

# Programowanie pod Windows

Wersja 1.2, marzec 2020

Uwaga: notatki są w fazie rozwoju. Brakujące elementy będą sukcesywnie uzupełniane. Dokument może być bez zgody autora rozpowszechniany, zabrania się jedynie czerpania z tego korzyści materialnych.

Wiktor Zychła

Instytut Informatyki  
Uniwersytetu Wrocławskiego

Wrocław 2003-2020



# Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>11</b>
1.1	Historia systemu operacyjnego Windows	11
1.2	Windows z punktu widzenia programisty	12
1.3	Narzędzia programistyczne	13
<b>2</b>	<b>Win32 API</b>	<b>17</b>
2.1	Fundamentalne idee Win32API	17
2.2	Okna	18
2.2.1	Tworzenie okien	18
2.2.2	Komunikaty	22
2.2.3	Okna potomne	25
2.2.3.1	Tworzenie okien potomnych	25
2.2.3.2	Aktywowanie i deaktywowanie okien potomnych	28
2.2.3.3	Komunikacja między oknem potomnym a macierzystym	29
2.2.4	Subclasowanie okien potomnych	31
2.2.5	Obsługa grafiki za pomocą GDI	34
2.2.5.1	Podstawy GDI	34
2.2.5.2	Uchwyty do kontekstów urządzeń	36
2.2.5.3	Własne kroje pisma	38
2.2.6	Tworzenie menu	38
2.3	Procesy, wątki, synchronizacja	40
2.3.1	Tworzenie wątków i procesów	40
2.3.2	Synchronizacja wątków	42
2.3.2.1	Zdarzenia	43
2.3.2.2	Mutexy	44
2.3.2.3	Semaforey	46
2.3.2.4	Sekcja krytyczna	48
2.4	Komunikacja między procesami	49
2.4.1	Charakterystyka protokołów sieciowych	49
2.4.2	Podstawy biblioteki Winsock	50
2.4.2.1	Gniazda asynchroniczne	56
2.5	Inne ważne elementy Win32API	57
2.5.1	Biblioteki ładowane dynamicznie	57
2.5.2	Różne przydatne funkcje Win32API	58
2.5.3	Zegary	59
2.5.4	Okna dialogowe	62
2.5.4.1	Powłoka systemu	66

<b>3</b>	<b>.NET</b>	<b>69</b>
3.1	Programowanie obiektowe . . . . .	69
3.2	C# - podstawowe elementy języka . . . . .	72
3.2.1	Pierwszy program . . . . .	72
3.2.2	Struktura kodu, operatory . . . . .	74
3.2.3	System typów, model obiektowy . . . . .	75
3.2.4	Typy proste a typy referencyjne, boxing i unboxing . . . . .	77
3.2.5	Klasy . . . . .	78
3.2.5.1	Pola . . . . .	80
3.2.5.2	Metody . . . . .	81
3.2.5.3	Przekazywanie parametrów do metod . . . . .	84
3.2.5.4	Konstruktory . . . . .	86
3.2.5.5	Właściwości (propercje) . . . . .	87
3.2.5.6	Stałe . . . . .	88
3.2.5.7	Indeksy . . . . .	88
3.2.5.8	Przeciążanie operatorów . . . . .	89
3.2.6	Struktury . . . . .	90
3.2.7	Dziedziczenie . . . . .	91
3.2.8	Niszczenie obiektów . . . . .	93
3.2.8.1	Destruktry . . . . .	93
3.2.8.2	Interfejs <i>IDisposable</i> . . . . .	94
3.2.9	Interfejsy . . . . .	96
3.2.10	Konwersje między typami . . . . .	99
3.2.10.1	Typy wbudowane . . . . .	100
3.2.10.2	Typy własne . . . . .	100
3.2.10.3	Dziedziczenie a konwersje . . . . .	102
3.2.11	Wyjątki . . . . .	103
3.2.12	Klasa <i>string</i> . . . . .	104
3.2.12.1	Podstawowe możliwości klasy <i>string</i> . . . . .	104
3.2.12.2	Formatowanie . . . . .	105
3.2.12.3	Encoding . . . . .	107
3.2.13	Delegaty i zdarzenia . . . . .	108
3.2.13.1	Delegaty . . . . .	108
3.2.13.2	Zdarzenia . . . . .	110
3.2.13.3	Uczeń Czarnoksiężnika . . . . .	111
3.2.14	Zestawy . . . . .	120
3.2.15	Refleksje . . . . .	121
3.2.16	Atrybuty . . . . .	123
3.2.16.1	Predefiniowane atrybuty . . . . .	125
3.2.17	Kod niebezpieczny . . . . .	125
3.2.18	Dokumentowanie kodu . . . . .	126
3.2.19	Dekompilacja kodu . . . . .	129
3.2.19.1	Dekompilacja do języka CIL . . . . .	131
3.2.19.2	Dekompilacja do C# . . . . .	132
3.2.19.3	Zabezpieczanie się przed dekompilacją . . . . .	133
3.2.20	Kolekcje niegeneryczne - <b>System.Collections</b> . . . . .	133
3.2.20.1	Tablice . . . . .	133
3.2.20.2	Tablice referencji . . . . .	134
3.2.20.3	Tablice wielowymiarowe . . . . .	135

3.2.20.4	<b>ArrayList</b>	137
3.2.20.5	Kolekcje silnie otypowane	138
3.2.20.6	Stos, kolejka	139
3.2.20.7	<b>Hashtable</b>	139
3.2.21	Enumerowalne kolekcje, interfejsy <b>IEnumerable</b> i <b>IEnumerator</b>	140
3.2.21.1	Sortowanie kolekcji	142
3.2.21.2	Opakowywanie enumeratorów	147
3.3	<b>C# 2.0</b>	152
3.3.1	Typy generyczne	152
3.3.1.1	Motywacja	152
3.3.1.2	Ograniczenia na typy generyczne	154
3.3.1.3	Ograniczenia typowe w praktyce	158
3.3.1.4	Abstrakcja operacji jako wymagania danego typu	159
3.3.1.5	Abstrakcja operacji jako zobowiązanie typu pomocniczego	160
3.3.1.6	Programowanie dynamiczne	161
3.3.1.7	Lokalna fabryka "dodawaczy"	162
3.3.1.8	Wykonywanie kodu generycznego	164
3.3.2	Kolekcje generyczne - <b>System.Collections.Generic</b>	164
3.3.2.1	Typy funkcyjne	166
3.3.2.2	Wykorzystanie typów funkcyjnych	168
3.3.3	Łatwiejsze enumerowanie z <b>yield</b>	168
3.3.4	Typy proste dopuszczające wartości <b>null</b>	172
3.4	<b>C# 3.0</b>	173
3.5	Biblioteka standardowa platformy .NET	173
3.5.1	Biblioteka funkcji matematycznych	173
3.5.2	Biblioteki wejścia/wyjścia	174
3.5.2.1	Struktura systemu plików	174
3.5.2.2	Obsługa danych w strumieniach	176
3.5.2.3	Szyfrowanie strumieni w locie	177
3.5.2.4	Strumień konsoli	179
3.5.3	Dynamiczne tworzenie kodu	179
3.5.4	Procesy, wątki	183
3.5.4.1	Procesy	183
3.5.4.2	Wątki	184
3.5.5	XML	185
3.5.5.1	<i>XmlTextReader</i> i <i>XmlTextWriter</i>	186
3.5.5.2	Obiekty DOM	187
3.5.5.3	XPath	188
3.5.5.4	Automatyczne tworzenie kodu do ładowania XML	189
3.5.5.5	Weryfikacja poprawności danych XML	192
3.5.6	Komunikacja między procesami	193
3.5.7	Wyrażenia regularne	196
3.5.7.1	Czym są <i>wyrażenia regularne</i>	196
3.5.7.2	Język wyrażeń regularnych	196
3.5.7.3	Dzielenie tekstu	196
3.5.7.4	Wyszukiwanie wzorca	197
3.5.7.5	Edycja, usuwanie tekstu	197
3.5.8	Serializacja	198
3.5.8.1	Serializacja binarna	198

