nazwę metody:

```
interface IBazowy
{
    void Metoda();
}
```

*class Implementacja : IBazowy // BŁĄD: brak implementacji metody*

```
{
    void InnaMetoda()
    {
        ...
    }
}
```

- listę argumentów:

```
interface IBazowy
{
    void Metoda(int Liczba, string Nazwa);
}
```

*class Implementacja : IBazowy // BŁĄD: brak implementacji metody*

```
{
    public void Metoda(string Nazwa, int Liczba)
    {
        ...
    }
}
```

Implementowana metoda może jednak zostać określona jako wirtualna, (choć w sygnaturze metody zdefiniowanej w interfejsie nie jest dozwolone użycie słowa kluczowego *virtual*):

```
interface IBazowy_1
{
    void Metoda1(int Liczba);
}
```

```
interface IBazowy_2
{
    string Metoda2();
}
```

```
class Implementacja : IBazowy_1, IBazowy_2
{
    public virtual void Metoda1(int Liczba)
    {
        ...
    }

    public string Metoda2()
    {
        ...
    }
}
```

### Jawna implementacja metod interfejsów

Alternatywnym sposobem implementacji metod interfejsów przez klasę jest ich tzw.: „jawna implementacja”, czyli wskazanie, jaką metodę z jakiego interfejsu implementujemy. W takim wypadku nie możemy użyć modyfikatora dostępu. Korzystanie z jawnej implementacji metod ma zastosowanie przede wszystkim w przypadku, gdy w interfejsach bazowych znajdują się metody o takiej samej nazwie:

```
interface IBazowy_1
{
    void Metoda();
}

interface IBazowy_2
{
    void Metoda();
}

class Implementacja : IBazowy_1, IBazowy_2
{
    void IBazowy_1.Metoda()
    {
        ...
    }

    void IBazowy_2.Metoda()
    {
        ...
    }
}
```

Jawna implementacja metod niesie za sobą pewne ograniczenia:

- wywołanie implementacji metody możliwe jest jedynie poprzez odwołanie za pomocą interfejsu:

```
interface IBazowy_1
{
    void Metoda1();
}

interface IBazowy_2
{
    void Metoda1();
}

class Implementacja : IBazowy_1, IBazowy_2
{
    void IBazowy_1.Metoda1()
    {
        ...
    }

    void Metoda2()
    {
        Metoda1(); // BŁĄD
        ((IBazowy_1)this).Metoda1();
    }
}
```

- metody nie mogą być implementowane jako wirtualne:

```
interface IBazowy
{
    void Metoda();
}

class Implementacja : IBazowy
{
    public virtual void IBazowy.Metoda() // BŁĄD
    {
        ...
    }
}
```

- nie można określić modyfikatora dostępu:

```
interface IBazowy
{
    void Metoda();
}
```

```
class Implementacja : IBazowy
{
    public void IBazowy.Metoda() // BŁĄD
    {
        ...
    }
}
```

- nie ma możliwości bezpośredniego odwołania się do implementowanej metody:

```
class Przykład
{
    interface IBazowy
    {
        void Metoda();
    }

    class Implementacja : IBazowy
    {
        void IBazowy.Metoda()
        {
            ...
        }
    }

    void Test(Implementacja imp)
    {
        imp.Metoda(); // BŁĄD
    }
}
```

- aby odwołać się do metody, należy odwołać się przez właściwy interfejs:

```
class Przykład
{
    interface IBazowy
    {
        void Metoda();
    }

    class Implementacja : IBazowy
    {
        void IBazowy.Metoda()
        {
            ...
        }
    }
}
```



```
    }

    void Test(Implementacja imp)
    {
        ((IBazowy)imp).Metoda();
    }
}
```

## Interfejs IDisposable

Pamięć jest zasobem, który jest współdzielony przez wszystkie programy. W przypadkach gdy wielkość wolnej pamięci spada poniżej pożądanej wielkości, możliwe jest wymuszenie odzyskania nieużywanej pamięci przez „kolekcjonera nieużytków”. Pamięć jednakże nie jest jedynym zasobem współdzielonym. Program może używać również uchwytów do plików oraz blokad, które ze względu na ograniczenia ilościowe mogą być trudniejsze do uzyskania niż pamięć.

W przypadku, gdy do zwalniania zasobów korzystamy z destruktorów, operacje zwolnienia mogą nastąpić w niekontrolowanym czasowo momencie, dlatego pozostawiono pewną furtkę, z której można skorzystać, aby wymusić natychmiastową akcję zwolnienia. Służy do tego celu metoda ***Dispose***, która znajduje się w interfejsie ***IDisposable***.

Aby móc skorzystać z tego mechanizmu należy:

- odziedziczyć z interfejsu ***IDisposable***,
- zaimplementować metodę ***Dispose***,
- upewnić się, że metoda ***Dispose*** może być wywoływana wiele razy,
- wywołać metodę ***SuppressFinalize***.

Przykład:

```
class Zasob : IDisposable
{
    private Component komponenty;
    private IntPtr uchwyt;
    private bool zwolniony = false;

    public Zasob()
    {
        // uchwyt = alokacja_niezarzadzalnego_elementu
        komponenty = new Component(...);
    }

    // niech destruktor zwalnia zasoby standardowo bez wymuszenia
    ~Zasob()
    {
        Dispose(false);
    }
}
```

```
// implementacja metody interfejsu IDisposable
public void Dispose()
{
    Dispose(true);
    // wymuś zwolnienie zasobów na "kolekcjonerze nieużytków"
    GC.SuppressFinalize(this);
}

// w zależności od parametru zwolnij nastąpi wymuszenie zwolnienia zasobów
// lub zwolnienie zasobów nastąpi w sposób standardowy, metoda może być
// wołana wielokrotnie (po pierwszym udanym wywołaniu znacznik zwolniony
// ustawiony zostanie na true)
protected virtual void Dispose(bool zwolnij)
{
    // jeżeli jeszcze nie zwolniono zasobów wykonaj instrukcje
    if (!this.zwolniony)
    {
        // jeżeli ma nastąpić wymuszone zwolnienie wywołaj Dispose dla komponentów
        if (zwolnij)
        {
            komponenty.Dispose();
        }

        // zwolnij uchwyt niezarządzalnego elementu
        Release(uchwyt);
        uchwyt = IntPtr.Zero;
        // ustaw znacznik zwolnienia, aby nie zwalniać zasobów ponownie
        this.zwolniony = true;
    }

    // pozwól na wielokrotne wywołanie metody Dispose, ale wyrzucić wyjątek
    // gdy obiekt był już zwolniony (zawsze sprawdzaj czy nastąpiło już
    // zwolnienie zasobów)
    public void WykonajCos()
    {
        if (this.zwolniony)
            throw new ObjectDisposedException(...);
    }
}
```

## Rozdział 10.

# Wyjątki

### Mechanizm wyjątków

Każdy dobry program powinien umieć obsługiwać przypadki wystąpienia różnych błędów. Mówi się, że nie ma takiego programu, który jest pozbawiony błędów. Są jedynie programy, w których błędy się nie ujawniają, co nie oznacza, że nie wystąpią w pewnych okolicznościach, o których nie pomyśleliśmy pisząc program.

Zanim stworzono mechanizm wyjątków, obsługa błędów była bardzo niewygodna. Aby poprawnie obsłużyć błędy mogące wystąpić w czasie działania programu, należało sprawdzać kody powrotu każdej funkcji i uzależniać od tego wykonywanie kolejnych kroków programu. Jednak sprawdzanie kodów powrotu na każdym kroku sprawiało, że program robił się zbyt złożony i mało przejrzysty (dlatego wielu programistów ignorowało zwracane przez funkcje kody). Często również same funkcje mimo wystąpienia błędu zwracały poprawny kod. Innym problemem była niejednolitość w numeracji kodów powrotu. Dla jednej funkcji 0 oznaczało poprawne zakończenie, a dla innej błąd.

Mając na uwadze problemy z obsługą błędów stworzono jednolity mechanizm obsługi wyjątków, który pozwolił na odseparowanie logiki funkcjonalnej programu od logiki obsługi błędów.

Wyjątek oznacza zarówno błąd jak i nieoczekiwane zachowanie, występujące w czasie działania programu. Mogą one być zgłaszane zarówno jako rezultat: błędnie napisanego przez nas kodu, odwołania się do funkcjonalności umieszczonej w jakiejś bibliotece, wystąpienia błędu zgłoszonego przez system operacyjny (np.: braku zasobów) lub wystąpienia nieoczekiwanych warunków.

W .NET Framework wyjątek jest obiektem, który dziedziczy z klasy wyjątku *System.Exception*.

### Bloki try i catch

Mechanizm obsługi wyjątków w C# realizowany jest za pomocą bloków *try* i *catch*. Blok *try* jest blokiem, w którym umieszcza się logikę funkcjonalną programu, natomiast obsługa błędów odbywa się w bloku *catch*.

Składnia obsługi wyjątków za pomocą bloków *try* i *catch* wygląda następująco:

```
try
{
    logika_funkcjonalna
}
catch (klasa_wyjatku Identyfikator)
{
    obsluga_bledow
}
gdzie:
```

*logika\_funkcjonalna*

ciąg instrukcji programu realizujący zadaną funkcjonalność (opcjonalne);

*klasa\_wyjatku*

klasa obsługująca wyjątek, musi być klasą **System.Exception** lub dziedziczącą z niej (wymagane);

*Identyfikator*

nazwa instancji klasy obsługi wyjątku (opcjonalne);

*obsługa\_bledów*

ciąg instrukcji programu wykonywany w przypadku wystąpienia błędu (opcjonalne).

Przykład:

```
try
{
    int x = 10, y = 0, z;

    z = x / y; // wyjątek: dzielenie przez zero
    z++;      // ta instrukcja nie zostanie już wykonana
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Dzielenie przez zero!");
}
```

W momencie wystąpienia, wyjątku następuje skok do bloku **catch**, a pozostałe instrukcje znajdujące się w bloku **try** nie zostają wykonane (w naszym przykładzie nie nastąpi już inkrementacja zmiennej *z*).

Ciąg instrukcji zawarty wewnątrz bloku **try** może zwrócić wiele różnych wyjątków, które mogą być obsługiwane przez wiele różnych klas. Ze względu na ten fakt, istnieje możliwość umieszczania za blokiem **try** wielu różnych bloków **catch**, obsługujących różne przypadki:

```
try
{
    int x, y = 0, z;

    Console.Write("Podaj numer: ");
    // jeżeli wprowadzimy literę wystąpi wyjątek FormatException
    x = Convert.ToInt32(Console.ReadLine());
    // jeżeli wprowadzimy poprawny numer wystąpi wyjątek DivideByZeroException
    z = x / y;
}
catch (FormatException)
{
    Console.WriteLine("Niepoprawny format.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Dzielenie przez zero.");
}
```

```
}
```

Umieszczając różne bloki *catch* należy pamiętać o właściwej kolejności. Wyjątki należy obsługiwać począwszy od szczegółowych do ogólnych (bazowa klasa *System.Exception* musi znaleźć się jako ostatnia):

```
try
{
    int x = 10, y = 0, z;

    z = x / y;
}
catch (Exception) // BŁĄD: niewłaściwa kolejność
{
    Console.WriteLine("Nieznany błąd!");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Dzielenie przez zero!");
}
```

Powinno być tak:

```
try
{
    int x = 10, y = 0, z;

    z = x / y;
}
catch (DivideByZeroException)
{
    Console.WriteLine("Dzielenie przez zero!");
}
catch (Exception)
{
    Console.WriteLine("Nieznany błąd!");
}
```

## Klasy wyjątków

Wyjątki mogą być obsługiwane przez różne klasy. Podstawową klasą dla wyjątków jest *System.Exception*. W .NET Framework zdefiniowano ogromną liczbę klas wyjątków.

Przyjrzymy się bazowej klasie wyjątków *System.Exception* (klasa ta, jak każda inna w C#, dziedziczy z klasy *System.Object*). Poniższa tabela przedstawia składniki klasy oraz nadpisane metody z klasy *System.Object* (nienadpisane metody nie są wymienione):

Nazwa składnika	Rodzaj składnika	Opis
<i>Exception</i>	konstruktor	Tworzy instancję klasy.
<i>HelpLink</i>	właściwość	Pozwala na ustawienie i odczytanie linku do pliku związanego z wyjątkiem.
<i>InnerException</i>	właściwość	Pozwala na odczytanie instancji klasy <b>Exception</b> , która spowodowała wystąpienie aktualnego wyjątku.
<i>Message</i>	właściwość	Pozwala na odczytanie opisu tekstowego dla aktualnego wyjątku.
<i>Source</i>	właściwość	Pozwala na ustawienie i odczytanie nazwy aplikacji lub obiektu, który spowodował wystąpienie aktualnego wyjątku.
<i>StackTrace</i>	właściwość	Pozwala na odczytanie łańcucha znaków, będącego reprezentacją sekwencji wywołań metod w czasie wystąpienia aktualnego wyjątku.
<i>TargetSite</i>	właściwość	Pozwala na odczytanie nazwy metody, która spowodowała wystąpienie aktualnego wyjątku.
<i>GetBaseException</i>	metoda wirtualna	W przypadku nadpisania jej w klasie potomnej, może zwrócić instancję klasy <b>Exception</b> , która stanowi pierwszy zwrócony wyjątek w całym łańcuchu wyjątków.
<i>GetObjectData</i>	metoda wirtualna	W przypadku nadpisania jej w klasie potomnej, może pozwolić na ustawienie <b>SerializationInfo</b> informacjami dotyczącymi wyjątku.
<i>ToString</i>	odziedziczona i nadpisana metoda z klasy <b>object</b>	Tworzy i zwraca łańcuch znaków reprezentujący aktualny wyjątek.
<i>HResult</i>	właściwość chroniona	Pozwala na ustawienie i odczytanie numerycznej wartości błędu skojarzonej z aktualnym wyjątkiem.

Ze względu na dużą wagę obsługi wyjątków, dobrze jest orientować się w podstawowych klasach wyjątków. Poniższa tabela przedstawia jedynie niektóre z tych klas:

Nazwa klasy wyjątku	Nazwa klasy bazowej	Przeznaczenie
<i>Exception</i>	<i>Object</i>	Bazowa klasa wyjątków, przechwytuje wszystkie błędy.
<i>SystemException</i>	<i>Exception</i>	Bazowa klasa wyjątków dla przestrzeni <b>System</b> .
<i>ArithmeticException</i>	<i>SystemException</i>	Błędy: arytmetyczne, rzutowania i konwersji.
<i>DivideByZeroException</i>	<i>ArithmeticException</i>	Błąd dzielenia przez zero.
<i>NotFiniteNumberException</i>	<i>ArithmeticException</i>	Liczba zmiennoprzecinkowa jest dodatnio lub ujemnie nieskończona.
<i>OverflowException</i>	<i>ArithmeticException</i>	Błąd przepełnienia w czasie działań arytmetycznych, rzutowania i konwersji.
<i>FormatException</i>	<i>SystemException</i>	Format argumentu nie jest zgodny ze specyfikacją wywołującej metody.
<i>IndexOutOfRangeException</i>	<i>SystemException</i>	Próba dostępu do elementu tablicy poprzez indeks spoza zakresu.
<i>InvalidCastException</i>	<i>SystemException</i>	Błąd rzutowania lub jawnej konwersji.

<i>OutOfMemoryException</i>	<i>SystemException</i>	Brak pamięci do kontynuowania działania programu.
<i>RankException</i>	<i>SystemException</i>	Niewłaściwa liczba wymiarów tablicy przekazana do metody.
<i>IOException</i>	<i>SystemException</i>	Błędy związane z obsługą plików.
<i>DirectoryNotFoundException</i>	<i>IOException</i>	Nie odnaleziono folderu.
<i>FileNotFoundException</i>	<i>IOException</i>	Nie odnaleziono pliku.
<i>PathTooLongException</i>	<i>IOException</i>	Zbyt długa ścieżka.
<i>FileLoadException</i>	<i>IOException</i>	Błąd wczytania pliku.
<i>EndOfStreamException</i>	<i>IOException</i>	Próba odczytu danych spoza końca strumienia.
<i>InternalBufferOverflowException</i>	<i>SystemException</i>	Przepełnienie bufora.

## Rzucanie wyjątków

W C# istnieje możliwość rzucania własnych wyjątków. Do tego celu służy instrukcja **throw**, po której należy wskazać instancję klasy rzucanego wyjątku.

Przykład:

```
throw new FileNotFoundException(NazwaPliku);
```

Jeżeli chcemy stworzyć własną klasę wyjątku musimy pamiętać, aby dziedziczyła ona z **System.Exception** (lub innej potomnej).

Przykładowy program:

```
using System;

class Test
{
    class NieAkceptowanaWartosc : Exception
    {
    }

    class Wartosci
    {
        public static void Procent(int Argument)
        {
            if (Argument < 0 || Argument > 100)
                throw new NieAkceptowanaWartosc(); // rzucamy wyjątek
        }
    }

    static void Main()
    {
        try
        {
            Wartosci.Procent(101);
        }
    }
}
```

```
    }  
    catch(NieAkceptowanaWartosc) // przechwytujemy nasz wyjątek  
    {  
        Console.WriteLine("Niepoprawna wartość.");  
    }  
}  
}
```

Wyjątki można rzucać również z bloku *catch*:

```
try  
{  
}  
catch (Exception ex)  
{  
    throw ex;  
}
```

Mechanizm ten nazywa się przerzucaniem wyjątku. W przypadku tego mechanizmu można zastosować również uproszczoną wersję zapisu:

```
try  
{  
}  
catch (Exception)  
{  
    throw;  
}
```

Samo przerzucenie wyjątku, bez wykonania innych instrukcji nie ma zbyt dużego sensu. Przyjrzyjmy się następującemu fragmentowi kodu:

```
try  
{  
    try  
    {  
        // instrukcje  
    }  
    catch (Exception)  
    {  
        throw; // przerzucenie  
    }  
}  
catch (Exception)  
{  
    // złapanie przerzuconego  
}
```

Jest on równoważny w zapisie takiemu fragmentowi:



```
try
{
    // instrukcje
}
catch (Exception)
{
    // złapanie wyjątku
}
```

Jak widać, w takim przypadku przerzucenie wyjątku jest operacją zbędną. Może się jednak zdarzyć, że będziemy chcieli obsłużyć wyjątek w jakiś określony sposób, a następnie przerzucić go na wyższy poziom lub rzucić inny rodzaj wyjątku. W takim wyjątku ma to sens:

```
try
{
    try
    {
        // instrukcje
    }
    catch (JakasKlasaException)
    {
        // operacje
        throw new InnaKlasaException();
    }
}
catch (InnaKlasaException)
{
}
```

## Blok finally

W przypadku wystąpienia wyjątku, dalsze wykonywanie instrukcji z bloku **try** nie jest kontynuowane, a sterowanie zostaje przekazane do bloku **catch**. Mechanizm wyjątków pozwala jednak na dołączenie jeszcze jednego bloku, który jest wykonywany zawsze jako ostatni, bez względu na to czy wystąpi wyjątek czy nie. Jest to blok **finally**. Blok ten współpracuje z blokiem **try** w podobny sposób jak blok **catch**:

```
try
{
}
catch (Exception)
{
}
finally
{
    // instrukcje końcowe
}
```

```
}
```

Blok *finally* jest przydatny, gdyż zapobiega sytuacji powielania się tych samych instrukcji w blokach *try* i *catch*. Zamiast zapisu:

```
try
{
    // ...
    // instrukcje np.: zwolnienie zasobów
}
catch (Exception)
{
    // obsługa wyjątku
    // instrukcje np.: zwolnienie zasobów
}
```

Stosujemy:

```
try
{
    // ...
}
catch (Exception)
{
    // obsługa wyjątku
}
finally
{
    // instrukcje np.: zwolnienie zasobów
}
```

## Przepełnienia arytmetyczne

Domyślnie w C# nie jest sprawdzane przepełnienie arytmetyczne. Do takiego przepełnienia może dojść w przypadku, gdy próbujemy przypisać jakiejś zmiennej wartość, która nie może być w niej przechowana. Przykładowo, jeżeli zmienna całkowita ma wartość maksymalną dopuszczalną dla typu, a my spróbujemy zwiększyć ją o jeden, nastąpi przepełnienie:

```
int numer = int.MaxValue;

Console.WriteLine(++numer); // przepełnienie arytmetyczne
```

W powyższym przykładzie zainicjowano zmienną całkowitą typu *int* maksymalną dodatnią wartością, czyli 2 147 483 647. Po wykonaniu operacji jednostkowego zwiększenia, nastąpiło przepełnienie, przez co zmienna przyjęła wartość -2 147 483 648, czyli minimalnej dopuszczalnej wartości.

Przepełnienia arytmetyczne wymagają jawnej kontroli, ze względu na to, że domyślnie nie są sprawdzane. W języku C# istnieje możliwość włączenia i wyłączenia mechanizmu sprawdzania przepełnień dla:

- całego programu – służy do tego opcja kompilacji **checked**;

- włączenie opcji:

```
csc /checked+ program.cs
```

- wyłączenie opcji:

```
csc /checked- program.cs
```

- pojedynczych wyrażeń – nawet w przypadku, gdy włączono lub wyłączono opcję sprawdzania dla całego programu istnieje możliwość włączenia/wyłączenia tej opcji dla pojedynczego wyrażenia;

- włączenie sprawdzania wyrażenia:

```
checked (wyrażenie)
```

Przykład:

```
int numer = int.MaxValue;
```

```
Console.WriteLine(checked(++numer));
```

- wyłączenie sprawdzania wyrażenia:

```
unchecked (wyrażenie)
```

Przykład:

```
int numer = int.MaxValue;
```

```
Console.WriteLine(unchecked(++numer));
```

- bloku instrukcji – nawet w przypadku, gdy włączono lub wyłączono opcję sprawdzania dla całego programu, istnieje możliwość włączenia/wyłączenia tej opcji dla bloku instrukcji;

- włączenie sprawdzania bloku:

```
checked
```

Przykład:

```
int numer = int.MaxValue;
```

```
checked
```

```
{  
    Console.WriteLine(++numer);  
    Console.WriteLine(--numer);  
}
```

- wyłączenie sprawdzania bloku:

```
unchecked
```

Przykład:

```
int numer = int.MaxValue;
```

```
unchecked
```

```
{  
    Console.WriteLine(++numer);  
    Console.WriteLine(--numer);  
}
```

# Rozdział 11.

## Przestrzenie nazw

### Deklarowanie przestrzeni nazw

Przestrzenie nazw grupują klasy w obrębie wspólnej logicznej struktury. Do deklarowania przestrzeni nazw służy słowo kluczowe *namespace*.

Składnia wygląda następująco:

```
namespace Identyfikator[.Identyfikator1[...]] {definicja_typów}
```

gdzie:

*Identyfikator*, *Identyfikator1*, ...

Nazwa przestrzeni nazw (wymagane);

*definicja\_typów*

Przestrzeń nazw może zawierać: inne przestrzenie nazw, klasy, struktury, interfejsy, delegacje i typy wyliczeniowe.

Przykład:

```
namespace Matematyka
{
    interface IDzialania
    {
        ...
    }

    class Macierze
    {
        ...
    }

    class Wielomiany
    {
        ...
    }

    ...
}
```

Przestrzenie nazw mogą być wielokrotnie deklarowane w różnych plikach. Ponowna deklaracja przestrzeni o takiej samej nazwie, nazywa się otwarciem przestrzeni nazw. Po otwarciu przestrzeni nazw, można do niej dołączać kolejne klasy:

```
namespace Matematyka
{
```

```
class LiczbyZespolone
{
    ...
}
```

```
namespace Matematyka
{
    class Pola
    {
        ...
    }
}
```

Przestrzenie nazw można również zagnieżdżać:

```
namespace Matematyka
{
    namespace Trygonometria
    {
    }
}
```

W przypadku zagnieżdżania przestrzeni nazw, można zastosować uproszczoną konwencję zapisu, która pozwala na zmniejszenie liczby linii kodu.

Powyższy przykład można zapisać w takiej formie:

```
namespace Matematyka.Trygonometria
{
}
```

Przestrzenie nazw są publiczne. Nie można stosować w stosunku do nich żadnych modyfikatorów:

```
public namespace Matematyka.Trygonometria // BŁĄD
{
    ...
}
```

Zaletą przestrzeni nazw jest możliwość zdefiniowania typów o takich samych nazwach w różnych przestrzeniach (w tej samej przestrzeni nie mogą istnieć dwa typy o takiej samej nazwie):

```
namespace Liczby1
{
    class Liczba
    {
        ...
    }
}
```

```
namespace Liczby2
{
    class Liczba
    {
        ...
    }
}
```

## Nazwy kwalifikowane

W przypadku używania klas wewnątrz tej samej przestrzeni, odwołanie do zdefiniowanego w przestrzeni typu odbywa się poprzez skróconą wersję zapisu nazwy typu. Takie odwołanie określa się mianem nazwy niekwalifikowanej:

```
namespace Test
{
    class Liczba
    {
        ...
    }

    class Test
    {
        static void Main()
        {
            Liczba lb = new Liczba();
        }
    }
}
```

W przypadku odwoływania się do typów zdefiniowanych w innej przestrzeni nazw, mamy do czynienia z odwołaniem się poprzez pełną nazwę kwalifikowaną, czyli całą ścieżkę począwszy od głównej przestrzeni nazw, poprzez podprzestrzenie aż do zdefiniowanego typu:

```
namespace Liczby
{
    public class Liczba
    {
        ...
    }
}

namespace Test
{
    static void Main()
    {
```

```
Liczba lb1 = new Liczba(); // BŁĄD: nieznany typ

Liczby.Liczba lb = new Liczby.Liczba(); // kwalifikowana nazwa typu
}
}
```

## Dyrektywa using

Dyrektywa **using** pozwala na skrócenie zapisu w przypadku odwoływania się do kwalifikowanej nazwy typu. Zamiast pisać przy każdym odwołaniu pełnej ścieżki, możemy odwołać się bezpośrednio do typu zdefiniowanego w innej przestrzeni nazw:

```
namespace Liczby
{
    public class Liczba
    {
        ...
    }
}

// ... inny plik ...
using Liczby;

namespace Test
{
    static void Main()
    {
        Liczba lb = new Liczba();
    }
}
```

Używając dyrektywy **using** należy pamiętać o tym, że:

- w przypadku zagnieżdżonych przestrzeni nazw należy użyć pełnej ścieżki począwszy od głównej przestrzeni poprzez kolejne zagnieżdżone aż do najbardziej zagnieżdżonej:

```
using System.Security.Cryptography;
```

- dyrektywa **using** musi znajdować się w globalnym zasięgu:

```
namespace Test
{
    class Test
    {
        static void Main()
        {
            Liczba lb = new Liczba();
        }
    }
}
```

```
}
```

*using Liczby; // BŁĄD: powinno być na początku*

- dyrektywa **using** może być umieszczona wewnątrz przestrzeni nazw:

```
namespace Test
{
    using Liczby;

    class Test
    {
        static void Main()
        {
            Liczba lb = new Liczba();
        }
    }
}
```

- dyrektywa **using** nie jest rekurencyjna:

```
namespace Matematyka
{
    namespace Figury
    {
        public class Kwadrat
        {
            ...
        }
    }

    public class Liczba
    {
        ...
    }
}

// ... inny plik ...
using Matematyka;

namespace Test
{
    class Test
    {
        static void Main()
        {
            Kwadrat kw = new Kwadrat(); // BŁĄD: nieznany typ
        }
    }
}
```



```
Liczba lb = new Liczba();  
}  
}  
}
```

- w przypadku, gdy w dwóch różnych przestrzeniach zadeklarowanych poprzez **using** znajdują się typy o tych samych nazwach, należy odwoływać się do nich używając nazwy kwalifikowanej:

```
namespace Przestrzen1  
{  
    class Klasa  
    {  
        ...  
    }  
}  
  
namespace Przestrzen2  
{  
    class Klasa  
    {  
        ...  
    }  
}  
  
// ... inny plik ...  
namespace Test  
{  
    using Przestrzen1;  
    using Przestrzen2;  
  
    class Test  
    {  
        static void Main()  
        {  
            Klasa k1 = new Klasa(); // BŁĄD: nie wiadomo o którą klasę chodzi  
            Przestrzen1.Klasa k1 = new Przestrzen1.Klasa();  
            Przestrzen2.Klasa k2 = new Przestrzen2.Klasa();  
        }  
    }  
}
```

## Tworzenie aliasów

Dyrektywę **using** można również użyć do tworzenia aliasów dla zagnieżdżonych przestrzeni nazw lub typów. Jeżeli alias dotyczy jedynie zagnieżdżonych przestrzeni nazw, dostęp do typu zdefiniowanego

w zagnieżdżonej przestrzeni wymaga podania aliasu oraz nazwy typu:

```
namespace Matematyka
{
    namespace Figury
    {
        public class Kwadrat
        {
            ...
        }
    }
}

// ... inny plik ...
namespace Test
{
    using MFigury = Matematyka.Figury; // alias do zagnieżdżonej przestrzeni

    class Test
    {
        static void Main()
        {
            // odwołanie się do typu poprzez alias
            MFigury.Kwadrat pkw = new MFigury.Kwadrat();
        }
    }
}
```

W przypadku, gdy alias dotyczy nazwy typu w zagnieżdżonej przestrzeni nazw, odwołanie się do niego wymaga podania samego aliasu:

```
namespace Matematyka
{
    namespace Figury
    {
        public class Kwadrat
        {
            ...
        }
    }
}

// ... inny plik ...
namespace Test
{
```

```
using MFKwadrat = Matematyka.Figury.Kwadrat; // alias do typu

class Test
{
    static void Main()
    {
        // odwołanie się do typu poprzez alias
        MFKwadrat pkw = new MFKwadrat();
    }
}
```

# Rozdział 12.

## Tablice

### Deklaracja tablic

Tablica jest szeregiem danych zawierającym określoną liczbę elementów tego samego typu. Elementy tablicy są składowane w jednym ciągłym bloku pamięci, co pozwala na szybki dostęp. Poza tym dostęp do elementów jest swobodny (tzn. można w danym momencie odczytać i zapisać dowolny element, dowolną ilość razy).

Składnia deklaracji tablicy wygląda następująco:

```
typ[] Identyfikator;
```

gdzie:

*typ*

Typ danych elementów tablicy (wymagane);

*Identyfikator*

Nazwa tablicy (wymagane).

Przykład:

```
int[] Liczby;
```

Należy pamiętać o jednej ważnej rzeczy. Zadeklarowanie zmiennej tablicowej nie tworzy instancji tablicy. Jest to jedynie zmienna referencyjna, która będzie powiązana z fizyczną tablicą dopiero po utworzeniu instancji tej tablicy.

Bazową klasą dla wszystkich tablic w języku C# jest klasa **System.Array**, więc w momencie utworzenia tablicy, można korzystać z metod i właściwości tej klasy.

### Wymiary tablic

Tablice mogą być jedno lub wielowymiarowe. W przypadku deklaracji tablic jednowymiarowych, używa się pustych nawiasów kwadratowych:

```
int[] Liczby; // deklaracja tablicy jednowymiarowej
```

W przypadku tablic wielowymiarowych w nawiasach kwadratowych należy wstawić znaki przecinka, które oddzielają poszczególne wymiary. Tak więc tablicę dwuwymiarową deklaruje się poprzez umieszczenie pojedynczego przecinka wewnątrz nawiasów, tablicę trójwymiarową poprzez umieszczenie dwóch przecinków, etc.:

```
int [,] LiczbyDw; // deklaracja tablicy dwuwymiarowej
```

```
int [,,] LiczbyTr; // deklaracja tablicy trójwymiarowej
```

## Tworzenie instancji tablic

Zadeklarowanie zmiennej tablicowej nie tworzy instancji tablicy (ponieważ jest to typ referencyjny). Aby utworzyć instancję tablicy należy posłużyć się operatorem *new*. Tworząc instancję tablicy należy podać jej rozmiar w każdym z wymiarów:

```
int[] Liczby = new int[20]; // jednowymiarowa tablica 20-sto elementowa
int[,] LiczbyD = new int[2, 2]; // dwuwymiarowa tablica 2x2 elementowa
```

Jeżeli nie podamy któregośkolwiek z rozmiarów w danym wymiarze, kompilator zgłosi błąd:

```
float[] Liczby = new float[]; // BŁĄD
double[,] LiczbyD = new double[10, ]; // BŁĄD
```

Po utworzeniu instancji tablicy, wszystkie elementy tablicy są inicjowane domyślnymi wartościami (tzn.: dla typu całkowitego elementy inicjowane są wartościami 0, dla typu zmiennoprzecinkowego wartościami 0.0, dla typu logicznego elementy inicjowane są wartością *false*, a dla typu referencyjnego elementy inicjowane są wartościami *null*).

Tworząc instancję tablicy, kompilator zawsze alokuje ciągły obszar pamięci, aby zapewnić swobodny i wydajny dostęp do poszczególnych elementów.

Tablice mogą być tworzone dynamicznie w czasie działania programu:

```
Console.WriteLine("Podaj rozmiar tablicy: ");
string str = Console.ReadLine();
int rozmiar = Convert.ToInt32(str);
int[] Liczby = new int[rozmiar];
```

## Dostęp do elementów

Aby uzyskać dostęp do określonego elementu tablicy, należy użyć operatora indeksowania (nawiasy kwadratowe) podając indeks w danym wymiarze. Indeks określa położenie elementu w ciągu i może przyjmować wartości od 0 do liczby elementów w danym wymiarze pomniejszoną o jeden (ze względu na numerację od zera). W przypadku tablicy jednowymiarowej w nawiasie podaje się jeden indeks:

```
int[] Liczby = new int[10];
Liczby[0] = 10; // przypisanie wartości pierwszemu elementowi
Liczby[9] = Liczby[0] + 10; // przypisanie wartości ostatniemu elementowi
```

W przypadku tablic wielowymiarowych, indeksy dotyczące poszczególnych wymiarów, oddziela się przecinkami:

```
int[,] LiczbyD = new int[10, 10];
LiczbyD[1][2] = 100;
LiczbyD[2][1] = LiczbyD[1][2] * 2;
```

W przypadku, gdy wartość podanego indeksu jest z poza zakresu (0..rozmiar-1), następuje wyrzucenie wyjątku *IndexOutOfRangeException*:

```
int[] Liczby = new int[10];
```

```
Liczby[99] = 100; // BŁĄD: indeks poza zakresem
```

W przypadku, gdy mamy wątpliwość, czy podany indeks należy do poprawnego zakresu, powinniśmy obsłużyć wyjątek:

```
try
{
    int[] Liczby = new int[10];
    string strLiczba;

    Console.Write("Podaj numer indeksu: ");
    strLiczba = Console.ReadLine();
    // nie wiadomo jaka będzie wartość indeksu dopóki nie zostanie podana
    Liczby[Convert.ToInt32(strLiczba)] = 100;
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Podano indeks z poza zakresu.");
}
```

## Inicjacja elementów tablic

W momencie tworzenia instancji tablicy, istnieje możliwość zainicjowania elementów tej tablicy wartościami początkowymi przy pomocy listy inicjacyjnej. Wartości początkowe podaje się w nawiasach {} oddzielając je przecinkami. Należy przy tym pamiętać o tym, aby podać wszystkie wartości początkowe:

```
int[] Liczby = new int[4]{1, 2, 3, 4};
int[,] LiczbyD = new int[2,2]{{1, 2}, {3, 4}};
int[] Liczby = new int[4]{1, 2};           // BŁĄD: muszą być wszystkie wartości
int[,] LiczbyD = new int[2, 2]{{1, 3}}; // BŁĄD: muszą być wszystkie wartości
```

Jeżeli instancja tablicy, tworzona jest w czasie deklaracji zmiennej tablicowej do inicjacji wartościami początkowymi, można użyć notacji skróconej w nawiasach {}. Liczba wartości podanych w tych nawiasach stanie się wtedy rozmiarem tablicy:

```
int[] Liczby = {1, 2, 3, 4};           // instancja tablicy czteroelementowej
int[] Liczby = {1, 2};                 // instancja tablicy dwuelementowej
int[,] Liczby = {{1, 2}, {3, 4}};      // instancja tablicy dwuwymiarowej 2x2
```

Notacji tej można używać jedynie w czasie deklaracji zmiennej. Jeżeli zmienna jest już zadeklarowana w czasie tworzenia nowego odnośnika do instancji, notacja skrócona nie jest dozwolona:

```
int[] Liczby;
Liczby = new int[4]{1, 2, 3, 4};
Liczby = {1, 2, 3, 4};           // BŁĄD
int[,] LiczbyD;
LiczbyD = new int[2, 2]{{1, 2}, {3, 4}};
LiczbyD = {{1, 2}, {3, 4}};     // BŁĄD
```

Z listy inicjacyjnej nie można korzystać w przypadku dynamicznego tworzenia tablic w czasie działania programu:

```
Console.Write("Podaj rozmiar tablicy: ");
string str = Console.ReadLine();
int rozmiar = Convert.ToInt32(str);
int[] Liczby = new int[rozmiar]{1, 2, 3, 4}; // BŁĄD: nie dozwolone
```

## Właściwości tablic

Klasa *System.Array*, zawiera wiele przydatnych właściwości, z których można korzystać w czasie operowania na tablicach. Do najważniejszych i najczęściej używanych należą:

- **Length** – właściwość tylko do odczytu, określająca liczbę wszystkich elementów tablicy we wszystkich wymiarach.

Przykład użycia:

```
int[] Liczby = new int[10];

for (int I = 0; I < Liczby.Length; I++)
    Liczby[I] = 9999;
```

- **Rank** – właściwość tylko do odczytu, określająca liczbę wymiarów tablicy.

Przykład użycia:

```
int[, ,] LiczbyT = new int[10, 10, 10];

if (LiczbyT.Rank == 3)
    Console.WriteLine("Tablica trójwymiarowa.");
```

## Metody operujące na tablicach

Klasa *System.Array*, zawiera wiele przydatnych metod, z których można korzystać w czasie operowania na tablicach. Do najważniejszych i najczęściej używanych należą:

- **Clear** – (metoda statyczna) przypisuje elementom tablicy z określonego przedziału wartości domyślne (0 dla typów całkowitych, 0.0 dla typów zmiennoprzecinkowych, *false* dla typu *bool* oraz *null* dla typów referencyjnych).

Przykład:

```
int[] Liczby = {1, 2, 3, 4, 5, 6, 7, 8, 9};

System.Array.Clear(Liczby, 0, Liczby.Length); // wypełnienie zerami
```

- **GetLength** – zwraca długość tablicy w określonym wymiarze.

Przykład:

```
int[,] LiczbyD = {{1, 2, 3, 4}, {5, 6, 7, 8}};
int rozmiar0 = LiczbyD.GetLength(0); // 2
int rozmiar1 = LiczbyD.GetLength(1); // 4
```

- **Sort** – (metoda statyczna) sortuje elementy tablicy. Metodę można użyć do sortowania tablicy

złożonej z elementów określonej klasy czy struktury, pod warunkiem, że implementuje ona interfejs **IComparable**.

Przykład:

```
int[] Liczby = {4, 2, 7, 1, 5, 6, 3, 9, 8};

System.Array.Sort(Liczby); // 1, 2, 3, 4, 5, 6, 7, 8, 9
```

- **Clone** – tworzy nową instancję tablicy i kopiuje wartości wszystkich elementów z klonowanej tablicy. Metoda ta tworzy jedynie „płytką kopię” elementów, co oznacza, że jeżeli elementami tablicy są referencje do jakiś obiektów, kopia tej tablicy będzie zawierać referencje do tych samych obiektów co klonowana tablica.

Przykład:

```
int[] Liczby = {4, 2, 7, 1, 5, 6, 3, 9, 8};
int[] KopiaLiczby = (int[])Liczby.Clone(); // kopia tablicy
```

- **IndexOf** – zwraca indeks pierwszego elementu, którego wartość jest zgodna z wartością argumentu przekazanego do tej metody. W przypadku nieznaalezienia elementu o określonej wartości w tablicy zwracana jest wartość -1:

Przykład:

```
int[] Liczby = {4, 2, 7, 4, 2, 7, 4, 2, 7};
int gdzie = System.Array.IndexOf(Liczby, 7); // 2
```

```
if (gdzie == -1)
    Console.WriteLine("Brak elementu w tablicy.");
```

- **LastIndexOf** – zwraca indeks ostatniego elementu, którego wartość jest zgodna z wartością argumentu przekazanego do tej metody. W przypadku nieznaalezienia elementu o określonej wartości w tablicy zwracana jest wartość -1:

Przykład:

```
int[] Liczby = {4, 2, 7, 4, 2, 7, 4, 2, 7};
int gdzie = System.Array.LastIndexOf(Liczby, 7); // 8
```

```
if (gdzie == -1)
    Console.WriteLine("Brak elementu w tablicy.");
```

- **GetValue** – zwraca wartość elementu o określonym indeksie.

Przykład:

```
int[] Liczby = {4, 2, 7, 4, 2, 7, 4, 2, 7};
```

```
if (Liczby.GetValue(1) == 2)
{
    ...
}
```

- **SetValue** – ustawia wartość elementu o określonym indeksie (najpierw podaje się wartość a następnie indeks).

Przykład:

```
int[] Liczby = {4, 2, 7, 4, 2, 7, 4, 2, 7};
```



```
Liczby.SetValue(100,1); // 4, 100, 7, 4, 2, 7, 4, 2, 7
```

- **Reverse** – odwraca porządek elementów w tablicy.

Przykład:

```
int[] Liczby = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
System.Array.Reverse(Liczby); // 9, 8, 7, 6, 5, 4, 3, 2, 1
```

## Zwracanie tablic z metod

Tablice jak inne zmienne mogą być zwracane z metod. W takim przypadku, w sygnaturze metody, określa się typ referencyjny dla odpowiedniej tablicy a z metody zwraca instancję tablicy.

Przykładowo:

```
class Test
{
    static int[] UtworzTablice(int Rozmiar)
    {
        return new int[Rozmiar];
    }

    static void Main()
    {
        int[] Liczby = UtworzTablice(10);
    }
}
```

Określenie rozmiaru tablicy w sygnaturze jest błędem (określamy tylko typ zwracanej wartości):

```
static int[10, 10] UtworzTablice(10) // BŁĄD
{
    return new int[10, 10];
}
```

Możliwe jest również zwracanie tablic o dynamicznym rozmiarze:

```
class Test
{
    static int[] UtworzTablice()
    {
        string str;

        Console.WriteLine("Podaj ilość elementów: ");
        str = Console.ReadLine();
        return new int[Convert.ToInt32(str)];
    }
}
```

```
static void Main()
{
    int[] Liczby = UtworzTablice();
}
}
```

## Przekazywanie tablic do metod

Tablice mogą być przekazywane do metod w postaci argumentów. Przekazując tablicę jako argument, tworzona jest kopia zmiennej referencyjnej będąca odnośnikiem do instancji tej samej tablicy (nie jest tworzona nowa instancja tablicy, przekazanie argumentu jest natychmiastowe).