3.5.8.2	Serializacja SOAP . . . . .	199
3.5.9	Wołanie kodu niezarządzanego . . . . .	200
3.5.9.1	Funkcje zwrotne . . . . .	202
3.5.10	Odśmiecacz . . . . .	203
3.6	Aplikacje okienkowe - <code>System.Windows.Forms</code> . . . . .	204
3.6.1	Tworzenie okien . . . . .	205
3.6.2	Okna potomne . . . . .	205
3.6.3	Zdarzenia . . . . .	207
3.6.3.1	Parametry zdarzeń . . . . .	209
3.6.3.2	Pokrywanie funkcji obsługi zdarzeń . . . . .	209
3.6.4	Okna dialogowe . . . . .	211
3.6.5	Subclassowanie okien . . . . .	211
3.6.5.1	Obsługa komunikatów własnych okien . . . . .	211
3.6.5.2	Obsługa komunikatów istniejących okien . . . . .	212
3.6.6	Komponenty wizualne . . . . .	213
3.6.6.1	ComboBox, ListBox . . . . .	213
3.6.6.2	ToolTip . . . . .	215
3.6.6.3	ListView . . . . .	215
3.6.6.4	TreeView . . . . .	217
3.6.7	Rozmieszczanie okien potomnych . . . . .	219
3.6.7.1	Kotwice i dokowanie . . . . .	219
3.6.7.2	Panele . . . . .	221
3.6.7.3	Splittery . . . . .	221
3.6.8	GDI+ . . . . .	222
3.6.8.1	Obiekt <b>Graphics</b> . . . . .	223
3.6.8.2	Kolory . . . . .	224
3.6.8.3	Czcionki . . . . .	224
3.6.8.4	Pędzle, szczotki . . . . .	224
3.6.8.5	Obrazki . . . . .	226
3.6.8.6	Podwójne buforowanie . . . . .	226
3.6.9	Zegary . . . . .	227
3.6.10	Menu . . . . .	228
3.6.10.1	Tworzenie menu . . . . .	228
3.6.10.2	Własne funkcje rysowania menu . . . . .	229
3.6.11	Schówek . . . . .	231
3.6.12	Drag & drop . . . . .	231
3.6.13	Tworzenie własnych komponentów . . . . .	231
3.6.13.1	Najprostszy komponent . . . . .	232
3.6.13.2	Komponent złożony . . . . .	233
3.6.13.3	Definiowanie własnych zdarzeń . . . . .	234
3.6.14	Typowe okna dialogowe . . . . .	236
3.6.14.1	Okna wyboru plików . . . . .	236
3.6.14.2	Okno wyboru czcionki . . . . .	236
3.6.14.3	Okno wyboru koloru . . . . .	237
3.7	Ciekawostki .NET . . . . .	237
3.7.1	Błąd odśmiecania we wczesnych wersjach Frameworka . . . . .	237
3.7.2	Dostęp do prywatnych metod klasy . . . . .	238
3.7.3	Informacje o systemie . . . . .	238
3.7.3.1	Informacje o konfiguracji systemu . . . . .	238

3.7.3.2	Informacje o środowisku graficznym . . . . .	239
3.7.3.3	Informacje o środowisku uruchomieniowym . . . . .	239
3.7.4	Własny kształt kursora myszy . . . . .	239
3.7.5	Własne kształty okien . . . . .	240
3.7.6	Podwójne buforowanie grafiki w GDI+ . . . . .	240
3.7.7	Sprawdzanie uprawnień użytkownika . . . . .	240
3.7.8	Ikona skojarzona z plikiem . . . . .	240
3.7.9	WMI . . . . .	241
3.8	Bazy danych i ADO.NET . . . . .	242
3.8.1	Interfejsy komunikacji z bazami danych . . . . .	242
3.8.1.1	ODBC . . . . .	243
3.8.1.2	OLE DB . . . . .	243
3.8.1.3	ADO i ADO.NET . . . . .	243
3.8.2	Manualne zakładanie bazy danych . . . . .	244
3.8.3	Nawiązywanie połączenia z bazą danych . . . . .	245
3.8.4	Pasywna wymiana danych . . . . .	246
3.8.5	Lokalne struktury danych . . . . .	247
3.8.6	Programowe zakładanie bazy danych . . . . .	251
3.8.7	Transakcje . . . . .	251
3.8.8	Typ <b>DataSet</b> . . . . .	252
3.8.9	Aktywna wymiana danych . . . . .	254
3.8.10	ADO.NET i XML . . . . .	255
3.8.11	Wiązanie danych z komponentami wizualnymi . . . . .	256
3.9	Dynamiczne WWW i ASP.NET . . . . .	258
3.9.1	Dlaczego potrzebujemy dynamicznego WWW . . . . .	258
3.9.2	Przegląd technologii dynamicznego WWW . . . . .	258
3.9.2.1	Common Gateway Interface . . . . .	258
3.9.2.2	Internet Server Application Programming Interface . . . . .	259
3.9.2.3	ASP . . . . .	259
3.9.3	Czym jest ASP.NET . . . . .	260
3.9.4	Pierwszy przykład w ASP.NET . . . . .	260
3.9.5	Łączenie stron ASP.NET z dowolnym kodem . . . . .	260
3.9.6	Kontrolki ASP.NET . . . . .	262
3.9.7	Inne przykłady ASP.NET . . . . .	263
3.9.7.1	Identyfikacja klienta . . . . .	263
3.9.7.2	Licznik odwiedzin strony . . . . .	263
3.9.8	Narzędzia wspomagające projektowanie stron ASP.NET . . . . .	266
3.9.8.1	Visual Studio .NET . . . . .	266
3.9.8.2	ASP.NET WebMatrix . . . . .	266
3.10	Inne języki platformy .NET . . . . .	266
3.10.1	VB.NET . . . . .	266
3.10.1.1	Przykładowa aplikacja w VB.NET . . . . .	267
3.10.1.2	Dynamiczne wiązanie . . . . .	268
3.10.2	ILAsm . . . . .	270
3.10.2.1	Informacje ogólne . . . . .	270
3.10.2.2	Pierwszy program w ILAsm . . . . .	270
3.10.2.3	Stałe i zmienne . . . . .	271
3.10.2.4	Instrukcje arytmetyczne . . . . .	272
3.10.2.5	Operacje warunkowe, skoki . . . . .	273

3.10.2.6	Metody i parametry . . . . .	274
3.10.2.7	Obiekty, pola, metody . . . . .	275
3.10.2.8	Polimorfizm . . . . .	275
3.10.2.9	Wyjątki . . . . .	277
3.10.3	Łączenie kodu z różnych języków . . . . .	278
3.10.3.1	Zasady łączenia kodu różnych języków . . . . .	278
3.10.3.2	Pułapki . . . . .	280
<b>A</b>	<b>Przykładowe aplikacje</b>	<b>285</b>
A.1	Animowany fraktalny zbiór Julii . . . . .	285
A.2	Bezpośredni dostęp do nośnika danych w Windows NT . . . . .	287



# Zamiast wstępu

## Dla kogo jest ten skrypt

Skrypt skierowany jest do programistów, którzy chcą dowiedzieć się jakich narzędzi i języków używać aby pisać programy pod Windows oraz jak wygląda sam system widziany oczami programisty. Powstał jako materiał pomocniczy do wykładu "Programowanie pod Windows", układ materiału odpowiada więc przebiegowi wykładu.

Zakładam, że czytelnik potrafi programować w C, wie co to jest kompilator, kod źródłowy i wynikowy, zna trochę C++ lub Javę. Dość dokładnie omawiam elementy języka C#, można więc rozdział poświęcony omówieniu tego języka potraktować jako mini-leksykon C#.

Poznanie nowych języków i metod programowania traktuję jako nie tylko pracę ale i bardzo uzależniające hobby. Ucząc się nowych rzeczy, czytam to co autor ma do powiedzenia na ich temat, a potem staram się dokładnie analizować listingi przykładowych programów. Niestety, bardzo często zdarza się, że kody przykładowych programów w książkach są koszmarnie długie! Autorzy przykładów być może kierują się przekonaniem, że przykładowy kod powinien wyczerpywać demonstrowane zagadnienie w sposób pełny, a ponadto zapoznać czytelnika przy okazji z paroma dodatkowymi, czasami niezwiązanymi z tematem, elementami. Tylko jak, chcąc nauczyć się czegoś szybko, znaleźć czas na analizę czasami kilkunastu stron kodu źródłowego, aby między 430 a 435 wierszem znaleźć interesujący mnie fragment?

Nie potrafię odpowiedzieć na to pytanie. Dlatego kody przykładowych programów w tym skrypcie są bardzo krótkie, czasami wręcz symboliczne. Zakładam bowiem, że programista który chce na przykład dowiedzieć się jak działa *ArrayList* nie potrzebuje jako przykładu 10 stron kodu źródłowego prostej aplikacji bazodanowej, tylko 10-15 linijek demonstrujących użycie tego a nie innego obiektu. Mimo to przeważająca większość przykładów to kompletne programy, gotowe do uruchomienia.

Zapraszam do lektury.

## Uwagi do wersji 1.0 i nowszych

Przygotowana w roku 2003 wersja 0.99 niniejszych notatek przez wiele lat z powodzeniem zaspokajała podstawowe potrzeby informacyjne w temacie programowania w systemie Windows. Wraz z wprowadzaniem nowych elementów do języka C# z czasem niedostatek informacyjny stał się mocno odczuwalny.

W 2017 rozpoczęto więc proces aktualizacji notatek, który ma za zadanie zasypanie luki jaką niewątpliwie był brak poruszenia tak ważnych tematów jak chociażby programowanie generyczne czy LINQ.