Przykład:

```
class Test
{
    static void Drukuj(int[] Argument)
    {
        for(int i = 0; i < Argument.Length; i++)
            Console.WriteLine(Argument[i]);
    }

    static void Main()
    {
        int[] Tablica = {0, 1, 2, 3};

        Drukuj(Tablica);
    }
}
```

Do metody można przekazać również nową instancję tablicy zamiast istniejącej. Należy jednak pamiętać, że czas istnienia tej instancji, ograniczony jest jedynie do czasu istnienia argumentu referencyjnego metody, do której przekazujemy nową instancję:

```
class Test
{
    static void Drukuj(int[] Argument)
    {
        for(int i = 0; i < Argument.Length; i++)
            Console.WriteLine(Argument[i]);
    }

    static void Main()
    {
        Drukuj(new int[4]{0, 1, 2, 3}); // poza metodą instancja nie jest dostępna
    }
}
```

```
}
```

## Tablica argumentów Main

W czasie uruchamiania programu pierwszą metodą jaka zostanie wykonana, jest metoda **Main**. Uruchamiając program z linii poleceń, można przekazać do programu dodatkowe parametry wejściowe. Parametry te przekazywane są do metody **Main** w postaci argumentu będącego tablicą napisów o długości równej ilości przekazanych do programu parametrów.

Przykładowy program:

```
using System;

class Test
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            Console.WriteLine("Parametry: ");

            foreach (string str in args)
                Console.WriteLine(str);
        }
        else
            Console.WriteLine("Brak parametrów.");
    }
}
```

Po skompilowaniu i uruchomieniu programu, możemy przekazać do niego parametry. Każdy parametr oddzielony jest znakiem spacji. Wyjątek stanowią parametry w cudzysłowach, w których spacja jest traktowana jako część jednego parametru:

```
test.exe -x ala.txt /s:s "Ala ma kota"
```

Wynik:

```
Parametry:
-x
ala.txt
/s:s
Ala ma kota
```

## Rozdział 13.

# Łańcuchy

### Klasa String

W języku C# typ *string* jest używany do przechowywania oraz operowania na ciągu znaków w standardzie Unicode. Typ *string* jest aliasem do klasy *System.String*, więc obie deklaracje zmiennych mogą być stosowane zamiennie:

```
string strNapis;  
System.String strInnyNapis;
```

Po zadeklarowaniu zmiennej tego typu można przypisać jej wartość (utworzyć instancję klasy *String*):

```
string str = "Test";
```

Do poszczególnych znaków łańcucha można dostać się za pomocą operatora indeksowania:

```
if (str[0] == 'T')  
{  
    ...  
}
```

Typ *string* jest jednak typem dość specyficznym. Przechowywany w łańcuchu tekst po utworzeniu instancji klasy *String* nie może ulec zmianie:

```
string str = "Test";  
str[0] = 'X'; // BŁĄD
```

Wszelkie zmiany w łańcuchu wymagają zastosowania odpowiednich metod, właściwości i operatorów, które w rezultacie utworzą nową i zmodyfikowaną instancję klasy *String*.

Przykładowo dla łączenia łańcuchów:

```
string Napis1 = "Jesteś", Napis2 = "Fajny", Napis3;  
  
// utworzenie instancji Napis3 i zainicjowanie jej wartością Napis1  
Napis3 = Napis1;  
  
// utworzenie nowej instancji Napis3 i zainicjowanie jej połączoną wartością  
// poprzedniej wartości Napis3, znakiem spacji i wartością Napis2  
Napis3 += " " + Napis2;
```

### Pola, właściwości i metody klasy String

Klasa *String* oferuje wiele różnych użytecznych metod, pól i właściwości, z których można korzystać w czasie operowania na łańcuchach. Do najważniejszych i najczęściej używanych należą:

- *Empty* – (statyczne pole) reprezentacja pustego łańcucha znaków.

Przykład:

```
static string Etykieta(int Numer)  
{
```

```
if (Numer == 0)
    return "Sól";

if (Numer == 1)
    return "Cukier";

return String.Empty;
}
```

- **Length** – (właściwość) zawiera aktualną liczbę znaków w łańcuchu (długość łańcucha).

Przykład:

```
string str = "Test";

if (str.Length > 0)
    Console.WriteLine("Łańcuch: " + str);
else
    Console.WriteLine("Łańcuch jest pusty.");
```

- **Compare** – (metoda statyczna) porównuje leksykalnie dwa łańcuchy znaków. Porównanie odbywa się w oparciu o porządek alfabetyczny. Zwracana wartość z porównania może być: ujemna (w przypadku, gdy pierwszy napis znajduje się przed drugim), zerowa (gdy oba napisy są tożsame), lub dodatnia (w przypadku, gdy pierwszy napis znajduje się za drugim).

Przykład:

```
string str1 = "Ala";
string str2 = "Kot";

int wyn = String.Compare(str1, str2); // ujemna wartość
```

Metoda domyślnie porównuje łańcuchy uwzględniając wielkość znaków. W przypadku, gdy wielkość znaków ma być pomijana, przy porównaniu należy użyć innej postaci metody, gdzie jako trzeci parametr przekazuje się wartość **true** (w przypadku, gdy wielkość znaków ma być pomijana) lub **false** (w przypadku, gdy wielkość znaków ma być uwzględniana).

Przykład:

```
string str1 = "ALA";
string str2 = "ala";

// są równe, pomijana jest wielkość znaków
int wyn1 = String.Compare(str1, str2, true);
```

- **Concat** – (metoda statyczna) łączy dwa lub więcej łańcuchów w jeden.

Przykład:

```
string str1 = "Ala";
string str2 = " ma kota";

string strNapis = String.Concat(str1, str2); // "Ala ma kota"
string[] napisy = {"Ala", " ma ", "kota"};
string Napis = String.Concat(napisy); // "Ala ma kota"
```

- **Contains** – (metoda) sprawdza, czy łańcuch zawiera określony napis.

Przykład:

```
string str1 = "Te meble nie wyglądają zbyt imponująco";
```

```
string str2 = "meble";  
bool bZawiera = str1.Contains(str2); // true
```

- **Copy** – (metoda statyczna) tworzy nową instancję klasy **String** z identyczną zawartością.

Przykład:

```
string str1 = "Test";  
string str2 = String.Copy(str1); // "Test"
```

- **Format** – (metoda statyczna) zamienia odnośniki w łańcuchu formatującym na odpowiednią postać tekstową.

Przykład:

```
int Numer = 100;  
string napis = String.Format("Cena wynosi zł", Numer);
```

- **IndexOf** – (metoda) zwraca położenie pierwszego znaku lub początku podłańcucha pasującego do wzorca w danym łańcuchu.

Przykład:

```
string str1 = "Te meble nie wyglądają zbyt imponująco";  
string str2 = "meble";  
int pos;
```

```
pos = str1.IndexOf(str2); // pos = 3  
pos = str1.IndexOf('e'); // pos = 1
```

- **IndexOfAny** – (metoda) zwraca położenie któregośkolwiek ze znaków spośród tych wyszukiwanych w łańcuchu.

Przykład:

```
string str = "Te meble nie wyglądają zbyt imponująco";  
int pos = str.IndexOfAny(new char[]{'a','e','i'}); // pos = 1
```

- **IsNullOrEmpty** – (metoda statyczna) sprawdza, czy łańcuch nie został utworzony (czy nie utworzono instancji) lub czy istniejący łańcuch jest pusty.

Przykład:

```
string str1 = null;  
string str2 = String.Empty;
```

```
if (String.IsNullOrEmpty(str1) && String.IsNullOrEmpty(str2))  
    str1 = str2 = "Test";
```

- **LastIndexOf** – (metoda) zwraca położenie ostatniego znaku lub początku ostatniego podłańcucha pasującego do wzorca w danym łańcuchu.

Przykład:

```
string str1 = "Te meble nie wyglądają zbyt imponująco";  
string str2 = "meble";  
int pos;
```

```
pos = str1.LastIndexOf(str2); // pos = 3  
pos = str1.LastIndexOf('e'); // pos = 11
```

- **LastIndexOfAny** – (metoda) zwraca ostatnie położenie któregośkolwiek ze znaków spośród tych

wyszukiwanych w łańcuchu.

Przykład:

```
string str = "Te meble nie wyglądają zbyt imponująco";  
int pos = str.LastIndexOfAny(new char[]{'a','e','i'}); // pos = 28
```

- **PadLeft** – (metoda) uzupełnia łańcuch do zadanej długości z lewej strony zadanymi znakami (domyślnie znakami spacji). Ilość dopisanych znaków jest równa podanej wartości pomniejszonej o długość łańcucha (jeżeli jest mniejsza nic nie jest dopisywane).

Przykład:

```
string str1 = "napis1";  
string str2 = "napis2";  
  
str1 = str1.PadLeft(15); // "          napis1"  
str2 = str2.PadLeft(15, '.'); // ".....napis2"
```

- **PadRight** – (metoda) uzupełnia łańcuch do zadanej długości z prawej strony zadanymi znakami (domyślnie znakami spacji). Ilość dopisanych znaków jest równa podanej wartości pomniejszonej o długość łańcucha (jeżeli jest mniejsza nic nie jest dopisywane).

Przykład:

```
string str1 = "napis1";  
string str2 = "napis2";  
  
str1 = str1.PadRight(15); // "napis1      "  
str2 = str2.PadRight(15, '.'); // "napis2....."
```

- **Remove** – (metoda) usuwa określoną liczbę znaków, począwszy od znaku o zadanym indeksie. Jeżeli nie zostanie określona liczba znaków, przyjmuje się, że usunięte zostaną znaki od podanego indeksu do końca łańcucha.

Przykład:

```
string str = "0123...789", wynik;  
  
wynik = str.Remove(4); // "0123"  
wynik = str.Remove(4,3); // "0123789"
```

- **Replace** – (metoda) zamienia wszystkie wystąpienia określonego znaku lub podłańcucha znaków na zadane.

Przykład:

```
string str = "0123...789", wynik;  
  
wynik = str.Replace('.', 'X'); // "0123XXX789"  
wynik = str.Replace("0123...", "Numer "); // "Numer 789"
```

- **Split** – (metoda) dzieli łańcuch znaków na podłańcuchy w oparciu o podane znaki separujące.

Przykład:

```
string str = "uruchom /a /b:cc plik.txt";  
string[] Komendy = str.Split(' ');  
  
foreach (string komenda in Komendy)
```

```
Console.WriteLine(komenda);
```

Wynik:

```
uruchom
/a
/b:cc
plik.txt
```

- **Substring** – (metoda) zwraca fragment łańcucha znaków poczynając od elementu o określonym indeksie początkowym i określonej długości znaków (jeżeli nie zostanie określona długość, zwracany jest fragment od zadanego indeksu do końca łańcucha).

Przykład:

```
string str = "Te meble nie wyglądają zbyt imponująco";
string wynik1 = str.Substring(3,5); // "meble"
string wynik2 = str.Substring(28); // "imponująco"
```

- **ToCharArray** – (metoda) kopiuje elementy łańcucha do tablicy znaków.

Przykład:

```
string str = "Te meble nie wyglądają zbyt imponująco";
char[] znaki = str.ToCharArray();
```

- **ToLower** – (metoda) zamienia wszystkie duże litery w łańcuchu na małe.

Przykład:

```
string str = "ALA MA KOTA";
string wynik = str.ToLower(); // "ala ma kota"
```

- **ToUpper** – (metoda) zamienia wszystkie małe litery w łańcuchu na duże.

Przykład:

```
string str = "ala ma kota";
string wynik = str.ToUpper(); // "ALA MA KOTA"
```

- **Trim** – (metoda) usuwa z początku i końca łańcucha zadane znaki (domyślnie, jeżeli nie określi się jakie znaki mają być usunięte, usuwane są tylko spacje).

Przykład:

```
string str = "    ...kot...    ";
string wynik1 = str.Trim(); // "...kot..."
string wynik2 = str.Trim(new char[]{' ', '.'}); // "kot"
```

- **TrimStart** – (metoda) usuwa z początku łańcucha zadane znaki (domyślnie, jeżeli nie określi się jakie znaki mają być usunięte, usuwane są tylko spacje).

Przykład:

```
string str = "    ...kot...    ";
string wynik1 = str.Trim(); // "...kot...    "
string wynik2 = str.Trim(new char[]{' ', '.'}); // "kot...    "
```

- **TrimEnd** – (metoda) usuwa z początku i końca łańcucha zadane znaki (domyślnie, jeżeli nie określi się jakie znaki mają być usunięte, usuwane są tylko spacje).

Przykład:

```
string str = "    ...kot...    ";
string wynik1 = str.Trim(); // "    ...kot..."
string wynik2 = str.Trim(new char[]{' ', '.'}); // "    ...kot"
```



## Budowanie łańcuchów – klasa **StringBuilder**

Klasa **StringBuilder** została stworzona w celu modyfikowania łańcuchów znaków, bez potrzeby tworzenia nowej instancji klasy. Wszelkie zmiany wykonywane na instancji klasy **StringBuilder** modyfikują łańcuch bezpośrednio.

Aby móc operować na łańcuchu za pomocą metod i właściwości tej klasy, należy utworzyć jej instancję:

```
string napis = "Testowy napis";  
StringBuilder sbn = new StringBuilder(napis);
```

Klasa **StringBuilder** oferuje wiele różnych użytecznych metod, pól i właściwości, z których można korzystać w czasie operowania na łańcuchach. Do najważniejszych i najczęściej używanych należą:

- **Capacity** – (właściwość) pozwala odczytać lub ustawić maksymalną liczbę znaków, jaka może być przechowywana w pamięci zaalokowanej dla obiektu.

Przykład:

```
string napis = "Testowy napis";  
StringBuilder sbn = new StringBuilder(napis);
```

```
sbn.Capacity = 50; // maksymalnie 50 znaków
```

- **Length** – (właściwość) pozwala odczytać lub ustawić aktualną liczbę znaków przechowywanych w pamięci.

Przykład:

```
string napis = "Testowy napis";  
StringBuilder sbn = new StringBuilder(napis);
```

```
sbn.Length = 5;  
napis = sbn.ToString(); // "Testo"
```

- **Append** – (metoda) pozwala na dołączenie do łańcucha napisowej reprezentacji dowolnego typu danych.

Przykład:

```
string napis = "Testowy napis: ";  
StringBuilder sbn = new StringBuilder(napis);
```

```
sbn.Append(1700); // "Testowy napis: 1700"  
sbn.Append(8.8d); // "Testowy napis: 17008,8"
```

- **AppendFormat** – (metoda) pozwala na dołączenie do łańcucha napisowej reprezentacji dowolnego typu danych w określonym formacie (za użyciem ciągów formatujących).

Przykład:

```
string napis = "Cena";  
StringBuilder sbn = new StringBuilder(napis);  
  
sbn.AppendFormat(" ", 1700, "zł"); // "Cena 1700 zł"
```

- **AppendLine** – (metoda) pozwala dołączyć do łańcucha znak przejścia do nowej linii.

Przykład:

```
string napis = "Cena";
StringBuilder sbn = new StringBuilder(napis);

sbn.AppendLine();
sbn.Append("Inna cena");
Console.WriteLine(sbn.ToString());
```

Wynik:

```
Cena
Inna cena
```

- **Insert** – (metoda) pozwala wstawić do łańcucha napisową reprezentację dowolnego typu danych.

Przykład:

```
string napis = "Testowy napis: ";
StringBuilder sbn = new StringBuilder(napis);

sbn.Insert(2, 1700); // "Tel700stowy napis: "
sbn.Insert(7, 8.8d); // "Tel700s8,8towy napis: "
```

- **Remove** – (metoda) pozwala usunąć z łańcucha określoną liczbę znaków, począwszy od zadanego indeksu początkowego.

Przykład:

```
string napis = "Testowy napis: ";
StringBuilder sbn = new StringBuilder(napis);

sbn.Remove(7, 8); // "Testowy"
```

- **Replace** – (metoda) pozwala na zamianę wszystkich wystąpień znaków lub podłańcuchów wewnątrz łańcucha.

Przykład:

```
string napis = "Testowy napis: ";
StringBuilder sbn = new StringBuilder(napis);

sbn.Replace(":", " ->"); // "Testowy napis -> "
```

# Rozdział 14.

## Kolekcje

### Wprowadzenie

W języku C# kolekcja jest grupą elementów, w której każdy element jest obiektem. W .NET Framework można znaleźć szeroką gamę różnego rodzaju kolekcji: list, kolejek, stosów czy słowników. Kolekcje te znajdują się w przestrzeni nazw ***System.Collections***.

Do najpopularniejszych klas definiujących kolekcje z przestrzeni ***System.Collections*** należą:

- ***ArrayList*** – reprezentuje tablicę, której rozmiar może zostać zwiększony, kiedy zajdzie taka potrzeba;
- ***BitArray*** – reprezentuje tablicę wartości bitowych;
- ***Hashtable*** – reprezentuje kolekcję par klucz/wartość;
- ***Queue*** – reprezentuje kolejkę (FIFO – first in, first out);
- ***SortedList*** – reprezentuje listę posortowanych elementów;
- ***Stack*** – reprezentuje stos (LIFO – last in, first out).

### Klasa ArrayList

Klasa ***ArrayList*** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na dynamicznych tablicach. Do najważniejszych i najczęściej używanych należą:

- ***Capacity*** – (właściwość) pozwala odczytać lub ustawić aktualną liczbę elementów kolekcji.

Przykład:

```
ArrayList arl = new ArrayList();
```

```
arl.Capacity = 10;
```

- ***Count*** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji.

Przykład:

```
ArrayList arl = new ArrayList(10);
```

```
if (arl.Count > 0)
{
    ...
}
```

- ***IsFixedSize*** – (właściwość) pozwala sprawdzić, czy kolekcja ma ustawioną liczbę elementów.

Przykład:

```
ArrayList arl = new ArrayList(10);
```

```
if (arl.IsFixedSize)
{
    ...
}
```

```
}
```

- **Add** – (metoda) dodaje element na końcu kolekcji.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("ala");
```

```
wyrazy.Add("ma");
```

```
wyrazy.Add("kota");
```

- **AddRange** – (metoda) pozwala dołączyć do kolekcji inną kolekcję elementów.

Przykład:

```
ArrayList wyrazy1 = new ArrayList();
```

```
ArrayList wyrazy2 = new ArrayList();
```

```
wyrazy1.Add("ala");
```

```
wyrazy2.Add("ma");
```

```
wyrazy2.Add("kota");
```

```
wyrazy1.AddRange(wyrazy2); // dodanie zawartości kolekcji wyrazy2
```

- **Clear** – (metoda) usuwa wszystkie elementy z kolekcji.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("ala");
```

```
wyrazy.Add("ma");
```

```
wyrazy.Add("kota");
```

```
wyrazy.Clear(); // usunięcie elementów
```

- **Contains** – (metoda) pozwala określić, czy w kolekcji znajduje się element.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("ala");
```

```
wyrazy.Add("ma");
```

```
wyrazy.Add("kota");
```

```
if (wyrazy.Contains("ma")) // sprawdzenie czy zawiera wyraz
```

```
{
```

```
...
```

```
}
```

- **GetRange** – (metoda) zwraca podzbiór elementów danej kolekcji. Zwrócony podzbiór zawiera określoną liczbę elementów, począwszy od określonego indeksu (indeks podaje się jako pierwszy argument, a długość jako drugi).

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("ala");  
wyrazy.Add("ma");  
wyrazy.Add("kota");
```

```
ArrayList podzbior = wyrazy.GetRange(1, 2); // "ma", "kota"
```

• **IndexOf** – (metoda) zwraca numer indeksu pierwszego elementu pasującego do wyszukiwanego wzorca.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("kot");  
wyrazy.Add("pies");  
wyrazy.Add("pies");
```

```
int pos = wyrazy.IndexOf("pies"); // pos = 1
```

• **Insert** – (metoda) pozwala na wstawienie do kolekcji elementu na określonej indeksie pozycję.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("kot");  
wyrazy.Add("pies");  
wyrazy.Insert(0, "mysz"); // wyrazy = {"mysz", "kot", "pies"}
```

• **InsertRange** – (metoda) pozwala na wstawienie do kolekcji elementów na określonej indeksie pozycję.

```
ArrayList wyrazy1 = new ArrayList();
```

```
wyrazy1.Add("kot");  
wyrazy1.Add("pies");
```

```
ArrayList wyrazy2 = new ArrayList();
```

```
wyrazy2.Add("mysz");  
wyrazy1.InsertRange(0, wyrazy2); // wyrazy1 = {"mysz", "kot", "pies"}
```

• **LastIndexOf** – (metoda) zwraca numer indeksu ostatniego elementu pasującego do wyszukiwanego wzorca.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("kot");  
wyrazy.Add("pies");  
wyrazy.Add("pies");
```

```
int pos = wyrazy.LastIndexOf("pies"); // pos = 2
```

- **Remove** – (metoda) usuwa z kolekcji pierwszy element pasujący do wzorca.

Przykład:

```
ArrayList wyrazy = new ArrayList();

wyrazy.Add("kot");
wyrazy.Add("pies");
wyrazy.Add("krowa");
wyrazy.Add("pies");
wyrazy.Remove("pies"); // wyrazy = {"kot", "krowa", "pies"}
```

- **RemoveAt** – (metoda) usuwa z kolekcji element o określonym indeksie.

Przykład:

```
ArrayList wyrazy = new ArrayList();

wyrazy.Add("kot");
wyrazy.Add("pies");
wyrazy.Add("krowa");
wyrazy.Add("pies");
wyrazy.RemoveAt(1); // wyrazy = {"kot", "krowa", "pies"}
```

- **RemoveRange** – (metoda) usuwa z kolekcji określoną ilość elementów, począwszy od określonego indeksu początkowego.

Przykład:

```
ArrayList wyrazy = new ArrayList();

wyrazy.Add("kot");
wyrazy.Add("pies");
wyrazy.Add("krowa");
wyrazy.Add("pies");
wyrazy.RemoveRange(0, 3); // wyrazy = {"pies"}
```

- **Reverse** – (metoda) odwraca porządek (kolejność) elementów w kolekcji.

Przykład:

```
ArrayList wyrazy = new ArrayList();

wyrazy.Add("kot");
wyrazy.Add("pies");
wyrazy.Add("krowa");
wyrazy.Reverse(); // wyrazy = {"krowa", "pies", "kot"}
```

- **SetRange** – (metoda) kopiuje do kolekcji elementy innej kolekcji nadpisując istniejące, począwszy od określonego indeksu.

Przykład:

```
ArrayList wyrazy1 = new ArrayList();
ArrayList wyrazy2 = new ArrayList();

wyrazy1.Add("kot");
```

```
wyrazyl.Add("pies");
wyrazyl.Add("krowa");
wyrazy2.Add("kot");
wyrazy2.Add("mysz");
wyrazyl.SetRange(1, wyrazy2); // wyrazyl = {"kot", "kot", "mysz"}
```

- **Sort** – (metoda) sortuje elementy w kolekcji.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("mysz");
wyrazy.Add("pies");
wyrazy.Add("kot");
wyrazy.Sort(); // wyrazy = {"kot", "mysz", "pies"}
```

- **ToArray** – (metoda) kopiuje elementy kolekcji do nowej tablicy.

Przykład:

```
ArrayList wyrazy = new ArrayList();
```

```
wyrazy.Add("mysz");
wyrazy.Add("pies");
wyrazy.Add("kot");
```

```
string[] tabwyrazy = (string [])wyrazy.ToArray(typeof(string));
```

## Klasa BitArray

Klasa **BitArray** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na tablicach bitowych. Do najważniejszych i najczęściej używanych należą:

- **Count** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji.

Przykład:

```
BitArray btr = new BitArray(10);
```

```
if (btr.Count > 0)
{
    ...
}
```

- **Length** – (właściwość) pozwala ustawić i odczytać aktualną liczbę elementów kolekcji.

Przykład:

```
BitArray btr = new BitArray(0);
```

```
if (btr.Length == 0)
    btr.Length = 10;
```

- **And** – (metoda) wykonuje kompleksowe działanie iloczynu logicznego na dwóch tablicach bitowych.

Przykład:

```
BitArray BA1 = new BitArray(4);
BitArray BA2 = new BitArray(4);

BA1[0] = BA1[1] = BA2[0] = BA2[2] = false;
BA1[2] = BA1[3] = BA2[1] = BA2[3] = true;
BA1.And(BA2); // BA1 = {false, false, false, true}
```

- **Get** – (metoda) odczytuje wartość bitową elementu o określonym indeksie.

Przykład:

```
BitArray BA = new BitArray(4);

BA[0] = BA[1] = BA[2] = false;
BA[3] = true;

if (BA.Get(3)) // lub: if (BA[3])
{
    ...
}
```

- **Not** – (metoda) wykonuje kompleksowe działanie negacji logicznej.

Przykład:

```
BitArray BA = new BitArray(4);

BA[0] = BA[1] = BA[2] = false;
BA[3] = true;

BA = BA.Not(); // BA = {true, true, true, false}
```

- **Or** – (metoda) wykonuje kompleksowe działanie sumy logicznej na dwóch tablicach bitowych.

Przykład:

```
BitArray BA1 = new BitArray(4);
BitArray BA2 = new BitArray(4);

BA1[0] = BA1[1] = BA2[0] = BA2[2] = false;
BA1[2] = BA1[3] = BA2[1] = BA2[3] = true;
BA1.Or(BA2); // BA1 = {false, true, true, true}
```

- **Set** – (metoda) ustawia wartość bitową elementu o określonym indeksie.

Przykład:

```
BitArray BA = new BitArray(4);

BA.Set(0, false); // lub: BA[0] = false;
BA.Set(1, false); // lub: BA[1] = false;
BA.Set(2, false); // lub: BA[2] = false;
BA.Set(3, true); // lub: BA[3] = true;
```

- **SetAll** – (metoda) ustawia wartość wszystkich elementów naadaną.



Przykład:

```
BitArray BA = new BitArray(4);
```

```
BA.SetAll(false);
```

• **Xor** – (metoda) wykonuje kompleksowe działanie różnicy symetrycznej na dwóch tablicach bitowych.

Przykład:

```
BitArray BA1 = new BitArray(4);
```

```
BitArray BA2 = new BitArray(4);
```

```
BA1[0] = BA1[1] = BA2[0] = BA2[2] = false;
```

```
BA1[2] = BA1[3] = BA2[1] = BA2[3] = true;
```

```
BA1.Xor(BA2); // BA1 = {false, true, true, false}
```

## Klasa Hashtable

Klasa **Hashtable** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na tablicach haszujących. Do najważniejszych i najczęściej używanych należą:

• **Count** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji (liczba par klucz/wartość).

Przykład:

```
Hashtable htb = new Hashtable();
```

```
if (htb.Count > 0)
```

```
{
```

```
...
```

```
}
```

• **Keys** – (właściwość) zawiera kolekcję kluczy tablicy haszującej.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");
```

```
htb.Add("Ania", "pies");
```

```
foreach (object obj in htb.Keys)
```

```
    Console.WriteLine(obj.ToString());
```

Wynik:

```
Ala
```

```
Ania
```

• **Values** – (właściwość) zawiera kolekcję wartości tablicy haszującej.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");  
htb.Add("Ania", "pies");  
  
foreach (object obj in htb.Values)  
    Console.WriteLine(obj.ToString());
```

Wynik:

```
kot  
pies
```

- **Add** – (metoda) dodaje element ze specyficznym kluczem i wartością na końcu kolekcji.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");  
htb.Add("Ania", "pies");
```

- **Clear** – (metoda) usuwa zawartość kolekcji.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");  
htb.Add("Ania", "pies");  
htb.Clear();
```

- **Contains** – (metoda) sprawdza czy kolekcja zawiera określony klucz.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");  
htb.Add("Ania", "pies");
```

```
if (htb.Contains("Ania"))  
{  
    ...  
}
```

- **ContainsKey** – (metoda) sprawdza czy kolekcja zawiera określony klucz.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");  
htb.Add("Ania", "pies");
```

```
if (htb.ContainsKey("Ala"))  
{  
    ...  
}
```

- ***ContainsValue*** – (metoda) sprawdza czy kolekcja zawiera określoną wartość.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");
```

```
htb.Add("Ania", "pies");
```

```
if (htb.Contains("kot"))  
{  
    ...  
}
```

- ***Remove*** – (metoda) usuwa element klucz/wartość z kolekcji.

Przykład:

```
Hashtable htb = new Hashtable();
```

```
htb.Add("Ala", "kot");
```

```
htb.Add("Ania", "pies");
```

```
htb.Remove("Ala");
```

## Klasa Queue

Klasa ***Queue*** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na kolejkach. Do najważniejszych i najczęściej używanych należą:

- ***Count*** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji.

Przykład:

```
Queue que = new Queue();
```

```
if (que.Count > 0)  
{  
    ...  
}
```

- ***Clear*** – (metoda) usuwa zawartość kolekcji.

Przykład:

```
Queue que = new Queue();
```

```
que.Enqueue("ala");
```

```
que.Clear();
```

- ***Contains*** – (metoda) sprawdza czy w kolekcji znajduje się określony element.

Przykład:

```
Queue que = new Queue();
```

```
que.Enqueue("ala");
```

```
if (que.Contains("ala"))  
{  
    ...  
}
```

• **Dequeue** – (metoda) usuwa element z kolekcji i zwraca jego wartość (jeżeli nie ma żadnego elementu w kolekcji, wywołanie tej metody zwróci wyjątek).

Przykład:

```
Queue que = new Queue();  
  
que.Enqueue("ala");  
  
object obj = que.Dequeue();  
  
Console.WriteLine(obj.ToString());
```

Wynik:

ala

• **Enqueue** – (metoda) dodaje element do kolejki.

Przykład:

```
Queue que = new Queue();  
  
que.Enqueue("ala");
```

• **Peek** – (metoda) zwraca wartość pierwszego elementu w kolejce nie usuwając go.

Przykład:

```
Queue que = new Queue();  
  
que.Enqueue("ala");  
  
object obj = que.Peek();  
  
Console.WriteLine(obj.ToString());
```

Wynik:

ala

• **ToArray** – (metoda) kopiuje elementy kolekcji do nowej tablicy.

Przykład:

```
Queue que = new Queue();  
  
que.Enqueue("ala");  
  
object[] objs = que.ToArray();
```

## Klasa SortedList

Klasa **SortedList** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na listach uporządkowanych. Do najważniejszych i najczęściej używanych należą:

- **Capacity** – (właściwość) pozwala odczytać lub ustawić aktualną liczbę elementów kolekcji.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Capacity = 10;
```

- **Count** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji.

Przykład:

```
SortedList slt = new SortedList();
```

```
if (slt.Count > 0)
```

```
{  
    ...  
}
```

- **Keys** – (właściwość) zawiera kolekcję kluczy listy uporządkowanej.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
foreach (object obj in slt.Keys)  
    Console.WriteLine(obj.ToString());
```

Wynik:

Ala

Ania

- **Values** – (właściwość) zawiera kolekcję wartości listy uporządkowanej.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
foreach (object obj in slt.Values)  
    Console.WriteLine(obj.ToString());
```

Wynik:

kot

pies

- **Add** – (metoda) dodaje element ze specyficznym kluczem i wartością na końcu kolekcji.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

- **Clear** – (metoda) usuwa zawartość kolekcji.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
slt.Clear();
```

- **Contains** – (metoda) sprawdza czy kolekcja zawiera określony klucz.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
if (slt.Contains("Ania"))
```

```
{
```

```
    ...
```

```
}
```

- **ContainsKey** – (metoda) sprawdza czy kolekcja zawiera określony klucz.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
if (slt.ContainsKey("Ala"))
```

```
{
```

```
    ...
```

```
}
```

- **ContainsValue** – (metoda) sprawdza czy kolekcja zawiera określoną wartość.

Przykład:

```
SortedList slt = new SortedList();
```

```
slt.Add("Ala", "kot");
```

```
slt.Add("Ania", "pies");
```

```
if (slt.ContainsValue("kot"))
```

```
{  
    ...  
}
```

- **GetByIndex** – (metoda) zwraca wartość elementu kolekcji o określonym indeksie.

Przykład:

```
SortedList slt = new SortedList();  
  
slt.Add("Ala", "kot");  
slt.Add("Ania", "pies");  
  
object obj = slt.GetByIndex(1); // "pies"
```

- **GetKey** – (metoda) zwraca klucz elementu kolekcji o określonym indeksie.

Przykład:

```
SortedList slt = new SortedList();  
  
slt.Add("Ala", "kot");  
slt.Add("Ania", "pies");  
  
object obj = slt.GetKey(1); // "Ania"
```

- **GetKeyList** – (metoda) zwraca listę kluczy kolekcji.

Przykład:

```
SortedList slt = new SortedList();  
  
slt.Add("Ala", "kot");  
slt.Add("Ania", "pies");  
  
IList lst = slt.GetKeyList();  
  
foreach (object obj in lst)  
    Console.WriteLine(obj.ToString());
```

- **GetValueList** – (metoda) zwraca listę wartości kolekcji.

Przykład:

```
SortedList slt = new SortedList();  
  
slt.Add("Ala", "kot");  
slt.Add("Ania", "pies");  
  
IList lst = slt.GetValueList();  
  
foreach (object obj in lst)  
    Console.WriteLine(obj.ToString());
```

- **IndexOfKey** – (metoda) zwraca indeks klucza o określonej wartości.

Przykład:

```
SortedList slt = new SortedList();

slt.Add("Ala", "kot");
slt.Add("Ania", "pies");

int indeks = slt.IndexOfKey("Ala"); // 0
```

- **IndexOfValue** – (metoda) zwraca indeks wartości o określonej wartości.

Przykład:

```
SortedList slt = new SortedList();

slt.Add("Ala", "kot");
slt.Add("Ania", "pies");

int indeks = slt.IndexOfValue("kot"); // 0
```

- **Remove** – (metoda) usuwa z kolekcji element o określonym kluczu.

Przykład:

```
SortedList slt = new SortedList();

slt.Add("Ala", "kot");
slt.Add("Ania", "pies");
slt.Remove("Ala"); // slt = {"Ania\kot"}
```

- **RemoveAt** – (metoda) usuwa z kolekcji element o określonym indeksie.

Przykład:

```
SortedList slt = new SortedList();

slt.Add("Ala", "kot");
slt.Add("Ania", "pies");
slt.RemoveAt(0); // slt = {"Ania\kot"}
```

- **SetByIndex** – (metoda) zamienia wartość elementu o określonym indeksie

Przykład:

```
SortedList slt = new SortedList();

slt.Add("Ala", "kot");
slt.Add("Ania", "pies");
slt.SetByIndex(0, "mysz"); // slt = {"Ala\mysz", "Ania\pies"}
```

## Klasa Stack

Klasa **Stack** oferuje wiele różnych metod i właściwości, z których można korzystać w czasie operowania na stosach. Do najważniejszych i najczęściej używanych należą:

- **Count** – (właściwość) pozwala odczytać aktualną liczbę elementów kolekcji.



Przykład:

```
Stack stk = new Stack();
```

```
if (stk.Count > 0)
{
    ...
}
```

- **Clear** – (metoda) usuwa zawartość kolekcji.

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala");
stk.Push("Ania");
stk.Clear();
```

- **Contains** – (metoda) sprawdza czy kolekcja zawiera określony element.

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala");
stk.Push("Ania");
```

```
if (stk.Contains("Ala"))
{
    ...
}
```

- **Peek** – (metoda) zwraca wartość pierwszego elementu w kolejce nie usuwając go.

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala");
stk.Push("Ania");
```

```
object obj = stk.Peek();
```

```
Console.WriteLine(obj.ToString());
```

Wynik:

```
Ania
```

- **Pop** – (metoda) usuwa element z kolekcji i zwraca jego wartość (jeżeli nie ma żadnego elementu w kolekcji, wywołanie tej metody zwróci wyjątek).

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala");
```

```
stk.Push("Ania");

object obj = stk.Pop(); // stk = {"Ala"}

Console.WriteLine(obj.ToString());
```

Wynik:

Ania

- **Push** – (metoda) dodaje element do kolekcji czyniąc go pierwszym.

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala"); // pierwszy
stk.Push("Ania"); // teraz ten jest pierwszy
```

- **ToArray** – (metoda) kopiuje elementy kolekcji do nowej tablicy.

Przykład:

```
Stack stk = new Stack();
```

```
stk.Push("Ala");
stk.Push("Ania");
```

```
object[] objs = stk.ToArray(); // objs = {"Ania", "Ala"}
```

## Rozdział 15.

### Data i czas

W .NET Framework zdefiniowany został osobny typ danych ***DateTime*** reprezentujący datę i czas. Do najważniejszych składników tego typu należą:

- ***MaxValue*** – (pole statyczne) stała wartość równa 23:59:59.9999999, 31 Grudzień 9999 r. (100 nanosekund przed 00:00:00, 1 Stycznia 10 000).

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
if (dt < DateTime.MaxValue)
{
    ...
}
```

- ***MinValue*** – (pole statyczne) stała wartość równa 00:00:00.0000000, 1 Styczeń 0001 r.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
if (dt > DateTime.MinValue)
{
    ...
}
```

- ***Date*** – (właściwość) zwraca samą datę (czas ustawiany jest na 00:00:00).

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
DateTime data = dt.Date; // data = [2004-12-18 00:00:00]
```

- ***Day*** – (właściwość) zwraca dzień miesiąca.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
int dzien = dt.Day; // dzien = 18
```

- ***DayOfWeek*** – (właściwość) zwraca dzień tygodnia.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
DayOfWeek dzientyg = dt.DayOfWeek; // dzientyg = DayOfWeek.Saturday
```

```
switch (dzientyg)
{
    case DayOfWeek.Monday:
        Console.WriteLine("Poniedziałek");
        break;
    case DayOfWeek.Tuesday:
        Console.WriteLine("Wtorek");
}
```

```
        break;
    case DayOfWeek.Wednesday:
        Console.WriteLine("Środa");
        break;
    case DayOfWeek.Thursday:
        Console.WriteLine("Czwartek");
        break;
    case DayOfWeek.Friday:
        Console.WriteLine("Piątek");
        break;
    case DayOfWeek.Saturday:
        Console.WriteLine("Sobota");
        break;
    case DayOfWeek.Sunday:
        Console.WriteLine("Niedziela");
        break;
}
```

- **DayOfYear** – (właściwość) zwraca dzień roku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
int dzienroku = dt.DayOfYear; // dzienroku = 353
```

- **Hour** – (właściwość) zwraca godzinę.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
int godzina = dt.Hour; // godzina = 9
```

- **Kind** – (właściwość) zwraca rodzaj reprezentacji czasu (lokalny, utc lub nieokreślony).

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
DateTimeKind rodzaj = dt.Kind; // rodzaj = DateTimeKind.Unspecified
```

```
switch (rodzaj)
{
    case DateTimeKind.Local:
        Console.WriteLine("Czas lokalny");
        break;
    case DateTimeKind.Utc:
        Console.WriteLine("Czas uniwersalny UTC");
        break;
    case DateTimeKind.Unspecified:
        Console.WriteLine("Czas nieokreślony (ani lokalny ani UTC)");
        break;
}
```

- **Millisecond** – (właściwość) zwraca liczbę milisekund.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
int milisekunda = dt.Millisecond; // milisekunda = 0
```

- **Minute** – (właściwość) zwraca minutę.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
int minuta = dt.Minute; // minuta = 10
```

- **Month** – (właściwość) zwraca miesiąc.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
int miesiac = dt.Month; // miesiac = 12
```

- **Now** – (właściwość statyczna) zwraca aktualną datę i czas.

Przykład:

```
DateTime dt = DateTime.Now;
```

- **Second** – (właściwość) zwraca sekundę.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
int sekunda = dt.Second; // sekunda = 0
```

- **Ticks** – (właściwość) zwraca liczbę tików (liczba 100 nanosekundowych interwałów od 00:00:00, 1 Stycznia 0001 r.).

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
long tiki = dt.Ticks; // tiki = 632389578000000000
```

- **Today** – (właściwość statyczna) zwraca aktualną datę (czas ustawiany jest na 00:00:00).

Przykład:

```
DateTime dt = DateTime.Today;
```

- **UtcNow** – (właściwość statyczna) zwraca aktualną datę i czas jako czas uniwersalny UTC.

Przykład:

```
DateTime dt = DateTime.UtcNow;
```

- **Year** – (właściwość) zwraca rok.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
int rok = dt.Year; // rok = 2004
```

- **Add** – (metoda) dodaje do **DateTime** wartość typu **TimeSpan** (reprezentującego interwał czasowy).

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
  
// sp = [2 dni, 1 godzina, 1 minuta, 1 sekunda]  
TimeSpan sp = new TimeSpan(2, 1, 1, 1);
```

```
dt = dt.Add(sp); // dt = [2004-12-20 10:11:01]
```

- **AddDays** – (metoda) dodaje do **DateTime** określoną liczbę dni.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddDays(100); // dt = [2005-03-28 09:10:00]
```

- **AddHours** – (metoda) dodaje do **DateTime** określoną liczbę godzin.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddHours(24); // dt = [2004-12-19 09:10:00]
```

- **AddMilliseconds** – (metoda) dodaje do **DateTime** określoną liczbę milisekund.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddMilliseconds(1); // dt = [2004-12-18 09:10:00.0010000]
```

- **AddMinutes** – (metoda) dodaje do **DateTime** określoną liczbę minut.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddMinutes(10); // dt = [2004-12-18 09:20:00]
```

- **AddMonths** – (metoda) dodaje do **DateTime** określoną liczbę miesięcy.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddMonths(12); // dt = [2005-12-18 09:10:00]
```

- **AddSeconds** – (metoda) dodaje do **DateTime** określoną liczbę sekund.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddSeconds(20); // dt = [2004-12-18 09:10:20]
```

- **AddTicks** – (metoda) dodaje do **DateTime** określoną liczbę tików.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddTicks(10000000); // dt = [2004-12-18 09:10:01]
```

- **AddYears** – (metoda) dodaje do **DateTime** określoną liczbę lat.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
```

```
dt = dt.AddYear(10); // dt = [2014-12-18 09:10:00]
```

- **DaysInMonth** – (metoda statyczna) zwraca liczbę dni w danym miesiącu określonego roku.

Przykład:

```
int liczbadni = DateTime.DaysInMonth(2004, 02); // liczbadni = 29
```

• **FromBinary** – (metoda statyczna) zwraca datę i czas dla określonej wartości interwałów.

Przykład:

```
DateTime dt = DateTime.FromBinary(632389578000000000);  
// dt = [2004-12-18 09:10:00]
```

• **GetDateTimeFormats** – (metoda) konwertuje datę i czas do wszystkich formatów łańcuchowych obsługiwanych przez **DateTime**.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
string[] formatydat = dt.GetDateTimeFormats();
```

```
foreach (string format in formatydat)  
    Console.WriteLine(format);
```

Wynik:

```
2004-12-18  
04-12-18  
18 grudnia 2004  
18 grudnia 2004  
18 grudnia 2004 09:10  
18 grudnia 2004 9:10  
18 grudnia 2004 09:10  
18 grudnia 2004 9:10  
18 grudnia 2004 09:10:00  
18 grudnia 2004 9:10:00  
18 grudnia 2004 09:10:00  
18 grudnia 2004 9:10:00  
2004-12-18 09:10  
2004-12-18 9:10  
04-12-18 09:10  
04-12-18 9:10  
2004-12-18 09:10:00  
2004-12-18 9:10:00  
04-12-18 09:10:00  
04-12-18 9:10:00  
18 grudnia  
18 grudnia  
2004-12-18T09:10:00.0000000  
2004-12-18T09:10:00.0000000  
Sat, 18 Dec 2004 09:10:00 GMT  
Sat, 18 Dec 2004 09:10:00 GMT  
2004-12-18T09:10:00  
09:10
```

```
9:10
09:10:00
9:10:00
2004-12-18 09:10:00Z
18 grudnia 2004 08:10:00
18 grudnia 2004 8:10:00
18 grudnia 2004 08:10:00
18 grudnia 2004 8:10:00
grudzień 2004
grudzień 2004
```

- ***IsLeapYear*** – (metoda statyczna) określa czy dany rok jest rokiem przestępnym.

Przykład:

```
if (DateTime.IsLeapYear(2004))
    Console.WriteLine("Przestępny");
```

Wynik:

```
Przestępny
```

- ***Parse*** – (metoda statyczna) konwertuje łańcuch znaków zawierający datę i czas na ***DateTime***.

Przykład:

```
DateTime dt = DateTime.Parse("2004-12-18 9:10:00");
// dt = [2004-12-18 09:10:00]
```

- ***SpecifyKind*** – (metoda statyczna) zwraca nową instancję ***DateTime*** w innej reprezentacji.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
DateTime dtutc = DateTime.SpecifyKind(dt, DateTimeKind.Utc);
```

- ***Subtract*** – (metoda) odejmuje od daty i czasu określoną wartość.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);

dt = dt.Subtract(new TimeSpan(36, 0, 0, 0));
// dt = [2004-02-16 09:10:00]
```

- ***ToBinary*** – (metoda) zwraca binarną reprezentację czasu.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
long bindt = dt.ToBinary(); // bindt = 632389578000000000
```

- ***ToLocalTime*** – (metoda) konwertuje czas UTC do czasu lokalnego.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0, DateTimeKind.Utc);
DateTime dtloc = dt.ToLocalTime(); // dtloc = [2004-12-18 10:10:00]
```

- ***ToLongDateString*** – (metoda) konwertuje datę na długi format łańcuchowy.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);
string strdt = dt.ToLongDateString(); // "18 grudnia 2004"
```



- ***ToLongTimeString*** – (metoda) konwertuje czas na długi format łańcuchowy.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
string strdt = dt.ToLongTimeString(); // "09:10:00"
```

- ***ToShortDateString*** – (metoda) konwertuje datę na krótki format łańcuchowy.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
string strdt = dt.ToShortDateString(); // "2004-12-18"
```

- ***ToShortTimeString*** – (metoda) konwertuje czas na krótki format łańcuchowy.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0);  
string strdt = dt.ToShortTimeString(); // "09:10"
```

- ***ToUniversalTime*** – (metoda) konwertuje czas lokalny do czasu UTC.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 18, 9, 10, 0,  
                           DateTimeKind.Local);  
  
DateTime dtutc = dt.ToUniversalTime();  
// dtutc = [2004-12-18 08:10:00]
```

# Rozdział 16.

## Foldery i pliki

### Wprowadzenie

Operacje wejścia/wyjścia należą do podstawowych operacji, które każdy programista powinien znać. Język C# zawiera bogatą funkcjonalność, pozwalającą na odczyt i zapis plików oraz strumieni danych jak również wykonywanie prostych operacji na plikach i folderach. Funkcjonalność ta realizowana jest przez zestaw klas znajdujący się w przestrzeni nazw **System.IO**.

Do najczęściej używanych klas należą:

- **Directory** – zawiera zestaw statycznych metod do obsługi folderów (tworzenie, przenoszenie, listowanie, usuwanie);
- **DirectoryInfo** – zawiera zestaw metod instancyjnych do obsługi folderów (tworzenie, przenoszenie, listowanie, usuwanie);
- **File** – zawiera zestaw metod statycznych do obsługi plików (tworzenie, otwieranie, zamykanie, kopiowanie, przenoszenie, usuwanie);
- **FileInfo** – zawiera zestaw metod instancyjnych do obsługi plików (tworzenie, otwieranie, zamykanie, kopiowanie, przenoszenie, usuwanie);
- **FileStream** – pozwala na obsługę strumieni plików;
- **StreamReader** – wczytuje dane ze strumienia;
- **StreamWriter** – zapisuje dane do strumienia;
- **BinaryReader** – wczytuje dane w postaci binarnej;
- **BinaryWriter** – zapisuje dane w postaci binarnej;
- **Path** – pozwala na wykonywanie operacji na ścieżkach.

### Klasa Directory

Klasa **Directory** oferuje wiele różnych użytecznych metod. Do najważniejszych i najczęściej używanych należą:

- **CreateDirectory** – (metoda statyczna) tworzy nowy folder.

Przykład:

```
string sciezka = @"C:\Test";

if (!Directory.Exists(sciezka))
    Directory.CreateDirectory(sciezka);
```

- **Delete** – (metoda statyczna) usuwa pusty folder.

Przykład:

```
string sciezka = @"C:\Test";

if (Directory.Exists(sciezka))
    Directory.Delete(sciezka);
```

- ***Exists*** – (metoda statyczna) sprawdza czy istnieje dany folder.

Przykład:

```
string sciezka = @"C:\Test";
bool jest = Directory.Exists(sciezka);

Console.WriteLine("Folder istnieje", sciezka, jest ? "": "nie ");
```

- ***GetCreationTime*** – (metoda statyczna) zwraca czas utworzenia folderu.

Przykład:

```
string sciezka = @"C:\Test";
DateTime dt;

if (Directory.Exists(sciezka))
    dt = Directory.GetCreationTime(sciezka);
```

- ***GetCreationTimeUtc*** – (metoda statyczna) zwraca czas UTC utworzenia folderu.

Przykład:

```
string sciezka = @"C:\Test";
DateTime dt;

if (Directory.Exists(sciezka))
    dt = Directory.GetCreationTimeUtc(sciezka);
```

- ***GetCurrentDirectory*** – (metoda statyczna) zwraca aktualny katalog roboczy aplikacji.

Przykład:

```
string sciezka = Directory.GetCurrentDirectory();
```

- ***GetDirectories*** – (metoda statyczna) zwraca listę folderów.

Przykład:

```
string[] foldery = Directory.GetDirectories(@"C:\");

foreach (string folder in foldery)
    Console.WriteLine(folder);
```

- ***GetDirectoryRoot*** – (metoda statyczna) zwraca katalog macierzysty.

Przykład:

```
Console.WriteLine(Directory.GetDirectoryRoot(@"c:\windows")); // "c:\"
```

- ***GetFiles*** – (metoda statyczna) zwraca listę plików w folderze.

Przykład:

```
string[] pliki = Directory.GetFiles(@"C:\");

foreach (string plik in pliki)
    Console.WriteLine(plik);
```

- ***GetFileSystemEntries*** – (metoda statyczna) zwraca listę folderów i plików.

Przykład:

```
string[] elementy = Directory.GetFileSystemEntries(@"C:\");

foreach (string element in elementy)
```

```
Console.WriteLine(element);
```

- **GetLastAccessTime** – (metoda statyczna) zwraca datę i czas ostatniego dostępu do folderu lub pliku.

Przykład:

```
DateTime lst = Directory.GetLastAccessTime(@"C:\readme.txt");
```

- **GetLastAccessTimeUtc** – (metoda statyczna) zwraca datę i czas UTC ostatniego dostępu do folderu lub pliku.

Przykład:

```
DateTime lst = Directory.GetLastAccessTimeUtc(@"C:\readme.txt");
```

- **GetLastWriteTime** – (metoda statyczna) zwraca datę i czas ostatniego zapisu w folderze lub pliku.

Przykład:

```
DateTime lst = Directory.GetLastWriteTime(@"C:\readme.txt");
```

- **GetLastWriteTimeUtc** – (metoda statyczna) zwraca datę i czas UTC ostatniego zapisu w folderze lub pliku.

Przykład:

```
DateTime lst = Directory.GetLastWriteTimeUtc(@"C:\readme.txt");
```

- **GetLogicalDrives** – (metoda statyczna) zwraca listę logicznych napędów (w formacie: „<LITERA\_NAPĘDU>:\”).

Przykład:

```
string[] napedy = Directory.GetLogicalDrives();
```

```
foreach(string naped in napedy)
```

```
    Console.WriteLine(naped);
```

- **GetParent** – (metoda statyczna) zwraca nazwę folderu nadrzędnego.

Przykład:

```
DirectoryInfo parent = Directory.GetParent(@"C:\Windows");
```

```
Console.WriteLine(parent.Name); // C:\
```

- **SetCreationTime** – (metoda statyczna) zmienia datę i czas utworzenia folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 0);
```

```
Directory.SetLastWriteTime(@"C:\readme.txt", dt);
```

- **SetCreationTimeUtc** – (metoda statyczna) zmienia datę i czas UTC utworzenia folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 0, DateTimeKind.Utc);
```

```
Directory.SetLastWriteTimeUtc(@"C:\readme.txt", dt);
```

- **SetCurrentDirectory** – (metoda statyczna) ustawia aktualny katalog roboczy aplikacji.

Przykład:

```
Directory.SetCurrentDirectory(@"C:\");
```

- ***SetLastAccessTime*** – (metoda statyczna) zmienia datę i czas ostatniego dostępu do folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 0);
```

```
Directory.SetLastAccessTime(@"C:\readme.txt", dt);
```

- ***SetLastAccessTimeUtc*** – (metoda statyczna) zmienia datę i czas UTC ostatniego dostępu do folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 0, DateTimeKind.Utc);
```

```
Directory.SetLastAccessTimeUtc(@"C:\readme.txt", dt);
```

- ***SetLastWriteTime*** – (metoda statyczna) zmienia datę i czas ostatniego zapisu do folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 0);
```

```
Directory.SetLastWriteTime(@"C:\readme.txt", dt);
```

- ***SetLastWriteTimeUtc*** – (metoda statyczna) zmienia datę i czas UTC ostatniego zapisu do folderu lub pliku.

Przykład:

```
DateTime dt = new DateTime(2004, 12, 19, 13, 22, 10, DateTimeKind.Utc);
```

```
Directory.SetLastWriteTimeUtc(@"C:\readme.txt", dt);
```

## Klasa **DirectoryInfo**

Klasa **DirectoryInfo** oferuje wiele różnych użytecznych właściwości i metod. Do najważniejszych i najczęściej używanych należą:

- ***Attributes*** – (właściwość) zwraca lub ustawia atrybuty folderu lub pliku (typ wyliczeniowy **FileAttributes**).

Typ wyliczeniowy **FileAttributes** dopuszcza następujące wartości:

- ***Archive*** – plik archiwalny (aplikacje używają tego statusu, aby zaznaczyć plik do zarchiwizowania lub usunięcia);
- ***Compressed*** – plik skompresowany;
- ***Device*** – urządzenie (nie używane);
- ***Directory*** – folder;
- ***Encrypted*** – plik lub folder zaszyfrowany;
- ***Hidden*** – plik ukryty (nie jest wyświetlany podczas standardowego listowania plików);
- ***Normal*** – plik nie ma ustawionych atrybutów;
- ***NotContentIndexed*** – plik nie indeksowany przez systemową usługę indeksowania;

- **Offline** – plik typu offline (dane nie są dostępne od razu);
- **ReadOnly** – plik tylko do odczytu;
- **ReparsePoint** – blok danych zdefiniowany przez użytkownika, który skojarzony jest z plikiem lub folderem;
- **SparseFile** – duże pliki zazwyczaj wypełnione zerami;
- **System** – plik systemowy (element systemu operacyjnego lub element wykorzystywany przez system operacyjny);
- **Temporary** – plik tymczasowy.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
dif.Attributes = FileAttributes.ReadOnly | FileAttributes.Archive;
```

- **CreationTime** – (właściwość) zwraca lub ustawia datę i czas utworzenia folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
DateTime dt = dif.CreationTime;
```

- **CreationTimeUtc** – (właściwość) zwraca lub ustawia datę i czas UTC utworzenia folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
DateTime dt = dif.CreationTimeUtc;
```

- **Exists** – (właściwość) sprawdza czy istnieje folder.

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Windows");
```

```
if (dif.Exists)
{
    ...
}
```

- **Extension** – (właściwość) zwraca rozszerzenie pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
string ext = dif.Extension; // ext = ".txt"
```

- **FullName** – (właściwość) zwraca pełną ścieżkę folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
Console.WriteLine(dif.FullName); // C:\readme.txt
```

- **LastAccessTime** – (właściwość) zwraca lub ustawia datę i czas ostatniego dostępu do folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
Console.WriteLine(dif.LastAccessTime.ToString());
```

• ***LastAccessTimeUtc*** – (właściwość) zwraca lub ustawia datę i czas UTC ostatniego dostępu do folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
Console.WriteLine(dif.LastAccessTimeUtc.ToString());
```

• ***LastWriteTime*** – (właściwość) zwraca lub ustawia datę i czas ostatniego zapisu w folderze lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
Console.WriteLine(dif.LastWriteTime.ToString());
```

• ***LastWriteTimeUtc*** – (właściwość) zwraca lub ustawia datę i czas ostatniego zapisu w folderze lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\readme.txt");
```

```
Console.WriteLine(dif.LastWriteTimeUtc.ToString());
```

• ***Name*** – (właściwość) zwraca nazwę folderu lub pliku.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Windows");
```

```
Console.WriteLine(dif.Name); // Windows
```

• ***Parent*** – (właściwość) zwraca nazwę folderu nadrzędnego.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Windows\System32");
```

```
Console.WriteLine(dif.Name); // Windows
```

• ***Root*** – (właściwość) zwraca nazwę nadrzędną.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Windows\System32");
```

```
Console.WriteLine(dif.Root); // C:\
```

• ***Create*** – (metoda) tworzy folder.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Test");
```

```
dif.Create();
```

• ***CreateSubdirectory*** – (metoda) tworzy podfolder.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Test");
```

```
dif.CreateSubdirectory("Test_1");
```

- **Delete** – (metoda) usuwa folder.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Test");
```

```
if (dif.Exists)
```

```
    dif.Delete();
```

- **GetDirectories** – (metoda) zwraca listę folderów.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\");
```

```
DirectoryInfo[] foldery = dif.GetDirectories();
```

```
foreach (DirectoryInfo folder in foldery)
```

```
    Console.WriteLine(folder.FullName);
```

- **GetFiles** – (metoda) zwraca listę plików.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\");
```

```
FileInfo[] pliki = dif.GetFiles();
```

```
foreach (FileInfo plik in pliki)
```

```
    Console.WriteLine(plik.FullName);
```

- **GetFileSystemInfos** – (metoda) zwraca listę folderów i plików.

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\");
```

```
FileSystemInfo[] folderypliki = dif.GetFileSystemInfos();
```

```
foreach (FileSystemInfo folderplik in folderypliki)
```

```
    Console.WriteLine(folderplik.FullName);
```

**MoveTo** – (metoda) przenosi folder w inne miejsce lub zmienia jego nazwę, jeżeli znajduje się na tym samym dysku logicznym

Przykład:

```
DirectoryInfo dif = new DirectoryInfo(@"C:\Test");
```

```
dif.MoveTo(@"C:\TestX");
```

## Klasa File

Klasa **File** oferuje wiele różnych użytecznych metod. Do najważniejszych i najczęściej używanych należą:

- **AppendAll** – (metoda statyczna) dodaje łańcuch na końcu pliku (jeżeli podany plik nie istnieje zostanie utworzony).

Przykład:

```
File.AppendAll(@"C:\readme.txt", "Tekst");
```



- ***AppendText*** – (metoda statyczna) otwiera strumień do zapisu dla istniejącego pliku i ustawia się na jego końcu.

Przykład:

```
StreamWriter swr = File.AppendText(@"C:\readme.txt");
```

```
swr.WriteLine("Tekst");
```

```
swr.Close();
```

- ***Copy*** – (metoda statyczna) kopiuje plik we wskazane miejsce.

Przykład:

```
File.Copy(@"C:\readme.txt", @"C:\nowy.txt"); // bez nadpisywania
```

```
File.Copy(@"C:\readme.txt", @"C:\nowy.txt", true); // z nadpisaniem
```

- ***Create*** – (metoda statyczna) tworzy plik i skojarzony z nim strumień plikowy.

Przykład:

```
FileStream fstr = File.Create(@"C:\readme.txt");
```

```
fstr.WriteByte(65);
```

```
fstr.Close();
```

- ***CreateText*** – (metoda statyczna) tworzy plik tekstowy UTF-8 i skojarzony z nim strumień do zapisu.

Przykład:

```
StreamWriter strw = File.Create(@"C:\readme.txt");
```

```
strw.Write("Tekst");
```

```
strw.Close();
```

- ***Delete*** – (metoda statyczna) usuwa plik.

Przykład:

```
File.Delete(@"C:\readme.txt");
```

- ***Exists*** – (metoda statyczna) sprawdza czy istnieje plik.

Przykład:

```
if (File.Exists(@"C:\readme.txt"))
```

```
{
```

```
...
```

```
}
```

- ***GetAttributes*** – (metoda statyczna) zwraca atrybuty pliku.

Przykład:

```
FileAttributes atr = File.GetAttributes(@"C:\readme.txt");
```

```
if ((atr & FileAttributes.Hidden) == FileAttributes.Hidden)
```

```
{
```

```
...
```

```
}
```

- ***GetCreationTime*** – (metoda statyczna) zwraca datę i czas utworzenia pliku.

Przykład:

```
DateTime dt = File.GetCreationTime(@"C:\readme.txt");
```

- **GetCreationTimeUtc** – (metoda statyczna) zwraca datę i czas UTC utworzenia pliku.

Przykład:

```
DateTime dt = File.GetCreationTimeUtc(@"C:\readme.txt");
```

- **GetLastAccessTime** – (metoda statyczna) zwraca datę i czas ostatniego dostępu do pliku.

Przykład:

```
DateTime dt = File.GetLastAccessTime(@"C:\readme.txt");
```

- **GetLastAccessTimeUtc** – (metoda statyczna) zwraca datę i czas UTC ostatniego dostępu do pliku.

Przykład:

```
DateTime dt = File.GetLastAccessTimeUtc(@"C:\readme.txt");
```

- **GetLastWriteTime** – (metoda statyczna) zwraca datę i czas ostatniego zapisu do pliku.

Przykład:

```
DateTime dt = File.GetLastWriteTime(@"C:\readme.txt");
```

- **GetLastWriteTimeUtc** – (metoda statyczna) zwraca datę i czas UTC ostatniego zapisu do pliku.

Przykład:

```
DateTime dt = File.GetLastWriteTimeUtc(@"C:\readme.txt");
```

- **Move** – (metoda statyczna) przenosi plik w inne miejsce lub zmienia jego nazwę.

Przykład:

```
File.Move(@"C:\readme.txt", @"C:\readme.bak");
```

- **Open** – (metoda statyczna) otwiera plik w określonym trybie otwarcia pliku (typ wyliczeniowy **FileMode**) i dostępu do pliku (typ wyliczeniowy **FileAccess**).

Typ wyliczeniowy **FileMode** może przyjmować wartości:

- **Append** – otwiera plik i ustawia się na jego końcu jeżeli istnieje, lub tworzy nowy plik, jeżeli nie istnieje;
- **Create** – utworzenie pliku (jeżeli plik istnieje zostanie nadpisany);
- **CreateNew** – utworzenie pliku (jeżeli plik istnieje wyrzucany jest wyjątek **IOException**);
- **Open** – otwarcie pliku (jeżeli plik nie istnieje wyrzucany jest wyjątek **FileNotFoundException**);
- **OpenOrCreate** – otwarcie pliku lub (jeżeli nie istnieje) utworzenie go;
- **Truncate** – otwarcie istniejącego pliku i wyzerowanie go.

Typ wyliczeniowy **FileAccess** może przyjmować wartości (jeżeli tryb dostępu nie zostanie określony, domyślnie przyjmowany jest dostęp do odczytu i zapisu):

- **Read** – dostęp tylko do odczytu;
- **ReadWrite** – dostęp do odczytu i zapisu;
- **Write** – dostęp tylko do zapisu.

Przykład:

```
using (FileStream fstr = File.Open(@"c:\readme.txt", FileMode.Create))
{
    byte[] znaki = new UTF8Encoding(true).GetBytes("Linia testowa.");
    fstr.Write(znaki, 0, znaki.Length);
}
```

```
using (FileStream fstr = File.Open(@"c:\readme.txt", FileMode.Open,
                                   FileAccess.Read))
{
    byte[] znaki = new byte[1024];
    UTF8Encoding kodowanie = new UTF8Encoding(true);

    while (fstr.Read(znaki, 0, znaki.Length) > 0)
        Console.WriteLine(kodowanie.GetString(znaki));
}
```

• **OpenRead** – (metoda statyczna) otwiera plik tylko do odczytu.

Przykład:

```
using (FileStream fstr = File.OpenRead(@"c:\readme.txt"))
{
    byte[] znaki = new byte[1024];
    UTF8Encoding kodowanie = new UTF8Encoding(true);

    while (fstr.Read(znaki, 0, znaki.Length) > 0)
    {
        Console.WriteLine(kodowanie.GetString(znaki));
    }
}
```

• **OpenText** – (metoda statyczna) otwiera plik w trybie tekstowym.

Przykład:

```
using (StreamReader sr = File.OpenText(@"c:\readme.txt"))
{
    string s = "";

    while ((s = sr.ReadLine()) != null)
        Console.WriteLine(s);
}
```

• **OpenWrite** – (metoda statyczna) otwiera plik tylko do zapisu.

Przykład:

```
using (FileStream fstr = File.OpenWrite(@"c:\readme.txt"))
{
    byte[] tekst = new UTF8Encoding(true).GetBytes("Jakiś tekst...");

    fstr.Write(tekst, 0, tekst.Length);
}
```

• **ReadAll** – (metoda statyczna) otwiera plik, wczytuje całą zawartość i zamyka plik.

Przykład:

```
string tekst = File.ReadAll(@"c:\readme.txt");
```

• **ReadAllBytes** – (metoda statyczna) otwiera plik w trybie binarnym, wczytuje całą zawartość i zamyka plik.

Przykład:

```
byte[] dane = File.ReadAllBytes(@"c:\readme.dat");
```

- ***ReadAllLines*** – (metoda statyczna) otwiera plik, wczytuje wszystkie linie i zamyka plik.

Przykład:

```
string[] linie = File.ReadAllLines(@"c:\readme.txt");
```

- ***SetAttributes*** – (metoda statyczna) ustawia atrybuty pliku.

Przykład:

```
string sciezka = @"c:\readme.txt";
```

```
File.SetAttributes(sciezka, File.GetAttributes(sciezka) |  
                    FileAttributes.Hidden);
```

- ***SetCreationTime*** – (metoda statyczna) ustawia datę i czas utworzenia pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 29, 0);
```

```
File.SetCreationTime(@"C:\readme.txt", dt);
```

- ***SetCreationTimeUtc*** – (metoda statyczna) ustawia datę i czas UTC utworzenia pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 29, 10, DateTimeKind.Utc);
```

```
File.SetCreationTimeUtc(@"C:\readme.txt", dt);
```

- ***SetLastAccessTime*** – (metoda statyczna) ustawia datę i czas ostatniego dostępu do pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 30, 0);
```

```
File.SetLastAccessTime(@"C:\readme.txt", dt);
```

- ***SetLastAccessTimeUtc*** – (metoda statyczna) ustawia datę i czas UTC ostatniego dostępu do pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 31, 0, DateTimeKind.Utc);
```

```
File.SetLastAccessTimeUtc(@"C:\readme.txt", dt);
```

- ***SetLastWriteTime*** – (metoda statyczna) ustawia datę i czas ostatniego zapisu do pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 31, 0);
```

```
File.SetLastWriteTime(@"C:\readme.txt", dt);
```

- ***SetLastWriteTimeUtc*** – (metoda statyczna) ustawia datę i czas UTC ostatniego zapisu do pliku.

Przykład:

```
DateTime dt = new DateTime(2005, 1, 1, 13, 31, 20, DateTimeKind.Utc);
```

```
File.SetLastWriteTimeUtc(@"C:\readme.txt", dt);
```

- **WriteAll** – (metoda statyczna) otwiera plik, zapisuje całą zawartość i zamyka plik.

Przykład:

```
File.WriteAll(@"c:\readme.txt", "Linia 1\r\nLinia 2\r\nLinia 3");
```

- **WriteAllBytes** – (metoda statyczna) otwiera plik w trybie binarnym, zapisuje całą zawartość i zamyka plik.

Przykład:

```
byte[] dane = new byte[128];
```

```
for (byte b = 0; b < 128; b++)
```

```
    dane[b] = b;
```

```
File.WriteAllBytes(@"c:\readme.dat", dane);
```

- **WriteAllLines** – (metoda statyczna) otwiera plik, zapisuje wszystkie linie i zamyka plik.

Przykład:

```
string[] linie = new string[4];
```

```
for (int i = 0; i < 4; i++)
```

```
    linie[i] = String.Format("Linia ", i + 1);
```

```
File.WriteAllLines(@"c:\readme.txt", linie);
```

## Klasa FileInfo

Klasa **FileInfo** oferuje wiele różnych użytecznych właściwości i metod. Do najważniejszych i najczęściej używanych należą:

- **Attributes** – (właściwość) zwraca lub ustawia atrybuty pliku (typ wyliczeniowy **FileAttributes**).

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
if ((fin.Attributes & FileAttributes.Hidden) ==
```

```
    FileAttributes.Hidden)
```

```
{
```

```
    ...
```

```
}
```

- **CreationTime** – (właściwość) zwraca lub ustawia datę i czas utworzenia pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
DateTime dt = fin.CreationTime;
```

- **CreationTimeUtc** – (właściwość) zwraca lub ustawia datę i czas UTC utworzenia pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
DateTime dt = fin.CreationTimeUtc;
```

- **Directory** – (właściwość) pobiera instancję folderu nadrzędnego.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
DirectoryInfo dif = fin.Directory;
```

- **DirectoryName** – (właściwość) pobiera łańcuch reprezentujący pełną ścieżkę do folderu.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

if (fin.Exists)
    Console.WriteLine(fin.DirectoryName);
```

- **Exists** – (właściwość) sprawdza czy istnieje plik.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

if (fin.Exists)
{
    ...
}
```

- **Extension** – (właściwość) zwraca rozszerzenie pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

Console.WriteLine(fin.Extension); // ".txt"
```

- **FullName** – (właściwość) zwraca pełną ścieżkę do folderu lub pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

Console.WriteLine(fin.FullName); // " c:\readme.txt "
```

- **LastAccessTime** – (właściwość) zwraca lub ustawia datę i czas ostatniego dostępu do pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
DateTime dt = fin.LastAccessTime;
```

- **LastAccessTimeUtc** – (właściwość) zwraca lub ustawia datę i czas UTC ostatniego dostępu do pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
DateTime dt = fin.LastAccessTimeUtc;
```

- **Length** – (właściwość) zwraca długość pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
long dl = fin.Length;
```

- **Name** – (właściwość) zwraca nazwę pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

Console.WriteLine(fin.Name); // "readme.txt"
```

- **AppendText** – (metoda) pozwala na dołączenie tekstu do pliku.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

if (fin.Exists)
{
    using (StreamWriter sw = fin.AppendText())
    {
        sw.WriteLine("Kolejna linia.");
        sw.WriteLine("Jeszcze inna linia.");
    }
}
```

- **CopyTo** – (metoda) kopiuje istniejący plik do nowego.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

fin.CopyTo(@"c:\readme.bak"); // bez nadpisywania
fin.CopyTo(@"c:\readme.bak", true); // z nadpisywaniem
```

- **Create** – (metoda) tworzy nowy plik.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
FileStream fstr = fin.Create();

fstr.Close();
```

- **CreateText** – (metoda) tworzy nowy plik tekstowy.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");

if (!fin.Exists)
{
    using (StreamWriter sw = fin.CreateText())
    {
        sw.WriteLine("Linia 1");
        sw.WriteLine("Linia 2");
    }
}
```

- **Delete** – (metoda) usuwa istniejący plik.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
if (fin.Exists)
    fin.Delete();
```

- **MoveTo** – (metoda) przenosi istniejący plik lub zmienia jego nazwę.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
if (fin.Exists)
    fin.MoveTo(@"c:\readme.bak");
```

- **Open** – (metoda) otwiera plik w określonym trybie otwarcia pliku (typ wyliczeniowy **FileMode**) i dostępu do pliku (typ wyliczeniowy **FileAccess**).

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
using (FileStream fstr = fin.Open(FileMode.Open))
{
    byte[] znaki = new byte[1024];
    UTF8Encoding kodowanie = new UTF8Encoding(true);

    while (fstr.Read(znaki, 0, znaki.Length) > 0)
        Console.WriteLine(kodowanie.GetString(znaki));
}
```

- **OpenRead** – (metoda) otwiera plik tylko do odczytu.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
using (FileStream fstr = fin.OpenRead())
{
    byte[] znaki = new byte[1024];
    UTF8Encoding kodowanie = new UTF8Encoding(true);

    while (fstr.Read(znaki, 0, znaki.Length) > 0)
        Console.WriteLine(kodowanie.GetString(znaki));
}
```

- **OpenText** – (metoda) otwiera plik w trybie tekstowym.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");
```

```
using (StreamReader sr = fin.OpenText())
{
    string s = "";

    while ((s = sr.ReadLine()) != null)
```



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

- **OpenWrite** – (metoda) otwiera plik tylko do zapisu.

Przykład:

```
FileInfo fin = new FileInfo(@"c:\readme.txt");  
  
using (FileStream fstr = fin.OpenWrite())  
{  
    byte[] tekst = new UTF8Encoding(true).GetBytes("Jakiś tekst...");  
  
    fstr.Write(tekst, 0, tekst.Length);  
}
```

## Klasa *FileStream*

Klasa ***FileStream*** oferuje wiele różnych użytecznych właściwości i metod. Do najważniejszych i najczęściej używanych należą:

- **CanRead** – (właściwość) sprawdza czy strumień wspiera operację odczytu.

Przykład:

```
FileStream fstr = new FileStream(@"c:\readme.txt",  
                                FileMode.OpenOrCreate,  
                                FileAccess.Read);  
  
if (fstr.CanRead)  
{  
    ...  
}
```

- **CanSeek** – (właściwość) sprawdza czy strumień wspiera operację zmiany pozycji.

Przykład:

```
FileStream fstr = File.Create(@"c:\readme.txt");  
  
if (fstr.CanSeek)  
{  
    ...  
}
```

- **CanWrite** – (właściwość) sprawdza czy strumień wspiera operację zapisu.

Przykład:

```
FileStream fstr = new FileStream(@"c:\readme.txt",  
                                FileMode.OpenOrCreate,  
                                FileAccess.Write);  
  
if (fstr.CanWrite)  
{
```

```
...  
}
```

- **Length** – (właściwość) zwraca długość strumienia.

Przykład:

```
FileStream fstr = new FileStream(@"c:\readme.txt",  
                                FileMode.Open,  
                                FileAccess.Read);  
  
long rozmiar = fstr.Length;
```

- **Name** – (właściwość) zwraca nazwę pliku skojarzonego ze strumieniem.

Przykład:

```
FileStream fstr = File.Open(@"c:\readme.txt", FileMode.Open);  
string Nazwa = fstr.Name;
```

- **Position** – (właściwość) zwraca lub ustawia pozycję w strumieniu.

Przykład:

```
FileStream fstr = File.Open(@"c:\readme.txt", FileMode.Open);  
  
fstr.Position = 100L;
```

- **Close** – (metoda) zamyka plik i strumień z nim skojarzony.

Przykład:

```
FileStream fstr = new FileStream(@"c:\readme.txt",  
                                FileMode.OpenOrCreate,  
                                FileAccess.Write);  
  
fstr.Close();
```

- **Read** – (metoda) pozwala na odczytanie bloku danych do bufora.

Przykład:

```
FileInfo fi = new FileInfo(@"c:\readme.txt");  
FileStream fstr = fi.OpenRead();  
int dlugosc = 100;  
byte[] znaki = new byte[dlugosc];  
int nBytesRead = fstr.Read(znaki, 0, dlugosc);
```

- **ReadByte** – (metoda) pozwala na odczytanie pojedynczego bajtu.

Przykład:

```
FileInfo fi = new FileInfo(@"c:\readme.txt");  
FileStream fstr = fi.OpenRead();  
int znak = fstr.ReadByte();
```

- **Seek** – (metoda) pozwala na ustalenie pozycji w strumieniu (względem określonej lokalizacji – typ wyliczeniowy **SeekOrigin**).

Typ wyliczeniowy **SeekOrigin** może przyjmować wartości:

- **Begin** – pozycja względem początku strumienia;
- **Current** – pozycja względem aktualnej pozycji strumienia;
- **End** – pozycja względem końca strumienia.

Przykład:

```
FileInfo fi = new FileInfo(@"c:\readme.txt");
FileStream fstr = fi.OpenRead();

fstr.Seek(0, SeekOrigin.Begin);
```

- **Write** – (metoda) pozwala na zapis bloku danych z bufora.

Przykład:

```
FileInfo fi = new FileInfo(@"c:\readme.txt");
FileStream fstr = fi.OpenWrite();
int dlugosc = 100;
byte[] znaki = new byte[dlugosc];

fstr.Write(znaki, 0, dlugosc);
```

- **WriteByte** – (metoda) pozwala na zapis pojedynczego bajta.

Przykład:

```
FileInfo fi = new FileInfo(@"c:\readme.txt");
FileStream fstr = fi.OpenWrite();
byte znak = 65;

fstr.WriteByte(znak);
```

## Klasa StreamReader

Klasa **StreamReader** oferuje wiele różnych użytecznych pól, właściwości i metod. Do najważniejszych i najczęściej używanych należą:

- **Null** – (pole statyczne) pusty strumień.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");

if (sr != StreamReader.Null)
{
    ...
}
```

- **EndOfStream** – (właściwość) sprawdza czy nastąpiło odwołanie poza strumień.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");

while (!sr.EndOfStream)
    Console.WriteLine(sr.ReadLine());
```

- **Close** – (metoda) zamyka plik i strumień z nim skojarzony.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");
```

```
sr.Close();
```

- **Peek** – (metoda) zwraca kolejny znak w strumieniu nie zmieniając pozycji w strumieniu.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");
```

```
while (sr.Peek() != 0)
    Console.WriteLine(sr.ReadLine());
```

- **Read** – (metoda) wczytuje kolejny znak lub zbiór znaków ze strumienia.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");
```

```
while (sr.Peek() != 0)
    Console.Write((char)sr.Read());
```

- **ReadBlock** – (metoda) wczytuje określoną liczbę znaków ze strumienia do bufora.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");
int dlugosc = 100;
char[] znaki = new char[dlugosc];
```

```
sr.ReadBlock(znaki, 0, dlugosc);
```

- **ReadLine** – (metoda) wczytuje kolejną linię ze strumienia.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");
```

```
while (!sr.EndOfStream)
    Console.WriteLine(sr.ReadLine());
```

- **ReadToEnd** – (metoda) wczytuje dane od aktualnej pozycji do końca strumienia.

Przykład:

```
StreamReader sr = File.OpenText(@"c:\readme.txt");

Console.WriteLine(sr.ReadToEnd());
```

## Klasa StreamWriter

Klasa **StreamWriter** oferuje wiele różnych użytecznych pól, właściwości i metod. Do najważniejszych i najczęściej używanych należą:

- **Null** – (pole statyczne) pusty strumień.

Przykład:

```
StreamWriter sr = new StreamWriter(@"c:\readme.txt");
```

```
if (sr != StreamWriter.Null)
{
    ...
}
```

```
}
```

- ***NewLine*** – (właściwość) zwraca lub ustawia znacznik przejścia do nowej linii.

Przykład:

```
StreamWriter sr = new StreamWriter(@"c:\readme.txt");
```

```
sr.NewLine = "\n";
```

- ***Close*** – (metoda) zamyka plik i strumień z nim skojarzony.

Przykład:

```
StreamWriter sr = new StreamWriter(@"c:\readme.txt");
```

```
sr.Close();
```

- ***Write*** – (metoda) zapisuje tekstowo dane do strumienia.

Przykład:

```
StreamWriter sr = new StreamWriter(@"c:\readme.txt");
```

```
byte bajt = 10;
```

```
long liczba = 1000000L;
```

```
char znak = 'A';
```

```
string napis = "Tekst";
```

```
sr.Write(bajt);
```

```
sr.Write(liczba);
```

```
sr.Write(znak);
```

```
sr.Write(napis);
```

```
sr.Close();
```

Zawartość pliku:

```
101000000ATEkst
```

- ***WriteLine*** – (metoda) zapisuje tekstowo dane do strumienia ze znacznikiem przejścia do nowej linii.

Przykład:

```
StreamWriter sr = new StreamWriter(@"c:\readme.txt");
```

```
byte bajt = 10;
```

```
long liczba = 1000000L;
```

```
char znak = 'A';
```

```
string napis = "Tekst";
```

```
sr.WriteLine(bajt);
```

```
sr.WriteLine(liczba);
```

```
sr.WriteLine(znak);
```

```
sr.WriteLine(napis);
```

```
sr.Close();
```

Zawartość pliku:

```
10
```

```
1000000
```

A

Tekst

## Klasa `BinaryReader`

Klasa ***BinaryReader*** oferuje wiele różnych użytecznych metod. Do najważniejszych i najczęściej używanych należą:

- ***Close*** – (metoda) zamyka strumień binarny.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
  
try  
{  
    int liczba = 0;  
  
    if (brd.PeekChar() != -1)  
        liczba = brd.ReadInt32();  
}  
catch (EndOfStreamException e)  
{  
    Console.WriteLine(e.Message);  
}  
finally  
{  
    brd.Close();  
}
```

- ***PeekChar*** – (metoda) zwraca kolejny dostępny znak nie zmieniając pozycji w strumieniu.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
  
while (brd.PeekChar() != -1)  
    Console.WriteLine(brd.ReadString());
```

- ***Read*** – (metoda) wczytuje kolejny znak ze strumienia.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
  
while (brd.PeekChar() != -1)  
    Console.Write((char)brd.Read());
```

- ***ReadBoolean*** – (metoda) wczytuje zmienną typu ***bool***.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
bool bCzyJest = brd.ReadBoolean();
```

- **ReadByte** – (metoda) wczytuje zmienną typu *byte*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
byte Bajt = brd.ReadByte();
```

- **ReadBytes** – (metoda) wczytuje zadaną liczbę bajtów.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
byte[] bajty = brd.ReadBytes(100);
```

- **ReadChar** – (metoda) wczytuje zmienną typu *char*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
char znak = brd.ReadChar();
```

- **ReadChars** – (metoda) wczytuje zadaną liczbę znaków.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
char[] znaki = brd.ReadChars(100);
```

- **ReadDecimal** – (metoda) wczytuje zmienną typu *decimal*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
decimal dLiczba = brd.ReadDecimal();
```

- **ReadDouble** – (metoda) wczytuje zmienną typu *double*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
double dLiczba = brd.ReadDouble();
```

- **ReadInt16** – (metoda) wczytuje zmienną typu *short*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
short sLiczba = brd.ReadInt16();
```

**ReadInt32** – (metoda) wczytuje zmienną typu *int*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));
```

```
int iLiczba = brd.ReadInt32();
```

- ***ReadInt64*** – (metoda) wczytuje zmienną typu *long*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
long lLiczba = brd.ReadInt64();
```

- ***ReadSByte*** – (metoda) wczytuje zmienną typu *sbyte*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
sbyte sBajt = brd.ReadSByte();
```

- ***ReadSingle*** – (metoda) wczytuje zmienną typu *float*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
float fLiczba = brd.ReadSingle();
```

- ***ReadString*** – (metoda) wczytuje zmienną typu *string*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
string Napis = brd.ReadString();
```

- ***ReadUInt16*** – (metoda) wczytuje zmienną typu *ushort*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
ushort usLiczba = brd.ReadUInt16();
```

- ***ReadUInt32*** – (metoda) wczytuje zmienną typu *uint*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
uint uiLiczba = brd.ReadUInt32();
```

- ***ReadUInt64*** – (metoda) wczytuje zmienną typu *ulong*.

Przykład:

```
BinaryReader brd =  
    new BinaryReader(File.Open(@"c:\readme.dat", FileMode.Open));  
ulong ulLiczba = brd.ReadUInt64();
```

## Klasa BinaryWriter

Klasa ***BinaryWriter*** oferuje wiele różnych użytecznych metod. Do najważniejszych i najczęściej używanych należą:

- ***Close*** – (metoda) zamyka strumień.



Przykład:

```
BinaryWriter brw =  
    new BinaryWriter(File.Open(@"c:\readme.dat", FileMode.Create));  
  
brw.Write("Test");  
brw.Close();
```

- **Seek** – (metoda) zmienia pozycję w strumieniu.