Po raz kolejny zapraszam do lektury jak również do dzielenia się uwagami, które w miarę możliwości będą uwzględniane przy przygotowywaniu kolejnych wersji.



# Rozdział 1

## Wprowadzenie

### 1.1 Historia systemu operacyjnego Windows

Na początku lat 80-tych pierwsze komputery osobiste pracowały pod kontrolą systemu operacyjnego MS-DOS. Swoim użytkownikom DOS oferował prosty interfejs, w którym polecenia systemowe i programy przywoływało się z *linii poleceń*. Programiści mieli do dyspozycji zbiór tzw. *przerwań* za pomocą których mogli sięgać do urządzeń wejścia/wyjścia. DOS był systemem jednozadaniowym, to znaczy, że w każdej chwili w systemie aktywny był tylko jeden proces<sup>1</sup>.

Pierwsza wersja interfejsu graficznego została zapowiedziana w roku 1983, zaś na rynek trafiła w listopadzie 1985. Windows 1.0 był odpowiedzią Microsoftu na graficzny interfejs jaki zaprojektowano w firmie Apple<sup>2</sup>. W 1987 roku pojawił się Windows 2.0, którego główną innowacją była możliwość nakładania się okien na siebie (w przeciwieństwie do okien ułożonych obok siebie w Windows 1.0). Oba systemy pracowały w trybie rzeczywistym procesorów 8086 mając dostęp do 1 MB pamięci. 22 maja 1990 roku pojawił się Windows 3.0, który potrafił już korzystać z trybu chronionego procesora 80386, mając dzięki temu dostęp aż do 16MB pamięci operacyjnej. Dwa lata później, w 1992, pojawił się Windows 3.1, który wprowadził nowe technologie: czcionki TrueType, OLE oraz obsługę multimediów. W czerwcu 1993 pojawiła się pierwsza wersja systemu Windows NT, którego jądro pracowało w trybie chronionym procesorów 80386, liniowym trybie adresowania i 32-bitowym trybie adresowania. Windows NT napisano niemal całkowicie od początku w C, dzięki czemu system ten był przenośny i pracował m.in. na platformach RISC-owych.

Wprowadzony na rynek w roku 1995 Windows 95, choć nieprzenośny i uboższy od NT o mechanizmy zabezpieczeń, zdobył dużą popularność jako system do użytku domowego. Pojawienie się tych dwóch systemów oznacza do dziś zasadniczą linię podziału Windows na dwie rodziny: rodzinę systemów opartych na jądrze NT (Windows NT, Windows 2000, Windows XP) oraz rodzinę opartą na uproszczonym jądrze, rozwijanym od czasów Windows 95 (Windows 95, Windows 98, Windows ME). Zapowiadana kolejna wersja systemu ma ostatecznie połączyć obie linie.

---

<sup>1</sup>Pewnym sposobem na pokonywanie tego ograniczenia było wykorzystanie przerywania zegara, dzięki czemu było możliwe wykonanie jakiegoś małego fragmentu kodu w regularnych odstępach czasu. Nie zmienia to jednak faktu, że DOS nie wspierał wielozadaniowości

<sup>2</sup>Miedzy Microsoftem a Apple regularnie toczyły się spory dotyczące praw do korzystania z różnych elementów interfejsu graficznego

## 1.2 Windows z punktu widzenia programisty

System operacyjny Windows zbudowany jest ze współpracujących ze sobą części zarządzających m.in. pamięcią, interakcją z użytkownikiem, urządzeniami wejścia-wyjścia. Z punktu widzenia programisty istotne jest w jaki sposób aplikacja może funkcjonować w systemie wchodząc w interakcje z różnymi jego składnikami. To czego potrzebuje programista, to informacje o tym w jaki sposób aplikacja ma komunikować się z systemem plików, jak obchodzić się z pamięcią, jak komunikować się z siecią itd.

Windows jest systemem operacyjnym zbudowanym warstwowo. Tylko najniższe warstwy systemu mogą operować na poziomie sprzętu - programista takiej możliwości nie ma (poza wczesnymi implementacjami Windows, w których taki dostęp jest możliwy). Oznacza to, że nie ma możliwości bezpośredniego odwołania się do pamięci ekranu, czy odczytania wartości z dowolnie wybranej komórki pamięci. Nie można bezpośrednio operować na strukturze dysku twardego, ani sterować głowicą drukarki. Zamiast tego programista ma do dyspozycji pewien ściśle określony zbiór *funkcji i typów danych*, za pomocą których program może komunikować się z systemem. O takim zbiorze funkcji i typów mówimy, że jest to **interfejs programowania** (*ang. Application Programming Interface, API*) jaki dany system udostępnia<sup>3</sup>.

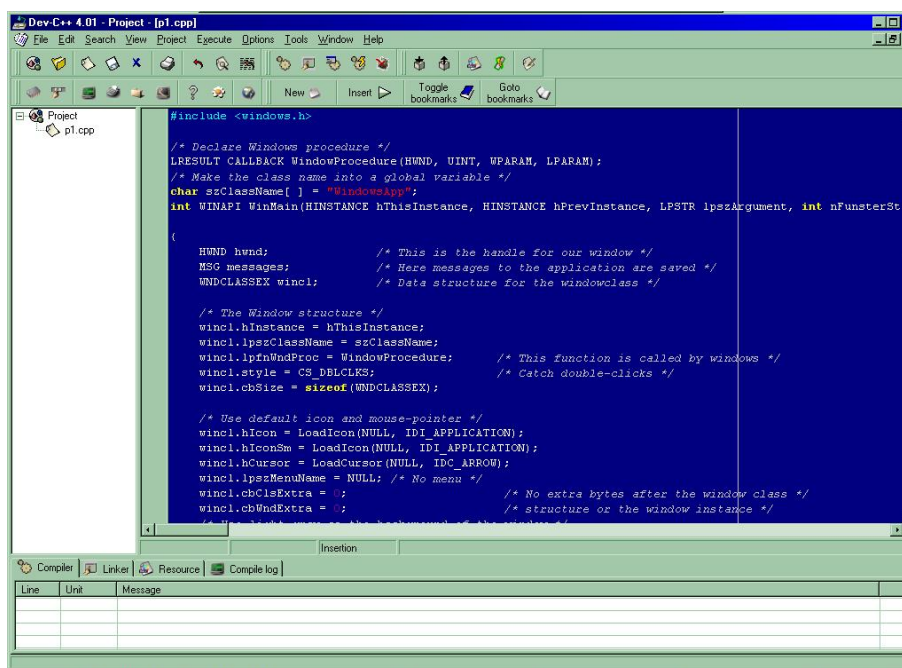
Dzięki takiej konstrukcji systemu operacyjnego programista nie musi martwić się na przykład o model karty graficznej jaki posiada użytkownik, bowiem z jego punktu widzenia oprogramowanie każdego możliwego typu karty graficznej wygląda dokładnie tak samo. To system operacyjny zajmuje się (tu: za pomocą sterownika) komunikacją z odpowiednimi częściami komputera i z punktu widzenia programisty robi to w sposób jednorodny. Co więcej, z punktu widzenia programisty wszelkie możliwe odmiany systemu operacyjnego Windows, choć bardzo różne "w środku", za zewnątrz wyglądają tak samo. Jeśli jakaś funkcja występuje we wszystkich odmianach systemu, to jej działanie jest identyczne, choć mechanizmy jakie pociągają za sobą wywołanie takiej funkcji w systemie operacyjnym mogą być zupełnie różne<sup>4</sup>.

Od pierwszej wersji systemu Windows, jego interfejs pozostaje w miarę jednolity, mimo że w międzyczasie przeszedł ewolucję i z systemu 16-bitowego stał się systemem 32-bitowym. Zasadniczo zmienił się sposób adresowania pamięci (w modelu 16-bitowym odwołania do pamięci miały postać *segment:offset* i były następnie tłumaczone na adresy fizyczne, model 32-bitowy zakłada 32-bitowe liniowe adresowanie pamięci, wykorzystujące odpowiednie możliwości procesorów 80386 i wyższych). Mimo tej zmiany interfejs programowania pozostał w dużej części nienaruszony. Wszystkie, nawet najnowsze, wersje systemu, pozwalają na korzystanie zarówno z nowego (**Win32**) jak i starego (**Win16**) interfejsu. Warto wiedzieć, że w systemach opartych na jądrze NT wywołania funkcji z Win16API przechodzą przez pośrednią warstwę tłumaczącą je na funkcje Win32API obsługiwane następnie przez system, zaś w systemach opartych na jądrze 16-bitowym (Windows 95, Windows 98) jest dokładnie odwrotnie - to funkcje z Win32API przechodzą przez warstwę tłumaczącą je na Win16API, które to z kolei funkcje są obsługiwane przez system operacyjny. Przyjmuje się że obie linie systemów wspierają Win32API, jednak sytuacja nie jest aż tak różowa - każdy z systemów obsługuje swój własny podzbiór Win32API. Część wspólna jest jednak na tyle pojemna, że jak już wcześniej wspomniano, możliwe jest pisanie programów, które działają na każdej odmianie systemu Windows.