Przykład:

```
BinaryWriter brw =  
    new BinaryWriter(File.Open(@"c:\readme.dat", FileMode.Create));  
  
brw.Write("Test");  
brw.Seek(0, SeekOrigin.Begin);  
brw.Write("Test");  
brw.Close();
```

- **Write** – (metoda) zapisuje dane do strumienia.

Przykład:

```
BinaryWriter brw =  
    new BinaryWriter(File.Open(@"c:\readme.dat", FileMode.Create));  
int iLiczba = 100;  
string napis = "Test";  
  
brw.Write(iLiczba);  
brw.Write(napis);  
brw.Close();
```

## Klasa Path

Klasa **Path** oferuje wiele różnych użytecznych metod. Do najważniejszych i najczęściej używanych należą:

- **ChangeExtension** – (metoda) zmienia rozszerzenie w ścieżce.

Przykład:

```
string sciezka = @"c:\readme.txt";  
string wynik = Path.ChangeExtension(sciezka, ".dat"); // c:\readme.dat
```

- **Combine** – (metoda) łączy dwie ścieżki.

Przykład:

```
string katalog = @"c:\windows\";  
string plik = "readme.txt";  
string sciezka = Path.Combine(katalog, plik); // c:\windows\readme.txt
```

- **GetDirectoryName** – (metoda) zwraca nazwę folderu dla ścieżki.

Przykład:

```
string sciezka = @"c:\windows\readme.txt";  
string wynik = Path.GetDirectoryName(sciezka); // c:\windows
```

- ***GetExtension*** – (metoda) zwraca rozszerzenie pliku.

Przykład:

```
string sciezka = @"c:\windows\readme.txt";  
string wynik = Path.GetExtension(sciezka); // .txt
```

- ***GetFileName*** – (metoda) zwraca nazwę pliku z rozszerzeniem.

Przykład:

```
string sciezka = @"c:\windows\readme.txt";  
string wynik = Path.GetFileName(sciezka); // readme.txt
```

- ***GetFileNameWithoutExtension*** – (metoda) zwraca nazwę pliku bez rozszerzenia.

Przykład:

```
string sciezka = @"c:\windows\readme.txt";  
string wynik = Path.GetFileNameWithoutExtension(sciezka); // readme
```

- ***GetFullPath*** – (metoda) zwraca pełną ścieżkę.

Przykład:

```
string sciezka = @"c:\readme.txt";  
string wynik = Path.GetFullPath(sciezka); // c:\readme.txt
```

- ***GetPathRoot*** – (metoda) zwraca nazwę folderu nadrzędnego.

Przykład:

```
string sciezka = @"c:\readme.txt";  
string wynik = Path.GetPathRoot(sciezka); // c:\
```

- ***GetTempFileName*** – (metoda) tworzy unikalną nazwę pliku tymczasowego.

Przykład:

```
string sciezka = Path.GetTempFileName();
```

- ***GetTempPath*** – (metoda) zwraca aktualną ścieżkę do systemowego katalogu tymczasowego.

Przykład:

```
string temp = Path.GetTempPath();
```

- ***HasExtension*** – (metoda) sprawdza czy ścieżka zawiera rozszerzenie.

Przykład:

```
string sciezka = @"c:\\";
```

```
if (!Path.HasExtension(sciezka))  
    sciezka = Path.Combine(sciezka, "readme.txt");
```

- ***IsPathRooted*** – (metoda) sprawdza czy ścieżka jest względna czy bezwzględna.

Przykład:

```
string sciezka = @"c:\readme.txt";
```

```
if (Path.IsPathRooted(sciezka))  
    Console.WriteLine(Path.GetPathRoot(sciezka));
```

## Rozdział 17.

# Debugowanie

### Wprowadzenie

Debugowanie, czyli śledzenie kodu, jest jedną z najczęściej wykonywanych przez programistę czynności w czasie tworzenia aplikacji. Wykonywane jest w celu wykrycia błędów logicznych aplikacji (błędy składniowe wykrywane są w czasie kompilacji). Dzięki tej czynności, możliwe jest obserwowanie zachowania obiektów i porównywanie ich faktycznych stanów z założonymi.

Visual Studio C#.NET 2005 zawiera zintegrowany debugger, który pozwala na zaawansowane i wygodne śledzenia kodu. Generując projekt zawsze domyślnie tworzone są dwie konfiguracje:

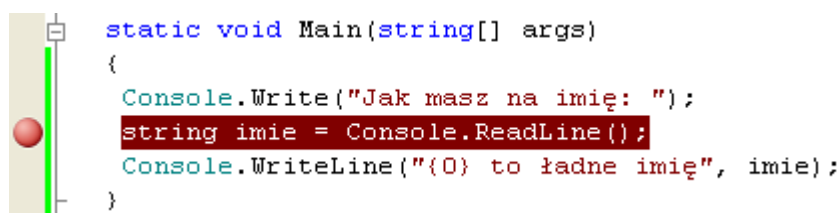
- **Debug** – (domyślna) generowana jest aplikacja zawierająca dodatkowe informacje używane w procesie śledzenia kodu (zbudowana aplikacja przeznaczona jest do testowania).
- **Release** – generowana jest aplikacja przeznaczona do dystrybucji (nie są dodawane informacje używane w czasie śledzenia kodu, dlatego nie jest możliwe śledzenie aplikacji zbudowanych w tej konfiguracji).

### Pułapki i śledzenie krokowe

Pułapka jest miejscem, w którym zatrzymuje się debugger w czasie śledzenia kodu. Pułapki zakłada się w miejscach, w których chcemy sprawdzić stan obiektów po wykonaniu pewnej liczby operacji (np.: w miejscu potencjalnego wystąpienia błędu logicznego).

Pułapki mogą być dodawane w dowolnym miejscu kodu zawierającego jakieś wyrażenie. Dodanie pułapki odbywa się poprzez naciśnięcie klawisza **F9** lub kliknięcie kursorem myszy na marginesie obok linii kodu. Założenie pułapki skutkuje pojawieniem się czerwonej kropki na marginesie (z lewej strony linii edytora). Ponowne naciśnięcie klawisza **F9** lub kliknięcie w czerwoną kropkę usuwa pułapkę. Pułapkę można ustawić lub usunąć również wybierając z menu **Debug** opcję **Toggle Breakpoint**.

Kompleksowe usunięcie wszystkich założonych pułapek, odbywa się przez naciśnięcie kombinacji klawiszy **Ctrl+Shift+F9** lub wybranie opcji **Clear All Breakpoints** z menu **Debug**.

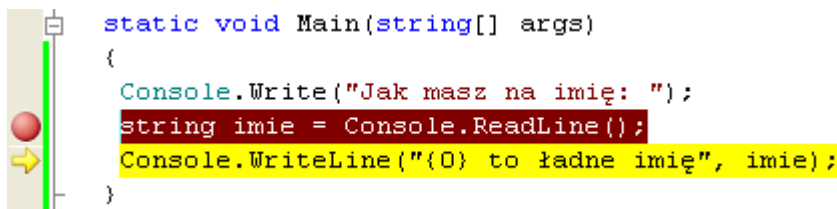


Rysunek 22. Wygląd założonej pułapki

Aby zdebugować program należy uruchomić go w trybie **Debug**. Wykonuje się to poprzez naciśnięcie klawisza **F5** lub wybranie opcji **Start** z menu **Debug**. Po uruchomieniu programu nastąpi jego wykonanie i jeżeli założone zostały pułapki, wykonanie kodu zatrzyma się na pierwszej z nich. Jeżeli chcemy kontynuować śledzenie kodu do następnej pułapki, wystarczy ponownie nacisnąć klawisz **F5**. W przypadku, gdy chcemy od pułapki wykonywać kod krokowo możemy nacisnąć klawisz **F10** lub

wybrać opcję **Step Over** z menu **Debug**, która spowoduje przejście do kolejnej instrukcji (wykonana zostanie aktualna instrukcja). Jeżeli interesuje nas co dzieje się w czasie wykonywania danej, linii możemy wejść głębiej (np. jeżeli następuje wyliczenie wyrażenia za pomocą jakiejś metody, możemy wejść do jej wnętrza) poprzez naciśnięcie klawisza **F11** lub wybranie opcji **Step Into** z menu **Debug**.

Aby przerwać proces debugowania kodu wystarczy nacisnąć kombinację klawiszy **Shift+F5** lub wybrać opcję **Stop Debugging** z menu **Debug**.



Rysunek 23. Przejście do następnej linii (Step Over)

## Okna śledzenia

W trakcie śledzenia kodu, możliwe jest obserwowanie i zmienianie wartości zmiennych. Najprostszym sposobem obserwowania wartości w czasie debugowania jest najechanie kursorem myszy na zmienną, wtedy pojawi się okno podpowiedzi, zawierające aktualną wartość zmiennej. Znacznie wygodniejszym sposobem obserwacji jest posłużenie się jednym z dostępnych okien wyświetlających wartości zmiennych i pozwalających je zmieniać.

Do podstawowych okien śledzenia należą:

- **Autos** – (okno zakotwiczone) wyświetla wartości zmiennych z poprzedniej i aktualnej linii.

Autos		
Name	Value	Type
imie	null	string

Rysunek 24. Okno Autos

- **Locals** – (okno zakotwiczone) wyświetla wartości zmiennych lokalnych (dla aktualnej metody).

Locals		
Name	Value	Type
args	{Dimensions:[0]}	string[]
imie	null	string

Rysunek 25. Okno Locals

- **Watch** – (okno zakotwiczone) wyświetla wartości zmiennych wskazanych przez użytkownika (zmienne można dodawać wpisując w polu **Name** nazwę zmiennej).

Watch 1		
Name	Value	Type
imie	null	string

Rysunek 26. Okno Watch

# Część II.

## Tworzenie aplikacji okienkowych

### Rozdział 1.

#### Podstawy Windows Forms

#### Wprowadzenie

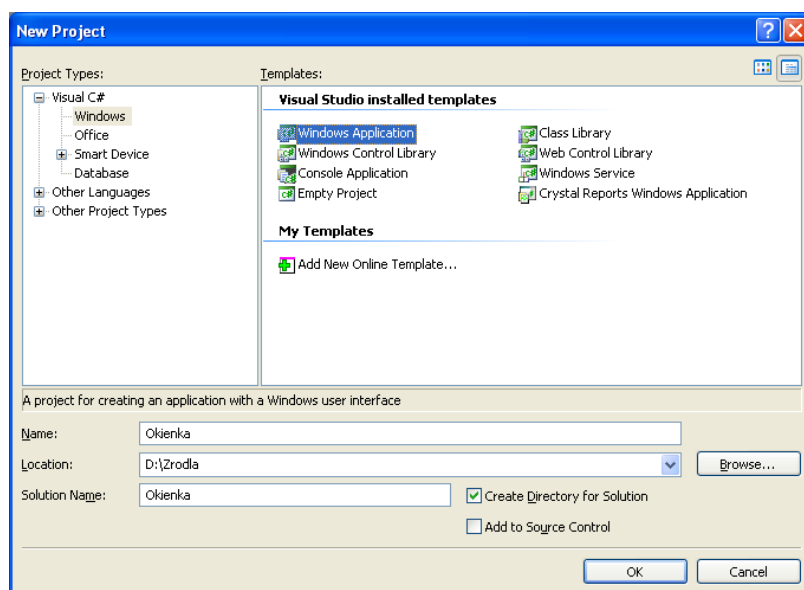
*Windows Forms* jest zbiorem elementów służących do tworzenia aplikacji okienkowych. Jednakże tworzenie aplikacji okienkowych to nie tylko tworzenie kodu, ale również projektowanie interfejsu użytkownika. Interfejs użytkownika jest jednym z podstawowych elementów aplikacji okienkowej i decyduje w znacznym stopniu o jej atrakcyjności. To właśnie interfejs jest warstwą, z którą styka się użytkownik (wchodzi w interakcje).

Środowisko Microsoft Visual Studio.NET 2005 wspiera proces tworzenia aplikacji okienkowych w pełnym zakresie. Zintegrowane narzędzia do projektowania interfejsu pozwalają na wygodne i szybkie budowanie aplikacji okienkowych.

#### Generowanie aplikacji Windows Forms

Aby wygenerować szablon projektu aplikacji *Windows Forms* należy:

1. Utworzyć nowy projekt wybierając opcję *Project...* z menu *File->New*.
2. W obszarze *Project Types* wybrać typ projektu *Windows*.
3. W obszarze *Templates* wybrać wzorzec *Windows Application*.
4. W obszarze *Name* wpisać nazwę projektu.
5. W obszarze *Location* wskazać ścieżkę katalogu głównego, w którym będzie umieszczony katalog projektu.
6. Upewniamy się, że w obszarze *Solution* wybrano z listy *Create new Solution*.
7. Naciskamy przycisk *Ok*.



Rysunek 27. Generowanie szablonu aplikacji okienkowej

Po wygenerowaniu szablonu aplikacji okienkowej zostanie utworzony projekt zawierający kod pozwalający na wyświetlenie okna głównego aplikacji.

## Rozdział 2.

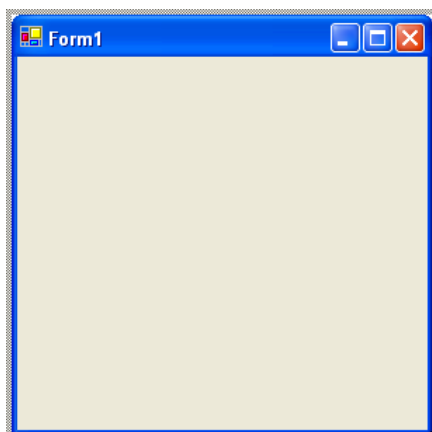
# Praca z formą

### Tworzenie formy

Forma jest podstawowym elementem interfejsu użytkownika aplikacji okienkowej Windows. Jest to forma jest podstawowym elementem interfejsu użytkownika aplikacji okienkowej Windows. Jest to projekt okna, które zostanie wyświetlone w celu prezentacji danych użytkownikowi oraz wczytywaniu danych od użytkownika.

Po wygenerowaniu szablonu aplikacji okienkowej, tworzona jest zawsze forma podstawowa. Na formie tej w trakcie projektowania interfejsu można umieszczać szereg różnych kontroltek (np.: menu, paski narzędzi, przyciski, etc.) oraz przypisywać im określone zachowania (np. reakcję na naciśnięcie przycisku czy wybranie opcji z menu).

Bazową klasą dla każdej formy jest klasa **Form**, znajdująca się w przestrzeni **System.Windows.Forms** (klasa **Form** dziedziczy z klasy **Control**).

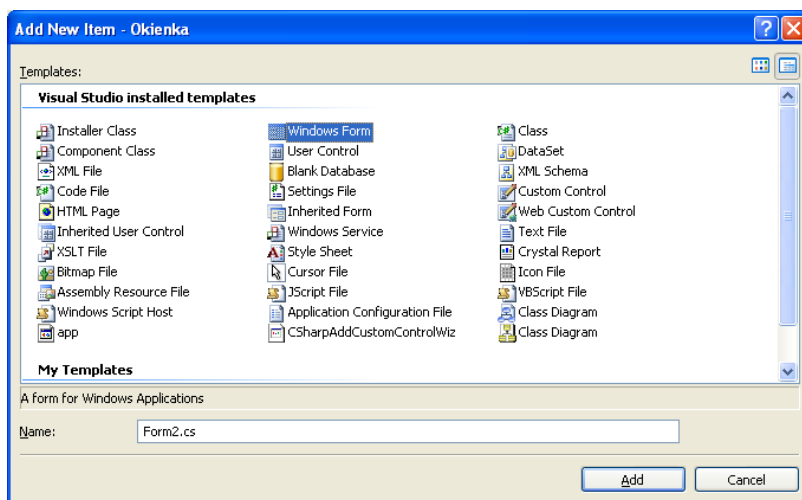


Rysunek 28. Wygenerowana forma

Oczywiście większość aplikacji posiada więcej niż jedno okno, więc jeżeli nasza aplikacja ma posługiwać się większą liczbą okien, musimy do projektu dodać nową formę.

W celu dodania nowej formy do projektu, należy wykonać następujące czynności:

1. W oknie **Solution Explorer** kliknąć prawym klawiszem myszy na nazwie projektu.
2. Z menu **Add** wybrać opcję **Windows Form...**
3. W oknie **Add New Item** w polu **Name** wpisać nazwę pliku dla nowej formy.
4. Nacisnąć przycisk **Add**.





Rysunek 29. Wygląd okna Add New Item

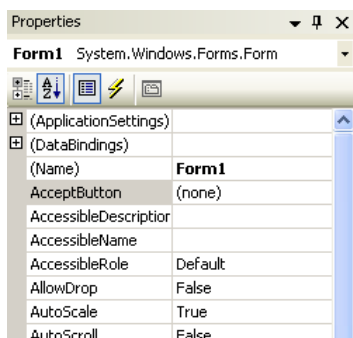
Każda forma posiada:

- **Właściwości** – pozwalające na zmianę wyglądu formy;
- **Metody** – pozwalające na zdefiniowanie zachowania formy;
- **Zdarzenia** – pozwalające na interakcję z użytkownikiem.

## Właściwości formy

Forma posiada szereg właściwości pozwalających na zmianę jej wyglądu. Właściwości form ustawia się za pośrednictwem okna **Properties Window**. Znając nazwę właściwości, którą chcemy zmienić, należy kliknąć na jej nazwę i po prawej stronie wpisać lub wybrać z listy określoną wartość właściwości.

Właściwości okna mogą być wyświetlane w grupach funkcyjnych (ikona ) lub w porządku alfabetycznym (ikona )



Rysunek 30. Fragment okna Properties z aktywną listą właściwości



Poniższe zestawienie zawiera opis najczęściej używanych właściwości formy oraz przyjmowanych wartości domyślnych:

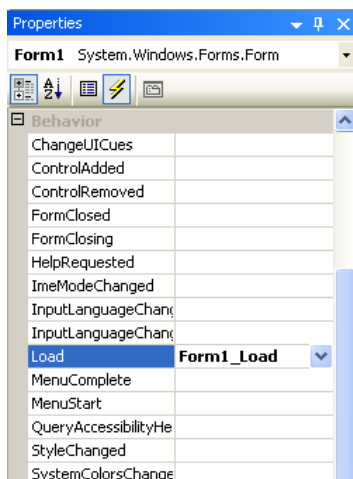
Właściwość	Opis	Wartość domyślna
<b>(Name)</b>	Nazwa formy (chodzi o nazwę klasy dla formy, której będziemy używać tworząc obiekty nie zaś napisu wyświetlanego na pasku tytułowym okna).	<b>Form1, Form2, ...</b>
<b>AcceptButton</b>	Określa, który przycisk ma pełnić rolę domyślnego przycisku akceptacji (reagować na	<b>(none)</b>



	naciśnięcie klawisza ENTER).	
<b>AllowDrop</b>	Określa, czy forma akceptuje komunikaty „przeciągnij i upuść”.	<b>False</b>
<b>AutoSize</b>	Określa, czy forma ma być automatycznie dopasowana do kontekstu.	<b>False</b>
<b>BackColor</b>	Określa kolor tła formy.	<b>Control</b>
<b>CancelButton</b>	Określa, który przycisk ma pełnić rolę domyślnego przycisku anulowania (reagować na naciśnięcie klawisza ESC).	<b>(none)</b>
<b>ControlBox</b>	Określa, czy forma wyświetla przyciski kontroli okna w pasku tytułowym. Menu może zawierać przyciski minimalizacji, maksymalizacji, pomocy oraz zamknięcia.	<b>True</b>
<b>Cursor</b>	Określa rodzaj kursora myszy wyświetlany w czasie, gdy znajdzie się on w obrębie formy.	<b>Default</b>
<b>Enabled</b>	Określa czy forma jest dostępna.	<b>True</b>
<b>Font</b>	Określa rodzaj czcionki używanej przez formę.	<b>Microsoft Sans Serif; 8,25pt</b>
<b>ForeColor</b>	Określa kolor tekstu i grafiki formy.	<b>ControlText</b>
<b>FormBorderStyle</b>	Określa wygląd okna (okno rozszerzalne, dialogowe, bez ramki, narzędziowe, etc.).	<b>Sizable</b>
<b>HelpButton</b>	Określa, czy okno posiada przycisk pomocy.	<b>False</b>
<b>Icon</b>	Określa ikonę dla formy.	<b>domyślna</b>
<b>IsMdiContainer</b>	Określa, czy forma pełni rolę kontenera MDI.	<b>False</b>
<b>Location</b>	Określa pozycję górnego lewego rogu formy.	<b>0,0</b>
<b>Locked</b>	Określa, czy kontrolki mogą być przesuwane i czy można zmieniać ich rozmiar.	<b>True</b>
<b>MaximizeBox</b>	Określa, czy forma posiada przycisk maksymalizacji na pasku tytułowym.	<b>True</b>
<b>MaximumSize</b>	Określa maksymalny rozmiar formy.	<b>0, 0 (czyli dowolny)</b>
<b>Menu</b>	Określa, które menu jest głównym dla okna.	<b>(none)</b>
<b>MinimizeBox</b>	Określa, czy forma posiada przycisk minimalizacji na pasku tytułowym.	<b>True</b>
<b>MinimumSize</b>	Określa minimalny rozmiar formy.	<b>0, 0</b>
<b>Opacity</b>	Określa poziom widoczności/przeźroczystości formy (100% widoczne, 0% przeźroczyste).	<b>100%</b>
<b>ShowInTaskBar</b>	Określa czy okno ma być widoczne w oknie zadań.	<b>True</b>
<b>Size</b>	Określa rozmiar początkowy formy.	<b>300; 300 (zaprojektowany)</b>
<b>StartPosition</b>	Określa pozycję pierwszego pojawienia się formy.	<b>WindowsDefaultLocation</b>
<b>Text</b>	Określa tytuł formy pojawiający się na pasku tytułowym.	<b>Form1, Form2, ...</b>
<b>TopMost</b>	Określa, czy forma jest oknem najbardziej widocznym.	<b>False</b>
<b>WindowState</b>	Określa sposób pojawienia się formy (normalna, zminimalizowana lub zmaksymalizowana).	<b>Normal</b>

## Obsługa zdarzeń

Forma posiada listę zdarzeń, które mogą zostać powiązane z metodą reagującą na wystąpienie określonego zdarzenia (np.: pojawienie się okna). Dodawanie metod reagujących na zdarzenie następuje za pośrednictwem okna **Properties Window**. Okno to pozwala zarówno na zmianę właściwości, jak i przypisywanie zdarzeniom metod. Tryb edycji właściwości jest aktywny w momencie naciśnięcia ikony **Properties** , natomiast tryb edycji zdarzeń w momencie naciśnięcia ikony **Events** .



Rysunek 31. Okno **Properties** w trybie edycji zdarzeń (na rysunku widać przypisaną zdarzeniu **Load** metodę **Form1\_Load**)

Poniższe zestawienie zawiera opis najczęściej używanych zdarzeń oraz opis przypadków ich występowania:

Zdarzenie	Opis
<b>Activated</b>	Występuje, gdy forma jest aktywowana przez użytkownika lub instrukcję.
<b>Click</b>	Występuje, gdy nastąpi kliknięcie na formę.
<b>Deactivate</b>	Występuje, gdy forma jest deaktywowana (traci focus).
<b>DoubleClick</b>	Występuje, gdy nastąpi podwójne kliknięcie na formę.
<b>Enter</b>	Występuje, gdy nastąpi wejście do formy.
<b>FormClosed</b>	Występuje, gdy forma zostanie zamknięta (po zamknięciu).
<b>FormClosing</b>	Występuje, gdy forma jest zamykana (przed zamknięciem).
<b>KeyDown</b>	Występuje, gdy nastąpi naciśnięcie klawisza (przekazywany jest kod klawisza).
<b>KeyPress</b>	Występuje, gdy nastąpi naciśnięcie klawisza (przekazywany jest znak klawisza).
<b>KeyUp</b>	Występuje, gdy nastąpi zwolnienie klawisza (przekazywany jest kod klawisza).
<b>Leave</b>	Występuje, gdy nastąpi opuszczenie formy.
<b>Load</b>	Występuje przed pierwszym pojawieniem się formy (zdarzenie to występuje przed wywołaniem metody <b>Show</b> i jest używane w przypadku, gdy istnieje potrzeba wykonania pewnych instrukcji jeszcze zanim pokaże się forma, zazwyczaj jest to dobry moment do przypisania domyślnych wartości formie i jej kontrolkom).
<b>MouseDown</b>	Występuje, gdy zostanie naciśnięty przycisk myszy.
<b>MouseMove</b>	Występuje, gdy kursor myszy jest poruszany w obszarze formy.
<b>MouseUp</b>	Występuje, gdy zostanie zwolniony przycisk myszy.

<b>Move</b>	Występuje, gdy forma zmienia położenie.
<b>Paint</b>	Występuje, gdy forma jest przerysowywana.
<b>Resize</b>	Występuje, gdy forma zmienia swój rozmiar.

Przykład ustawienia tytułu okna:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.Text = "Aplikacja testowa";
}
```

## Metody formy

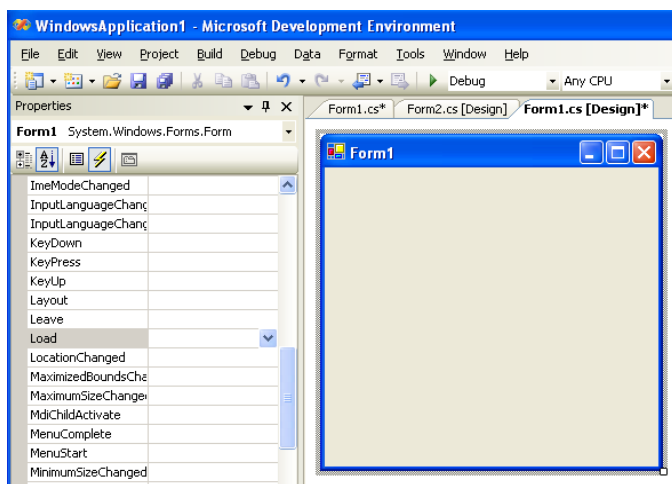
Forma posiada szereg metod, które pozwalają na definiowanie zachowania formy. Poniższa tabela zawiera zestawienie najczęściej używanych metod:

Metoda	Opis
<b>Activate</b>	Aktywuje formę.
<b>Close</b>	Zamyka formę.
<b>Focus</b>	Sprawia, że forma uzyskuje focus.
<b>Hide</b>	Ukrywa formę.
<b>Refresh</b>	Wymusza odświeżenie (odrysowanie) całej formy i jej kontrolek.
<b>Show</b>	Pokazuje formę.
<b>ShowDialog</b>	Pokazuje formę jako modalne okno dialogowe.
<b>Update</b>	Wymusza odrysowanie widocznej części formy.

## Przykładowa aplikacja

Napiszmy sobie teraz prostą aplikację złożoną z dwóch form (głównej i dodatkowej), które po uruchomieniu aplikacji pojawią się na ekranie obok siebie:

1. Generujemy nowy projekt aplikacji okienkowej (patrz „Generowanie aplikacji Windows Forms”).
2. Dodajemy nową formę (patrz „Tworzenie formy”).
3. Klikamy na projekcie formy podstawową (**Form1**) i w oknie **Properties** wybieramy zdarzenie **Load**.



Rysunek 32. Okno projektu formy **Form1** oraz okno **Properties** z zaznaczonym zdarzeniem **Load**

4. Tworzymy metodę do zdarzenia i łączymy ją ze zdarzeniem podwójnie klikając na puste pole po prawej stronie nazwy zdarzenia **Load**. Zostanie utworzona metoda **Form1\_Load**, która będzie wywoływana po wystąpieniu zdarzenia **Load**:

```
private void Form1_Load(object sender, EventArgs e)
{
}
```

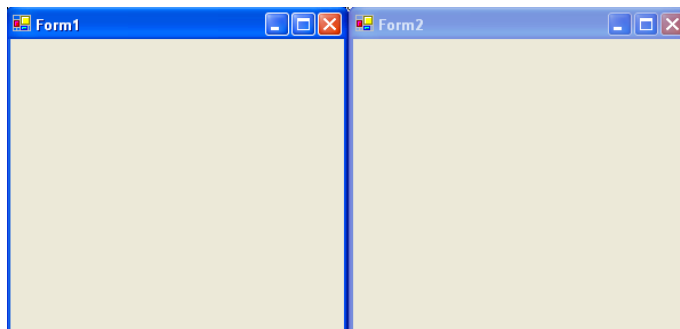
5. Wpisujemy kod, który utworzy nowe okno i ustawi jego początkowe położenie z prawej strony okna głównego. Pełna metoda wygląda tak:

```
private void Form1_Load(object sender, EventArgs e)
{
    Form2 frm = new Form2(); // tworzymy instancję nowej formy

    frm.Show(); // wyświetlamy formę
    frm.Location = new Point(this.Location.X + this.Size.Width,
                             this.Location.Y); // zmieniamy jej położenie
}
```

Jak widać, w pierwszej kolejności musieliśmy utworzyć instancję dla nowej formy i użyć metody **Show**, aby ukazała się ona na ekranie. Następnie wykorzystując właściwość **Location**, zmieniliśmy jej położenie początkowe na zgodne w pionie z formą podstawową i przesunięte o szerokość formy podstawowej w poziomie (do tego celu posłużyliśmy się dodatkowo właściwością **Size**).

6. Po skompilowaniu i uruchomieniu otrzymujemy wynik działania programu.



Rysunek 33. Wynik działania programu

## Rozdział 3.

# Korzystanie z prostych kontrolek

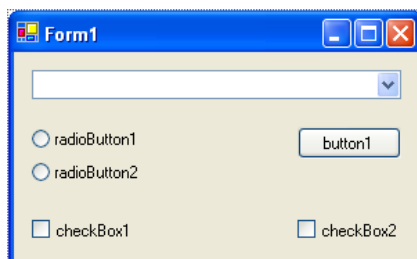
### Dodawanie kontrolek do formy

Kontrolki są obiektami, które są umieszczane na formie. Przykładami kontrolek są: przyciski, pola tekstowe, etykiety, listy rozwijane, drzewa, itp.

Dodanie kontrolki do formy odbywa się poprzez wybór z okna **Toolbox** właściwej kontrolki i przeciągnięcie jej na formę. Rozmiar i położenie kontrolki można dopasować zarówno w czasie przeciągania przed lub też po upuszczeniu jej na formę.

Jeżeli chcemy dopasować rozmiar i położenie w czasie upuszczania, wystarczy chwycić kontrolkę i umieścić kursor myszy w miejscu, gdzie ma się znajdować lewa górna krawędź kontrolki. Następnie trzymając lewy klawisz myszy wciśnięty poruszać się w prawy dolny róg. Kiedy rozmiar jest właściwy zwalniamy lewy klawisz.

Jeżeli upuszczamy kontrolkę bez dopasowywania jej w czasie przeciągania, wystarczy kliknąć lewym klawiszem myszy w miejscu, w którym ma się znajdować lewa górna krawędź kontrolki, a wszelkie operacje przesunięcia i dopasowywania rozmiaru wykonać później.

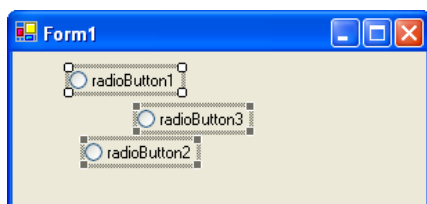


Rysunek 34. Projekt formy z przykładowymi kontrolkami (zaraz po umieszczeniu ich na formie)

### Organizowanie kontrolek na formie

Po umieszczeniu kontrolek na formie można je dowolnie formatować. W celu ich równomiernego rozmieszczenia oraz dopasowania wielkości, można posłużyć się jedną z wielu opcji formatowania, dostępnych w menu **Format** środowiska Visual Studio C#.NET 2005.

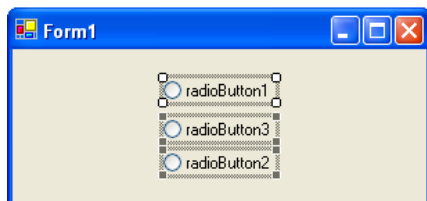
W pierwszej kolejności należy zaznaczyć kontrolkę lub grupę kontrolek (grupę wybiera się trzymając wciśnięty klawisz **Shift** i wskazując kolejne kontrolki). Następnie wybieramy opcje formatowania. Grupę kontrolek można również przesuwąć na formatce w taki sam sposób jak pojedyncze kontrolki.



Rysunek 35. Przykładowa grupa kontrolek przygotowanych do sformatowania

Menu **Format** zawiera dużą ilość opcji pogrupowanych w podmenu, które służą do formatowania grupy kontroltek:

- **Align** – pozwala na wyrównanie kontroltek do jednej linii,
- **Make Same Size** – pozwala na wyrównanie rozmiarów kontroltek,
- **Horizontal Spacing** – pozwala na zmianę poziomego rozmiaru odstępu między kontrolkami,
- **Vertical Spacing** – pozwala na zmianę pionowego rozmiaru odstępu między kontrolkami,
- **Center in Form** – pozwala na umieszczenie kontroltek w centrum formy,
- **Order** – pozwala na zmianę porządku kontroltek,
- **Lock Controls** – pozwala na zablokowanie kontroltek do modyfikacji.



Rysunek 36. Wyrównanie do lewej i umieszczenie w centrum

## Wspólne cechy kontroltek

Ze względu na to, że wszystkie kontrolki dziedziczą z klasy **Control**, posiadają zbiór wspólnych cech, które mają wspólne znaczenie.

### Właściwości

Poniższe zestawienie zawiera najczęściej używane właściwości kontroltek (dostępnych z okna **Properties**):

Właściwość	Opis
<b>(Name)</b>	Nazwa obiektu utworzonej kontrolki, którego będziemy używać.
<b>Anchor</b>	Określa, które brzegi są zakotwiczone do formy i dla których ma zostać zachowana odległość od krawędzi formy (określone brzegi będą rozszerzane wraz z formą w taki sposób, aby odległość od krawędzi formy była zachowana).
<b>AutoSize</b>	Określa, czy kontrolka będzie automatycznie dopasowywana do kontekstu (np.: im dłuższy tekst tym większy rozmiar kontrolki).
<b>BackColor</b>	Określa kolor tła danej kontrolki.
<b>BackgroundImage</b>	Określa jaki obrazek pełniący rolę tła ma być wyświetlany w kontrolce.
<b>ContextMenuStrip</b>	Określa rodzaj menu kontekstowego skojarzonego z kontrolką.
<b>Cursor</b>	Określa rodzaj kursora myszy, jaki pojawi się, gdy znajdzie się on w obszarze kontrolki.
<b>Dock</b>	Określa rodzaj zakotwiczenia kontrolki (np.: wypełnienie całego wnętrza).
<b>Enabled</b>	Określa, czy kontrolka jest dostępna w trybie do edycji.
<b>FlatStyle</b>	Określa wygląd kontrolki oraz jej zachowanie. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Standard</b> – (domyślny) kontrolka z krawędziami 3d</li> <li>• <b>System</b> – wygląd kontrolki uzależniony jest od systemu operacyjnego</li> </ul>

	<ul style="list-style-type: none"> <li>• <b>Flat</b> – kontrolka płaska</li> <li>• <b>Popup</b> – kontrolka jest płaska do momentu, gdy kursor nie znajdzie się w jej obszarze, wtedy się podnosi</li> </ul>
<b>Font</b>	Określa czcionkę używaną przez kontrolkę.
<b>ForeColor</b>	Określa kolor tekstu i grafiki używany przez kontrolkę.
<b>Location</b>	Określa położenie lewej górnej krawędzi kontrolki.
<b>Locked</b>	Określa, czy kontrolka może być modyfikowana na etapie projektowania.
<b>MaximumSize</b>	Określa maksymalny rozmiar kontrolki.
<b>MinimumSize</b>	Określa minimalny rozmiar kontrolki.
<b>Modifiers</b>	Określa modyfikator dostępu dla obiektu kontrolki.
<b>RightToLeft</b>	Określa, czy kontrolka ma stosować tryb z prawej do lewej dla języków stosujących taki zapis.
<b>Size</b>	Określa rozmiar kontrolki w pikselach.
<b>Text</b>	Określa tekst jaki zawiera kontrolka (np.: etykieta przycisku).
<b>Visible</b>	Określa, czy kontrolka jest widoczna po wyświetleniu formy.

## Zdarzenia

Poniższe zestawienie zawiera najczęściej używane zdarzenia dla kontrolek (dostępne z okna *Properties*):

Zdarzenie	Opis
<b>AutoSizeChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>AutoSize</b> .
<b>BackColorChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>BackColor</b> .
<b>BackgroundImageChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>BackgroundImage</b> .
<b>Click</b>	Występuje, gdy nastąpi kliknięcie na kontrolkę.
<b>CursorChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>Cursor</b> .
<b>DockChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>Dock</b> .
<b>EnabledChanged</b>	Występuje, gdy nastąpi zmiana trybu dostępności kontrolki.
<b>Enter</b>	Występuje, gdy kontrolka staje się aktywną kontrolką formy.
<b>FontChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>Font</b> .
<b>ForeColorChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>ForeColor</b> .
<b>KeyDown</b>	Występuje, gdy naciśnięty zostanie klawisz (przekazywany jest kod klawisza).
<b>KeyPress</b>	Występuje, gdy naciśnięty zostanie klawisz (przekazywany jest znak klawisza).
<b>KeyUp</b>	Występuje, gdy zwolniony zostanie klawisz (przekazywany jest kod klawisza).
<b>Leave</b>	Występuje, gdy kontrolka przestaje być aktywną kontrolką formy.
<b>MouseClick</b>	Występuje, gdy nastąpi kliknięcie myszą.
<b>MouseDoubleClick</b>	Występuje, gdy nastąpi podwójne kliknięcie myszą.
<b>MouseDown</b>	Występuje, gdy nastąpi naciśnięcie klawisza myszy.
<b>MouseEnter</b>	Występuje, gdy kursor znajdzie się w obszarze kontrolki.
<b>MouseHover</b>	Występuje, gdy kursor myszy zatrzyma się w obszarze kontrolki po wystąpieniu zdarzenia <b>MouseEnter</b> .

<b>MouseLeave</b>	Występuje, gdy kursor opuści obszar kontrolki.
<b>MouseMove</b>	Występuje, gdy nastąpi zmiana pozycji kursora myszy w obszarze kontrolki.
<b>MouseUp</b>	Występuje, gdy nastąpi zwolnienie klawisza myszy.
<b>Resize</b>	Występuje, gdy zmieniany jest rozmiar kontrolki.
<b>SizeChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>Size</b> .
<b>TextChanged</b>	Występuje, gdy zmianie ulegnie właściwość <b>Text</b> .
<b>Validated</b>	Występuje, gdy kontrolka została sprawdzona (zdarzenie może być wykorzystane do przetwarzania sprawdzonych wcześniej wartości charakterystycznej dla kontrolki np.: formatu danych).
<b>Validating</b>	Występuje, gdy kontrolka jest sprawdzana. Aby anulować sprawdzanie należy ustawić właściwość <b>Cancel</b> na wartość <b>True</b> (zdarzenie może być wykorzystane do sprawdzenia poprawności wartości charakterystycznej dla kontrolki np.: formatu danych).
<b>VisibleChanged</b>	Występuje, gdy nastąpi zmiana widoczności kontrolki.

Podstawowym zdarzeniem dla kontrolki jest zdarzenie **Click**. Aby stworzyć metodę obsługującą to zdarzenie wystarczy dwukrotnie kliknąć na kontrolkę w projekcie formy, w której się znajduje. Zostanie wtedy utworzona pusta metoda, którą możemy wypełnić kodem wykonującym jakąś akcję. Metoda wywoływana będzie zawsze, gdy nastąpi kliknięcie na kontrolkę. Przykładowo dla przycisku o nazwie `button1` zostanie wygenerowane zdarzenie:

```
private void button1_Click(object sender, EventArgs e)
```

Zdarzenie może występować z określonymi parametrami, które są przekazywane do metody je obsługującej w postaci argumentów wejściowych.

Do argumentów tych należą:

- obiekt, który wysłał zdarzenie (`sender`);
- obiekt (klasa **EventArgs** lub klasa potomna tej klasy) zawierający parametry wywołania zdarzenia (np.: dla zdarzenia **KeyDown** jest to klasa **KeyEventArgs**, która zawiera parametry dotyczące kodu naciśniętego klawisza).

## Metody

Poniższe zestawienie zawiera najczęściej używane metody dla kontrolki:

Metoda	Opis
<b>Focus</b>	Sprawia, że kontrolka uzyskuje focus.
<b>Hide</b>	Ukrywa kontrolkę.
<b>Invalidate</b>	Wymusza odrysowanie kontrolki.
<b>Refresh</b>	Wymusza odrysowanie kontrolki i wszystkich innych kontrolki, dla których dana kontrolka jest macierzystą.
<b>Show</b>	Pokazuje kontrolkę.
<b>Update</b>	Wymusza odrysowanie obszaru kontrolki.

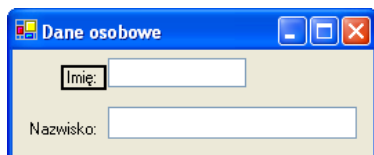
## Etykieta tekstowa



Etykieta tekstowa (kontrolka **A Label**) standardowo używana jest do prezentowania tekstu, który umieszczany jest na formie. Zazwyczaj pełni ona rolę informacyjną i służy przede wszystkim do wyświetlania nazwy skojarzonej z nią kontrolki.

## Właściwości

Jeżeli etykieta spełnia jedynie rolę opisową dla skojarzonej z nią kontrolki, wystarczy zmienić jej właściwość **Text**, wpisując treść opisową w oknie **Properties**.



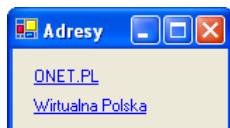
Rysunek 37. Projekt formy z etykietami w roli opisującej przeznaczenie pól tekstowych

Oczywiście, jeżeli zajdzie taka potrzeba treść opisu czy informacji może być dowolnie modyfikowana w trakcie działania programu. W takim wypadku należy jedynie odwołując się do obiektu kontrolki, ustawić wartość właściwości **Text** w kodzie. Przykładowo dla kontrolki, dla której nazwa obiektu (pole **Name**) ma wartość **label1** (domyślnie dla pierwszej etykiety), zmiana treści opisowej wygląda następująco:

```
label1.Text = "Nowa treść opisu lub informacji";
```

## Etykieta łączy

Etykieta łączy (kontrolka **A LinkLabel**) jest specjalną odmianą etykiety tekstowej, która służy do wyświetlania łączy hipertekstowych.



Rysunek 38. Projekt formy z etykietami łączy

## Właściwości

Kontrolka etykiety łączy (**LinkLabel**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>ActiveLinkColor</b>	Określa kolor etykiety łączy w czasie kliknięcia.	<b>255; 0; 0</b>
<b>DisabledLinkColor</b>	Określa kolor etykiety łączy zablokowanej przez ustawienie właściwości kontrolki <b>Enabled</b> na <b>false</b> .	<b>143; 140; 127</b>
<b>LinkArea</b>	Określa zakres aktywności łączy w tekście etykiety (przykładowo ustawienie wartości <b>Start</b> na 0 i <b>Length</b> na 1 sprawi, że tylko pierwszy znak tekstu będzie łączy).	<b>Start = 0</b> <b>Length = długość</b>
<b>LinkBehavior</b>	Określa zachowanie łączy (np. czy ma być wyświetlane jako zawsze podkreślone lub niepodkreślone). Dozwolone są wartości:	<b>SystemDefault</b>

	<ul style="list-style-type: none"> <li>• <b><i>SystemDefault</i></b> – domyślny (podkreślony)</li> <li>• <b><i>AlwaysUnderline</i></b> – zawsze podkreślony</li> <li>• <b><i>HoverUnderline</i></b> – podkreślony po najechaniu na łącze kursorem</li> <li>• <b><i>NeverUnderline</i></b> – nigdy nie podkreślony</li> </ul>	
<b><i>LinkColor</i></b>	Określa domyślny kolor wyświetlania etykiety łącza.	<b><i>0; 0; 255</i></b>
<b><i>LinkVisited</i></b>	Określa czy etykieta łącza ma wyświetlać łącze tak jakby było już wcześniej odwiedzone.	<b><i>false</i></b>
<b><i>VisitedLinkColor</i></b>	Określa kolor etykiety łącza po kliknięciu (po odwiedzeniu łącza).	<b><i>128; 0; 128</i></b>

## Zdarzenia

Podstawowym zdarzeniem dla tej kontrolki jest zdarzenie ***LinkClicked***, które występuje, gdy nastąpi kliknięcie na etykietę łącza. Jeżeli chcemy, aby takie kliknięcie skutkowało jakąś akcją, musimy stworzyć metodę obsługującą to zdarzenie.


Przykładowo, dla formy widocznej na rysunku, obsługa zdarzeń dla obu obiektów etykiet (domyślnie ***linkLabel1*** i ***linkLabel2***) może wyglądać następująco:

```
private void linkLabel1_LinkClicked(object sender,
                                   LinkLabelLinkClickedEventArgs e)
{
    Process.Start("www.onet.pl");
}

private void linkLabel2_LinkClicked(object sender,
                                   LinkLabelLinkClickedEventArgs e)
{
    Process.Start("www.wp.pl");
}
```

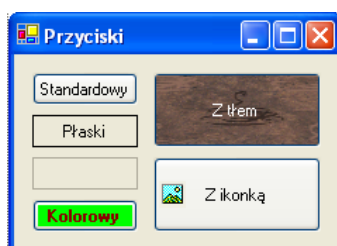
W przykładzie wykorzystaliśmy metodę ***Start*** z klasy ***Process***, która otwiera stronę o podanym adresie w domyślnej przeglądarce systemowej. Klasa ***Process*** wchodzi w skład przestrzeni ***System.Diagnostics*** (należy tu pamiętać o dodaniu przestrzeni nazw za pomocą dyrektywy ***using***).

## Przycisk

Przycisk (kontrolka  **Button**) standardowo używany jest do wywoływania skojarzonej z nim akcji.

Przycisk charakteryzuje się przede wszystkim tym, że posiada własną etykietę, wskazującą na rodzaj wykonywanej akcji oraz tym, że może zostać naciśnięty (co spowoduje wywołanie skojarzonej akcji).

W zależności od ustawionych właściwości, przycisk może zmieniać swój wygląd i zachowanie.



Rysunek 39. Projekt formy z kilkoma przyciskami różniącymi się właściwościami

## Właściwości

Kontrolka przycisku (**Button**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>BorderColor</b>	Dla przycisków płaskich (używających stylu <i>Flat</i> ) określa kolor tekstu obwoluty przycisku.	<b>ControlText</b>
<b>BorderSize</b>	Dla przycisków płaskich (używających stylu <i>Flat</i> ) określa grubość obwoluty przycisku.	<b>1</b>
<b>DialogResult</b>	Określa wartość dialogową zwracaną przez przycisk. Właściwość ta używana jest przy korzystaniu z okien dialogowych, o których mowa będzie w dalszej części książki.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>None</b> – brak wartości dialogowej</li> <li>• <b>OK</b> – przycisk akceptacji</li> <li>• <b>Cancel</b> – przycisk rezygnacji</li> <li>• <b>Abort</b> – przycisk przerwania</li> <li>• <b>Retry</b> – przycisk ponowienia</li> <li>• <b>Ignore</b> – przycisk ignorowania</li> <li>• <b>Yes</b> – przycisk potwierdzenia</li> <li>• <b>No</b> – przycisk zaprzeczenia</li> </ul>	<b>None</b>
<b>Image</b>	Pozwala na wskazanie obrazka, który ma być wyświetlany na przycisku. Obrazek może pełnić zarówno rolę ikonki przycisku jak i tła.	<b>(none)</b>
<b>ImageAlign</b>	Określa rodzaj wyrównania obrazka.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>TopLeft</b> – górny lewy</li> <li>• <b>TopCenter</b> – górny centralny</li> <li>• <b>TopRight</b> – górny prawy</li> <li>• <b>MiddleLeft</b> – środkowy lewy</li> <li>• <b>MiddleCenter</b> – środkowy centralny</li> <li>• <b>MiddleRight</b> – środkowy prawy</li> <li>• <b>BottomLeft</b> – dolny lewy</li> <li>• <b>BottomCenter</b> – dolny centralny</li> <li>• <b>BottomRight</b> – dolny prawy</li> </ul>	<b>MiddleCenter</b>

<b><i>TextImageRelation</i></b>	Określa relację między tekstem etykiety przycisku a obrazkiem.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>Overlay</i></b> – przenikanie</li> <li>• <b><i>ImageAboveText</i></b> – obrazek nad tekstem</li> <li>• <b><i>TextAboveImage</i></b> – tekst nad obrazkiem</li> <li>• <b><i>ImageBeforeText</i></b> – obrazek przed tekstem</li> <li>• <b><i>TextBeforeImage</i></b> – tekst przed obrazkiem</li> </ul>	<b><i>Overlay</i></b>
---------------------------------	--	-----------------------

## Zdarzenia

Podstawowym zdarzeniem dla tej kontrolki jest zdarzenie **Click**, które występuje gdy nastąpi naciśnięcie przycisku. Jeżeli chcemy, aby po naciśnięciu przycisku wywołana została jakaś akcja, musimy stworzyć metodę obsługującą to zdarzenie (wystarczy podwójne kliknięcie na przycisk).

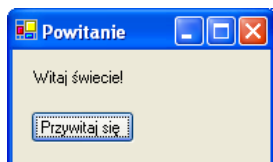
## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Powitanie”, zawierającą dwie kontrolki: pustą etykietę tekstową oraz przycisk, w którym dla zdarzenia kliknięcia stworzymy metodę ustawiającą tekst etykiety na wartość „Witaj świecie!”:

1. Tworzymy prostą aplikację **Windows Forms** i zmieniamy tytuł formy (wartość domyślną właściwości **Text** „Form1” na wartość „Powitanie”)
2. Wstawiamy kontrolkę **Label** i usuwamy wartość domyślną właściwości **Text** („label1”)
3. Wstawiamy kontrolkę **Button** i zmieniamy wartość domyślną właściwości **Text** („button1”) na wartość „Przywitaj się”
4. Klikamy podwójnie na kontrolce przycisku i dla wygenerowanej metody do zdarzenia **Click**, wpisujemy instrukcję zmieniającą wartość tekstową etykiety tak, aby metoda wyglądała następująco:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Witaj świecie!"; // ustaw tekst etykiety
}
```

5. Po uruchomieniu aplikacji i naciśnięciu przycisku, wyświetli się tekst powitalny.

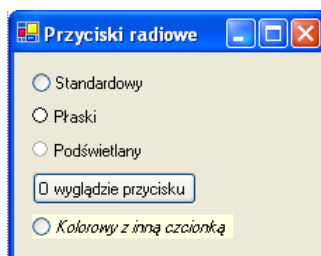


Rysunek 40. Wygląd okna aplikacji po naciśnięciu przycisku

## Przycisk radiowy

Przycisk radiowy (kontrolka **RadioButton**) standardowo używany jest do wybierania jednej z dostępnych opcji należącej do jakiejś grupy.

W zależności od ustawionych właściwości przycisk może zmieniać swój wygląd i zachowanie.



Rysunek 41. Projekt formy z kilkoma przyciskami radiowymi różniącymi się właściwościami

## Właściwości

Kontrolka przycisku radiowego (**RadioButton**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>Appearance</b>	Określa, czy przycisk radiowy ma być wyświetlany w postaci kółka czy też jako standardowy przycisk.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Normal</b> – wygląd kółka</li> <li>• <b>Button</b> – wygląd przycisku</li> </ul>	<b>Normal</b>
<b>AutoCheck</b>	Określa, czy po kliknięciu przycisk będzie zaznaczony	<b>True</b>
<b>Checked</b>	Określa, czy przycisk jest zaznaczony (czy została wybrana opcja).	<b>False</b>
<b>TextImageRelation</b>	Określa relację między tekstem etykiety przycisku a obrazkiem (dopuszczalne wartości – patrz Przycisk).	<b>Overlay</b>

Najważniejszą i najczęściej wykorzystywaną właściwością przycisku radiowego jest **Checked**. Dzięki tej właściwości można sprawdzić, który przycisk został zaznaczony. Ze zmianą tej właściwości powiązane jest również odpowiednie zdarzenie **CheckedChanged**, które występuje, jeżeli właściwość ta ulegnie zmianie (przycisk zostanie zaznaczony lub nastąpi zaznaczenie innego przycisku co spowoduje odznaczenie tego, którego dotyczy zdarzenie).

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Wybór powitania” zawierającą pięć kontroltek: pustą etykietę tekstową, trzy przyciski radiowe z tekstami: „Witam serdecznie”, „Miło mi poznać” oraz „Cześć i czołem”, a także standardowy przycisk z tekstem „Powitaj”, w którym dla zdarzenia kliknięcia stworzymy metodę ustawiającą tekst etykiety na wartość zgodną z wybraną opcją (zaznaczonym jednym z trzech przycisków radiowych).

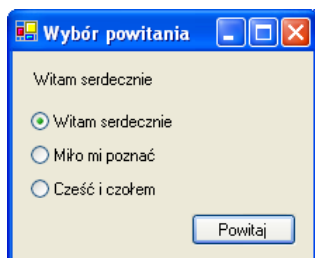
1. Tworzymy prostą aplikację **Windows Forms** i zmieniamy tytuł formy (wartość domyślną właściwości **Text** „Form1” na wartość „Wybór powitania”).
2. Wstawiamy kontrolkę **Label** i usuwamy wartość domyślną właściwości **Text** („label1”).
3. Wstawiamy kolejno trzy kontrolki **RadioButton** i zmieniamy ich domyślną właściwość **Text** („radioButton1”, „radioButton2”, „radioButton3”) na odpowiednio: „Witam serdecznie”, „Miło mi poznać” oraz „Cześć i czołem”.
4. Wstawiamy kontrolkę **Button** i zmieniamy wartość domyślną właściwości **Text** („button1”) na

wartość „Powitaj”.

5. Klikamy podwójnie na kontrolce przycisku i dla wygenerowanej metody do zdarzenia **Click** wpisujemy instrukcję zmieniającą wartość tekstową etykiety na zgodną z wybraną opcją tak, aby metoda wyglądała następująco:

```
private void button1_Click(object sender, EventArgs e)
{
    if (radioButton1.Checked) // jeżeli wybrano pierwszą opcję
        label1.Text = radioButton1.Text; // ustaw tekst etykiety na tekst opcji
    else if (radioButton2.Checked) // jeżeli wybrano drugą opcję
        label1.Text = radioButton2.Text; // ustaw tekst etykiety na tekst opcji
    else if (radioButton3.Checked) // jeżeli wybrano trzecią opcję
        label1.Text = radioButton3.Text; // ustaw tekst etykiety na tekst opcji
}
```

6. Po uruchomieniu aplikacji, wybraniu jednej z trzech opcji, a następnie naciśnięciu przycisku wyświetli się tekst powitalny.

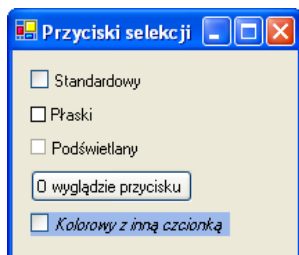


Rysunek 42. Wygląd okna aplikacji po wybraniu pierwszej opcji i naciśnięciu przycisku

## Przycisk selekcji

Przycisk selekcji (kontrolka ☒ **CheckBox**) standardowo używany jest do wybierania dowolnej ilości opcji wchodzących w skład grupy. Kontrolka ta różni się od przycisku radiowego tym, że jeżeli w danej grupie znajduje się kilka opcji, możliwe jest wybranie ich dowolnej ilości.

W zależności od ustawionych właściwości, przycisk może zmieniać swój wygląd i zachowanie.



Rysunek 41. Projekt formy z kilkoma przyciskami radiowymi różniącymi się właściwościami

## Właściwości

Kontrolka przycisku selekcji (**CheckBox**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
------------	------	------------------

<b>Appearance</b>	Określa, czy przycisk radiowy ma być wyświetlany w postaci kółka czy też jako standardowy przycisk. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Normal</b> – wygląd kółka</li> <li>• <b>Button</b> – wygląd przycisku</li> </ul>	<b>Normal</b>
<b>AutoCheck</b>	Określa czy po kliknięciu przycisk będzie zaznaczony.	<b>True</b>
<b>CheckAlign</b>	Określa rodzaj położenia pola selekcji w przycisku. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>TopLeft</b> – górny lewy</li> <li>• <b>TopCenter</b> – górny centralny</li> <li>• <b>TopRight</b> – górny prawy</li> <li>• <b>MiddleLeft</b> – środkowy lewy</li> <li>• <b>MiddleCenter</b> – środkowy centralny</li> <li>• <b>MiddleRight</b> – środkowy prawy</li> <li>• <b>BottomLeft</b> – dolny lewy</li> <li>• <b>BottomCenter</b> – dolny centralny</li> <li>• <b>BottomRight</b> – dolny prawy</li> </ul>	<b>MiddleLeft</b>
<b>Checked</b>	Określa, czy przycisk jest zaznaczony (czy została wybrana opcja).	<b>False</b>
<b>CheckState</b>	Określa stan selekcji pola w przycisku. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Unchecked</b> – pole nie jest zaznaczone</li> <li>• <b>Checked</b> – pole jest zaznaczone</li> <li>• <b>Indeterminate</b> – pole jest zaznaczone pośrednio</li> </ul>	<b>Unchecked</b>
<b>TextImageRelation</b>	Określa relację między tekstem etykiety przycisku a obrazkiem (dopuszczalne wartości – patrz Przycisk).	<b>Overlay</b>
<b>ThreeState</b>	Określa, czy przycisk selekcji może przyjmować trzy stany pola selekcji (zaznaczone, niezaznaczone oraz zaznaczone pośrednio).	<b>False</b>

Najważniejszą i najczęściej wykorzystywaną właściwością przycisku selekcji jest **Checked**. Dzięki tej właściwości można sprawdzić, który przycisk został zaznaczony. Ze zmianą tej właściwości powiązane jest również odpowiednie zdarzenie **CheckedChanged**, które występuje, jeżeli właściwość ta ulegnie zmianie.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Zakupy”, zawierającą pięć kontroltek: pustą etykietę tekstową, trzy przyciski selekcji z tekstami: „chleb”, „mleko” oraz „masło”, a także standardowy przycisk z tekstem „Wybierz”, w którym dla zdarzenia kliknięcia stworzymy metodę ustawiającą tekst zawierający listę zakupów. Jeżeli nie zostanie wybrane nic, wyświetlony zostanie tekst „Lista zakupów jest pusta”, jeżeli zaś zostaną wybrane jakieś produkty, powinien zostać wyświetlony tekst zaczynający się od „Do kupienia:”, a następnie powinny pojawić się zaznaczone produkty

oddzielone przecinkami.

1. Tworzymy prostą aplikację **Windows Forms** i zmieniamy tytuł formy (zmieniamy wartość domyślną właściwości **Text** „Form1” na wartość „Zakupy”).
2. Wstawiamy kontrolkę **Label** i usuwamy wartość domyślną właściwości **Text** („label1”).
3. Wstawiamy kolejno trzy kontrolki **CheckBox** i zmieniamy ich domyślną właściwość **Text** („checkBox1”, „checkBox2”, „checkBox3”) na odpowiednie: „chleb”, „mleko” oraz „masło”.
4. Wstawiamy kontrolkę **Button** i zmieniamy wartość domyślną właściwości **Text** („button1”) na wartość „Wybierz”.
5. Klikamy podwójnie na kontrolce przycisku i dla wygenerowanej metody do zdarzenia **Click** wpisujemy instrukcję zmieniającą wartość tekstową na przedstawiającą odpowiedni tekst, tak aby metoda wyglądała następująco:

```
private void button1_Click(object sender, EventArgs e)
{
    // zmienna pomocnicza określająca czy jakaś opcja została wybrana
    bool bPrzecinek = false;
    label1.Text = "Do kupienia: "; // początek zdania

    if (checkBox1.Checked) // jeżeli wybrano pierwszą opcję
    {
        label1.Text += checkBox1.Text; // dodaj do tekstu nazwę opcji
        bPrzecinek = true; // zaznacz, że na liście już jest jakaś opcja
    }

    if (checkBox2.Checked) // jeżeli wybrano drugą opcję
    {
        if (bPrzecinek) // jeżeli na liście jest jakaś opcja
            label1.Text += ", "; // to wstaw przed kolejną przecinek

        label1.Text += checkBox2.Text; // dodaj do tekstu nazwę opcji
        bPrzecinek = true; // zaznacz, że na liście już jest jakaś opcja
    }

    if (checkBox3.Checked) // jeżeli wybrano trzecią opcję
    {
        if (bPrzecinek) // jeżeli na liście jest jakaś opcja
            label1.Text += ", "; // to wstaw przed kolejną przecinek

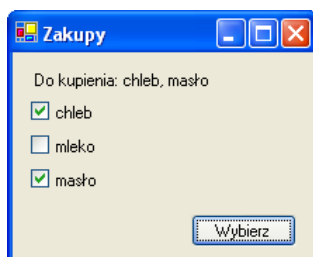
        label1.Text += checkBox3.Text; // dodaj do tekstu nazwę opcji
        bPrzecinek = true; // zaznacz, że na liście już jest jakaś opcja
    }

    if (!bPrzecinek) // jeżeli nie ma żadnej opcji
        label1.Text = "Lista zakupów jest pusta"; // wtedy ustaw inny tekst
```



}

6. Po uruchomieniu aplikacji, wybraniu dwóch opcji: „chleb” oraz „masło”, a następnie naciśnięciu przycisku, wyświetli się tekst z listą zakupów.



Rysunek 44. Wygląd okna aplikacji po wybraniu pierwszej oraz trzeciej opcji i naciśnięciu przycisku

## Pole tekstowe

Pole tekstowe (kontrolka `TextBox`) służy do wczytywania prostych danych tekstowych, wprowadzanych przez użytkownika. Dane te dostępne są poprzez właściwość **Text**.

### Właściwości

Kontrolka przycisku selekcji (**TextBox**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>AcceptsReturn</b>	Pozwala na używanie klawisza akceptacji jako dopuszczalnego znaku wejściowego wewnątrz kontrolki w trybie wieloliniowej edycji tekstu.	<b>False</b>
<b>AcceptsTab</b>	Pozwala na używanie klawisza tabulacji jako dopuszczalnego znaku wejściowego wewnątrz kontrolki w trybie wieloliniowej edycji tekstu.	<b>False</b>
<b>BorderStyle</b>	Określa, rodzaj obramowania dla pola tekstowego. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Fixed3D</b> – standardowe obramowanie 3D</li> <li>• <b>FixedSingle</b> – pojedyncza linia</li> <li>• <b>None</b> – brak obramowania</li> </ul>	<b>Fixed3D</b>
<b>CharacterCasing</b>	Określa, konwencję wyświetlania wprowadzonego tekstu. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Normal</b> – litery nie są zamieniane</li> <li>• <b>Upper</b> – małe litery zamieniane są na duże</li> <li>• <b>Lower</b> – duże litery zamieniane są na małe</li> </ul>	<b>Normal</b>
<b>HideSelection</b>	Określa, czy zaznaczony tekst wewnątrz kontrolki ma być ukrywany po jej opuszczeniu.	<b>True</b>
<b>Lines</b>	Zawiera poszczególne linie tekstu w trybie wieloliniowej edycji tekstu (właściwość ta jest tablicą łańcuchów).	
<b>Multiline</b>	Określa, czy pole tekstowe może pracować w trybie wieloliniowej edycji tekstu.	<b>True</b>

<b>PasswordChar</b>	Określa rodzaj znaku wyświetlanego w miejsce wprowadzanych danych w trybie jednoliniowej edycji tekstu.	
<b>ReadOnly</b>	Określa dostępność pola tekstowego w trybie tylko do odczytu.	<b>False</b>
<b>ScrollBars</b>	Określa, że pole tekstowe w trybie wieloliniowej edycji tekstu powinno wyświetlać paski przewijania. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Horizontal</b> – pasek poziomy</li> <li>• <b>Vertical</b> – pasek pionowy</li> <li>• <b>Both</b> – pasek poziomy i pionowy</li> </ul>	<b>None</b>
<b>WordWrap</b>	Określa, czy w trybie wieloliniowej edycji tekstu linie mają być zawijane.	<b>True</b>

Najważniejszą i najczęściej wykorzystywaną właściwością pola tekstowego w trybie jednoliniowej edycji jest **Text**. Dzięki tej właściwości, można pobrać wartość wprowadzoną przez użytkownika. Ze zmianą wartości tej właściwości powiązane jest również odpowiednie zdarzenie **TextChanged**, które występuje, jeżeli właściwość ta ulegnie zmianie. W trybie wieloliniowej edycji tekstu korzysta się przede wszystkim z właściwości: **Lines** oraz **Multiline**.

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Adresy IP”, która będzie wyświetlać listę adresów IP w polu tekstowym pozwalającym na wyświetlenie wielu linii tekstu w trybie tylko do odczytu jedynie wtedy, gdy użytkownik wpisze wyraz „adresy” w polu tekstowym służącym do wczytywania hasła:

1. Tworzymy aplikację **Windows Forms** o tytule „Adresy IP”.
2. Dodajemy kontrolkę **Label**, która będzie służyć do opisanie przeznaczenia pierwszego pola tekstowego i ustawiamy jej właściwość **Text** na wartość „Hasło”.
3. Dodajemy kontrolkę **TextBox**, która będzie służyć do wczytywania hasła i ustawiamy jej właściwości:
  - a) **Multiline** na **False**
  - b) **PasswordChar** na \*
4. Dodajemy kontrolkę **Label**, która będzie służyć do opisanie przeznaczenia drugiego pola tekstowego i ustawiamy jej właściwość **Text** na wartość „Adresy”.
5. Dodajemy kontrolkę **TextBox**, która będzie służyć do wyświetlania adresów IP i ustawiamy jej właściwości:
  - a) **ReadOnly** na **True**
  - b) **Multiline** na **True**
  - c) **ScrollBars** na **Vertical**
6. Tworzymy metodę obsługującą zdarzenie zmiany tekstu (**TextChanged**) dla pola tekstowego wczytywania hasła i sprawdzamy hasło. Jeżeli będzie poprawne, wprowadzamy dane do pola zawierającego adresy a jeżeli nie czyścimy je:

```
private void textBox1_TextChanged(object sender, EventArgs e)
```

```

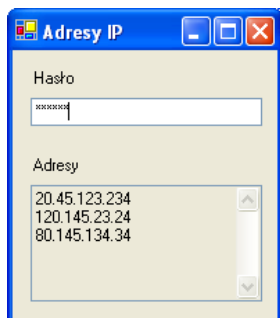
{
    if (textBox1.Text == "adresy") // proste sprawdzenie hasła
    {
        string[] dane = new string[3]; // utworzenie listy łańcuchów

        dane[0] = "20.45.123.234"; // wypełnienie listy danymi
        dane[1] = "120.145.23.24";
        dane[2] = "80.145.134.34";

        textBox2.Lines = dane; // wyświetlenie listy adresów
    }
    else
        // wyczyszczenie zawartości pola wyświetlającego adresy
        textBox2.Clear();
}

```

7. Po uruchomieniu aplikacji i wpisaniu hasła „adresy” nastąpi wyświetlenie listy adresów.



Rysunek 45. Wygląd okna aplikacji po wprowadzeniu poprawnego hasła

## Pole tekstowe z wzorcem

Pole tekstowe z wzorcem (kontrolka `[# MaskedTextBox]`) jest odmianą pola tekstowego, pozwalającego określić format akceptowanych danych wejściowych (tzw. maskę pola).

### Właściwości

Kontrolka przycisku selekcji (*MaskedTextBox*) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>AsciiOnly</i>	Określa, że dozwolone są jedynie znaki ASCII.	<i>False</i>
<i>BeepOnError</i>	Określa, czy po próbie wprowadzenia znaku niepasującego do wzorca, ma zostać wysłany sygnał dźwiękowy sygnalizacji błędu.	<i>False</i>
<i>BorderStyle</i>	Określa rodzaj obramowania dla pola tekstowego z wzorcem (patrz pole tekstowe).	<i>Fixed3D</i>
<i>HidePromptOnLeave</i>	Określa, czy znak oczekiwania będzie wyświetlany po wyjściu z kontrolki.	<i>True</i>
<i>HideSelection</i>	Określa, czy zaznaczony tekst wewnątrz kontrolki ma	<i>True</i>

	być ukrywany po jej opuszczeniu.	
<b><i>IncludeLiterals</i></b>	Określa, czy napis dostępny poprzez właściwość <b><i>Text</i></b> ma mieć dołączane również wartości literalne.	<b><i>True</i></b>
<b><i>IncludePrompt</i></b>	Określa, czy napis dostępny poprzez właściwość <b><i>Text</i></b> ma mieć dołączony również znak oczekiwania.	<b><i>True</i></b>
<b><i>InputText</i></b>	Tekst wprowadzony do kontrolki bez formatowania.	
<b><i>InsertMode</i></b>	Określa tryb nadpisywania danych w kontrolce. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>InsertKeyMode</i></b> – wstawienie znaku</li> <li>• <b><i>On</i></b> – włączone</li> <li>• <b><i>Off</i></b> – wyłączony</li> </ul>	<b><i>InsertKeyMode</i></b>
<b><i>Mask</i></b>	Określa wzorzec dla wprowadzanych danych. Przykładowo: <b><i>00/00/0000</i></b> – oznacza datę w formacie dzień, numer miesiąca, rok Wzorzec musi składać się z jednego lub więcej znaków określających format akceptowanych danych wejściowych. Dozwolone są następujące znaki: <ul style="list-style-type: none"> <li>• <b><i>0</i></b> – cyfra 0..9, pole obowiązkowe</li> <li>• <b><i>9</i></b> – cyfra 0..9, pole nie obowiązkowe</li> <li>• <b><i>#</i></b> – cyfra 0..9 lub spacja, pole nie obowiązkowe</li> <li>• <b><i>L</i></b> – litera, pole obowiązkowe</li> <li>• <b><i>?</i></b> – litera, pole nie obowiązkowe</li> <li>• <b><i>&amp;</i></b> – znak, pole obowiązkowe</li> <li>• <b><i>C</i></b> – znak, pole nie obowiązkowe</li> <li>• <b><i>A</i></b> – znak alfanumeryczny, pole obowiązkowe</li> <li>• <b><i>a</i></b> – znak alfanumeryczny, pole nie obowiązkowe</li> <li>• <b><i>.</i></b> – znak separacji dziesiętnej</li> <li>• <b><i>,</i></b> – znak separacji tysięcznej</li> <li>• <b><i>:</i></b> – znak separacji czasu</li> <li>• <b><i>/</i></b> – znak separacji daty</li> <li>• <b><i>\$</i></b> – symbol waluty</li> <li>• <b><i>&lt;</i></b> – konwersja wszystkich znaków na małe litery</li> <li>• <b><i>&gt;</i></b> – konwersja wszystkich znaków na duże litery</li> <li>• <b><i> </i></b> – zablokowanie poprzedniej operacji konwersji znaków na małe lub duże litery</li> <li>• <b><i>pozostale</i></b> – literały</li> </ul>	
<b><i>PasswordChar</i></b>	Określa rodzaj znaku wyświetlanego w miejsce wprowadzanych danych.	
<b><i>PromptChar</i></b>	Określa rodzaj znaku oczekiwania.	–
<b><i>ReadOnly</i></b>	Określa dostępność pola tekstowego z wzorcem	<b><i>False</i></b>

	w trybie tylko do odczytu.	
--	----------------------------	--

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<i>InputTextChanged</i>	Występuje, gdy nastąpi zmiana właściwości <i>InputText</i> .
<i>OutputTextChanged</i>	Występuje, gdy nastąpi zmiana właściwości <i>OutputText</i> .
<i>MaskInputRejected</i>	Występuje, gdy następuje próba wprowadzenia danych niezgodnych z wzorcem.

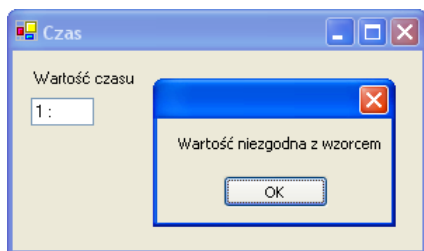
## Przykładowa aplikacja

Stworzmy sobie prostą aplikację *Windows Forms* z tytułem „Czas”, która będzie wczytywać czas w formacie "00:00" i wyświetlać komunikaty o błędzie, jeżeli wprowadzona wartość nie będzie zgodna z wzorcem:

1. Tworzymy aplikację *Windows Forms* o tytule „Czas”.
2. Dodajemy kontrolkę *Label*, która będzie służyć do opisanego przeznaczenia pola tekstowego z wzorcem i ustawiamy jej właściwość *Text* na wartość: „Wartość czasu”.
3. Dodajemy kontrolkę *MaskedTextBox* i wartość właściwości *Mask* na "00:00".
4. Tworzymy metodę obsługującą zdarzenie odrzucenia wartości z powodu niezgodności z wzorcem *MaskInputRejected* i wyświetlamy komunikat.


```
private void maskedTextBox1_MaskInputRejected(object sender,
                                             MaskInputRejectedEventArgs e)
{
    MessageBox.Show("Wartość niezgodna z wzorcem");
}
```

5. Po uruchomieniu aplikacji i wpisaniu znaku niezgodnego z wzorcem (np.: litery) wyświetlony zostanie komunikat o błędzie:



Rysunek 46. Wygląd okna po wpisaniu znaku niezgodnego z wzorcem (na pierwszym planie widać okno z komunikatem)

## Lista prosta

Lista prosta (kontrolka  *ListBox*) pozwala na wyświetlanie listy elementów, które użytkownik może wybierać poprzez ich wskazanie. Lista prosta pozwala zarówno na selekcję pojedynczego elementu, jak również wielu elementów.

## Właściwości

Kontrolka listy prostej (**ListBox**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>ColumnWidth</b>	Określa szerokość kolumn listy.	<b>0</b>
<b>IntegralHeight</b>	Określa, czy lista powinna zawierać pełne elementy.	<b>True</b>
<b>ItemHeight</b>	Określa wysokość (w pikselach) wiersza danych listy rysowanej przez użytkownika.	<b>13</b>
<b>Items</b>	Zawiera kolekcję elementów listy ( <b>ListItemCollection</b> )	
<b>Multicolumn</b>	Określa, czy elementy listy mają być wyświetlane w wielu kolumnach.	<b>False</b>
<b>ScrollAlwaysVisible</b>	Określa, czy paski przewijania mają być zawsze widoczne.	<b>False</b>
<b>SelectionMode</b>	Określa rodzaj selekcji elementów listy. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>None</b> – brak selekcji elementów</li> <li>• <b>One</b> – selekcja pojedyncza</li> <li>• <b>Multisimple</b> – prosta selekcja wielu elementów</li> <li>• <b>Multiextended</b> – rozszerzona selekcja wielu elementów (użytkownik może używać klawiszy selekcji: SHIFT, CTRL oraz strzałek)</li> </ul>	<b>One</b>
<b>Sorted</b>	Określa, czy elementy listy mają być sortowane.	<b>False</b>

Lista prosta posiada również grupę właściwości, które dostępne są jedynie na poziomie kodu programu i związane są przede wszystkim z operacjami zarządzania selekcją elementów listy.

Do właściwości tych należą:

Właściwość	Opis
<b>SelectedIndex</b>	Odczytuje lub zapisuje indeks aktualnie wybranego elementu w liście.
<b>SelectedItem</b>	Odczytuje lub zapisuje aktualnie wybrany element listy.
<b>SelectedItems</b>	Zwraca kolekcję aktualnie wybranych elementów listy.

## Zdarzenia

Najważniejszym zdarzeniem dla listy prostej jest zdarzenie **SelectedIndexChanged**, czyli zmiany aktualnie wybranego elementu (zmiany wartości właściwości **SelectedIndex**).

## Kolekcja elementów

Najważniejszą i najczęściej wykorzystywaną właściwością listy jest kolekcja elementów **Items**. Dzięki tej właściwości, można zarządzać kolekcją elementów listy (dodawać, usuwać i zmieniać wartości). Funkcjonalność kolekcji elementów listy jest realizowana za pomocą klasy **ListItemCollection** (kolekcja elementów **ListItem**).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<i>Count</i>	Zwraca liczbę elementów kolekcji.
<i>IsReadOnly</i>	Sprawdza czy kolekcja dostępna jest w trybie tylko do odczytu.
<i>Item</i>	Zwraca element kolekcji o określonym indeksie.

Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<i>Add</i>	Pozwala na dodanie elementu do kolekcji.
<i>AddRange</i>	Pozwala na dodanie kilku elementów do kolekcji.
<i>Clear</i>	Pozwala na usunięcie wszystkich elementów z kolekcji.
<i>Contains</i>	Pozwala na sprawdzenie, czy w kolekcji znajduje się określony element.
<i>FindByText</i>	Pozwala na znalezienie w kolekcji elementu zawierającego podany tekst.
<i>FindByValue</i>	Pozwala na znalezienie w kolekcji elementu zawierającego podaną wartość.
<i>IndexOf</i>	Zwraca indeks podanego elementu kolekcji.
<i>Remove</i>	Pozwala na usunięcie z kolekcji podanego elementu.
<i>RemoveAt</i>	Pozwala na usunięcie z kolekcji elementu o określonym indeksie.

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Pliki”, która będzie wyświetlać listę plików znajdujących się w jakimś folderze. Ścieżkę do folderu wczytywać będziemy za pomocą pola tekstowego o nazwie „Ścieżka”. Wyświetlenie plików nastąpić będzie po naciśnięciu przycisku „Wyświetl”, o ile zostanie podana poprawna ścieżka. W przypadku podania niepoprawnej ścieżki wyświetlony zostanie komunikat o błędzie:

1. Tworzymy aplikację **Windows Forms** o tytule „Pliki”.
2. Dodajemy kontrolkę **Label**, która będzie służyć do opisanego przeznaczenia pola tekstowego do wczytywania ścieżki i ustawiamy jej właściwość **Text** na wartość „Ścieżka”.
3. Dodajemy kontrolkę **TextBox**, która będzie służyć do wczytywania ścieżki.
4. Dodajemy kontrolkę **Label**, która będzie służyć do opisanego przeznaczenia listy wyświetlającej pliki i ustawiamy jej właściwość **Text** na wartość „Lista plików w folderze”.
5. Dodajemy kontrolkę **ListBox**, która będzie służyć do wyświetlania listy plików w folderze.
6. Dodajemy kontrolkę **Button**, która będzie służyć do wypełniania naszej listy listą plików dostępnych w danym folderze i ustawiamy jej właściwość **Text** na wartość „Wypełnij”.
7. Wybieramy właściwość formy **AcceptButton** i wybieramy nasz przycisk, dzięki czemu wprowadzenie nazwy folderu i naciśnięcie klawisza **ENTER**, będzie równoważne naciśnięciu przycisku „Wypełnij”.
8. Tworzymy metodę obsługującą zdarzenie kliknięcia przycisku **Click** i dodajemy odpowiednią funkcjonalność (poprzez dwukrotne naciśnięcie na przycisk w projekcie).

```
private void button1_Click(object sender, EventArgs e)
{
    string NazwaFolderu = textBox1.Text.Trim(); // usuwamy spacje
```

```

if (!Directory.Exists(NazwaFolderu)) // jeżeli nie istnieje
{
    // komunikat wyświetlany w przypadku niezalezienia folderu
    MessageBox.Show("Podana ścieżka nie jest poprawna.");
    textBox1.Text = ""; // usuwamy wprowadzoną ścieżkę
    textBox1.Focus(); // aktywujemy kontrolkę (ponieważ straciła focus)

    return; // przerywamy dalsze wykonywanie
}

listBox1.Items.Clear(); // usuwamy poprzednią zawartość kolekcji

// pobieramy listę plików dla wprowadzonego folderu
string[] ListaPlikow = Directory.GetFiles(NazwaFolderu);

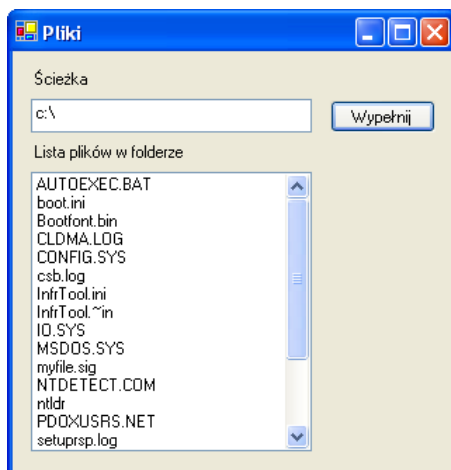
// wypełniamy listę samymi nazwami plików (bez ścieżki)
foreach (string Nazwa in ListaPlikow)
    listBox1.Items.Add(Path.GetFileName(Nazwa));
}

```

9. Ponieważ używaliśmy funkcjonalności związanej z plikami, musimy włączyć do listy używanych przestrzeni nazw przestrzeń **System.IO** (rozwijamy sekcję **Using directives** i wpisujemy odpowiednią deklarację):


```
using System.IO;
```

10. Po uruchomieniu aplikacji, wpisaniu dowolnej istniejącej ścieżki do folderu np. C:\ i naciśnięciu przycisku „Wypełnij”, zostanie wyświetlona lista plików:



Rysunek 47. Wygląd okna po wpisaniu ścieżki c:\ i naciśnięciu przycisku (widoczna jest lista plików)

## Lista selekcji

Lista selekcji (kontrolka  **CheckedListBox**) pozwala na wyświetlanie listy elementów, które można selekcjonować w podobny sposób jak pola selekcji.



## Właściwości

Kontrolka listy selekcji (*CheckedListBox*) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>ColumnWidth</i>	Określa szerokość kolumn listy selekcji.	<i>0</i>
<i>IntegralHeight</i>	Określa, czy lista powinna zawierać pełne elementy.	<i>True</i>
<i>Items</i>	Zawiera kolekcję elementów listy ( <i>ListItemCollection</i> ). Kolekcja elementów omówiona została przy liście prostej.	
<i>Multicolumn</i>	Określa, czy elementy listy mają być wyświetlane w wielu kolumnach.	<i>False</i>
<i>ScrollAlwaysVisible</i>	Określa, czy paski przewijania mają być zawsze widoczne.	<i>False</i>
<i>SelectionMode</i>	Określa rodzaj selekcji elementów listy (patrz lista prosta).	<i>One</i>
<i>Sorted</i>	Określa, czy elementy listy mają być sortowane.	<i>False</i>

Lista selekcji posiada również grupę właściwości, które dostępne są jedynie na poziomie kodu programu i związane są przede wszystkim z operacjami zarządzania selekcją elementów listy.

Do właściwości tych należą:

Właściwość	Opis
<i>CheckedItems</i>	Zwraca kolekcję elementów, dla których zaznaczone zostały pola selekcji.
<i>SelectedIndex</i>	Odczytuje lub zapisuje indeks aktualnie wybranego elementu w liście.
<i>SelectedItem</i>	Odczytuje lub zapisuje aktualnie wybrany element listy.
<i>SelectedItems</i>	Zwraca kolekcję aktualnie wybranych elementów listy.

## Zdarzenia

Najważniejszym zdarzeniem dla listy selekcji jest zdarzenie *SelectedIndexChanged*, czyli zmiany aktualnie wybranego elementu (zmiany wartości właściwości *SelectedIndex*). Selekcja w przypadku listy selekcji, nie oznacza zaznaczenia pola selekcji, tylko wybranie elementu (jak w przypadku listy prostej). Kolekcja elementów z zaznaczonymi polami selekcji dostępna jest za pomocą właściwości *CheckedItems*.

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację *Windows Forms* z tytułem „Lista zakupów”, która zawierać będzie listę standardowych produktów spożywczych. Jeżeli użytkownik wyselekcjonuje dane produkty (zaznaczając przycisk selekcji w liście), to po naciśnięciu przycisku „Lista”, powinien otrzymać komunikat z listą produktów, które ma kupić.

1. Tworzymy aplikację *Windows Forms* o tytule „Lista zakupów”.
2. Dodajemy kontrolkę *Label*, która będzie służyć do opisanego przeznaczenia listy wyświetlającej produkty i ustawiamy jej właściwość *Text* na wartość „Produkty”.

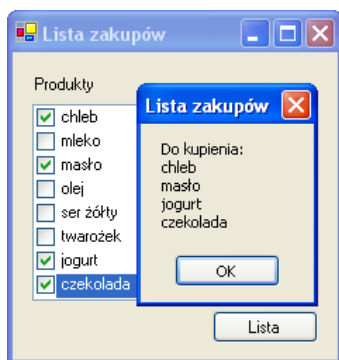
3. Dodajemy kontrolkę **CheckedListBox**, która będzie służyć do wyświetlania listy najczęściej kupowanych produktów.
4. Dodajemy kontrolkę **Button**, która będzie służyć do wyświetlania wybranych produktów i ustawiamy jej właściwość **Text** na wartość „Lista”.
5. Wybieramy właściwość formy **AcceptButton** i wybieramy nasz przycisk, dzięki czemu wprowadzenie nazwy folderu i naciśnięcie klawisza **ENTER**, będzie równoważne naciśnięciu przycisku „Lista”.
6. Tworzymy metodę obsługującą zdarzenie kliknięcia przycisku **Click** i dodajemy odpowiednią funkcjonalność (poprzez dwukrotne kliknięcie na przycisk w projekcie).

```
private void button1_Click(object sender, EventArgs e)
{
    string lista = "Do kupienia:\n"; // początek komunikatu

    // dołącz do komunikatu każdy z zaznaczonych na liście produktów
    for (int i = 0; i < checkedListBox1.CheckedItems.Count; i++)
    {
        lista += checkedListBox1.CheckedItems[i].ToString();
        lista += "\n"; // nowa linia
    }


    // wyświetl komunikat z tytułem okna "Lista zakupów"
    MessageBox.Show(lista, "Lista zakupów");
}
```

7. Po uruchomieniu aplikacji, zaznaczeniu dowolnych produktów i naciśnięciu przycisku „Lista”, zostanie wyświetlony komunikat z listą produktów do kupienia:



Rysunek 48. Wygląd okna po wybraniu listy produktów i wciśnięciu przycisku „Lista”

## Lista rozwijana

Lista rozwijana (kontrolka  **ComboBox**), pozwala na wyświetlanie rozwijanej listy elementów, które użytkownik może wybierać poprzez ich wskazanie. Lista rozwijana zajmuje niewiele miejsca i stanowi rodzaj hybrydy między listą a polem tekstowym (pozwala wprowadzać dane w taki sam sposób jak pole tekstowe lub wybierać wartość pola z elementów listy).

## Właściwości

Kontrolka listy rozwijanej (**ComboBox**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>DropDownHeight</b>	Określa wysokość w pikselach okna zawierającego listę rozwijaną.	<b>106 (bez rozszerzania)</b>
<b>DropDownStyle</b>	Określa styl i zachowanie listy rozwijanej. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Simple</b> – można modyfikować tekst wyświetlany w oknie, można wybierać wartości z listy tylko za pomocą klawiszy strzałek</li> <li>• <b>DropDown</b> – można modyfikować tekst wyświetlany w oknie, można swobodnie wybierać wartości z listy</li> <li>• <b>DropDownList</b> – nie można modyfikować tekstu wyświetlanego w oknie, można wybierać swobodnie wartości z listy</li> </ul>	<b>DropDown</b>
<b>DropDownWidth</b>	Określa szerokość w pikselach okna listy rozwijanej.	<b>121 (bez rozszerzania)</b>
<b>Items</b>	Zawiera kolekcję elementów listy ( <b>ListItemCollection</b> ). Kolekcja elementów omówiona została przy liście prostej.	
<b>Sorted</b>	Określa, czy elementy listy mają być sortowane.	<b>False</b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<b>DropDown</b>	Występuje, gdy lista zostanie rozwinięta.
<b>DropDownClosed</b>	Występuje, gdy nastąpi zwinięcie listy rozwijanej.
<b>SelectedIndexChanged</b>	Występuje, gdy zmieni się indeks wyświetlanego elementu.
<b>SelectedValueChanged</b>	Występuje, gdy zmieni się wyświetlana wartość wyświetlanego.
<b>SelectionChangeCommitted</b>	Występuje, gdy zostanie wybrany element z listy i lista zostanie zwinięta.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Napędy”, która pozwalać będzie na wybranie z listy rozwijanej symbolu napędu. Lista rozwijana zawierać będzie jedynie symbole napędów dostępnych w systemie. Lista powinna być dostępna jedynie w trybie selekcji bez możliwości wprowadzania wartości. Jeżeli użytkownik wybierze symbol z listy powinien otrzymać komunikat informujący go, jaki symbol napędu został wybrany.

1. Tworzymy aplikację **Windows Forms** o tytule „Napędy”.
2. Dodajemy kontrolkę **Label**, która będzie służyć do opisanego przeznaczenia listy rozwijanej

wyświetlającej aktualnie wybrany symbol napędu i ustawiamy jej właściwość **Text** na wartość „Symbol napędu”.

3. Dodajemy kontrolkę **ComboBox**, która będzie służyć do wyświetlania listy napędów dostępnych w systemie i ustawiamy właściwość **DropDownStyle** na **DropDownList**.

4. Wypełniamy zawartość listy rozwijanej przed pojawieniem się formy. W tym celu tworzymy metodę dla zdarzenia **Load** formy:

```
private void Form1_Load(object sender, EventArgs e)
{
    // pobieramy listę symboli napędów dostępnych w systemie
    string[] lista = Directory.GetLogicalDrives();

    // czyścimy zawartość listy rozwijanej (na wszelki wypadek)
    comboBox1.Items.Clear();

    // wypełniamy symbolami napędów
    foreach (string napęd in lista)
        comboBox1.Items.Add(napęd);
}
```

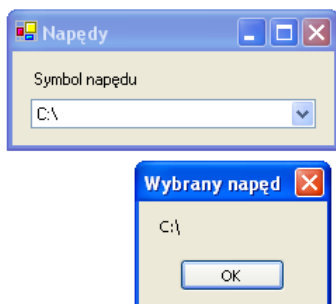
5. Ponieważ używaliśmy funkcjonalności związanej z folderami musimy włączyć do listy używanych przestrzeni nazw, przestrzeń **System.IO** (rozwijamy sekcję **Using directives** i wpisujemy odpowiednią deklarację):

```
using System.IO;
```

6. W przypadku, gdy użytkownik wybierze symbol z listy, powinien pojawić się komunikat wyświetlający wybrany symbol napędu. Tworzymy metodę obsługującą zdarzenie zmiany indeksu elementu listy **SelectedIndexChanged** i dodajemy odpowiednią funkcjonalność


```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show(comboBox1.Text, "Wybrany napęd");
}
```

7. Po uruchomieniu aplikacji i wybraniu symbolu, zostanie wyświetlony odpowiedni komunikat:



Rysunek 49. Wygląd okna po wybraniu symbolu napędu

## Pole grupujące

Pole grupujące (kontrolka  **GroupBox**) służy do grupowania kontroltek, które można zaliczyć do

wspólnej grupy (z punktu widzenia logiki aplikacji np.: opcje wyświetlania). Kontrolkę wykorzystuje się głównie do grupowania innych kontroltek. Pozwala to na ukrywanie lub blokowanie interakcji z użytkownikiem (wyszarzanie) wszystkich kontroltek należących do grupy w czasie działania programu.

## Właściwości

Kontrolka nie posiada charakterystycznych dla siebie właściwości i zdarzeń, które można wykorzystać. Do najczęściej używanych właściwości zalicza się właściwość **Visible**, która pozwala na ukrycie pola grupującego wraz z należącymi do tego pola kontrolkami, oraz właściwość **Enabled**, która pozwala zablokować interakcję z użytkownikiem dla pola grupy, wraz z należącymi do tego pola kontrolkami.

## Przykładowa aplikacja

Stworzmy sobie prostą aplikację **Windows Forms** z tytułem „Lista folderów i plików”, która składać się będzie z trzech grup: „Opcje wyświetlania” (pozwalające na wybór opcji wyświetlania: folderów i plików, samych plików lub samych folderów), „Napęd” (pozwalające na wybór symbolu napędu, dla którego wyświetlane będą foldery i pliki) oraz „Lista” (wyświetlające w zależności od wybranych opcji wyświetlania listę folderów i plików dla wybranego napędu).

1. Tworzymy aplikację **Windows Forms** o tytule „Lista folderów i plików”.
2. Dodajemy kontrolkę **GroupBox**, która będzie służyć do grupowania opcji wyświetlania i ustawiamy jej właściwość **Text** na wartość „Opcje wyświetlania” (możemy również ustawić właściwość **Dock** na **Top**).
3. Umieszczamy na obszarze tego pola grupującego kontrolkę **CheckBox** i ustawiamy jej właściwości **Text** na „Pokaż foldery” (możemy również ustawić właściwość **Checked** na **True**).
4. Umieszczamy na obszarze tego pola grupującego kontrolkę **CheckBox** (obok lub pod poprzednią kontrolką **CheckBox**) i ustawiamy jej właściwości **Text** na „Pokaż pliki” (możemy również ustawić właściwość **Checked** na **True**).
5. Dodajemy kontrolkę **GroupBox** (pod poprzednią kontrolką **GroupBox**), która będzie służyć do wyświetlania listy rozwijanej z napędami dostępnymi w systemie i ustawiamy jej właściwość **Text** na wartość „Napęd” (możemy również ustawić właściwość **Dock** na **Top**).
6. Umieszczamy na obszarze tego pola grupującego kontrolkę **ComboBox**, która będzie służyć do wyświetlania listy napędów dostępnych w systemie i ustawiamy właściwość **DropDownStyle** na **DropDownList**.
7. Wypełniamy zawartość listy rozwijanej przed pojawieniem się formy. W tym celu tworzymy metodę dla zdarzenia **Load** formy:

```
private void Form1_Load(object sender, EventArgs e)
{
    // pobieramy listę symboli napędów dostępnych w systemie
    string[] lista = Directory.GetLogicalDrives();

    // czyścimy zawartość listy rozwijanej (na wszelki wypadek)
    comboBox1.Items.Clear();
}
```

```
// wypełniamy symbolami napędów
foreach (string naped in lista)
    comboBox1.Items.Add(naped);
}
```

8. Ponieważ używaliśmy funkcjonalności związanej z folderami i plikami musimy włączyć do listy używanych przestrzeni nazw, przestrzeń **System.IO** (rozwijamy sekcję *Using directives* i wpisujemy odpowiednią deklarację):

```
using System.IO;
```

9. Dodajemy kontrolkę **GroupBox** (pod poprzednią kontrolką **GroupBox**), która będzie służyć do wyświetlania listy folderów i plików dla wybranego napędu i ustawiamy jej właściwość **Text** na wartość „Lista” (możemy również ustawić właściwość **Dock** na **Fill**).

10. Umieszczamy na obszarze tego pola grupującego kontrolkę **ListBox**, która będzie służyć do wyświetlania listy folderów i plików dla wybranego napędu.

11. W przypadku, gdy użytkownik wybierze symbol napędu powinno nastąpić odświeżenie listy folderów i plików dla wybranego napędu. W czasie wyświetlania należy sprawdzić, jakie opcje wyświetlania zostały wybrane i w zależności od nich wyświetlamy foldery i pliki. Tworzymy metodę obsługującą zdarzenie zmiany indeksu elementu listy **SelectedIndexChanged** i dodajemy odpowiednią funkcjonalność:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (checkBox1.Text.Length == 0)    // jeżeli nie ma symbolu napędu
        return;                      // kończymy działanie metody

    listBox1.Items.Clear(); // czyścimy listę

    // jeżeli zaznaczono opcję wyświetlania folderów dodajemy je do listy
    if (checkBox1.Checked)
    {
        string[] foldery = Directory.GetDirectories(comboBox1.Text);

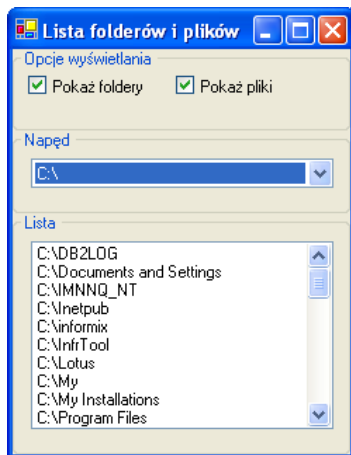
        foreach (string folder in foldery)
            listBox1.Items.Add(folder);
    }

    // jeżeli zaznaczono opcję wyświetlania plików dodajemy je do listy
    if (checkBox2.Checked)
    {
        string[] pliki = Directory.GetFiles(comboBox1.Text);

        foreach (string plik in pliki)
            listBox1.Items.Add(plik);
    }
}
```


12. Należy jeszcze wykonać jakąś akcję w przypadku zmiany opcji po wyświetleniu listy folderów i plików. Ponieważ powinna zostać wykonana ta sama funkcjonalność co w przypadku wyboru symbolu napędu z listy wystarczy, że do zdarzeń **CheckedChanged** obu kontroltek **CheckBox** podepiemy metodę **comboBox1\_SelectedIndexChanged** (wybierając w oknie **Properties** dla tych zdarzeń odpowiednią metodę z listy).

13. Po uruchomieniu aplikacji i wybraniu symbolu zostanie wyświetlona lista folderów i plików dla tego napędu



Rysunek 50. Wygląd okna po wybraniu symbolu napędu

## Pole obrazkowe

Pole obrazkowe (kontrolka  **PictureBox**) służy do wyświetlania obrazków zarówno wchodzących w skład zasobów programu jak i zapisanych w plikach zewnętrznych (np.: BMP czy JPEG).

## Właściwości

Kontrolka pola obrazkowego (**PictureBox**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>Image</b>	Określa obrazek wyświetlany w kontrolce (obrazek wchodzi w skład zasobów programu).	
<b>ImageLocation</b>	Określa ścieżkę do pliku zawierającego obrazek, który będzie wyświetlany w kontrolce.	
<b>InitialImage</b>	Określa początkowy rodzaj obrazka (przed załadowaniem właściwego obrazka).	<b>System.Drawing.Bitmap</b>
<b>SizeMode</b>	Określa, w jaki sposób obrazek jest wyświetlany. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>AutoSize</b> – kontrolka przyjmuje rozmiar zgodny z wielkością obrazka</li> <li>• <b>CenterImage</b> – wewnątrz kontrolki wyświetlony zostanie środek obrazka w przypadku, gdy jego rozmiar jest większy od rozmiaru kontrolki</li> <li>• <b>Normal</b> - wewnątrz kontrolki wyświetlona zostanie lewa górna część obrazka</li> </ul>	<b>Normal</b>

	<p>w przypadku, gdy jego rozmiar jest większy od rozmiaru kontrolki</p> <ul style="list-style-type: none"> <li>• <b>StretchImage</b> – wielkość obrazka zostanie dopasowana do wielkości kontrolki bez zachowania proporcji obrazka</li> <li>• <b>Zoom</b> – wielkość obrazka zostanie dopasowana do wielkości kontrolki z zachowaniem proporcji obrazka</li> </ul>	
<b>WaitOnLoad</b>	Określa, czy dalsze przetwarzanie ma zostać wstrzymane do czasu wczytania obrazka.	<b>False</b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

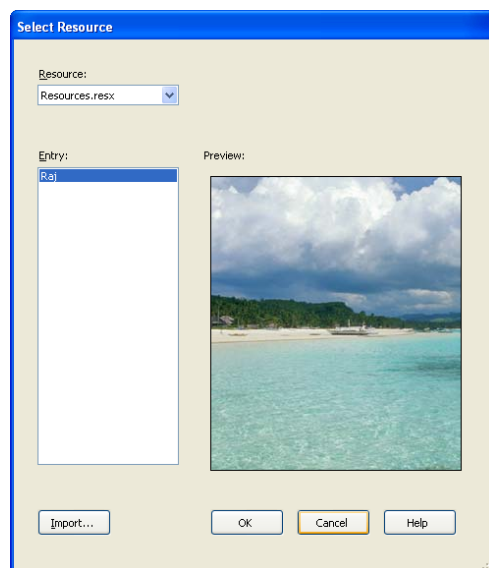
Zdarzenie	Opis
<b>LoadCompleted</b>	Zachodzi, gdy zakończy się proces wczytywania obrazka.
<b>LoadProgressChanged</b>	Zachodzi, gdy zmienia się poziom zaawansowania procesu wczytywania obrazka.
<b>SizeModeChanged</b>	Zachodzi, gdy nastąpi zmiana właściwości <b>SizeMode</b> .

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Program testowy”, która będzie wyświetlać po uruchomieniu okno powitalne (tzw. splash screen). Okno powitalne będzie widoczne przez 3 sekundy na ekranie, po czym zniknie.

1. Tworzymy aplikację **Windows Forms** o tytule „Program testowy”.
2. Dodajemy do projektu nową formę, która będzie pełnić rolę okna powitalnego i zmieniamy następujące właściwości:
  - a) **FormBorderStyle** na **None** (dzięki czemu nie będzie wyświetlana ramka okna),
  - b) **StartPosition** na **CenterScreen** (dzięki czemu okno powitalne będzie wyświetlane na środku ekranu),
  - c) **TopMost** na **True** (dzięki czemu będzie zawsze widoczne na pierwszym planie).
3. Na formie okna powitalnego umieszczamy kontrolkę **PictureBox** i zmieniamy następujące właściwości:
  - a) **Dock** na **Fill** (dzięki czemu obrazek wypełni okno powitalne).
  - b) **Image** importując jakiś obrazek, który będzie wyświetlany w oknie powitalnym (naciskamy przycisk trzech kropek, co spowoduje pojawienie się okna importu obrazów, następnie naciskamy klawisz **Import** i wskazujemy lokalizację obrazka, który ma być wyświetlany w oknie powitalnym, a na końcu naciskamy przycisk **OK**).





Rysunek 51. Okno importu obrazków kontrolki PictureBox z przykładowym zdjęciem

c) **SizeMode** na **StretchImage** (dzięki czemu obrazek zostanie dopasowany do rozmiarów okna).

4. Aby okno powitalne zamknęło się automatycznie po upływie wyznaczonego czasu, skorzystamy z kontrolki **Timer** (która zostanie omówiona bardziej szczegółowo w dalszej części książki).

5. Ustawiamy własność **Interval** kontrolki **Timer** na 3000 (milisekund) oraz dodajemy obsługę dla zdarzenia **Tick**. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void timer1_Tick(object sender, EventArgs e)
{
    Close(); // zamknięcie okna
}
```

6. Dodajemy obsługę zdarzenia wyświetlenia formy okna powitalnego, która rozpocznie proces odmierzenia czasu, po którym okno zostanie zamknięte w momencie, w którym pojawi się ono na ekranie. Metoda obsługująca to zdarzenie powinna wyglądać tak:

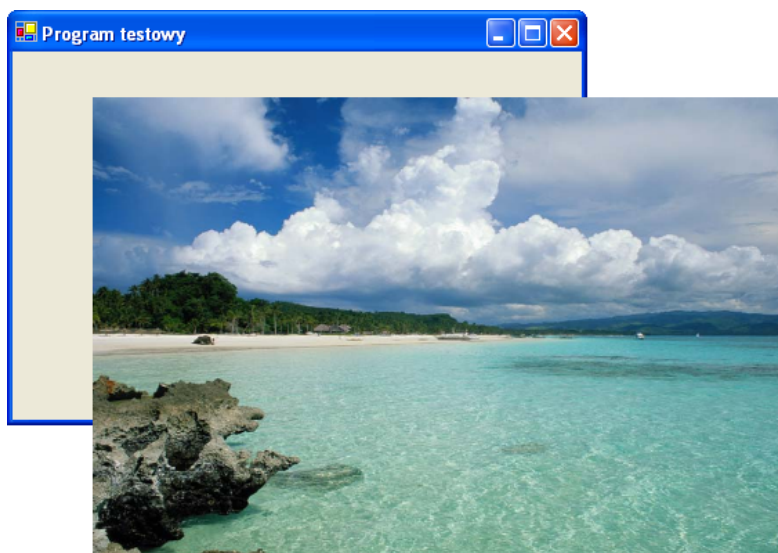
```
private void Form2_Load(object sender, EventArgs e)
{
    timer1.Start(); // zaczynamy odliczanie czasu
}
```

7. W oknie podstawowym aplikacji musimy jeszcze dodać funkcjonalność pozwalającą na wyświetlenie okna powitalnego. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void Form1_Load(object sender, EventArgs e)
{
    Form2 splash = new Form2();


    splash.Show();
}
```

8. Po uruchomieniu aplikacji wyświetlone zostanie okno powitalne, które zamknie się automatycznie po upływie 3 sekund.



Rysunek 52. Okno aplikacji wraz z oknem powitalnym (tutaj zdjęcie)

## Panel

Panel (kontrolka  **Panel**) służy jako kontener dla innych kontroltek. Wykorzystuje się ją głównie do tworzenia bardziej złożonych okien, zawierających dużą liczbę kontroltek.

## Właściwości

Kontrolka nie posiada charakterystycznych dla siebie właściwości i zdarzeń, które można wykorzystać. Do najczęściej używanych właściwości zalicza się właściwość **Visible**, która pozwala na ukrycie panela wraz z należącymi do niego kontrolkami oraz właściwość **Enabled**, która pozwala zablokować interakcję z użytkownikiem dla panelu wraz z należącymi do niego kontrolkami.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Obszary”, która składać się będzie z pięciu obszarów, które będą miały rozmiar proporcjonalny do rozmiarów okna. Trzy z tych obszarów powinny wyświetlać się z lewej strony jeden pod drugim, a dwa z prawej również jeden pod drugim. W każdym obszarze umieścimy kontrolkę wyświetlającą obrazek.

1. Tworzymy aplikację **Windows Forms** o tytule „Obszary”.
2. Dodajemy lewy panel i ustawiamy jego właściwość **Dock** na **Left**.
3. Dodajemy prawy panel i ustawiamy jego właściwość **Dock** na **Fill**.
4. Do lewego obszaru dodajemy trzy kontrolki **PictureBox** i ustawiamy ich właściwość **Dock** na kolejno: **Top**, **Bottom**, **Fill** (pierwsza będzie wyświetlana u góry, druga na dole a trzecia w środku).
5. Do prawego obszaru dodajemy dwie kontrolki **PictureBox** i ustawiamy ich właściwość **Dock** na kolejno: **Top**, **Fill** (pierwsza będzie wyświetlana u góry, druga w pozostałej części okna).
6. Dla każdej z pięciu kontroltek **PictureBox** ustawiamy właściwość **SizeMode** na **StretchImage**.
7. Dla każdej z pięciu kontroltek **PictureBox** importujemy inny obrazek (właściwość **Image**).
8. Po uruchomieniu aplikacji wyświetlone zostanie okno z pięcioma obrazkami w zdefiniowanych obszarach (zmiana rozmiaru okna proporcjonalnie dopasuje rozmiar poszczególnych obrazków).



Rysunek 53. Okno aplikacji z obszarami wyświetlającymi obrazki

## Pasek postępu

Pasek postępu (kontrolka `ProgressBar`) służy do prezentowania poziomu zaawansowania w wykonywaniu jakiejś czynności (np. przetwarzania).

### Właściwości

Kontrolka zakładek (***ProgressBar***) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b><i>Maximum</i></b>	Określa maksymalną wartość (górny przedział).	<b><i>100</i></b>
<b><i>Minimum</i></b>	Określa minimalną wartość (dolny przedział).	<b><i>0</i></b>
<b><i>Step</i></b>	Określa wielkość pojedynczego kroku inkrementacyjnego.	<b><i>10</i></b>
<b><i>Style</i></b>	Określa styl wyświetlania paska postępu. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>Blocks</i></b> – wyświetlane są bloki</li> <li>• <b><i>Continuous</i></b> – wyświetlany jest prostokąt ciągły</li> </ul>	<b><i>Blocks</i></b>
<b><i>Value</i></b>	Określa aktualną wartość postępu, której wartość musi zawierać się w przedziale <b><i>Minimum</i></b> i <b><i>Maximum</i></b> .	<b><i>0</i></b>

### Przykładowa aplikacja

Stwórzmy sobie prostą aplikację ***Windows Forms*** z tytułem „Postęp”, która będzie prezentować postęp w odliczaniu procentowej liczby sekund dla jednej minuty, po której nastąpi zamknięcie okna.

1. Tworzymy aplikację ***Windows Forms*** o tytule „Postęp”.
2. Dodajemy kontrolkę paska postępu i ustawiamy jego właściwości:
  - a) ***Minimum*** na ***0***
  - b) ***Maximum*** na ***60***

c) *Step* na 1

d) *Value* na 0

3. Dodajemy kontrolkę **Timer**, która posłuży nam do przesuwania paska co jakiś przedział czasu (u nas będzie to co sekundę).

4. Ustawiamy własność **Interval** kontrolki **Timer** na 1000 (czyli 1s) oraz dodajemy obsługę dla zdarzenia **Tick**. Metoda obsługująca to zdarzenie powinna wyglądać tak:

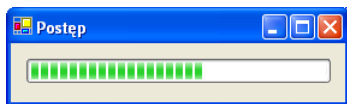
```
private void timer1_Tick(object sender, EventArgs e)
{
    progressBar1.PerformStep(); // zwiększamy o jeden krok

    // jeżeli dojdziemy do końca skali (100%) zamykamy aplikację
    if (progressBar1.Value == progressBar1.Maximum)
        Close();
}
```

5. Dodajemy obsługę zdarzenia rozpoczynającego proces odmierzenia czasu, po którym okno zostanie zamknięte (po minucie). Metoda obsługująca to zdarzenie powinna wyglądać tak:


```
private void Form1_Load(object sender, EventArgs e)
{
    timer1.Start(); // zaczynamy odliczanie czasu
}
```

6. Po uruchomieniu aplikacji wyświetlone zostanie okno z paskiem postępu. Kiedy pasek dojdzie do końca okno zamknie się automatycznie.



Rysunek 55. Okno aplikacji z paskiem postępu

## Suwak

Suwak (kontrolka  **TrackBar**) służy do wizualnego ustalania wartości liczbowej z zadanego przedziału liczbowego za pomocą wskaźnika.

## Właściwości

Kontrolka suwaka (**TrackBar**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>LargeChange</b>	Określa liczbę przesunięć suwaka po naciśnięciu klawiszy PAGE UP i PAGE DOWN lub też naciśnięcia myszy.	<b>5</b>
<b>Maximum</b>	Określa maksymalną wartość (górny przedział).	<b>10</b>
<b>Minimum</b>	Określa minimalną wartość (dolny przedział).	<b>0</b>
<b>Orientation</b>	Określa położenie suwaka w oknie. Właściwość ta może przyjąć wartości:	<b>Horizontal</b>

	<ul style="list-style-type: none"> <li>• <b>Horizontal</b> – położenie poziome</li> <li>• <b>Vertical</b> – położenie pionowe</li> </ul>	
<b>SmallChange</b>	Określa liczbę przesunięć suwaka po naciśnięciu klawiszy LEWY i PRAWY.	<b>1</b>
<b>TickFrequency</b>	Określa liczbę znaczników w przedziale.	<b>1</b>
<b>TickStyle</b>	<p>Określa sposób wyświetlania suwaka.</p> <p>Właściwość ta może przyjąć wartości:</p> <ul style="list-style-type: none"> <li>• <b>None</b> – znaczniki graficzne nie są wyświetlane</li> <li>• <b>TopLeft</b> – suwak ułożony jest strzałką do góry, znaczniki są widoczne u góry</li> <li>• <b>BottomRight</b> – suwak ułożony jest strzałką do dołu, znaczniki są widoczne na dole</li> <li>• <b>Both</b> – wyświetlany jest prostokątny suwak, znaczniki widoczne są u góry oraz na dole (dwa rzędy)</li> </ul>	<b>BottomRight</b>

## Zdarzenia

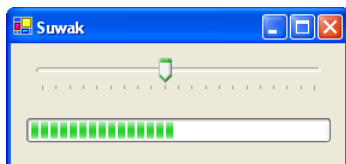
Najczęściej wykorzystywane zdarzenie charakterystyczne dla tej kontrolki to zdarzenie **Scroll**, które występuje w momencie zmiany położenia suwaka.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Suwak”, która będzie pozwalać na określanie wartości paska postępu na podstawie położenia suwaka.

1. Tworzymy aplikację **Windows Forms** o tytule „Suwak”.
2. Dodajemy kontrolkę suwaka i ustawiamy następujące właściwości:
  - a) **Minimum** na **0**
  - b) **Maksimum** na **20**
3. Dodajemy kontrolkę paska postępu i ustawiamy jego właściwości:
  - a) **Minimum** na **0**
  - b) **Maximum** na **20**
  - c) **Step** na **1**
  - d) **Value** na **0**
4. Dodajemy obsługę zdarzenia przesunięcia suwaka (**Scroll**). Metoda obsługująca to zdarzenie powinna wyglądać tak:
 

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    progressBar1.Value = trackBar1.Value;
}
```
5. Po uruchomieniu aplikacji, wyświetlone zostanie okno z suwakiem i paskiem postępu. Kiedy zmienione zostanie położenie suwaka, wyświetlony zostanie odpowiadający temu prostokąt na pasku postępu.



Rysunek 56. Okno aplikacji z suwakiem i paskiem postępu

## Kalendarz

Kalendarz (kontrolka  `DateTimePicker`) służy do wyboru daty i czasu.

### Właściwości

Kontrolka kalendarza (***DateTimePicker***) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b><i>CalendarFont</i></b>	Określa rodzaj czcionki kalendarza.	<b><i>Microsoft Sans Serif; 8,25pt</i></b>
<b><i>CalendarForeColor</i></b>	Określa kolor czcionki kalendarza.	<b><i>ControlText</i></b>
<b><i>CalendarMonthBackground</i></b>	Określa kolor tła dla kalendarza.	<b><i>Window</i></b>
<b><i>CalendarTitleBackColor</i></b>	Określa kolor tła paska tytułu kalendarza.	<b><i>ActiveCaption</i></b>
<b><i>CalendarTitleForeColor</i></b>	Określa kolor czcionki paska tytułu kalendarza.	<b><i>ActiveCaptionText</i></b>
<b><i>CalendarTrailingForeColor</i></b>	Określa kolor czcionki dla nieaktywnych dni w kalendarzu.	<b><i>GrayText</i></b>
<b><i>MaxDate</i></b>	Określa maksymalną dozwoloną wartość dla daty.	<b><i>9998-12-31</i></b>
<b><i>MinDate</i></b>	Określa minimalną dozwoloną wartość dla daty.	<b><i>1753-01-01</i></b>
<b><i>ShowCheckBox</i></b>	Określa, czy przycisk selekcji jest wyświetlany w kalendarzu.	<b><i>False</i></b>
<b><i>ShowUpDown</i></b>	Określa, czy zamiast przycisku listy rozwijanej mają być wyświetlane przyciski inkrementacji i dekrementacji daty.	<b><i>False</i></b>
<b><i>Value</i></b>	Określa wartość daty i czasu aktualnie ustawionej dla kalendarza.	

### Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<b><i>CloseUp</i></b>	Występuje w momencie zamknięcia kalendarza.
<b><i>DropDown</i></b>	Występuje w momencie rozwinięcia kalendarza.
<b><i>ValueChanged</i></b>	Występuje w momencie zmiany wartości właściwości <b><i>Value</i></b> .

### Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Sekundy”, która będzie wyliczać liczbę sekund od wybranej daty. Jeżeli wybierzemy wcześniejszą datę, wyświetlony zostanie komunikat

informujący ile sekund upłynęło od danego dnia. Jeżeli wybierzemy datę późniejszą wyświetlony zostanie komunikat informujący ile sekund pozostało do danego dnia.

1. Tworzymy aplikację **Windows Forms** o tytule „Sekundy”.
2. Dodajemy kontrolkę kalendarza.
3. Dodajemy obsługę zdarzenia zamknięcia kalendarza (**CloseUp**). Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void dateTimePicker1_CloseUp(object sender, EventArgs e)
{
    DateTime DataWyb = dateTimePicker1.Value; // data wybrana
    DateTime DataAkt = DateTime.Now; // data aktualna
    string napis;

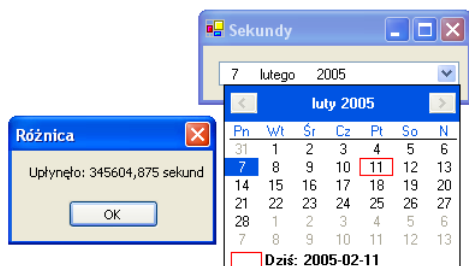
    // jeżeli wybrana jest późniejsza od aktualnej
    if (DataWyb > DataAkt)
    {
        TimeSpan roznica = DataWyb - DataAkt;

        napis = "Pozostało: " + roznica.TotalSeconds.ToString() + " sekund";
    }
    else // jeżeli wybrana jest wcześniejsza lub równa aktualnej
    {
        TimeSpan roznica = DataAkt - DataWyb;

        napis = "Upłynęło: " + roznica.TotalSeconds.ToString() + " sekund";
    }


    MessageBox.Show(napis, "Różnica");
}
```

4. Po uruchomieniu aplikacji, wyświetlone zostanie okno z kalendarzem. Jeżeli wybierzemy jakąś datę, wyświetlony zostanie komunikat informujący ile sekund upłynęło od wybranej daty lub ile pozostało.



Rysunek 57. Okno aplikacji z kalendarzem po wybraniu daty

## Pole numeryczne

Pole numeryczne (kontrolka  **NumericUpDown**) służy do określania wartości numerycznej za pomocą



strzałek inkrementacji i dekrementacji.

## Właściwości

Kontrolka pola numerycznego (*NumericUpDown*) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>DecimalPlaces</i>	Określa liczbę miejsc po przecinku do wyświetlenia.	<i>0</i>
<i>Hexadecimal</i>	Określa czy pole numeryczne ma wyświetlać liczby w formacie szesnastkowym.	<i>False</i>
<i>Increment</i>	Określa o ile ma się zwiększać/zmniejszać liczba.	<i>1</i>
<i>InterceptArrowKeys</i>	Określa, czy klawisze strzałek górna oraz dolna pozwalają na sterowanie inkrementacją i dekrementacją wartości kontrolki.	<i>True</i>
<i>Maximum</i>	Określa maksymalną wartość liczbową dozwoloną dla kontrolki.	<i>100</i>
<i>Minimum</i>	Określa minimalną wartość liczbową dozwoloną dla kontrolki.	<i>0</i>
<i>ThousandsSeparator</i>	Określa, czy w czasie wyświetlania ma być wstawiany separator tysięcy.	<i>False</i>
<i>UpDownAlign</i>	Określa położenie przycisków inkrementacji i dekrementacji.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <i>Left</i> – położenie z lewej</li> <li>• <i>Right</i> – położenie z prawej</li> </ul>	<i>Right</i>
<i>Value</i>	Określa aktualną wartość numeryczną kontrolki.	<i>0</i>

## Zdarzenia

Najczęściej wykorzystywane zdarzenie charakterystyczne dla tej kontrolki to zdarzenie *ValueChanged*, które występuje w momencie zmiany wartości właściwości *Value*.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację *Windows Forms* z tytułem „Szczęśliwa liczba”, która będzie wyliczać na podstawie daty urodzenia liczbę z przedziału 0..9 związaną z tym dniem (wyliczenie tej liczby polega na sumowaniu wszystkich cyfr występujących w dacie urodzenia).

1. Tworzymy aplikację *Windows Forms* o tytule „Szczęśliwa liczba”.
2. Dodajemy kontrolkę pola grupującego i ustawiamy wartość właściwości *Text* na „Data urodzenia”.
3. Wewnątrz tego pola umieszczamy trzy kontrolki pola numerycznego na rok, miesiąc i dzień urodzenia oraz odpowiadające im etykiety tekstowe z nazwami: „Rok”, „Miesiąc”, „Dzień”.
4. Ustawiamy dla kontrolki pola numerycznego wartości minimum i maksimum zgodne z przeznaczeniem (właściwości *Minimum* oraz *Maximum*). Czyli dla roku np.: 1900..9999 dla miesiąca 1..12 i dla dnia 1..31.



5. Dodajemy przycisk do wywołania akcji wyliczenia liczby o nazwie „Wylicz szczęśliwą liczbę”

6. Dodajemy obsługę zdarzenia naciśnięcia przycisku (**Click**). Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void button1_Click(object sender, EventArgs e)
{
    int Rok = (int)numericUpDown1.Value;
    int Miesiac = (int)numericUpDown2.Value;
    int Dzień = (int)numericUpDown3.Value;
    int Liczba;
    int Podzielnik = 1000; // początkowo 1000, dla roku 4 cyfrowego
    int Suma = 0;

    for (int i = 0; i < 4; i++) // operacja powtórzy się 4 razy dla roku
    {
        Liczba = Rok / Podzielnik; // kolejne cyfry
        Suma += Liczba; // dodajemy je
        Rok -= Liczba * Podzielnik; // eliminujemy z działania zsumowane
        Podzielnik /= 10; // zmniejszamy podzielnik
    }

    Podzielnik = 10; // podzielnik dla 2 cyfrowego miesiąca

    for (int i = 0; i < 2; i++) // operacja powtórzy się 2 razy dla miesiąca
    {
        Liczba = Miesiac / Podzielnik;
        Suma += Liczba;
        Miesiac -= Liczba * Podzielnik;
        Podzielnik /= 10;
    }

    Podzielnik = 10; // podzielnik dla 2 cyfrowego dnia

    for (int i = 0; i < 2; i++) // operacja powtórzy się 2 razy dla dnia
    {
        Liczba = Dzień / Podzielnik;
        Suma += Liczba;
        Dzień -= Liczba * Podzielnik;
        Podzielnik /= 10;
    }

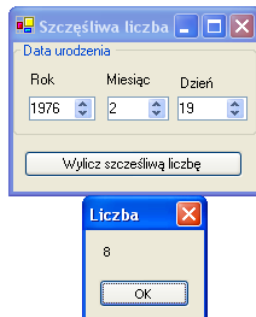
    // jeżeli liczba jest dwucyfrowa robimy z niej jednocyfrową
    while (Suma > 9)
        Suma = (Suma / 10) + (Suma % 10);
}
```

```

    MessageBox.Show(Suma.ToString(), "Liczba");
}


```

7. Po uruchomieniu aplikacji, wyświetlone zostanie okno pozwalające na wyliczenie liczby związanej z datą urodzenia.



Rysunek 58. Okno wyliczania szczęśliwej liczby po wskazaniu daty i naciśnięciu na przycisk

## Lista obrazów


Lista obrazów (kontrolka  **ImageList**) służy do budowania listy obrazów wykorzystywanych przez inne kontrolki (np.: listy złożonej, drzewa, menu, paska narzędzi).

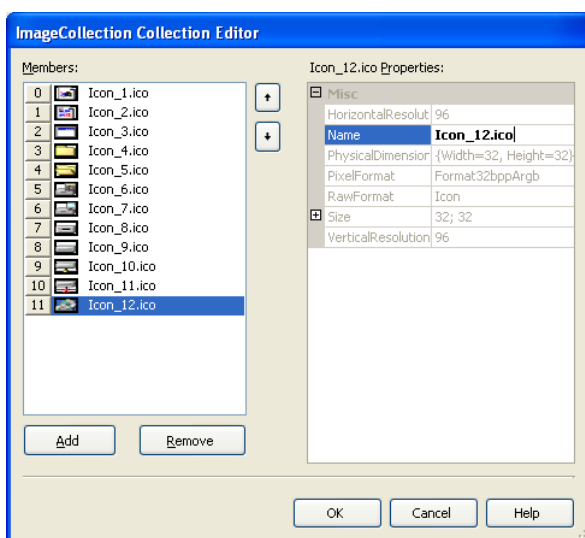
### Właściwości

Kontrolka zakładek (**ImageList**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>ColorDepth</b>	Określa liczbę kolorów dla obrazów w kolekcji.	<b>Depth8Bit</b>
<b>Images</b>	Zawiera kolekcję obrazów kolekcji.	
<b>ImageSize</b>	Określa wspólną wielkość obrazów kolekcji.	<b>16; 16</b>
<b>TransparentColor</b>	Określa, który kolor będzie traktowany jako przezroczysty.	<b>Transparent</b>

### Kolekcja obrazów

Zarządzanie kolekcją obrazów odbywa się za pomocą okna edytora kolekcji „ImageCollection Collection Editor” (który uruchamia się poprzez naciśnięcie przycisku  znajdującego się przy właściwości **Images**).



Rysunek 59. Okno edytora kolekcji obrazów

Funkcjonalność kolekcji obrazów jest realizowana za pomocą klasy **ImageCollection** (kolekcja elementów **Image**).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<b>Count</b>	Zwraca liczbę obrazków w kolekcji.
<b>Empty</b>	Sprawdza, czy kolekcja obrazków jest pusta.
<b>IsReadOnly</b>	Sprawdza, czy kolekcja dostępna jest w trybie tylko do odczytu.
<b>Item</b>	Zwraca element kolekcji (obrazek) o określonym indeksie.


Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<b>Add</b>	Pozwala na dodanie obrazka do kolekcji.
<b>AddStrip</b>	Pozwala na dodanie obrazka zawierającego kilka elementów do kolekcji.
<b>Clear</b>	Pozwala na usunięcie wszystkich obrazków z kolekcji.
<b>Contains</b>	Pozwala na sprawdzenie czy w kolekcji znajduje się określony obrazek.
<b>IndexOf</b>	Zwraca indeks podanego obrazka z kolekcji.
<b>Remove</b>	Pozwala na usunięcie z kolekcji określonego obrazka.
<b>RemoveAt</b>	Pozwala na usunięcie z kolekcji obrazka o określonym indeksie.

## Rozdział 4.

# Korzystanie z zaawansowanych kontrolek

### Zakładki

Zakładki (kontrolka  `TabControl`) służą do grupowania dużej liczby kontrolek na małym obszarze okna. Dzięki zakładkom interfejs użytkownika staje się bardziej przejrzysty i łatwiejszy w obsłudze.