W pierwszej wersji systemu do dyspozycji programistów oddano około 450 funkcji. W ostatnich wersjach ich liczba znacząco wzrosła (mówi się o tysiącach funkcji), głównie dlatego, że

<sup>3</sup>Taka konstrukcja oprogramowania, w której wewnętrzne mechanizmy funkcjonowania jakiegoś fragmentu oprogramowania są ukryte, zaś dostęp do jego funkcji jest możliwy za pomocą jakiegoś interfejsu, jest powszechnie stosowany w nowoczesnym oprogramowaniu. Istnieją setki specjalizowanych interfejsów programowania przeróżnych bibliotek (DirectX, OpenGL), protokołów (sieć, ODBC, OLEDB), czy programów (MySQL).

<sup>4</sup>Na przykład funkcje do operacji na systemie plików czy rejestrze systemu w systemach opartych na jądrze NT muszą dodatkowo wykonać pracę związaną ze sprawdzaniem przywilejów użytkownika.



Rysunek 1.1: DevC++ pozwala pisać programy w C i wspiera Win32API.

znacząco wzrosła liczba możliwości jakimi nowe odmiany systemu dysponują. Każda kolejna warstwa, zbudowana nad Win32API, musi z konieczności być w jakiś sposób ograniczona. MFC, VCL, QT, GTK czy środowisko uruchomieniowe .NET Framework nie są tu wyjątkami: zdarzają się sytuacje, kiedy zachodzi konieczność sięgnięcia "głębiej" niż pozwalają na to wymienione interfejsy, aż do poziomu Win32API. Zrozumienie zasad Win32API pozwala więc przewyżczać ograniczenia interfejsów wyższego poziomu<sup>5</sup>. Pełna dokumentacja wszystkich funkcji systemowych dostępnych we wszystkich interfejsach zaprojektowanych przez Microsoft oraz mnóstwo artykułów z poradami na temat programowania pod Windows dostępna jest on-line pod adresem <http://msdn.microsoft.com>.

### 1.3 Narzędzia programistyczne

Repertuar języków programowania, które pozwalają na pisanie programów pod Windows jest bogaty i każdy znajdzie tu coś dla siebie. Win32API przygotowano jednak z myślą o języku C i to właśnie pisząc programy w języku C można od systemu Windows otrzymać najwięcej. Programiści mają do wyboru nie tylko *Microsoft Visual C++*, który jest częścią *Visual Studio*, ale także kilka niezłych darmowych kompilatorów rozpowszechnianych na licencji GNU (wśród nich wyróżnia się DevC++, do pobrania ze strony <http://www.bloodshed.net>).

Dużą popularność zdobył sobie język *Delphi* zaprojektowany przez firmę *Borland* jako rozszerzenie *Pascala*. Wydaje się jednak, że znaczenie tego języka będzie coraz mniejsze. Marginalizuje się również znaczenie wielu innych interfejsów takich jak MFC czy VCL.

Pojawienie się języka Java, zaprojektowanego przez firmę Sun, oznaczało dla społeczności programistów nową epokę. Projektantom Javy przyświecała idea *Jeden język - wiele platform*, zgodnie z którą programy napisane w Javie miały być przenośne między różnymi systemami operacyjnymi. W praktyce okazało się, że Java nie nadaje się do pisania dużych aplikacji, osadzonych

<sup>5</sup>Tak będziemy mówić o interfejsach zbudowanych na Win32API

w konkretnych systemach operacyjnych. Na przykład oprogramowanie interfejsu użytkownika w Javie polega na skorzystaniu z komponentów specyficznych dla Javy, nie zaś dla konkretnego systemu operacyjnego. Odpowiadając na zarzuty programistów o ignorowanie istnienia w systemach operacyjnych specjalizowanych komponentów, Microsoft przygotował swoją wersję Javy, którą wyposażył w bibliotekę WFC (*Windows Foundation Classes*), związującą *Visual J++* z platformą Windows. W 1997 Sun wytoczył Microsoftowi proces, który ostatecznie doprowadził do zaniechania przez Microsoft rozwijania *J++* i podjęcia pracy nad nowym językiem, pozbawionym wad Javy, który osadzony byłby na nowej platformie, pozbawionej wad środowiska uruchomieniowego Javy. Prace te zaowocowały pojawieniem się w okoliach roku 2000 pierwszych testowych wersji środowiska uruchomieniowego, nazwanego .NET Framework, dla którego zaprojektowano nowy język nazwany C#. Dla wielu programistów używających Javy jedną z kropel w kielichu goryczy jest niezgodność semantyczna zachowania się maszyn wirtualnych pochodzących z różnych źródeł<sup>6</sup>.

.NET Framework opiera się na idei odwrotnej niż Java. Ta idea to *Jedna platforma - wiele języków*. Specyfikacja języka pośredniego, nazwanego *IL* (*Intermediate Language*) jest otwarta dla wszystkich twórców kompilatorów. Co otrzymują w zamian? Wspólny system typów, pozwalający na komunikację programów pochodzących z różnych języków, rozbudowaną bibliotekę funkcji, wspólny mechanizm obsługi wyjątków oraz odśmiecacz. Ze swojej strony Microsoft przygotował 5 języków programowania platformy .NET. Są to:

- C#, w pełni obiektowy język programowania o składni C-podobnej
- J++, Java dla platformy .NET
- C++, który w nowej wersji potrafi korzystać z dobrodziejstw platformy .NET
- VB.NET, nowa wersja Visual Basica o znacznie większych możliwościach niż poprzednia wersja
- IL Assembler, niskopoziomowy język programowania w kodzie pośrednim platformy .NET

Poza Microsoftem pojawiają się kompilatory innych języków dla platformy .NET. W tej chwili dostępne są m.in.:

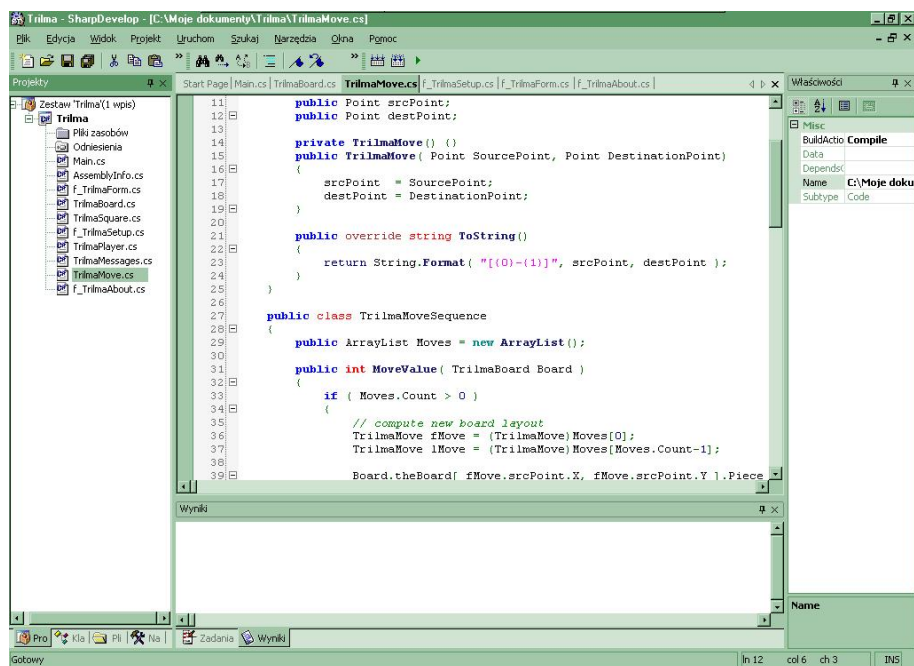
- Ada
- COBOL
- Perl
- Python
- SmallTalk
- SML.NET

Trwają prace nad NET ową wersją Prologa, Delphi oraz wielu innych języków.

Kompilatory dla trzech języków (C#, VB.NET, IL Assembler) wchodzi w skład środowiska uruchomieniowego .NET Framework, czyli są **darmowe**. Również bez wnoszenia opłat można pobrać ze stron Microsoftu pakiet dla J++. Sam .NET Framework można pobrać również bezpłatnie ze strony <http://msdn.microsoft.com/netframework/downloads/howtoget.asp>. Pakiet instalacyjny zajmuje około 20MB. Programiści mogą pobrać *.NET Framework SDK*, który oprócz

---

<sup>6</sup>Zdarza się również, że maszyny wirtualne tego samego producenta zachowują się inaczej na różnych systemach operacyjnych



Rysunek 1.2: SharpDevelop oferuje m.in. autouzupełnianie kodu i wizualny edytor form.