### Właściwości

Kontrolka zakładek (***TabControl***) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b><i>Alignment</i></b>	Określa położenie przycisków przełączania między zakładkami.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>Top</i></b> – przyciski umieszczone u góry</li> <li>• <b><i>Bottom</i></b> – przyciski umieszczone na dole</li> <li>• <b><i>Left</i></b> – przyciski umieszczone z lewej</li> <li>• <b><i>Right</i></b> – przyciski umieszczone z prawej</li> </ul>	<b><i>Top</i></b>
<b><i>HotTrack</i></b>	Określa, czy zakładki zmieniają wygląd w momencie najechania na nie kursorem.	<b><i>False</i></b>
<b><i>ImageList</i></b>	Lista ikon skojarzona z zakładkami.	<b><i>(None)</i></b>
<b><i>Multiline</i></b>	Określa, czy przyciski przełączania zakładek mają być wyświetlane w wielu liniach (jeżeli nie zmieszczą się w jednej).	<b><i>False</i></b>
<b><i>Padding</i></b>	Określa rozmiar dodatkowego obszaru dodawanego wokół tekstu i obrazu w zakładce.	<b>6; 3</b>
<b><i>RaiseEnterLeaveEventsForTab</i></b>	Określa, czy zdarzenia wejścia i opuszczenia mają być przekazywane do strony zakładki.	<b><i>False</i></b>
<b><i>ShowToolTips</i></b>	Określa, czy podpowiedzi mają być wyświetlane w zakładkach.	<b><i>False</i></b>
<b><i>SizeMode</i></b>	Określa, w jaki sposób ma być dopasowany rozmiar zakładek.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>Normal</i></b> – rozmiar standardowy</li> <li>• <b><i>FillToRect</i></b> – rozmiar dopasowany do prostokąta</li> <li>• <b><i>Fixed</i></b> – rozmiar określony na stałe</li> </ul>	<b><i>Normal</i></b>
<b><i>TabIndex</i></b>	Określa indeks aktywnej zakładki.	<b>0</b>
<b><i>TabPage</i></b>	Zawiera kolekcję stron zakładek ( <b><i>TabPageCollection</i></b> ).	
<b><i>TabStop</i></b>	Określa, czy klawisz TAB może być używany	<b><i>True</i></b>

	do przełączania między zakładkami.	
--	------------------------------------	--

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń, charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<i>Deselected</i>	Występuje po deaktywacji zakładki.
<i>Deselecting</i>	Występuje przed deaktywacją zakładki.
<i>Selected</i>	Występuje po aktywacji zakładki.
<i>Selecting</i>	Występuje przed aktywacją zakładki.
<i>TabIndexChanged</i>	Występuje, gdy zmieni się wartość właściwości <i>TabIndex</i> .

## Kolekcja stron

Najważniejszą i najczęściej wykorzystywaną właściwością zakładek, jest kolekcja stron *TabPage*. Dzięki tej właściwości można zarządzać kolekcją zakładek (dodawać, usuwać, przełączać, etc.).

Funkcjonalność kolekcji elementów listy rozwijanej jest realizowana za pomocą klasy *TabPageCollection* (kolekcja elementów *TabPage*).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<i>Count</i>	Zwraca liczbę elementów kolekcji.
<i>IsReadOnly</i>	Sprawdza, czy kolekcja dostępna jest w trybie tylko do odczytu.
<i>Item</i>	Zwraca element kolekcji o określonym indeksie.

Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<i>Add</i>	Pozwala na dodanie elementu do kolekcji.
<i>AddRange</i>	Pozwala na dodanie kilku elementów do kolekcji.
<i>Clear</i>	Pozwala na usunięcie wszystkich elementów z kolekcji.
<i>Contains</i>	Pozwala na sprawdzenie, czy w kolekcji znajduje się określony element.
<i>IndexOf</i>	Zwraca indeks podanego elementu kolekcji.
<i>Remove</i>	Pozwala na usunięcie z kolekcji podanego elementu.
<i>RemoveAt</i>	Pozwala na usunięcie z kolekcji elementu o określonym indeksie.

## Przykładowa aplikacja

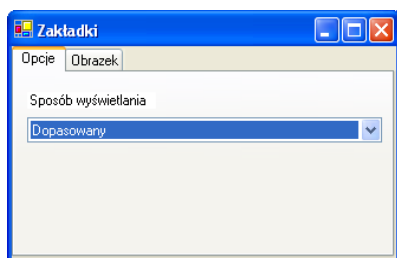
Stworzymy sobie prostą aplikację *Windows Forms* z tytułem „Zakładki”, która zawierać będzie dwie zakładki: jedna pozwalająca na wybór opcji wyświetlenia obrazka i druga wyświetlająca obrazek w całym dostępnym obszarze zakładki (z uwzględnieniem ustawionych opcji).

1. Tworzymy aplikację *Windows Forms* o tytule „Zakładki”.
2. Dodajemy kontrolkę zakładek i ustawiamy jej właściwość *Dock* na *Fill* (ponieważ standardowo po dodaniu tej kontrolki będzie zawierała dwie zakładki, więc nie musimy dodawać nowych zakładek).

3. Ustawiamy się na pierwszej zakładce, klikamy na stronę tej zakładki i zmieniamy nazwę zakładki na „Opcje” (właściwość **Text**).
4. Na stronie opcji dodajemy kolejno kontrolki:
  - a) etykiety tekstowej i ustawiamy jej właściwość **Text** na „Sposób wyświetlania”,
  - b) listy rozwijanej i tworzymy listę dostępnych wartości wyświetlania (Normalny, Automatyczny, Wycentrowany, Dopasowany, Powiększony), dodatkowo ustawiamy właściwość **DropDownStyle** na **DropDownList**.
5. Ustawiamy się na drugiej zakładce, klikamy na stronę tej zakładki i zmieniamy nazwę tej zakładki na „Obrazek”.
6. Dodajemy kontrolkę obrazka, ustawiamy jej właściwość **Dock** na **Fill** i wczytujemy jakiś obrazek.
7. Dodajemy obsługę zdarzenia **SelectedIndexChanged**, która występować będzie zawsze, gdy nastąpi przełączenie między zakładkami. Metoda obsługująca to zdarzenie powinna wyglądać tak:


```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{
    // wyświetlanie obrazka jedynie w momencie aktywacji zakładki "Obrazek"
    if (tabControl1.SelectedTab.Text != "Obrazek")
        return;

    // w zależności od wybranego z listy trybu wyświetlania obrazek zostanie
    // wyświetlony zgodnie z tym trybem
    switch (comboBox1.Text)
    {
        case "Normalny":
            pictureBox1.SizeMode = PictureBoxSizeMode.Normal;
            break;
        case "Automatyczny":
            pictureBox1.SizeMode = PictureBoxSizeMode.AutoSize;
            break;
        case "Wycentrowany":
            pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
            break;
        case "Dopasowany":
            pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
            break;
        case "Powiększony":
            pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
            break;
    }
}
```
8. Przełączamy się na pierwszą zakładkę.
9. Po uruchomieniu aplikacji, wyświetlone zostanie okno z zakładkami.



Rysunek 54. Okno aplikacji z zakładkami (widok pierwszej aktywnej)

## Drzewo

Drzewo (kontrolka  **TreeView**) pozwala na wyświetlanie listy elementów, będących ze sobą w relacji hierarchicznej (np.: drzewo folderów).

## Właściwości

Kontrolka drzewa (**TreeView**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>DrawMode</b>	Określa sposób rysowania drzewa (przez system lub użytkownika). Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Normal</b> – wszystkie elementy rysowane są przez system</li> <li>• <b>OwnerDrawText</b> – wszystkie elementy za wyjątkiem tekstu są rysowane przez system</li> <li>• <b>OwnerDrawAll</b> – wszystkie elementy rysowane są przez użytkownika</li> </ul>	<b>Normal</b>
<b>FullRowSelect</b>	Określa, czy podświetlenie elementu jest widoczne na całej szerokości.	<b>False</b>
<b>HideSelection</b>	Określa, czy podświetlenie wybranego elementu ma być widoczne po utraceniu przez kontrolkę focusa.	<b>True</b>
<b>HotTracking</b>	Określa, czy element ma być podkreślany w momencie, gdy kursor myszy znajdzie się w obszarze jego selekcji.	<b>False</b>
<b>ImageIndex</b>	Określa domyślny numer indeksu obrazka ze skojarzonej z kontrolką kolekcji obrazków.	<b>(none)</b>
<b>ImageKey</b>	Określa domyślną nazwę obrazka ze skojarzonej z kontrolką kolekcją obrazków.	<b>(none)</b>
<b>ImageList</b>	Zawiera nazwę obiektu kolekcji obrazów zawierającą ikony.	<b>(none)</b>
<b>Indent</b>	Określa szerokość odstępu w pikselach od etykiety elementu.	<b>19</b>
<b>ItemHeight</b>	Określa wysokość każdego elementu.	<b>16</b>
<b>LabelEdit</b>	Określa, czy użytkownik może edytować etykiety elementów.	<b>False</b>
<b>LineColor</b>	Określa kolor linii łączących elementy.	<b>Black</b>

<b>Nodes</b>	Zawiera kolekcję elementów drzewa ( <b>TreeNodeCollection</b> ).	
<b>PathSeparator</b>	Określa separator ścieżki używany w łańcuchu ścieżki zwracanym przez właściwość <b>FullPath</b> .	
<b>Scrollable</b>	Określa, czy drzewo wyświetla paski przewijania w momencie, gdy elementy nie mieszczą się w obszarze widoczności.	<b>True</b>
<b>SelectedImageIndex</b>	Określa numer indeksu obrazka dla wybranego elementu ze skojarzonej z kontrolką kolekcji obrazków.	<b>(none)</b>
<b>SelectedImageKey</b>	Określa nazwę obrazka dla wybranego elementu ze skojarzonej z kontrolką kolekcją obrazków.	<b>(none)</b>
<b>ShowLines</b>	Określa, czy linie łączące elementy mają być wyświetlane.	<b>True</b>
<b>ShowNodeToolTips</b>	Określa, czy dla elementów mają być wyświetlane podpowiedzi.	<b>False</b>
<b>ShowPlusMinus</b>	Określa, czy przyciski rozwijania/zwijania (plus/minus) elementów nadrzędnych mają być widoczne.	<b>True</b>
<b>ShowRootLines</b>	Określa, czy linie między elementami nadrzędnymi mają być widoczne.	<b>True</b>
<b>Sorted</b>	Określa, czy elementy mają być posortowane.	<b>False</b>
<b>TabStop</b>	Określa, czy użytkownik może używać klawisza TAB dla tej kontrolki.	<b>True</b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<b>AfterCheck</b>	Zachodzi, gdy okienko selekcji skojarzone z elementem zostanie zaznaczone lub odznaczone.
<b>AfterCollapse</b>	Zachodzi, gdy nastąpi zwinięcie listy elementów potomnych.
<b>AfterExpand</b>	Zachodzi, gdy nastąpi rozwinięcie listy elementów potomnych.
<b>AfterLabelEdit</b>	Zachodzi, gdy nastąpi zmiana etykiety elementu.
<b>AfterSelect</b>	Zachodzi, gdy nastąpi wybranie elementu.
<b>BeforeCheck</b>	Zachodzi przed zaznaczeniem lub odznaczeniem okienka selekcji skojarzonego z elementem.
<b>BeforeCollapse</b>	Zachodzi przed zwinięciem listy elementów potomnych.
<b>BeforeExpand</b>	Zachodzi przed rozwinięciem listy elementów potomnych.
<b>BeforeLabelEdit</b>	Zachodzi przed zmianą etykiety elementu.
<b>BeforeSelect</b>	Zachodzi przed wybraniem elementu.
<b>DrawNode</b>	Zachodzi w trybie rysowania użytkownika, gdy element ma być wyświetlony.
<b>NodeMouseClick</b>	Zachodzi, gdy nastąpi kliknięcie myszą na elemencie.
<b>NodeMouseDoubleClick</b>	Zachodzi, gdy nastąpi podwójne kliknięcie myszą na elemencie.
<b>NodeMouseHover</b>	Zachodzi, gdy mysz znajdzie się w obszarze elementu.



## Kolekcja elementów drzewa

Funkcjonalność kolekcji elementów drzewa (dostępna poprzez właściwość **Nodes**) jest realizowana za pomocą klasy **TreeNodeCollection** (kolekcja elementów **TreeNode**).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<b>Count</b>	Zwraca liczbę elementów kolekcji.
<b>Item</b>	Zwraca element kolekcji o określonym indeksie.

Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<b>Add</b>	Pozwala na dodanie elementu do kolekcji
<b>AddAt</b>	Pozwala na dodanie elementu do kolekcji na określoną pozycję
<b>Clear</b>	Pozwala na usunięcie wszystkich elementów z kolekcji
<b>Contains</b>	Pozwala na sprawdzenie czy w kolekcji znajduje się określony element
<b>IndexOf</b>	Zwraca indeks podanego elementu kolekcji
<b>Remove</b>	Pozwala na usunięcie z kolekcji podanego elementu
<b>RemoveAt</b>	Pozwala na usunięcie z kolekcji elementu o określonym indeksie

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Drzewo folderów”, która będzie wyświetlać pełną listę hierarchiczną folderów na dysku C:\. Dla każdego z elementów drzewa powinna być wyświetlana odpowiednia ikona.

1. Tworzymy aplikację **Windows Forms** o tytule „Drzewo folderów”.
2. Dodajemy kontrolkę drzewa i ustawiamy jej właściwość **Dock** na **Fill**.
3. Dodajemy kontrolkę listy obrazów i wczytujemy dwie ikony: jedna symbolizująca folder otwarty i druga symbolizująca folder zamknięty.
4. Kojarzmy listę obrazów z drzewem, wybierając z listy rozwijanej właściwości **ImageList** dla tego drzewa odpowiednią listę obrazów (w naszym wypadku jest tylko jedna).
5. W zależności od kolejności ikon na liście zmieniamy właściwości **ImageIndex** (lub **ImageKey**) na numer indeksu ikony folderu zamkniętego oraz **SelectedImageIndex** (lub **SelectedImageKey**) na indeks ikony folderu otwartego.
6. Ponieważ będziemy korzystać z funkcjonalności związanej z folderami, musimy włączyć do listy używanych przestrzeni nazw przestrzeń **System.IO**:

```
using System.IO;
```

7. Tworzymy metodę rekurencyjną o nazwie **DodajFoldery**, która służyć będzie do wypełniania pojedynczej gałęzi drzewa.

```
private void DodajFoldery(string Sciezka, TreeNode Korzen)
{
    try
    {
```

```
// pobieramy listę folderów
string[] Lista = Directory.GetDirectories(Sciezka);

// dla każdego folderu budujemy jego poddrzewo
for (int i = 0; i < Lista.Length; i++)
{
    // tworzymy korzeń poddrzewa (będzie on liściem, jeżeli nie
    // będzie miał elementów potomnych)
    TreeNode Lisc = new TreeNode(Path.GetFileName(Lista[i]));

    // dodajemy elementy do poddrzewa
    Korzen.Nodes.Add(Lisc);
    // wywołujemy rekurencyjnie tą samą metodę w celu wypełnienia
    // poddrzewa następnego poziomu
    DodajFoldery(Lista[i], Lisc);
}
}
catch (Exception)
{
}
}
```

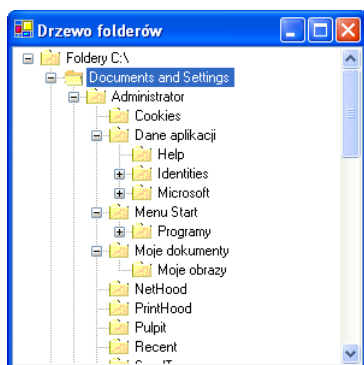
8. Korzystając z napisanej wcześniej metody **DodajFoldery**, tworzymy punkt początkowy, który rozpocznie proces rekurencyjnego tworzenia drzewa folderów w momencie utworzenia formy (zdarzenie **Load**). Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void Form1_Load(object sender, EventArgs e)
{
    TreeNode Korzen = new TreeNode(@"Foldery C:\"); // inicjacja korzenia

    treeView1.Nodes.Clear(); // usunięcie elementów (o ile jakieś są)
    treeView1.Nodes.Add(Korzen); // utworzenie korzenia


    // wywołanie metody rekurencyjnego tworzenia drzewa folderów
    DodajFoldery(@"c:\", Korzen);
}
```

9. Po uruchomieniu aplikacji, (co może trochę potrwać ze względu na czas potrzebny na zbudowanie struktury) wyświetlone zostanie okno wyświetlające drzewo folderów na dysku C:\



Rysunek 60. Okno drzewa folderów na dysku C:\

## Lista złożona

Lista złożona (kontrolka  **ListView**) pozwala na wyświetlanie listy elementów w różnych widokach. Elementy listy mogą być wybierane poprzez ich wskazanie (lista pozwala zarówno na selekcję pojedynczego elementu jak również wielu elementów).

## Właściwości

Kontrolka listy złożonej (**ListView**) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>Activation</b>	Określa rodzaj aktywowania elementu listy oraz odpowiedzi na aktywację.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Standard</b> – użytkownik musi podwójnie kliknąć na element w celu jego aktywacji, w czasie przesuwania wskaźnika kursora nad elementem nie ma odpowiedzi</li> <li>• <b>OneClick</b> – użytkownik musi kliknąć na element w celu jego aktywacji, w czasie przesuwania wskaźnika kursora nad elementem zmienia się on na wskaźnik dłoni, element listy zmienia swój kolor</li> <li>• <b>TwoClick</b> – użytkownik musi podwójnie kliknąć na element w celu jego aktywacji, w czasie przesuwania wskaźnika kursora nad elementem następuje zmiana jego koloru</li> </ul>	<b>Standard</b>
<b>Alignment</b>	Określa rodzaj wyrównywania elementów listy.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>Default</b> – kiedy użytkownik przenosi element pozostaje on w miejscu upuszczenia</li> <li>• <b>Left</b> – elementy wyrównywane są do lewej krawędzi</li> <li>• <b>Top</b> – elementy wyrównywane są do górnej krawędzi</li> <li>• <b>SnapToGrid</b> – elementy wyrównywane są do</li> </ul>	<b>Top</b>

	niewidzialnego grida (tabeli), kiedy użytkownik przenosi element, zostaje on przesunięty do najbliższej linii grida	
<b>AllowColumnReorder</b>	Określa, czy użytkownik może zmieniać kolejność kolumn w widoku szczegółowym.	<b>False</b>
<b>AutoArrange</b>	Określa, czy ikony są porządkowane automatycznie.	<b>True</b>
<b>CheckBoxes</b>	Określa, czy przy elementach listy będą wyświetlane przyciski selekcji.	<b>False</b>
<b>Columns</b>	Zawiera kolekcję kolumn listy dla widoku szczegółowego ( <b>ColumnHeaderCollection</b> ).	
<b>FullRowSelect</b>	Określa, czy podświetlenie elementu jest widoczne na całej szerokości.	<b>False</b>
<b>GridLines</b>	Określa, czy w widoku szczegółowym mają być widoczne linie oddzielające elementy kolekcji.	<b>False</b>
<b>Groups</b>	Zawiera kolekcję grup elementów ( <b>ListViewGroupCollection</b> ).	
<b>HeaderStyle</b>	Określa styl nagłówka kolumny w widoku szczegółowym.  Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b>None</b> – nagłówek kolumny nie jest wyświetlany w widoku szczegółowym</li> <li>• <b>Nonclickable</b> – kolumna nie reaguje na kliknięcie myszy</li> <li>• <b>Clickable</b> – kolumna funkcjonuje jak przycisk i może odpowiadać wykonaniem jakiejś akcji (np.: sortowaniem po kliknięciu)</li> </ul>	<b>Clickable</b>
<b>HideSelection</b>	Określa, czy podświetlenie wybranego elementu ma być widoczne po utraceniu przez kontrolkę focusa.	<b>True</b>
<b>HoverSelection</b>	Określa, czy możliwe jest użycie kółka myszy w czasie selekcji elementów.	<b>False</b>
<b>IsBackgroundImageTiled</b>	Określa, czy obrazek tła będzie wyświetlany na całej powierzchni tła kontrolki.	<b>False</b>
<b>Items</b>	Zawiera kolekcję elementów listy ( <b>ListViewItemCollection</b> ).	
<b>LabelEdit</b>	Określa, czy użytkownik może edytować etykiety tekstowe elementów listy.	<b>False</b>
<b>LabelWrap</b>	Określa, czy etykiety tekstowe będą zawijane, jeżeli nie zmieszczą się w jednej linii.	<b>True</b>
<b>LargeImageList</b>	Zawiera nazwę obiektu kolekcji obrazów zawierającą duże ikony (wykorzystywane w widoku dużych ikon).	<b>(none)</b>
<b>Multiselect</b>	Określa, czy dozwolona jest selekcja wielu elementów.	<b>True</b>
<b>OwnerDraw</b>	Określa, czy użytkownik będzie samodzielnie rysował listę (domyślnie rysowana jest przez system).	<b>False</b>
<b>Scrollable</b>	Określa, czy lista wyświetla paski przewijania w momencie, gdy elementy nie mieszczą się w obszarze widoczności.	<b>True</b>
<b>ShowGroups</b>	Określa, czy elementy będą wyświetlane w grupach.	<b>True</b>
<b>ShowItemToolTips</b>	Określa, czy dla elementów mają być wyświetlane podpowiedzi.	<b>False</b>

<b><i>SmallImageList</i></b>	Zawiera nazwę obiektu kolekcji obrazów zawierającą małe ikony (wykorzystywane w widoku małych ikon, widoku listy, widoku szczegółowym).	<b><i>(none)</i></b>
<b><i>Sorting</i></b>	Określa rodzaj sortowania dla listy. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>None</i></b> – brak sortowania</li> <li>• <b><i>Ascending</i></b> – sortowanie rosnące</li> <li>• <b><i>Descending</i></b> – sortowanie malejące</li> </ul>	<b><i>None</i></b>
<b><i>StateImageList</i></b>	Zawiera nazwę obiektu kolekcji obrazów, która wykorzystywana jest do odwzorowywania stanów.	<b><i>(none)</i></b>
<b><i>TabStop</i></b>	Określa, czy użytkownik może używać klawisza TAB dla tej kontrolki.	<b><i>True</i></b>
<b><i>TileSize</i></b>	Rozmiar prostokątnego obszaru dla elementu, który używany jest w widoku pokrywającym.	<b><i>0; 0</i></b>
<b><i>View</i></b>	Określa rodzaj widoku listy. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>LargeIcon</i></b> – widok dużych ikon</li> <li>• <b><i>Details</i></b> – widok szczegółowy</li> <li>• <b><i>SmallIcon</i></b> – widok małych ikon</li> <li>• <b><i>List</i></b> – widok listy</li> <li>• <b><i>Tile</i></b> – widok pokrywający</li> </ul>	<b><i>LargeIcon</i></b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

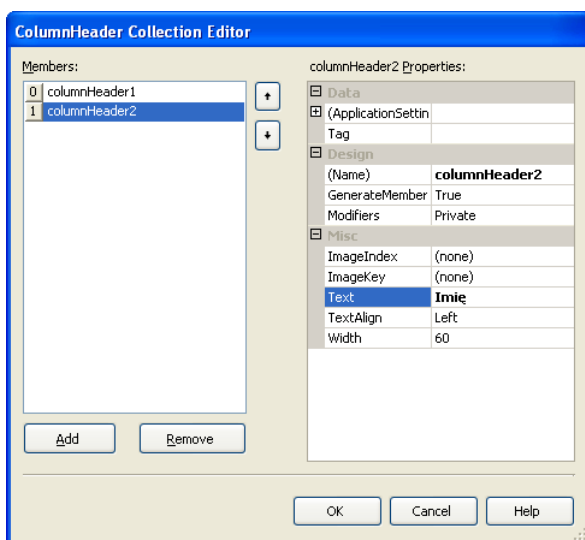
<b>Zdarzenie</b>	<b>Opis</b>
<b><i>AfterLabelEdit</i></b>	Zachodzi, gdy nastąpi zmiana etykiety elementu.
<b><i>BeforeLabelEdit</i></b>	Zachodzi przed zmianą etykiety elementu.
<b><i>ColumnClick</i></b>	Zachodzi, gdy nastąpi kliknięcie na nagłówek kolumny.
<b><i>ColumnRearranged</i></b>	Zachodzi, gdy zmienia się położenie nagłówka kolumny.
<b><i>ColumnWidthChanged</i></b>	Zachodzi, gdy zmieni się szerokość nagłówka kolumny.
<b><i>ColumnWidthChanging</i></b>	Zachodzi w czasie zmiany szerokości nagłówka kolumny.
<b><i>DrawColumnHeader</i></b>	Zachodzi, gdy nagłówek kolumny ma zostać narysowany (w trybie rysowania przez użytkownika).
<b><i>DrawItem</i></b>	Zachodzi, gdy główny element wiersza z listy ma zostać narysowany (w trybie rysowania przez użytkownika).
<b><i>DrawSubItems</i></b>	Zachodzi, gdy pozostałe elementy wiersza z listy (poza głównym) mają zostać narysowane (w trybie rysowania przez użytkownika).
<b><i>ItemActivate</i></b>	Zachodzi, gdy element listy staje się aktywny.
<b><i>ItemCheck</i></b>	Zachodzi, gdy przycisk selekcji związany z elementem ma zostać zaznaczony lub odznaczony.
<b><i>ItemChecked</i></b>	Zachodzi, gdy przycisk selekcji związany z elementem zostanie zaznaczony lub odznaczony.
<b><i>ItemDrag</i></b>	Zachodzi, gdy element jest przeciągany.
<b><i>ItemMouseHover</i></b>	Zachodzi, gdy mysz znajdzie się w obszarze elementu.

<b><i>ItemSelectionChanged</i></b>	Zachodzi, gdy nastąpi zmiana stanu wybrania elementu.
<b><i>SelectedIndexChanged</i></b>	Zachodzi, gdy nastąpi zmiana wartości właściwości <b><i>SelectedIndex</i></b> .

## Kolumny w widoku szczegółowym

Lista złożona w trybie widoku szczegółowego przyjmuje postać tabeli. Kolumny tabeli określają sposób wyświetlania elementów wiersza tabeli. Kolumny można zdefiniować zarówno w czasie projektowania aplikacji jak również na poziomie kodu.

Definiowanie kolumn w czasie projektowania aplikacji odbywa się z użyciem kreatora kolumn „ColumnHeader Collection Editor”.



Rysunek 61. Okno kreatora kolumn

## Kolekcja elementów listy

Najważniejszą i najczęściej wykorzystywaną właściwością listy jest kolekcja elementów ***Items***. Dzięki tej właściwości można zarządzać kolekcją elementów listy (dodawać, usuwać i zmieniać wartości).

Funkcjonalność kolekcji elementów listy jest realizowana za pomocą klasy ***ListViewItemCollection*** (kolekcja elementów ***ListViewItem***).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<b><i>Count</i></b>	Zwraca liczbę elementów kolekcji.
<b><i>IsReadOnly</i></b>	Sprawdza, czy kolekcja dostępna jest w trybie tylko do odczytu.
<b><i>Item</i></b>	Zwraca element kolekcji o określonym indeksie.

Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<b><i>Add</i></b>	Pozwala na dodanie elementu do kolekcji.
<b><i>AddRange</i></b>	Pozwala na dodanie kilku elementów do kolekcji.
<b><i>Clear</i></b>	Pozwala na usunięcie wszystkich elementów z kolekcji.

<b>Contains</b>	Pozwala na sprawdzenie, czy w kolekcji znajduje się określony element.
<b>IndexOf</b>	Zwraca indeks podanego elementu kolekcji.
<b>Remove</b>	Pozwala na usunięcie z kolekcji podanego elementu.
<b>RemoveAt</b>	Pozwala na usunięcie z kolekcji elementu o określonym indeksie.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Pliki”, która będzie prezentować listę plików na dysku C:\ wraz z opisem zawierającym informacje o jego rozmiarze i dacie utworzenia. Dane będą prezentowane w widoku szczegółowym.

1. Tworzymy aplikację **Windows Forms** o tytule „Pliki”.
2. Dodajemy kontrolkę listy i ustawiamy jej właściwości:
  - a) **Dock** na **Fill**
  - b) **View** na **Details**
  - c) **FullRowSelect** na **True**
3. Definiujemy następujące nagłówki kolumn dla listy (korzystając z kreatora kolumn):
  - a) „Nazwa” o szerokości 150
  - b) „Rozmiar” o szerokości 100 z formatowaniem do prawej
  - c) „Data” o szerokości 80
4. Ponieważ będziemy korzystać z funkcjonalności związanej z plikami, musimy włączyć do listy używanych przestrzeni nazw przestrzeń **System.IO**:
5. Wypełniamy zawartość listy przed pojawieniem się formy. W tym celu tworzymy metodę dla zdarzenia **Load** formy:

```
using System.IO;

private void Form1_Load(object sender, EventArgs e)
{
    string[] lista = Directory.GetFiles(@"c:\"); // pobieramy listę plików

    // czyścimy zawartość listy
    listView1.Items.Clear();

    // dodajemy do wyświetlanej listy każdy plik z informacjami o nim
    foreach (string plik in lista)
    {
        ListViewItem lvi = new ListViewItem(); // tworzymy element kolekcji
        FileInfo fin = new FileInfo(plik); // obiekt informacji o pliku

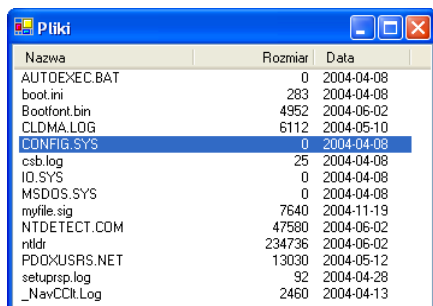
        // ustawiamy nazwę głównego elementu wyświetlanej listy
        lvi.Text = Path.GetFileName(plik);

        // dodajemy kolejny element wiersza (rozmiar)
        lvi.SubItems.Add(fin.Length.ToString());
        // dodajemy kolejny element wiersza (data utworzenia)
```

```
lvi.SubItems.Add(fin.CreationTime.ToShortDateString());


listView1.Items.Add(lvi); // dodajemy element do kolekcji
}
}
```

6. Po uruchomieniu aplikacji wyświetlone zostanie okno wyświetlające listę plików na dysku C:\ wraz z dodatkowymi informacjami



Rysunek 62. Widok okna z listą plików na dysku C:\

## Kontener podzielnika obszarów

Kontener podzielnika obszarów (kontrolka  `SplitContainer`) pozwala na budowanie okna zawierającego oddzielone od siebie widoki o zmieniającym się dynamicznie rozmiarze. Rozmiar widoku ustala się za pomocą tzw. podzielników (spliterów).

### Właściwości

Kontrolka podzielnika (*SplitContainer*) charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b><i>FixedPanel</i></b>	Określa, który z panelów kontenera ma zachować swój rozmiar w momencie zmiany rozmiaru kontenera.	<b><i>None</i></b>
<b><i>IsSplitterFixed</i></b>	Określa, czy podzielnik jest zablokowany (nie jest możliwa zmiana rozmiaru paneli w kontenerze).	<b><i>False</i></b>
<b><i>Orientation</i></b>	Określa położenie podzielnika obszarów. Właściwość ta może przyjąć wartości: <ul style="list-style-type: none"> <li>• <b><i>Vertical</i></b> – pionowe</li> <li>• <b><i>Horizontal</i></b> - poziome</li> </ul>	<b><i>Vertical</i></b>
<b><i>Panel1</i></b>	Zawiera właściwości dla lewego panela w kontenerze.	
<b><i>Panel1MinSize</i></b>	Określa minimalny rozmiar obszaru lewego panela.	<b>25</b>
<b><i>Panel2</i></b>	Zawiera właściwości dla prawego panela w kontenerze.	
<b><i>Panel2MinSize</i></b>	Określa minimalny rozmiar obszaru prawego panela.	<b>25</b>
<b><i>SplitterDistance</i></b>	Określa odległość podzielnika od lewej krawędzi kontenera.	
<b><i>SplitterIncrement</i></b>	Określa wielkość w pikselach, o jaki zostanie zwiększony/zmniejszony rozmiar obszaru paneli po	<b>1</b>



	przesunięciu podzielnika.	
<i>SplitterWidth</i>	Określa szerokość podzielnika.	4

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<i>Panel1</i>	Zdarzenia dla lewego panela.
<i>Panel2</i>	Zdarzenia dla prawego panela.
<i>SplitterMoved</i>	Występuje, gdy nastąpi przesunięcie podzielnika.
<i>SplitterMoving</i>	Występuje przed przesunięciem podzielnika.

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Eksploracja”, w której połączymy i odpowiednio zmodyfikujemy dwa poprzednie przykłady w jeden, czyli drzewo wyświetlające foldery na dysku C:\ oraz listę plików. Drzewo umieścimy w lewym panelu, a listę w prawym. Modyfikacja polegać będzie na stworzeniu zależności między aktywnym katalogiem w drzewie a listą plików, która zawierać będzie w tym przykładzie wszystkie pliki z aktywnego katalogu.

1. Tworzymy aplikację **Windows Forms** o tytule „Eksploracja”.
2. Dodajemy kontrolkę **SplitContainer**, która będzie miała dwa obszary.
3. Klikamy na lewy obszar, wstawiamy kontrolkę drzewa i zmieniamy jej właściwość **Dock** na **Fill**.
4. Dodajemy kontrolkę listy obrazów i wczytujemy dwie ikony: jedna symbolizująca folder otwarty i druga symbolizująca folder zamknięty.
5. Kojarzmy listę obrazów z drzewem (właściwości **ImageList**).
6. Zmieniamy właściwości **ImageIndex** na numer indeksu ikony folderu zamkniętego oraz **SelectedImageIndex** na indeks ikony folderu otwartego.
7. Ponieważ będziemy korzystać z funkcjonalności związanej z folderami i plikami musimy włączyć do listy używanych przestrzeni nazw przestrzeń **System.IO**:

```
using System.IO;
```

8. Tworzymy metodę rekurencyjną o nazwie **DodajFoldery**, która służyć będzie do wypełniania pojedynczej gałęzi drzewa (identycznie jak wcześniej).

```
private void DodajFoldery(string Sciezka, TreeNode Korzen)
{
    try
    {
        string[] Lista = Directory.GetDirectories(Sciezka);

        for (int i = 0; i < Lista.Length; i++)
        {
            TreeNode Lisc = new TreeNode(Path.GetFileName(Lista[i]));

            Korzen.Nodes.Add(Lisc);
            DodajFoldery(Lista[i], Lisc);
        }
    }
}
```

```
    }  
    }  
    catch (Exception)  
    {  
    }  
}
```

9. Korzystając z napisanej wcześniej metody **DodajFoldery**, tworzymy punkt początkowy, który rozpocznie proces rekurencyjnego tworzenia drzewa folderów w momencie utworzenia formy (zdarzenie **Load**). Metoda obsługująca to zdarzenie powinna wyglądać tak (identycznie jak wcześniej):

```
private void Form1_Load(object sender, EventArgs e)  
{  
    TreeNode Korzen = new TreeNode(@"Foldery C:\");  
  
    treeView1.Nodes.Clear();  
    treeView1.Nodes.Add(Korzen);  
  
    DodajFoldery(@"c:\", Korzen);  
}
```

10. Klikamy na prawy obszar, wstawiamy kontrolkę listy i zmieniamy jej właściwości:

- a) **Dock** na **Fill**
- b) **View** na **Details**
- c) **FullRowSelect** na **True**

11. Definiujemy następujące nagłówki kolumn dla listy (korzystając z kreatora kolumn):

- a) „Nazwa” o szerokości 150
- b) „Rozmiar” o szerokości 100 z formatowaniem do prawej
- c) „Data” o szerokości 80

12. Dodajemy metodę do zdarzenia wybrania elementu z drzewa **AfterSelect**. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)  
{  
    // parametr zdarzenia zawierający aktualnie wybrany element  
    TreeNode Node = e.Node;  
    string sciezka = "";  
  
    // tworzymy ścieżkę do folderu w drzewie budując ją od elementu  
    // aktywnego do korzenia  
    while (Node != null)  
    {  
        // do każdego elementu dodajemy separator ścieżki (za wyjątkiem  
        // korzenia drzewa)  
        if (sciezka.Length > 0)  
            sciezka = "\\\" + sciezka;
```

```

// jeżeli element nie jest korzeniem (nie ma rodzica) dodajemy jego
// etykietę do ścieżki, gdyż etykieta jest nazwą foldera
if (Node.Parent != null)
    sciezka = Node.Text + sciezka;
else
    sciezka = "C:" + sciezka; // w innym wypadku dodajemy symbol dysku

// poruszamy się w górę drzewa budując ścieżkę
Node = Node.Parent;
}

string[] lista = Directory.GetFiles(sciezka);

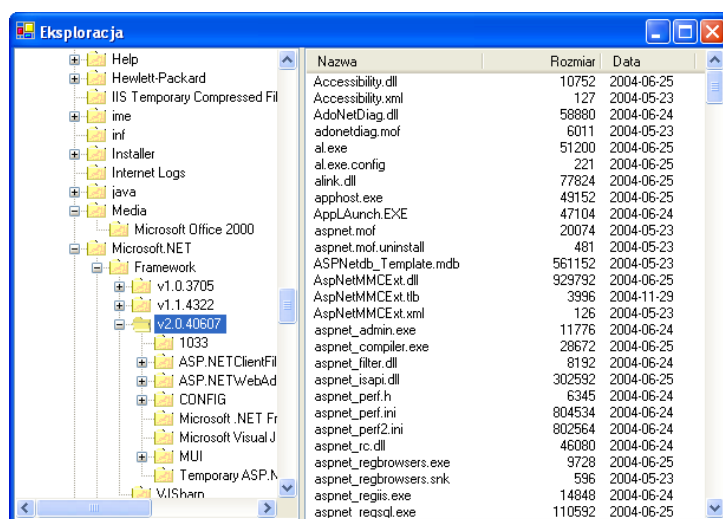
listView1.Items.Clear();

foreach (string plik in lista)
{
    ListViewItem lvi = new ListViewItem();
    FileInfo fin = new FileInfo(plik);

    lvi.Text = Path.GetFileName(plik);
    lvi.SubItems.Add(fin.Length.ToString());
    lvi.SubItems.Add(fin.CreationTime.ToShortDateString());
    listView1.Items.Add(lvi);
}
}


```

13. Po uruchomieniu aplikacji, (co może trochę potrwać ze względu na czas potrzebny na zbudowanie struktury) wyświetlone zostanie okno wyświetlające drzewo folderów na dysku C:\ oraz listę plików wraz z informacjami o nich dla aktualnie wybranego folderu.



Rysunek 63. Widok okna eksploracji dysku C:\

## Czasomierz

Czasomierz (kontrolka  **Timer**) służy do wykonywania określonych czynności po upływie określonego czasu (podanej liczby milisekund).

### Właściwości

Najważniejszą właściwością dla kontrolki czasomierza (**Timer**) jest właściwość **Interval**, która określa odstęp czasu w milisekundach po upływie którego zostanie wywołane zdarzenie **Tick** (domyślnie 100).

### Zdarzenia

Kontrolka ta posiada tylko jedno zdarzenie **Tick**, które zachodzi po upływie założonego czasu (określonego za pomocą właściwości **Interval**) od momentu startu czasomierza.

### Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Zegar cyfrowy”, która będzie wyświetlać aktualny czas.

1. Tworzymy aplikację **Windows Forms** o tytule „Zegar cyfrowy”.
2. Dodajemy etykietę tekstową i ustawiamy właściwości.
  - a) **Text** na *pusty*
  - b) **Font** na *Arial, 36 pkt*
3. Dodajemy kontrolkę **Timer** i ustawiamy jej właściwość **Interval** na 1000.
4. Dodajemy metodę do zdarzenia **Tick** dla czasomierza, która będzie wyświetlać aktualny czas.

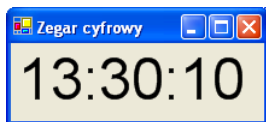
Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

5. Dodajemy metodę do zdarzenia **Load**, w której nasz czasomierz zacznie działanie. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void Form1_Load(object sender, EventArgs e)
{
    timer1.Start();
}
```

6. Po uruchomieniu aplikacji, wyświetlone zostanie okno zegara cyfrowego, który rozpocznie wyświetlanie aktualnego czasu.



Rysunek 64. Widok okna zegara cyfrowego

## Rozdział 5.

# Interakcja z użytkownikiem

### Współpraca z myszą

W czasie pracy użytkownika z aplikacją, każde użycie myszy powoduje, wysłanie zdarzeń odpowiadających tej czynności. W zależności od potrzeb, można tworzyć metody obsługujące zdarzenia:

- kliknięcia (*MouseClick*)
- podwójnego kliknięcia (*MouseDoubleClick*)
- naciśnięcia dowolnego przycisku myszy (*MouseDown*)
- wejścia w obszar kontrolki (*MouseEnter*)
- zatrzymania się w obszarze kontrolki (*MouseHover*)
- opuszczenia obszaru kontrolki (*MouseLeave*)
- zmiany pozycji kursora (*MouseMove*)
- zwolnienia przycisku myszy (*MouseUp*)

### Parametry zdarzenia

W przypadku zdarzeń: *MouseClick*, *MouseDoubleClick*, *MouseDown*, *MouseMove* oraz *MouseUp* wraz ze zdarzeniem przekazywane są dodatkowe parametry w obiekcie klasy *MouseEventArgs*. W pozostałych przypadkach wykorzystywana jest bazowa klasa parametrów zdarzenia *EventArgs*.

Klasa *MouseEventArgs* zawiera informacje o położeniu i stanie myszy:

Właściwość	Opis
<i>Button</i>	Określa rodzaj wciśniętego przycisku. Właściwość może przyjąć jedną z wartości listy wyliczanej <i>MouseButtons</i> : <ul style="list-style-type: none"> <li>• <i>None</i> – nie został naciśnięty żaden z przycisków</li> <li>• <i>Left</i> – naciśnięto lewy przycisk</li> <li>• <i>Middle</i> – naciśnięto środkowy przycisk</li> <li>• <i>Right</i> – naciśnięto prawy przycisk</li> <li>• <i>XButton1</i> – naciśnięto pierwszy przycisk nawigacji (Microsoft IntelliMouse)</li> <li>• <i>XButton2</i> – naciśnięto drugi przycisk nawigacji (Microsoft IntelliMouse)</li> </ul>
<i>Clicks</i>	Określa liczbę kliknięć i zwolnień przycisku myszy .
<i>Delta</i>	Określa parametr przesunięcia kółka myszy.
<i>X</i>	Określa wartość punktu x wskaźnika myszy.
<i>Y</i>	Określa wartość punktu y wskaźnika myszy.

### Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Mysz”, która będzie wyświetlać informacje dotyczące położenia myszy. Aplikacja będzie składać się z dwóch obszarów: dolnego wyświetlającego informację dotyczącą myszy i górnego, który będzie reagować na zdarzenie zmiany położenia kursora myszy.

1. Tworzymy aplikację **Windows Forms** o tytule „Mysz”.
2. Dodajemy dwa panele i zmieniamy ich właściwości:
  - **Dock** na **Bottom** dla pierwszego i **Full** dla drugiego
  - **BorderStyle** na **Fixed3D**
3. Dodajemy etykietę tekstową w obszarze dolnego panela, która wyświetlać będzie status myszy.
4. Dla górnego obszaru dodajemy obsługę do zdarzenia **MouseMove**. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void panel2_MouseMove(object sender, MouseEventArgs e)
{
    label1.Text = String.Format("Współrzędne x= y=", e.X, e.Y);
}
```

5. Po uruchomieniu aplikacji wyświetlone zostanie okno, w którym można obserwować dane o położeniu kursora myszy w górnym obszarze



Rysunek 65. Widok okna obserwacji położenia myszy

## Współpraca z klawiaturą

W czasie pracy użytkownika z aplikacją, każde użycie klawiatury powoduje wysłanie zdarzeń odpowiadających tej czynności. W zależności od potrzeb można tworzyć metody obsługujące zdarzenia:

- naciśnięcia klawisza (**KeyDown**, **KeyPress**)
- zwolnienia klawisza (**KeyUp**)

### Parametry zdarzenia

W przypadku zdarzeń: **KeyDown** oraz **KeyUp** wraz ze zdarzeniem przekazywane są dodatkowe parametry w obiekcie klasy **KeyEventArgs**. W przypadku zdarzenia **KeyPress** wraz ze zdarzeniem przekazywane są dodatkowe parametry w obiekcie klasy **KeyPressEventArgs**.