środowiska uruchomieniowego zawiera setki przykładów i tysiące stron dokumentacji technicznej. *.NET Framework SDK* to około 120MB. Samo środowisko uruchomieniowe można zainstalować na systemach Windows począwszy od Windows 98. .NET Framework SDK, podobnie jak Visual Studio .NET wymagają już co najmniej Windows 2000, jednak rozwijane w Windows 2000 programy dadzą się oczywiście uruchomić w Windows 98 z zainstalowanym środowiskiem uruchomieniowym .NET (pod warunkiem nie wykorzystywania klas specyficznych dla Windows 2000, np. *FileSystemWatcher*).

Do dyspozycji programistów oddano oczywiście nową wersję środowiska developerskiego *Visual Studio .NET* (oczywiście ono nie jest już darmowe). Dostępne są za to środowiska darmowe, rozwijane poza Microsoftem. Najlepiej zapowiada się *SharpDevelop* (do pobrania ze strony <http://www.icsharpcode.net>).

Specyfikacja platformy .NET jest publiczna, ogłoszona poprzez ECMA-International (*European Computer Manufacturer Association International*, <http://www.ecma-international.org>), nic więc dziwnego, że powstają wersje pod inne niż Windows systemy operacyjne. Najbardziej zaawansowany jest w tej chwili projekt Mono (<http://www.go-mono.com>), dostępny na kilka systemów operacyjnych (w tym Linux i Windows).

Platforma .NET jest dobrze udokumentowana, powstają coraz to nowe strony, gdzie developerzy dzielą się przykładowymi kodami i wskazówkami. Warto zaglądać na <http://msdn.microsoft.com>, <http://www.c-sharpcorner.com>, <http://www.gotdotnet.com> czy <http://www.codeproject.com>.





## Rozdział 2

# Win32 API

### 2.1 Fundamentalne idee Win32API

Interfejs programowania Win32API można podzielić na spójne podzbiory funkcji przeznaczonych do podobnych celów. Dokumentacja systemu mówi o 6 kategoriach:

**Usługi podstawowe** Ta grupa funkcji pozwala aplikacjom na korzystanie z takich możliwości systemu operacyjnego jak zarządzanie pamięcią, obsługa systemu plików i urządzeń zewnętrznych, zarządzanie procesami i wątkami.

**Biblioteka Common Controls** Ta część Win32API pozwala obsługiwać zachowanie typowych okien potomnych, takich jak proste pola edycji i comboboxy czy skomplikowane ListView i TreeView.

**GDI** GDI (*Graphics Device Interface*) dostarcza funkcji i struktur danych, które mogą być wykorzystane do tworzenia efektów graficznych na urządzeniach wyjściowych takich jak monitory czy drukarki. GDI pozwala rysować kształty takie jak linie, krzywe oraz figury zamknięte, pozwala także na rysowanie tekstu.

**Usługi sieciowe** Za pomocą tej grupy funkcji można obsługiwać warstwę komunikacji sieciowej, na przykład tworzyć współdzielone zasoby sieciowe czy diagnozować stan konfiguracji sieciowej.

**Interfejs użytkownika** Ta grupa funkcji dostarcza środków do tworzenia i zarządzania interfejsem użytkownika: tworzenia okien i interakcji z użytkownikiem. Zachowanie i wygląd tworzonych okien jest uzależnione od właściwości tzw. *klas okien*.

**Powłoka systemu** To funkcje pozwalające aplikacjom integrować się z powłoką systemu, na przykład uruchomić dany dokument ze skojarzoną z nim aplikacją, dowiadywać się o ikony skojarzone z plikami i folderami czy odczytywać położenie ważnych folderów systemowych.

Programowanie systemu Windows wymaga przyswojenia sobie trzech istotnych elementów.

Po pierwsze - wszystkie elementy interfejsu użytkownika, pola tekstowe, przyciski, comboboxy, radiobuttony<sup>1</sup>, wszystkie one z punktu widzenia systemu są oknami. Jak zobaczymy, Windows traktuje wszystkie te elementy w sposób jednorodny, przy czym niektóre okna mogą być tzw. *oknami potomnymi* innych okien. Windows traktuje okna potomne w sposób szczególny,

---

<sup>1</sup>'Angielskawe' brzmienie tych terminów może być trochę niezręczne, jednak ich polskie odpowiedniki bywają przerażające. Pozostaniemy więc przy terminach powszechnych wśród programistów.

zawsze umieszczając je w obszarze okna macierzystego oraz automatycznie przesuwając je, gdy użytkownik przesuwa okno macierzyste<sup>2</sup>.

Po drugie - z perspektywy programisty wszystkie okna zachowują się prawie dokładnie tak samo jak z perspektywy użytkownika. Użytkownik, za pomocą myszy, klawiatury lub innego wskaźnika, wykonuje różne operacje na widocznych na pulpicie oknach. Każde zdarzenie w systemie, bez względu na źródło jego pochodzenia, powoduje powstanie tzw. komunikatu, czyli pewnej informacji mającej swój cel i niosącej jakąś określoną informację. Programista w kodzie swojego programu tak naprawdę zajmuje się obsługiwaniem komunikatów, które powstają w systemie przez interakcję użytkownika<sup>3</sup>.

Po trzecie - do identyfikacji obiektów w systemie, takich jak okna, obiekty GDI, pliki, biblioteki, wątki itd., Windows korzysta z tzw. *uchwyty* (czyli 32-bitowych identyfikatorów). Mnóstwo funkcji Win32API przyjmuje jako jeden z parametrów uchwyt (czyli identyfikator) obiektu systemowego, przez co wykonanie takiej funkcji odnosi się do wskazanego przez ten uchwyt obiektu. W języku C różne uchwyty zostały różnie nazwane (HWND, HDC, HPEN, HBRUSH, HICON, HANDLE itd.) choć tak naprawdę są one najczęściej wskaźnikami na miejsce w pamięci gdzie znajduje się pełny opis danego obiektu. Z perspektywy programisty, są one, jak już powiedziano, unikatowymi identyfikatorami obiektów systemowych.

Dokładne poznanie i zrozumienie trzech wymienionych wyżej elementów stanowi istotę poznania i zrozumienia Win32API. Idee które leżą u podstaw wyżej wymienionych elementów są jednakowe we wszystkich wersjach systemu Windows i z dużą dozą prawdopodobieństwa można powiedzieć, że nie ulegną zasadniczym zmianom w kolejnych wersjach systemu. Programista może oczywiście znać mniej lub więcej funkcji Win32API, umieć posługiwać się mniejszą lub większą ilością komunikatów, znać mniej lub więcej typów uchwytów, jednak bez zrozumienia zasad, wedle jakich wszystkie te elementy składają się na funkcjonowanie systemu operacyjnego Windows, programista pisząc program będzie często bezradny.

## 2.2 Okna

### 2.2.1 Tworzenie okien

Zarządzanie oknami i tworzenie grafiki to jedno z najważniejszych zadań przy programowaniu pod Windows, wymagające bardzo dokładnego poznania. Interfejs użytkownika jest pierwszym elementem programu, z jakim styka się użytkownik, co więcej - interfejs jest tym elementem, któremu użytkownik zwykle poświęca najwięcej czasu i uwagi. Programista musi więc bardzo dokładnie poznać możliwości jakimi dysponuje w tym zakresie system operacyjny.

Przeanalizujemy bardzo prosty programi Windowsowy, który na pulpicie pokaże okno.

```
/*
 *
 * Tworzenie okna aplikacji
 *
 */
#include <windows.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

int WINAPI WinMain(HINSTANCE hInstance,
```

<sup>2</sup>To dość ważne. Gdyby programista musiał dbać o przesuwanie się okien potomnych za przesuwanym się oknem macierzystym, byłoby to niesłychanie niewygodne.