Klasa **KeyEventArgs** zawiera informacje o kodzie naciśniętego lub zwolnionego klawisza oraz statusie klawiszy specjalnych:

Właściwość	Opis
------------	------

<b>Alt</b>	Określa, czy w tym samym czasie naciśnięty został klawisz ALT (kombinacje z ALT).
<b>Control</b>	Określa, czy w tym samym czasie naciśnięty został klawisz CTRL (kombinacje z CTRL).
<b>Handled</b>	Określa, czy zostało obsłużone zdarzenie naciśnięcia/zwolnienia klawisza
<b>KeyCode</b>	Określa kod pierwszego naciśniętego/zwolnionego klawisza (zawiera tylko informacje o przeznaczeniu klawisza specjalnego ALT, CTRL, SHIFT). Właściwość może przyjąć jedną z wartości listy wyliczanej <b>Keys</b> (w dużej części etykieta stałej klawisza nosi nazwę zgodną z samym klawiszem lub jego funkcją np.: dla klawisza A jest to <i>A</i> ).
<b>KeyData</b>	Określa kod naciśniętego/zwolnionego klawisza (zawiera informacje o przeznaczeniu klawisza specjalnego oraz jego symbol ALT, CTRL, SHIFT). Właściwość może przyjąć jedną z wartości listy wyliczanej <b>Keys</b> .
<b>KeyValue</b>	Określa wartość numeryczną dla właściwości <b>KeyData</b> .
<b>Modifiers</b>	Określa, czy w tym samym czasie naciśnięty został jeden z klawiszy: ALT, CTRL lub SHIFT (kombinacje z ALT, CTRL, SHIFT).
<b>Shift</b>	Określa, czy w tym samym czasie naciśnięty został klawisz SHIFT (kombinacje z SHIFT)

Klasa **KeyPressEventArgs** zawiera informacje o znaku naciśniętego klawisza oraz status obsłużenia naciśnięcia przycisku:

Właściwość	Opis
<b>Handled</b>	Określa, czy zostało obsłużone zdarzenie naciśnięcia klawisza.
<b>KeyChar</b>	Zawiera znak, odpowiadający naciśniętemu klawiszowi.

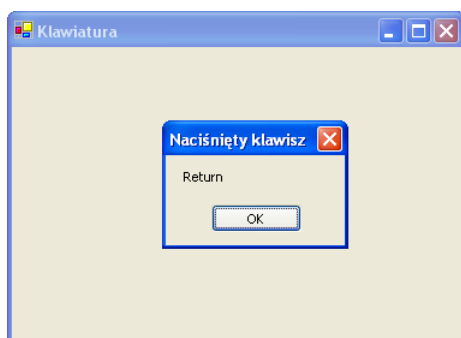
## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Klawiatura”, która będzie wyświetlać informacje dotyczące naciśniętych klawiszy.

1. Tworzymy aplikację **Windows Forms** o tytule „Klawiatura”.
2. Dodajemy obsługę zdarzenia **KeyDown**. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    MessageBox.Show(e.KeyData.ToString(), "Naciśnięty klawisz");
}
```

3. Po uruchomieniu aplikacji i naciśnięciu klawisza, zostanie wyświetlone okno z informacją o naciśniętym przycisku.





Rysunek 66. Widok okna aplikacji po naciśnięciu klawisza


## Korzystanie z menu, paska narzędzi i paska stanu


Menu, pasek narzędzi i pasek stanu wykorzystują wspólną funkcjonalność kontrolki **ToolStrip**, dlatego zostaną omówione razem.

Menu jest elementem interfejsu użytkownika, pozwalającym na swobodny dostęp do funkcji aplikacji. Menu pozwala na zgrupowanie w jednym miejscu podobnych i powiązanych ze sobą funkcji aplikacji.

Istnieją dwa typy menu:

- podstawowe (kontrolka  **MenuStrip**) występuje w postaci paska z rozwijanymi listami funkcji należącymi do wspólnej grupy.
- kontekstowe (kontrolka  **ContextMenuStrip**) występuje w postaci listy funkcji pojawiającej się w momencie kliknięcia prawego klawisza myszy.

Pasek narzędzi (kontrolka  **ToolStrip**) składa się z przycisków szybkiego dostępu do funkcji aplikacji. Pasek ten może być zakotwiczany w dowolnym miejscu okna, ale najczęściej umieszcza się go pod paskiem menu.

Pasek stanu (kontrolka  **StatusStrip**) służy do prezentowania specyficznych danych dla programu (np.: statusu klawiszy, położenia, opcji, liczby stron, postępu przetwarzania dokumentu, etc.).

### Właściwości

Kontrolka **ToolStrip** charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>AllowDrop</b>	Określa, czy kontrolka przyjmować będzie komunikaty o zdarzeniach przeciągnięcia i upuszczenia.	<b>False</b>
<b>AllowItemReorder</b>	Określa, czy możliwe jest porządkowanie elementów po wciśnięciu klawisza ALT.	<b>False</b>
<b>AllowMerge</b>	Określa, czy możliwe jest łączenie elementów.	<b>False</b>
<b>AutoRelocate</b>	Określa, czy kontrolka zmienia swoje położenie w zależności od zmian rozmiaru.	<b>False</b>
<b>CanOverflow</b>	Określa, czy elementy mogą się przepełnić.	<b>False</b>
<b>DropShadowEnabled</b>	Dostępne tylko w menu kontekstowym. Określa, czy okno menu kontekstowego ma rzucać cień.	<b>True</b>
<b>GripMargin</b>	Określa parametry symbolu zakotwiczania.	<b>{Left = 2, Top = 2, Right = 2, Bottom = 2}</b>
<b>GripStyle</b>	Określa zachowanie symbolu zakotwiczania. Właściwość może przyjąć jedną z wartości: <ul style="list-style-type: none"> <li>• <b>Hidden</b> – symbol zakotwiczenia jest ukryty</li> <li>• <b>Visible</b> – symbol zakotwiczenia jest widoczny</li> </ul>	<b>Hidden</b>
<b>ImageList</b>	Określa listę ikon skojarzoną z kontrolką.	<b>(none)</b>



<b>Items</b>	Zawiera kolekcję elementów kontrolki ( <b>ToolStripItemCollection</b> ).	
<b>MdiWindowListItem</b>	Dostępne tylko w menu podstawowym. Określa listę funkcji, jaka będzie widoczna w oknach potomnych aplikacji MDI.	<b>(none)</b>
<b>SaveSettings</b>	Określać, czy dla kontrolki mają być zapisywane ustawienia.	<b>False</b>
<b>ShowItemToolTips</b>	Określa, czy dla elementów będą wyświetlane podpowiedzi.	<b>True</b>
<b>Stretch</b>	Określa, czy kontrolka dopasowuje się do rozmiaru kontenera.	<b>False</b>
<b>TextDirection</b>	Określa kierunek tekstu etykiet tekstowych elementów. Właściwość może przyjąć jedną z wartości: <ul style="list-style-type: none"> <li>• <b>Inherit</b> – kierunek odziedziczony z kontrolki rodzica</li> <li>• <b>Horizontal</b> – kierunek poziomy</li> <li>• <b>Vertical90</b> – tekst odwrócony o 90 stopni</li> <li>• <b>Vertical270</b> – tekst odwrócony o 270 stopni</li> </ul>	<b>Horizontal</b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<b>Closed</b>	Dostępne tylko w menu kontekstowym. Występuje, gdy menu kontekstowe zostanie zamknięte.
<b>Closing</b>	Dostępne tylko w menu kontekstowym. Występuje przed zamknięciem menu kontekstowego.
<b>ItemAdded</b>	Występuje, gdy do kolekcji zostanie dodany nowy element.
<b>ItemClicked</b>	Występuje, gdy nastąpi kliknięcie na elemencie kolekcji menu.
<b>ItemRemoved</b>	Występuje, gdy z kolekcji zostanie usunięty jakiś element.
<b>MenuActivate</b>	Dostępne tylko w menu podstawowym. Występuje, gdy menu zostanie uaktywnione.
<b>Opened</b>	Dostępne tylko w menu kontekstowym. Występuje, gdy menu kontekstowe zostanie uaktywnione.

## Kolekcja elementów

Najważniejszą i najczęściej wykorzystywaną właściwością listy jest kolekcja elementów **Items**. Dzięki tej właściwości, można zarządzać kolekcją elementów menu, paska narzędzi oraz paska statusu.

Funkcjonalność kolekcji elementów listy jest realizowana za pomocą klasy **ToolStripItemCollection** (kolekcja elementów **ToolStripItem**).

Najczęściej wykorzystywane właściwości tej kolekcji przedstawiono poniżej:

Właściwość	Opis
<b>Count</b>	Zwraca liczbę elementów kolekcji.
<b>IsReadOnly</b>	Sprawdza czy kolekcja dostępna jest w trybie tylko do odczytu.
<b>Item</b>	Zwraca element kolekcji o określonym indeksie.

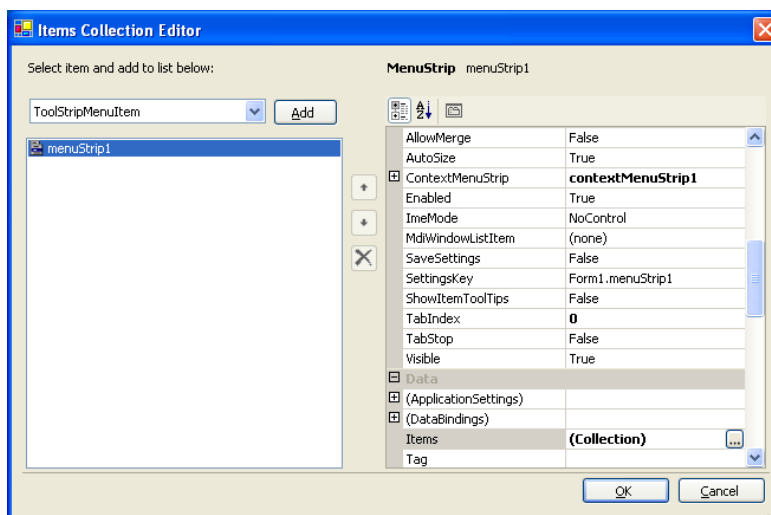
Najczęściej wykorzystywane metody tej kolekcji przedstawiono poniżej:

Metoda	Opis
<i>Add</i>	Pozwala na dodanie elementu do kolekcji.
<i>AddRange</i>	Pozwala na dodanie kilku elementów do kolekcji.
<i>Clear</i>	Pozwala na usunięcie wszystkich elementów z kolekcji.
<i>Contains</i>	Pozwala na sprawdzenie, czy w kolekcji znajduje się określony element.
<i>IndexOf</i>	Zwraca indeks podanego elementu kolekcji.
<i>Insert</i>	Pozwala na wstawienie do kolekcji elementu na określoną indeksem pozycję.
<i>Remove</i>	Pozwala na usunięcie z kolekcji podanego elementu.
<i>RemoveAt</i>	Pozwala na usunięcie z kolekcji elementu o określonym indeksie.

## Zarządzanie kolekcją elementów

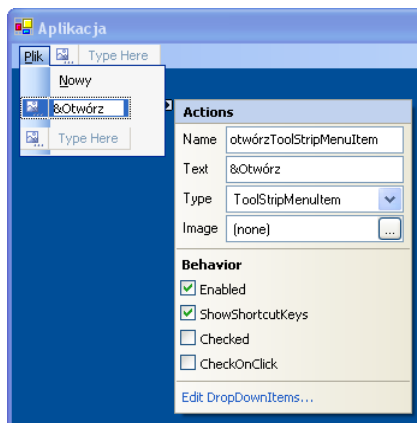
Definiowanie kolekcji elementów w czasie projektowania aplikacji, może odbywać się na dwa sposoby:

- za pomocą okna kreatora „Items Collection Editor”, które dostępne jest poprzez właściwość *Items*.



Rysunek 67. Okno kreatora elementów menu

- poprzez bezpośrednią modyfikację kontrolki w projekcie okna.



Rysunek 68. Projektowanie kontrolki menu

Znacznie wygodniej korzysta się z mechanizmu bezpośredniej modyfikacji kontrolki w projekcie okna.

Nadanie nazwy elementowi kolekcji wymaga jedynie wpisania w miejscu pojawiającego się tekstu „Type Here” właściwego tekstu etykiety. Jeżeli kontrolka powiązana jest z listą obrazków, można wskazać ikonę dla danej etykiety klikając z jej lewej strony (ikonka obrazka) i wskazać właściwą ikonę. Z prawej strony etykiety tekstowej, znajduje się przycisk rozwinięcia, którego naciśnięcie powoduje pojawienie się okna „Actions”. Okno to pozwala na szybką modyfikację podstawowych właściwości elementu. W zależności od rodzaju kontrolki, okno to może zawierać inne grupy podstawowych właściwości. Przykładowo, dla menu możliwe jest określenie rodzaju elementu menu (standardowo *ToolStripMenuItem*, można również wybrać inne elementy takie jak separator *ToolStripSeparator*, czy lista rozwijana *ToolStripComboBox*).

Oczywiście, jeżeli wskażemy utworzony element (po nadaniu etykiety tekstowej) możemy modyfikować jego właściwości w oknie *Properties*.

Definiując etykietę tekstową dla grupy, podgrupy lub funkcji można wskazać również aktywny znak skrótu, który pozwala na uaktywnienie elementu z klawiatury. Będzie on wyszczególniony w etykiecie poprzez podkreślenie. Definiowanie aktywnego znaku odbywa się poprzez dodanie znaku „&” przed znakiem pełniącym tę rolę w etykiecie.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację *Windows Forms* z tytułem „Interakcja”, która będzie reagować odpowiednimi komunikatami wyświetlanymi w pasku statusu na wybierane przez użytkownika opcje z menu.

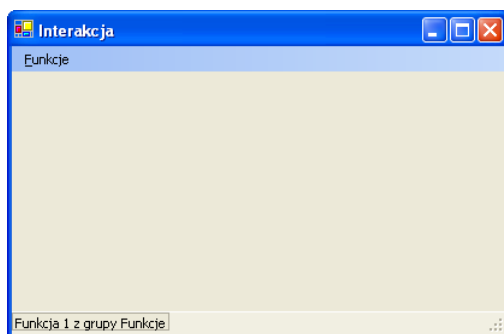
1. Tworzymy aplikację *Windows Forms* o tytule „Interakcja”.
2. Dodajemy kontrolkę *MenuStrip* i tworzymy menu np.: menu z grupą o nazwie „Funkcje” zawierającą dwa elementy „Funkcja 1” i „Funkcja 2”.
3. Dodajemy kontrolkę *StatusStrip* i tworzymy element typu *StatusStripPanel*.
4. Dodajemy metody obsługujące zdarzenia kliknięcia *Click* dla obu opcji z menu „Funkcje”.

Metody obsługujące te zdarzenia powinny wyglądać tak:

```
private void funkcjaToolStripMenuItem_Click(object sender, EventArgs e)
{
    statusStripPanel1.Text = "Funkcja 1 z grupy Funkcje";
}

private void funkcja2ToolStripMenuItem_Click(object sender, EventArgs e)
{
    statusStripPanel1.Text = "Funkcja 2 z grupy Funkcje";
}
```

5. Po uruchomieniu aplikacji i wybraniu jednej z funkcji w menu w pasku statusu wyświetlony zostanie odpowiedni komunikat, informujący jaka opcja i z jakiej grupy menu została wybrana.



*Rysunek 69. Okno aplikacji po wybraniu pierwszej funkcji*

## Rozdział 6.

# Korzystanie z okien dialogowych

### Tworzenie okien dialogowych

Okna dialogowe są szczególnym rodzajem okien aplikacji. Wykorzystywane są przede wszystkim do wprowadzania i wyprowadzania dodatkowych informacji w czasie interakcji z użytkownikiem.

Tworzenie okna dialogowego odbywa się w identyczny sposób jak tworzenie nowej formy (patrz Tworzenie Formy). Różnica polega na tym, że formy pełniące rolę okien dialogowych, muszą mieć ustawioną właściwość **FormBorderStyle** na **FixedDialog**. Okna dialogowe nie powinny również zawierać menu, pasków narzędzi i pasków statusu. Nie powinny również mieć widocznych przycisków minimalizacji i maksymalizacji okna.

Każde okno dialogowe powinno poza tym zawierać co najmniej jeden przycisk z ustawioną właściwością **DialogResult**, która wykorzystywana jest do określania stanu wyjściowego dla okna dialogowego.

Wyświetlenie okna dialogowego odbywa się poprzez wywołanie metody **ShowDialog**. Metoda ta może zwrócić jedną z wartości listy wyliczeniowej **DialogResult** (wynikającej z wartości zwracanej przez naciśnięcie jednego z przycisków tego okna):

```
OknoDialogowe dlg = new OknoDialogowe();

if (dlg.ShowDialog() == DialogResult.OK)
{
    // zaakceptowano dane
}
```

### Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Dialogi”, która będzie zawierać pole tekstowe, którego wartość będziemy wprowadzać jedynie przy pomocy okna dialogowego (w oknie głównym edycja nie będzie możliwa). Okno dialogowe będzie zawierać pole tekstowe do edycji danych oraz dwa przyciski: anulujący i akceptujący.

1. Tworzymy aplikację **Windows Forms** o tytule „Dialogi”.
2. Dodajemy kontrolkę pola tekstowego i ustawiamy jej właściwość **ReadOnly** na **True**.
3. Dodajemy etykietę tekstową opisującą przeznaczenie pola tekstowego o nazwie „Tekst”.
4. Dodajemy przycisk, który będzie służył do wywoływania okna dialogowego i nadajemy mu nazwę „Zmień”.
5. Dodajemy nową formę o tytule „Wprowadzanie danych” i ustawiamy jej właściwości:
  - a) **FormBorderStyle** na **FixedDialog**
  - b) **MaximizeBox** na **False**
  - c) **MinimizeBox** na **False**
6. Na nowej formie umieszczamy kontrolkę pola tekstowego i zmieniamy jej właściwość **Modifiers**

na **Public** (dzięki temu będziemy mogli z okna głównego odczytać wartość pola tekstowego wprowadzonego w oknie dialogowym).

7. Dodajemy etykietę tekstową opisującą pole tekstowe o nazwie „Tekst”.

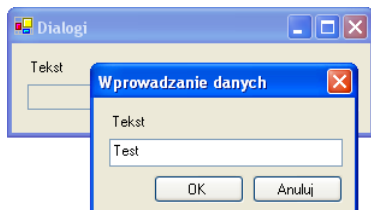
8. Dodajemy dwa przyciski: anulowania (ustawiamy etykietę na „Anuluj” oraz właściwość **DialogResult** na **Cancel**) oraz akceptacji (ustawiamy etykietę na „OK” oraz właściwość **DialogResult** na **OK**)

9. Przechodzimy do głównego okna i tworzymy metodę obsługującą zdarzenie kliknięcia na przycisk „Zmień”. Metoda obsługująca to zdarzenie powinna wyglądać tak:

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 dlg = new Form2();

    if (dlg.ShowDialog() == DialogResult.OK)
        textBox1.Text = dlg.textBox1.Text;
}
```

10. Po uruchomieniu aplikacji i naciśnięciu klawisza „Zmień” pojawi się okno dialogowe, gdzie będzie można wprowadzić tekst. Po naciśnięciu w oknie dialogowym przycisku „OK” zostanie on przepisany do pola tekstowego okna głównego.



Rysunek 70. Okno aplikacji po naciśnięciu przycisku „Zmień”

## Wspólne okna dialogowe

Wspólne okna dialogowe są oknami, które wykorzystywane są przez różne aplikacje do wykonywania podobnych czynności. Oferują one standardowy interfejs dostępowy do wspólnej funkcjonalności wymaganej w aplikacjach Windows. Okna te stanowią część wspólną systemu operacyjnego.

Należą do nich przede wszystkim okna pozwalające na:

- wybór pliku ( OpenFileDialog oraz SaveFileDialog )
- wybór określonego folderu ( FolderBrowserDialog )
- wybór koloru ( ColorDialog )
- wybór czcionki ( FontDialog )

### Okna wyboru pliku

Okna wyboru pliku pozwalają na wskazywanie lokalizacji plików, które mają być przetwarzane. Do wskazywania istniejących plików wykorzystuje się okno **OpenFileDialog** natomiast do określania lokalizacji do zapisu dla nowych plików okno **SaveFileDialog**.

### Właściwości

Okna wyboru pliku charakteryzują się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<b>AddExtension</b>	Określa, czy rozszerzenie pliku ma być automatycznie dodawane do nazwy.	<b>True</b>
<b>CheckFileExists</b>	Określa, czy przed wyjściem z okna dialogowego ma być sprawdzane istnienie wskazanego pliku.	<b>True (dla OpenFileDialog), False (dla SaveFileDialog)</b>
<b>CheckPathExists</b>	Określa, czy przed wyjściem z okna dialogowego ma być sprawdzone istnienie lokalizacji pliku.	<b>True</b>
<b>CreatePrompt</b>	Tylko dla <b>SaveFileDialog</b> . Określa, czy przed założeniem nowego pliku ma być wyświetlane ostrzeżenie. Działa jedynie, gdy właściwość <b>ValidateName</b> ustawiono na <b>True</b> .	<b>False</b>
<b>DefaultExt</b>	Określa domyślne rozszerzenie pliku.	
<b>FileName</b>	Zawiera nazwę wskazanego pliku.	
<b>Filter</b>	Zawiera listę rozszerzeń plików wyświetlaną w oknie dialogowym. Lista ta musi mieć format napisowy składający się z oddzielonych znakiem „ ” sekcji. Każda sekcja składa się z pary "opis   nazwa rozszerzenia". Przykładowa lista dwóch rozszerzeń z opisami:  <code>"Pliki C# *.cs Wszystkie pliki *.*"</code>	
<b>FilterIndex</b>	Określa numer sekcji na liście rozszerzeń domyślnie wyświetlanego w oknie dialogowym. Pierwszy element ma indeks równy 1.	<b>1</b>
<b>InitialDirectory</b>	Określa domyślny folder startowy.	
<b>Multiselect</b>	Tylko dla <b>OpenFileDialog</b> . Określa, czy możliwe jest wskazanie więcej niż jednego pliku.	<b>False</b>
<b>OverwritePrompt</b>	Tylko dla <b>SaveFileDialog</b> . Określa, czy wyświetlane będzie ostrzeżenie przed nadpisaniem istniejącego pliku.	<b>True</b>
<b>ShowHelp</b>	Określa, czy przycisk pomocy będzie widoczny.	<b>False</b>
<b>Title</b>	Zawiera napis, który będzie wyświetlany na pasku tytułowym okna.	
<b>ValidateName</b>	Określa, czy będzie sprawdzana poprawność nazw plików.	<b>True</b>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<b>FileOk</b>	Występuje, gdy użytkownik naciśnie przycisk „Otwórz” lub „Zapisz”.
<b>HelpRequest</b>	Występuje, gdy użytkownik naciśnie przycisk „Pomoc”.

## Okno wyboru folderu

Okno wyboru folderu (**FolderBrowserDialog**) pozwala wskazać lokalizację do folderu wybranego

z drzewa.

### Właściwości

Okno wyboru folderu charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>Description</i>	Zawiera tekst wyświetlany w pasku tytułowym okna.	
<i>RootFolder</i>	Określa położenie folderu nadrzędnego, dla którego realizowana będzie funkcja wskazania lokalizacji. W oknie zostanie wyświetlone wyłącznie drzewo dla wskazanego folderu nadrzędnego.	<i>Desktop</i>
<i>SelectedPath</i>	Zawiera lokalizację do aktualnie wskazanego folderu.	
<i>ShowNewFolderButton</i>	Określa, czy w oknie będzie dostępny przycisk utworzenia nowego folderu.	<i>True</i>

### Okno wyboru koloru

Okno wyboru koloru (*ColorDialog*) pozwala wybrać kolor z palety dostępnych kolorów.

### Właściwości

Okno wyboru folderu charakteryzuje się specyficznymi właściwościami pozwalającymi na zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>AllowFullOpen</i>	Określa dostępność przycisku „Definiuj kolory niestandardowe”.	<i>True</i>
<i>AnyColor</i>	Określa czy możliwa jest selekcja dowolnego koloru.	<i>False</i>
<i>Color</i>	Zawiera wybrany kolor.	<i>Black</i>
<i>FullOpen</i>	Określa, czy wyświetlane będzie pełne okno razem z sekcją kolorów niestandardowych.	<i>False</i>
<i>ShowHelp</i>	Określa, czy przycisk pomocy będzie widoczny.	<i>False</i>
<i>SolidColorOnly</i>	Określa, czy dozwolone są jedynie kolory pełne.	<i>False</i>

### Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<i>HelpRequest</i>	Występuje, gdy użytkownik naciśnie przycisk „Pomoc”.

### Okno wyboru czcionki

Okno wyboru koloru (*ColorDialog*) pozwala wybrać kolor z palety dostępnych kolorów.

### Właściwości

Okno wyboru koloru charakteryzuje się specyficznymi właściwościami pozwalającymi na



zdefiniowanie wyglądu i zachowania:

Właściwość	Opis	Wartość domyślna
<i>AllowScriptChange</i>	Określa, czy dozwolona jest zmiana zbioru znaków.	<i>True</i>
<i>AllowSimulations</i>	Określa, czy możliwa jest symulacja czcionek GDI.	<i>True</i>
<i>AllowVectorFonts</i>	Określa, czy możliwa jest selekcja czcionek wektorowych.	<i>True</i>
<i>AllowVerticalFonts</i>	Określa, czy możliwa jest selekcja czcionek wertykalnych.	<i>True</i>
<i>Color</i>	Zawiera wybrany kolor czcionki.	<i>Black</i>
<i>Font</i>	Zawiera wybraną czcionkę.	<i>Microsoft Sans Serif; 8,25pt</i>
<i>FontMustExists</i>	Określa, czy w przypadku braku czcionki ma być wyświetlana informacja.	<i>False</i>
<i>MaxSize</i>	Określa maksymalny rozmiar w punktach, jaki może zostać wybrany.	<i>0</i>
<i>MinSize</i>	Określa minimalny rozmiar w punktach, jaki może zostać wybrany.	<i>0</i>
<i>ScriptOnly</i>	Określa, czy z listy czcionek mają być wykluczone czcionki symboliczne i OEM.	<i>False</i>
<i>ShowApply</i>	Określa, czy przycisk „Zastosuj” będzie widoczny.	<i>False</i>
<i>ShowColor</i>	Określa, czy wybrany kolor będzie wyświetlany.	<i>False</i>
<i>ShowEffects</i>	Określa, czy będzie możliwa selekcja właściwości czcionki takich jak: podkreślenie, przekreślenie, kolor.	<i>True</i>
<i>ShowHelp</i>	Określa, czy przycisk pomocy będzie widoczny.	<i>False</i>

## Zdarzenia

Do najczęściej wykorzystywanych zdarzeń charakterystycznych dla tej kontrolki należą:

Zdarzenie	Opis
<i>Apply</i>	Występuje, gdy użytkownik naciśnie przycisk „Zastosuj”.
<i>HelpRequest</i>	Występuje, gdy użytkownik naciśnie przycisk „Pomoc”.

## Przykładowa aplikacja

Stwórzmy sobie prostą aplikację **Windows Forms** z tytułem „Notatnik”, która będzie uproszczoną wersją notatnika. Aplikacja powinna umożliwiać tworzenie nowych plików tekstowych, wczytywanie istniejących plików z dowolnej lokalizacji, zapisywanie plików w określonej lokalizacji. Powinna również być możliwa zmiana właściwości wyświetlanego w oknie tekstu (zmianę czcionki, rozmiaru, etc.) oraz tła notatnika (poprzez wybór koloru).

1. Tworzymy aplikację **Windows Forms** o tytule „Notatnik”.
2. Dodajemy menu i tworzymy dwie grupy „Plik” oraz „Opcje”.
3. Dodajemy do menu „Plik” następujące pozycje: „Nowy”, „Otwórz”, „Zapisz” oraz „Koniec” (przed wstawieniem elementu „Koniec” możemy wstawić separator wybierając typ elementu menu jako *ToolStripSeparator* zamiast *ToolStripMenuItem*).
4. Dodajemy do menu „Opcje” następujące pozycje: „Kolor tła”, „Czcionka”.
5. Dodajemy kontrolkę pola tekstowego i ustawiamy jej właściwości:

- a) **Dock** na **Fill**
- b) **Multiline** na **True**
- c) **ScrollBars** na **Both**

6. Ponieważ będziemy korzystać z funkcjonalności związanej z plikami musimy włączyć do listy używanych przestrzeni nazw przestrzeń **System.IO** (rozwijamy sekcję *Using directives* i wpisujemy odpowiednią deklarację):

```
using System.IO;
```

7. Dodajemy obsługę wybrania funkcji z menu (obsługa zdarzenia **Click** dla elementu menu). Poszczególne metody powinny wyglądać następująco:

a) dla funkcji „Nowy”:

```
private void nowyToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.Text = "";
}
```

b) dla funkcji „Otwórz”:

```
private void otwórzToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();

    // ustawiamy właściwości okna dialogowego
    dlg.Title = "Otwórz plik tekstowy";
    dlg.Filter = "Pliki tekstowe|.txt|Wszystkie pliki|*.*";
    dlg.FilterIndex = 1;

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        // korzystamy ze strumienia odczytu
        StreamReader read = new StreamReader(dlg.FileName);

        // wczytujemy całą zawartość i przekazujemy do kontrolki
        textBox1.Text = read.ReadToEnd();
        read.Close();
    }
}
```

c) dla funkcji „Zapisz”:

```
private void zapiszToolStripMenuItem_Click(object sender, EventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();

    // ustawiamy właściwości okna dialogowego
    dlg.Title = "Zapisz plik tekstowy";
    dlg.Filter = "Pliki tekstowe|.txt|Wszystkie pliki|*.*";
    dlg.FilterIndex = 1;
```

```
dlg.DefaultExt = "txt";

if (dlg.ShowDialog() == DialogResult.OK)
{
    StreamWriter write = new StreamWriter(dlg.FileName);

    write.Write(textBox1.Text);
    write.Close();
}
}
```

d) dla funkcji „Koniec”:

```
private void koniecToolStripMenuItem_Click(object sender,
                                           EventArgs e)
{
    Close();
}
```

e) dla funkcji „Kolor tła”:

```
private void kolortlaToolStripMenuItem_Click(object sender,
                                              EventArgs e)
{
    ColorDialog dlg = new ColorDialog();

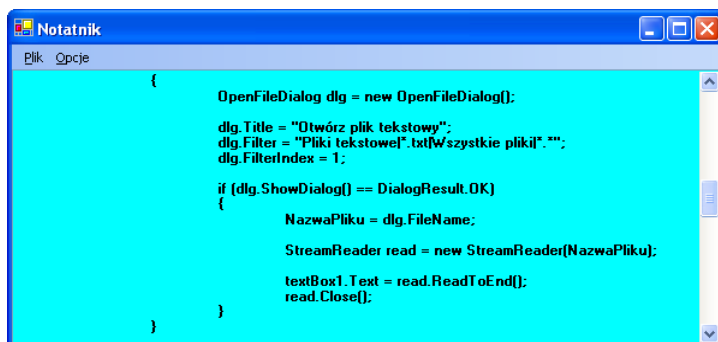
    if (dlg.ShowDialog() == DialogResult.OK)
        textBox1.BackColor = dlg.Color;
}
```

f) dla funkcji „Czcionka”:

```
private void czcionkaToolStripMenuItem_Click(object sender,
                                              EventArgs e)
{
    FontDialog dlg = new FontDialog();

    if (dlg.ShowDialog() == DialogResult.OK)
        textBox1.Font = dlg.Font;
}
```

8. Po uruchomieniu aplikacji będziemy mogli korzystać z prostych funkcji notatnika.



*Rysunek 71. Okno aplikacji notatnika z wczytanym plikiem, zmienionym tłem i czcionką*

## Rozdział 7.

# Tworzenie aplikacji MDI

### Tworzenie aplikacji MDI

Aplikacja może przyjąć styl interfejsu pojedynczego dokumentu **SDI** (*Single Document Interface*) lub wielu dokumentów **MDI** (*Multiple Document Interface*). Aplikacje SDI wykorzystują jeden główny widok okna i pozwalają na pracę z pojedynczym dokumentem. Aplikacje MDI wykorzystują mechanizm tworzenia widoków w oknach potomnych należących do okna głównego (kontenera widoków) i pozwalają na pracę z wieloma dokumentami (każdy dokument otwarty jest w osobnym oknie potomnym).

Tworzenie aplikacji MDI polega na zdefiniowaniu okna głównego, pełniącego rolę kontenera widoków oraz szablonu okna potomnego pełniącego rolę widoku.

W celu zdefiniowania okna kontenera należy ustawić jego właściwość **IsMdiContainer** na **True**. Po zdefiniowaniu szablonu okna widoku (utworzeniu nowej formy i umieszczeniu na niej kontrolki), możemy tworzyć nowe widoki wewnątrz kontenera w standardowy sposób. Należy jednak pamiętać przy tym, że są to okna potomne, więc wymagane jest ustawienie właściwości **MdiParent** na odpowiadającą oknu rodzica:

```
OknoPotomne nowe = new OknoPotomne(); // typ Form

nowe.MdiParent = rodzic; // rodzic - okno rodzica
nowe.Show();
```

Ze względu na to, że aplikacja MDI może mieć wiele widoków, zawsze kiedy istnieje potrzeba odwołania się do aktywnego okna widoku możemy skorzystać z właściwości okna rodzica **ActiveMdiChild**:

```
Form AktywnyWidok = rodzic.ActiveMdiChild;
```

Jeżeli potrzebujemy dostać się do listy wszystkich okien potomnych, możemy skorzystać z właściwości **MdiChildren**.

Przykładowo:

```
foreach (Form potomne in rodzic.MdiChildren)
{
    // operacje na oknie potomnym
}
```

Przyjęło się, że funkcjonalność tworzenia nowego okna widoku, umieszcza się w menu „Plik” pod pozycją „Nowy”. Przyjęło się również, że funkcjonalność związaną z zarządzaniem oknami widoków umieszcza się w menu „Okno”. Najczęściej wykonywaną czynnością dla zestawu widoków jest ich porządkowanie. Wykorzystuje się do tego celu metodę okna rodzica **LayoutMdi**. Metoda ta porządkuje okna widoków zgodnie z wartością argumentu wejściowego listy wyliczeniowej **MdiLayout**.

Poniższa tabela zawiera dopuszczalne wartości listy wyliczeniowej **MdiLayout**:

Etykieta wartości	Opis
<i>ArrangeIcons</i>	Ikony okien są porządkowane w obrębie powierzchni kontenera.
<i>Cascade</i>	Okna umieszczane są kaskadowo jedno pod drugim z odstępem.
<i>TileHorizontal</i>	Okna umieszczane są jedno pod drugim poziomo wypełniając całą powierzchnię kontenera.
<i>TileVertical</i>	Okna umieszczane są jedno obok drugiego pionowo wypełniając całą powierzchnię kontenera.

Przykład:

```
rodzic.LayoutMdi(MdiLayout.Cascade);
```

Kontrolka menu okna kontenera posiada specjalną właściwość **MdiWindowListItem**, dzięki której można określić, która grupa menu będzie wyświetlała listę widocznych okien potomnych.

## Przykładowa aplikacja

Stworzymy sobie prostą aplikację **Windows Forms** z tytułem „Notatnik MDI”, która będzie wersją MDI naszego wcześniejszego przykładu. Aplikacja powinna zachowywać się tak jak we wcześniejszym przykładzie, z tą jednak różnicą, że każdy plik tekstowy powinien być otwierany w osobnym oknie potomnym. Dla uproszczenia w tym przykładzie nie będziemy realizować funkcji zmiany koloru tła oraz czcionki.

1. Tworzymy aplikację **Windows Forms** o tytule „Notatnik MDI”.
2. Dla formy podstawowej ustawiamy właściwość **IsMdiContainer** na **True**.
3. Dodajemy menu i tworzymy dwie grupy: „Plik” oraz „Okno”.
4. Ustawiamy właściwość **MdiWindowListItem** dla menu na **oknoToolStripMenuItem**
5. Dodajemy do menu „Plik” następujące pozycje: „Nowy”, „Otwórz”, „Zapisz” oraz „Koniec” (przed wstawieniem elementu „Koniec” możemy wstawić separator).
6. Dodajemy do menu „Okno” następujące pozycje: „Zamknij”, „Zamknij wszystkie” oraz separator.
7. Ponieważ będziemy korzystać z funkcjonalności związanej z plikami musimy włączyć do listy używanych przestrzeni nazw, przestrzeń **System.IO** (rozwijamy sekcję **Using directives** i wpisujemy odpowiednią deklarację):

```
using System.IO;
```

8. Dodajemy nową formę o tytule „Bez nazwy” i umieszczamy na niej kontrolkę pola tekstowego, dla której ustawiamy właściwości:

- a) **Dock** na **Fill**
- b) **Multiline** na **True**
- c) **ScrollBars** na **Both**

9. Dodajemy metodę Wypełnij do nowej formy, którą będziemy wykorzystywać do wypełniania pola tekstowego oraz metodę Zawartosc, która będzie zwracać zawartość pola tekstowego (możemy zamiast tego upublicznić atrybut kontrolki pola tekstowego i bezpośrednio modyfikować zawartość pola tekstowego lub stworzyć właściwość):

```
public void Wypełnij(string Tekst)
{
```

```
    textBox1.Text = Tekst;
}
```

```
public string Zawartosc()
{
    return textBox1.Text;
}
```

10. Dodajemy obsługę wybrania funkcji z menu okna głównego (obsługa zdarzenia **Click** dla elementu menu). Poszczególne metody powinny wyglądać następująco:

a) dla funkcji „Nowy”:

```
private void nowyToolStripMenuItem_Click(object sender, EventArgs e)
{
    // tworzymy nowe okno widoku
    Form2 widok = new Form2();

    // wskazujemy okno rodzica
    widok.MdiParent = this;
    widok.Show();
}
```

b) dla funkcji „Otwórz”:

```
private void otwórzToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();

    dlg.Title = "Otwórz plik tekstowy";
    dlg.Filter = "Pliki tekstowe|.txt|Wszystkie pliki|*.*";
    dlg.FilterIndex = 1;

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        StreamReader read = new StreamReader(dlg.FileName);
        Form2 widok = new Form2();

        widok.MdiParent = this;
        widok.Text = dlg.FileName; // ustawiamy tytuł okna
        widok.Show();
        // wczytujemy całą zawartość i przekazujemy do okna widoku za pomocą
        // naszej metody Wypelnij
        widok.Wypelnij(read.ReadToEnd());

        read.Close();
    }
}
```

c) dla funkcji „Zapisz”:

```
private void zapiszToolStripMenuItem_Click(object sender, EventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();

    dlg.Title = "Zapis plik tekstowy";
    dlg.Filter = "Pliki tekstowe|*.txt|Wszystkie pliki|*.*";
    dlg.FilterIndex = 1;
    dlg.DefaultExt = "txt";

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        StreamWriter write = new StreamWriter(dlg.FileName);
        // pobieramy obiekt aktywnego okna
        Form2 aktywne = (Form2)this.ActiveMdiChild;

        // ustawiamy tytuł okna na nazwę nowego pliku
        aktywne.Text = dlg.FileName;
        // zapisujemy zawartość okna potomnego korzystając z
        // metody Zawartosc
        write.Write(aktywne.Zawartosc());
        write.Close();
    }
}
```

d) dla funkcji „Koniec”:

```
private void koniecToolStripMenuItem_Click(object sender, EventArgs e)
{
    Close();
}
```

e) dla funkcji „Zamknij”:

```
private void closeToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form aktywne = this.ActiveMdiChild;

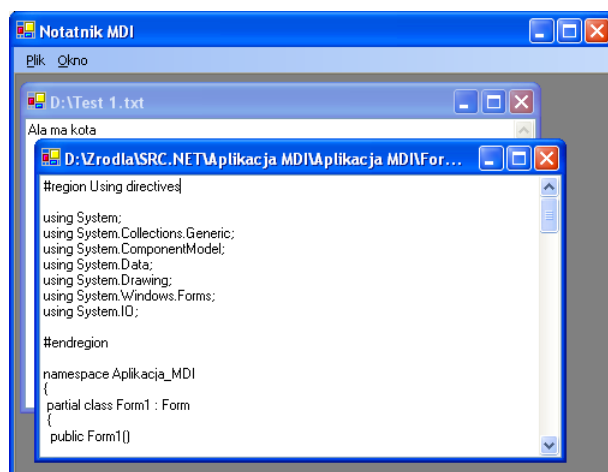
    aktywne.Close();
}
```

f) dla funkcji „Zamknij wszystkie”:

```
private void zamknijwszystkieToolStripMenuItem_Click(object sender,
                                                    EventArgs e)
{
    foreach (Form frm in this.MdiChildren)
        frm.Close();
}
```



11. Po uruchomieniu aplikacji będziemy mogli korzystać z prostych funkcji notatnika w architekturze MDI.



Rysunek 72. Okno aplikacji notatnika z otwartymi dwoma oknami

# Źródła

- C# Language Specification, Standard ECMA-334 2<sup>nd</sup> edition December 2002, ECMA International.
- MSDN Library, <http://msdn.microsoft.com/library/default.asp>, © 2005 Microsoft Corporation. All rights reserved.