<sup>3</sup>I nie tylko - komunikaty mogą mieć swoje źródło w samym systemie. Komunikaty wysyłają do siebie na przykład okna i okna potomne, źródłem komunikatów mogą być zegary itd.

```

        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nShowCmd)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;         /* Komunikaty okna */
    WNDCLASSEX wincl;     /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance = hInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure; // wskaźnik na funkcję obsługi okna
    wincl.style = CS_DBLCLKS;
    wincl.cbSize = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;
    wincl.cbClsExtra = 0;
    wincl.cbWndExtra = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0, szClassName,
        "Przykład",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        512, 512,
        HWND_DESKTOP, NULL,
        hInstance, NULL );

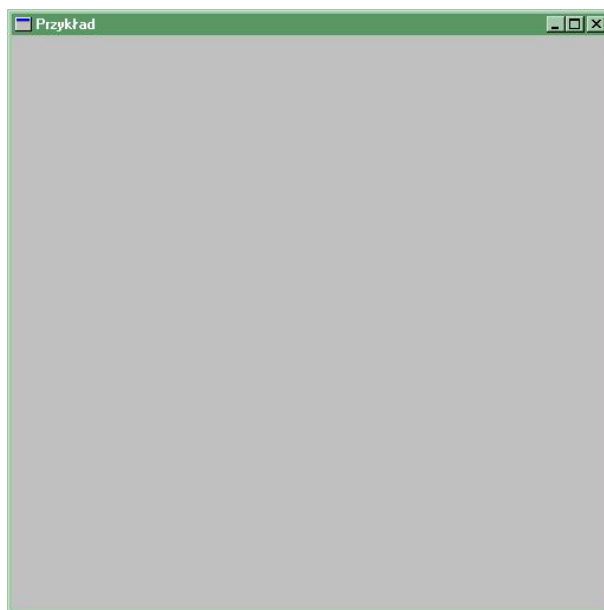
    ShowWindow(hwnd, nShowCmd);
    /* Pętla obsługi komunikatów */
    while(GetMessage(&messages, NULL, 0, 0))
    {
        /* Tłumacz kody rozszerzone */
        TranslateMessage(&messages);
        /* Obsłuż komunikat */
        DispatchMessage(&messages);
    }

    /* Zwróć parametr podany w PostQuitMessage( ) */
    return messages.wParam;
}

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

```

Z punktu widzenia syntaktyki - jest to zwykły program w języku C. Być może rozczarowujące jest to, że program ten jest aż tak długi. Okazuje się jednak, że prościej się po prostu nie da.



Rysunek 2.1: Efekt działania pierwszego przykładowego programu

Jeżeli w jakimkolwiek innym języku programowania lub przy użyciu jakichś bibliotek da się napisać prostszy program tworzący okno (a jak zobaczymy w rozdziale 3.6.1 analogiczny program w C# zajmuje mniej więcej 10 linii kodu), będzie to zawsze oznaczało, że część kodu jest po prostu ukryta przed programistą.

Z tego właśnie powodu mówimy, że interfejs Win32API jest "najbliżej" systemu operacyjnego jak tylko jest to możliwe (czasem mówi się też, że jest on "najniższym" interfejsem programowania). Każda inna biblioteka umożliwiająca tworzenie okien **musi** korzystać z funkcji Win32API, opakowując je ewentualnie w jakiś własny interfejs programowania.

Wielu programistów znających bardzo dobrze Win32API uważa to za jego największą zaletę. To właśnie bowiem Win32API daje największą kontrolę nad tym jak wygląda okno i jak się zachowuje.

Ale wróćmy do naszego programu. Pierwsza ważna różnica między programem Windowsowym a zwykłym programem w języku C, to brak funkcji **main**, zastąpionej przez **WinMain**. Tradycyjnie funkcja ta ma następujący prototyp:

```
int
WINAPI
WinMain(

    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nShowCmd
);
```

W tej deklaracji

- **WINAPI** oznacza konwencję przekazywania parametrów do funkcji. Zwykle w którymś z plików nagłówkowych znajdziemy po prostu **#define WINAPI \_\_stdcall**<sup>4</sup>

<sup>4</sup>O innych konwencjach przekazywania parametrów do funkcji (**\_stdcall**, **\_cdecl**, **\_pascal**) warto poczytać, ponieważ niezgodność konwencji bywa źródłem problemów przy łączeniu bibliotek napisanych w różnych językach, np. Delphi i Visual Basicu.

- **hInstance**, jak sugeruje typ, jest uchwytem. W tym przypadku jest to uchwyt do bieżącej instancji aplikacji.
- **hPrevInstance** to uchwyt do poprzedniej instancji tej aplikacji. W Win16API za pomocą tego uchwytu można było zidentyfikować istniejącą już w systemie instancję aplikacji i uaktywnić ją w razie potrzeby. W Win32API ten parametr jest zawsze równy NULL i zachowano go tylko ze względów historycznych. Do identyfikowania innych instancji aplikacji w Win32API należy użyć jakichś trwałych obiektów, na przykład **Mutexów**<sup>5</sup>.
- **lpCmdLine** to lista parametrów programu. W programie Windowsowym, w przeciwieństwie do zwykłego programu w języku C, wszystkie parametry przekazywane są w tej jednej tablicy. Oznacza to, że programista musi sam zatroszczyć się o wyłowienie kolejnych parametrów z listy. Inaczej też niż w zwykłym programie w C można uzyskać informację o lokalizacji bieżącej aplikacji w systemie plików: zamiast odczytać zerowy parametr na liście parametrów, programista woła funkcję API **GetModuleFileName**.
- Windows może aktywować okno na różne sposoby, m.in.:
  - SW\_HIDE, ukrywa okno
  - SW\_MINIMIZE, okno jest zminimalizowane
  - SW\_RESTORE, SW\_SHOWNORMAL, aktywuje okno w jego oryginalnych rozmiarach
  - SW\_SHOW, aktywuje okno w jego bieżących rozmiarach
  - SW\_SHOWMAXIMIZED, okno jest zmaksymalizowane

**nShowCmd** sugeruje aplikacji sposób pokazania głównego okna. Programista może oczywiście tę informację zlekceważyć, jednak nie jest to dobrą praktyką.

Druga ważna różnica między programem Windowsowym a zwykłym programem w języku C, to mnóstwo nowych funkcji i struktur od jakich roi się w programie Windowsowym. Zauważmy, że samo utworzenie okna jest procesem o tyle skomplikowanym, że wymaga wcześniej utworzenia tzw. *klasy okna*. Chodzi o to, by wszystkie okna o podobnych właściwościach mogły mieć tę samą funkcję obsługi komunikatów (o komunikatach za chwilę). Na przykład wszystkie przyciski są oknami utworzonymi na bazie klasy **BUTTON**, wskazującej na odpowiednią funkcję obsługi zachowań przycisku. Aplikacja może tworzyć dowolną ilość okien bazujących na tej samej klasie, za każdym razem konkretyzując pewne dodatkowe cechy każdego nowego okna.

Aby zarejestrować w systemie nową klasę okna należy skorzystać z funkcji

```
ATOM RegisterClassEx(
    CONST WNDCLASSEX *lpwcx
);
```

Klasa okna utworzona przez aplikację jest automatycznie wyrejestrowywana przy zakończeniu aplikacji. Okna tworzy się za pomocą funkcji

```
HWND CreateWindowEx(
    DWORD dwExStyle, // rozszerzony styl okna
    LPCTSTR lpClassName, // nazwa klasy okna
    LPCTSTR lpWindowName, // nazwa okna
    DWORD dwStyle, // styl okna
```

---

<sup>5</sup>Więcej o Mutexach na stronie 44

```

int x, // pozycja okna
int y,
int nWidth, // szerokość
int nHeight, // wysokość
HWND hWndParent, // uchwyt okna macierzystego
HMENU hMenu, // uchwyt menu lub identyfikator okna potomnego
HINSTANCE hInstance, // instancja aplikacji
LPVOID lpParam
)

```

Zapamiętajmy przy okazji prawidłowość: wiele funkcji API istnieje w dwóch wariantach, podstawowym i rozszerzonym. Bardzo często funkcje podstawowe oczekują pewnej ściśle określonej ilości parametrów, natomiast funkcje rozszerzone oczekują jednego parametru, którym jest struktura z odpowiednio wypełnionymi polami<sup>6</sup>.

## 2.2.2 Komunikaty

W przykładzie z poprzedniego rozdziału widzieliśmy, że funkcja obsługi okna zajmuje się obsługą komunikatów docierających do okna. Komunikaty pełnią w systemie Windows główną rolę jako środek komunikacji między różnymi obiektami. Jeżeli gdziekolwiek w systemie dzieje się coś, co wymaga poinformowania jakiegoś innego obiektu, najprawdopodobniej ta informacja przepłynie w postaci komunikatu.

Obsługą komunikatów, ich rozdzielaniem do odpowiednich obiektów zajmuje się jądro systemu. W praktyce każde okno ma swoją własną *kolejkę komunikatów*, w której system umieszcza kolejne komunikaty, które mają swoje źródło gdzieś w systemie, a ich przeznaczeniem jest dane okno.

Programista może kazać oknu przechwytywać odpowiednie komunikaty, może również inicjować komunikaty i kierować je do wybranych okien. W funkcji obsługi komunikatów programista sam decyduje o tym, na które komunikaty okno powinno reagować. Najczęściej są to komunikaty typowe. Programista nie ma obowiązku reagować na wszystkie możliwe komunikaty.

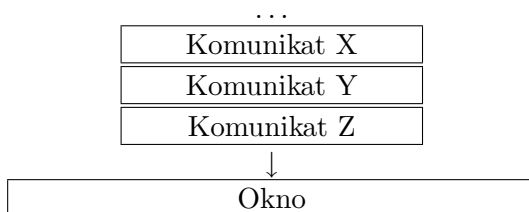


Tabela 2.1: Z każdym oknem system kojarzy kolejkę komunikatów dla niego przeznaczonych

Oto lista ważniejszych komunikatów, jakie mogą docierać do okna.

**WM\_CHAR** Dociera do aktywnego okna po tym, jak komunikat WM\_KEYDOWN zostanie przetłumaczony w funkcji TranslateMessage().

**chCharCode = (TCHAR) wParam;** Znakowy kod wciśniętego klawisza.

**lKeyData = lParam;** Ilość powtórzeń, kody rozszerzone.

**WM\_CLOSE** Dociera do aktywnego okna przed jego zamknięciem. Jest to chwila kiedy można jeszcze anulować zamknięcie okna.

<sup>6</sup>Nie jest to jednak regułą

**WM\_COMMAND** Dociera do aktywnego okna przy wyborze pozycji z menu lub jako powiadomienie od okna potomnego.

**wNotifyCode = HIWORD(wParam);** Kod powiadomienia.

**wID = LOWORD(wParam);** Identyfikator pozycja menu lub okna potomnego.

**hwndCtl = (HWND) lParam;** Uchwyt okna potomnego.

**WM\_CREATE** Dociera do okna po jego utworzeniu za pomocą `CreateWindow()` ale przed jego pierwszym pojawieniem się. Jest zwykle wykorzystywany na tworzenie okien potomnych, inicjowanie menu czy inicjowanie podsystemów OpenGL, DirectX itp.

**lpcs = (LPCREATESTRUCT) lParam;** Informacje o utworzonym oknie.

```
typedef struct tagCREATESTRUCT { // cs
    LPVOID    lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCTSTR   lpzName;
    LPCTSTR   lpzClass;
    DWORD     dwExStyle;
} CREATESTRUCT;
```

**WM\_KEYDOWN** Dociera do aktywnego okna gdy zostanie naciśnięty klawisz *niesystemowy* (czyli dowolny klawisz bez wciśniętego klawisza ALT).

**nVirtKey = (int) wParam;** Kod klawisza.

**lKeyData = lParam;** Ilość powtórzeń, kody rozszerzone.

**WM\_KEYUP** Dociera do aktywnego okna gdy zostanie zwolniony klawisz *niesystemowy* (czyli dowolny klawisz bez wciśniętego klawisza ALT).

**nVirtKey = (int) wParam;** Kod klawisza.

**lKeyData = lParam;** Ilość powtórzeń, kody rozszerzone.

**WM\_KILLFOCUS** Dociera do aktywnego okna przed przekazaniem aktywności innemu oknu.

**hwndGetFocus = (HWND) wParam;** Uchwyt okna, które stanie się aktywne.

**lKeyData = lParam;** Ilość powtórzeń, kody rozszerzone.

**WM\_LBUTTONDOWN** Dociera do aktywnego okna gdy jego obszar zostanie dwukliknięty.

**fwKeys = wParam;** Informuje o tym, czy jednocześnie są wciśnięte klawisze systemowe: SHIFT, CTRL.

**xPos = LOWORD(lParam);** Współrzędna X dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**yPos = HIWORD(lParam);** Współrzędna Y dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**WM\_LBUTTONDOWN** Dociera do aktywnego okna gdy jego obszar zostanie kliknięty za pomocą lewego przycisku.

**fwKeys = wParam;** Informuje o tym, czy jednocześnie są wciśnięte klawisze systemowe: SHIFT, CTRL.

**xPos = LOWORD(lParam);** Współrzędna X dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**yPos = HIWORD(lParam);** Współrzędna Y dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**WM\_LBUTTONUP** Dociera do aktywnego okna gdy użytkownik zwalnia lewy przycisk myszy, a wskaźnik znajduje się nad obszarem klienckim okna.

**fwKeys = wParam;** Informuje o tym, czy jednocześnie są wciśnięte klawisze systemowe: SHIFT, CTRL.

**xPos = LOWORD(lParam);** Współrzędna X dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**yPos = HIWORD(lParam);** Współrzędna Y dwuklikniętego punktu względem punktu w lewym górnym rogu obszaru klienckiego okna.

**WM\_MOVE** Dociera do okna po tym jak zmieniło się jego położenie.

**xPos = LOWORD(lParam);** Nowa współrzędna X okna.

**yPos = HIWORD(lParam);** Nowa współrzędna Y okna.

**WM\_PAINT** Dociera do okna gdy jego obszar kliencki wymaga odrysowania. Więcej o tym komunikacie na stronie 34.

**WM\_SIZE** Dociera do okna, gdy zmienił się jego rozmiar.

**nWidth = LOWORD(lParam);** Nowa szerokość okna.

**nHeight = HIWORD(lParam);** Nowa wysokość okna.

**WM\_QUIT** Powoduje zakończenie pętli komunikatów i tym samym zakończenie aplikacji.

**nExitCode = (int) wParam;** Kod zakończenia.

**WM\_SYSCOLORCHANGE** Dociera do wszystkich okien po tym, gdy zmienią się ustawienia kolorów pulpitu.

**WM\_TIMER** Dociera do aktywnego okna od ustawionego przez aplikację zegara. Więcej o zegarach na stronie 59.

**wTimerID = wParam;** Identyfikator zegara.

**tmprc = (TIMERPROC \*) lParam;** Adres funkcji obsługi zdarzenia.

**WM\_USER** Pozwala użytkownikowi definiować własne komunikaty. Użytkownik tworzy komunikat za pomocą funkcji



```

UINT RegisterWindowMessage(

    LPCTSTR lpString
);

```

Zaproponowana w przykładzie konstrukcja pętli obsługi komunikatów jest bardzo charakterystyczna.

```

/* Pętla obsługi komunikatów */
while(GetMessage(&messages, NULL, 0, 0))
{
    /* Tłumacz kody rozszerzone */
    TranslateMessage(&messages);
    /* Obsłuż komunikat */
    DispatchMessage(&messages);
}

```

Funkcja **GetMessage** czeka na pojawienie się komunikatu w kolejce komunikatów, zaś **DispatchMessage** wysyła komunikat do funkcji obsługi komunikatów.

Funkcja **GetMessage** jest jednak funkcją *blokującą*, to znaczy że wykonanie programu zostanie wstrzymane na tak długo, aż jakaś wiadomość pojawi się w kolejce komunikatów okna aplikacji. Najczęściej aplikacja wstrzymywana jest na kilka czy kilkanaście milisekund, bowiem komunikaty napływają do okna dość często, oznacza to jednak, że część cennego czasu aplikacja marnuje na biernym oczekiwaniu na komunikaty.

Takie zachowanie nie byłoby wskazane dla aplikacji, która miałaby działać w sposób ciągły, na przykład tworząc grafikę czy inne efekty w czasie rzeczywistym. Rozwiązaniem jest zastosowanie innej postaci pętli obsługi komunikatów, alternatywnej dla pokazanej powyżej, wykorzystującej nieblokującą funkcję **PeekMessage**, która po prostu sprawdza czy w kolejce komunikatów jest jakiś komunikat, a jeśli nie - oddaje sterowanie do pętli obsługi komunikatów. Wybór pomiędzy oboma funkcjami (a co za tym idzie - między dwoma możliwościami konstrukcji pętli obsługi komunikatów) należy do programisty.

```

/* Pętla obsługi komunikatów */
while (TRUE)
{
    /* Sprawdź czy są jakieś komunikaty do obsłużenia */
    if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
    {
        if (msg.message == WM_QUIT)
            break ;

        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    else
    {
        // "czas wolny" aplikacji do wykorzystania do innych celów
        // niż obsługa komunikatów -
    }
}

```

## 2.2.3 Okna potomne

### 2.2.3.1 Tworzenie okien potomnych

Główne okno aplikacji, jak również każde kolejne okno z którym styka się użytkownik, zwykle posiada jakieś okna potomne (zwane inaczej *kontrolkami*), za pomocą których użytkownik mógłby komunikować się z aplikacją.

Dwa najprostsze rodzaje okien potomnych to pole tekstowe i przycisk. Okazuje się jednak, że klasa okna (na przykład klasa **BUTTON** definiująca przyciski), tak naprawdę definiuje nie

*jeden* typ okna potomnego, ale całą rodzinę okien potomnych, różniących się właściwościami. Odpowiedni styl okna podaje się jako jeden z parametrów do funkcji **CreateWindow**.

Zobaczmy prosty przykład tworzenia okien potomnych o różnych stylach:

```
/*
 *
 * Tworzenie okien potomnych
 *
 */
#include <windows.h>
#include <string.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

struct
{
    TCHAR * szClass;
    int     iStyle ;
    TCHAR * szText ;
} button[] =
{
    "BUTTON" , BS_PUSHBUTTON      , "PUSHBUTTON",
    "BUTTON" , BS_AUTOCHECKBOX    , "CHECKBOX",
    "BUTTON" , BS_RADIOBUTTON     , "RADIOBUTTON",
    "BUTTON" , BS_GROUPBOX        , "GROUPBOX",
    "EDIT"   , WS_BORDER          , "TEXTBOX",
    "STATIC" , WS_BORDER          , "STATIC",
};

#define NUM (sizeof button / sizeof button[0])

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;         /* Komunikaty okna */
    WNDCLASSEX wincl;     /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc    = WindowProcedure;  // wskaźnik na funkcję obsługi okna
    wincl.style          = CS_DBLCLKS;
    wincl.cbSize         = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName   = NULL;
    wincl.cbClsExtra     = 0;
    wincl.cbWndExtra     = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0,
        szClassName,
        "PRZYKLAD",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
```

```

        CW_USEDEFAULT,
        HWND_DESKTOP,
        NULL,
        hThisInstance,
        NULL
    );

    ShowWindow(hwnd, nFunsterStil);
    /* Pętla obsługi komunikatów */
    while(GetMessage(&messages, NULL, 0, 0))
    {
        /* Tłumacz kody rozszerzone */
        TranslateMessage(&messages);
        /* Obsługa komunikatów */
        DispatchMessage(&messages);
    }

    /* Zwróć parametr podany w PostQuitMessage( ) */
    return messages.wParam;
}

int xSize, ySize;

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                  WPARAM wParam, LPARAM lParam)
{
    static HWND hwndButton[NUM];
    static int  cxChar, cyChar;
    static RECT r;
    HDC        hdc;
    int         i;
    PAINTSTRUCT ps;

    TCHAR       szFormat[] = TEXT ("%16s Akcja: %04X, ID:%04X, hWnd:%08X");
    TCHAR       szBuffer[80];

    switch (message)
    {
        case WM_CREATE :
            cxChar = LOWORD (GetDialogBaseUnits ()) ;
            cyChar = HIWORD (GetDialogBaseUnits ()) ;

            for (i = 0 ; i < NUM ; i++)
                hwndButton[i] = CreateWindow ( button[i].szClass,
                                                button[i].szText,
                                                WS_CHILD | WS_VISIBLE | button[i].iStyle,
                                                cxChar, cyChar * (1 + 2 * i),
                                                20 * cxChar, 7 * cyChar / 4,
                                                hwnd, (HMENU) i,
                                                ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;

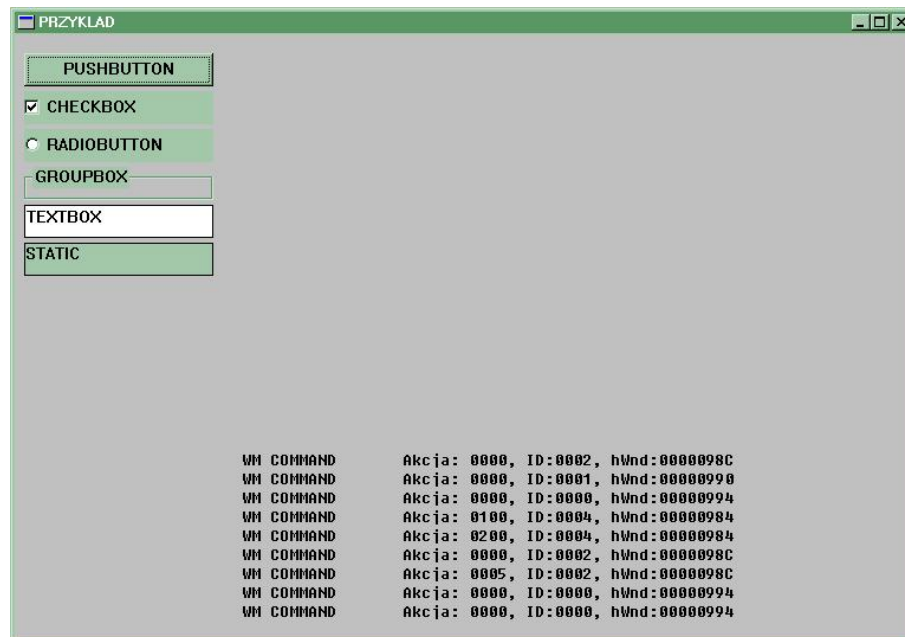
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_SIZE:
            xSize = LOWORD(lParam);
            ySize = HIWORD(lParam);

            r.left   = 24 * cxChar ;
            r.top    = 2 * cyChar ;
            r.right  = LOWORD (lParam) ;
            r.bottom = HIWORD (lParam) ;

            break;
        case WM_COMMAND:
            hdc = GetDC (hwnd);

            ScrollWindow (hwnd, 0, -cyChar, &r, &r) ;
    }
}

```



Rysunek 2.2: Okna potomne komunikują się z oknem macierzystym za pomocą powiadomień

```

SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

SetBkMode (hdc, TRANSPARENT) ;
TextOut (hdc, 24 * cxChar, cyChar * (r.bottom / cyChar - 1),
        szBuffer,
        wprintf (szBuffer, szFormat,
        "WM_COMMAND",
        HIWORD (wParam), LOWORD (wParam), lParam ));

ReleaseDC( hwnd, hdc );
return DefWindowProc(hwnd, message, wParam, lParam);
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps);
    EndPaint( hwnd, &ps );
    break;

default:
    return DefWindowProc(hwnd, message, wParam, lParam);
}
return 0;
}

```

### 2.2.3.2 Aktywowanie i deaktywowanie okien potomnych

Programista może w każdej chwili uaktywnić bądź deaktywować okno<sup>7</sup> za pomocą funkcji

```

BOOL EnableWindow(
    HWND hWnd,          // uchwyt okna
    BOOL bEnable // aktywacja bądź deaktywacja
);

```

<sup>7</sup>Okno potomne, które jest nieaktywne zwykle ma szary kolor i nie przyjmuje fokusa.

### 2.2.3.3 Komunikacja między oknem potomnym a macierzystym

Komunikacja między oknem potomnym a oknem macierzystym odbywa się za pomocą komunikatów przesyłanych między nimi. Komunikaty te pojawiają się w oknie macierzystym jako WM\_COMMAND z dodatkowymi informacjami na temat powiadomienia od okna potomnego.

Spójrzmy przykładowo na powiadomienia, jakie oknu macierzystemu przysyła przycisk:

- BN\_CLICKED : 0, przycisk został naciśnięty
- BN\_PAINT : 1, przycisk powinien zostać narysowany
- BN\_PUSHED : 2, przycisk został wciśnięty
- BN\_UNPUSHED : 3, przycisk został wyciśnięty
- BN\_DISABLE : 4, przycisk został deaktywowany
- BN\_DBLCLK : 5, przycisk został podwójnie naciśnięty
- BN\_SETFOCUS : 6, przycisk otrzymał fokusa
- BN\_KILLFOCUS : 7, przycisk stracił fokusa

Pole tekstowe przysyła oknu macierzystemu następujące powiadomienia:

- EN\_SETFOCUS : 0x100, Pole tekstowe otrzymało fokusa
- EN\_KILLFOCUS : 0x200, Pole tekstowe straciło fokusa
- EN\_CHANGE : 0x300, Pole tekstowe zmieni zawartość
- EN\_UPDATE : 0x400, Pole tekstowe zmieniło zawartość
- EN\_ERRSPACE : 0x500, Pole tekstowe nie może zaalokować pamięci
- EN\_MAXTEXT : 0x501, Pole tekstowe przekroczyło rozmiar przy wskawianiu tekstu
- EN\_HSCROLL : 0x601, Pole tekstowe jest skrolowane w poziomie
- EN\_VSCROLL : 0x602, Pole tekstowe jest skrolowane w pionie

Okno główne może żądać od okien potomnych wykonania właściwych im operacji. Każda klasa okna potomnego charakteryzuje się specyficznymi możliwościami. Okno główne wysyła do okien potomnych takie żądania za pomocą funkcji:

```
LRESULT SendMessage(
    HWND hWnd, // uchwyt okna
    UINT Msg, // komunikat
    WPARAM wParam, // parametr
    LPARAM lParam // parametr
);
```

Możliwości okien potomnych są naprawdę duże. Wspomnijmy tylko o kilku, natomiast pełna ich lista dostępna jest w dokumentacji. Na przykład do pola tekstowego można wysłać komunikat:

```
EM_FINDTEXT
wParam = (WPARAM) (UINT) fuFlags;
lParam = (LPARAM) (FINDTEXT FAR *) lpFindText;
```

gdzie:

- `fuFlags` : zero, `FT_MATCHCASE` lub `FT_WHOLEWORD`
- `lpFindText` : wskaźnik do struktury `FINDTEXT` zawierającej informacje o szukanym tekście
- wynik : -1 jeśli nie znaleziono tekstu, w przeciwnym razie indeks pozycji szukanego tekstu oraz około 30 innych, odpowiadających m.in. za kolor, ograniczenie długości, przesuwanie zawartości, undo itd.

Do comboboxa można wysyłać komunikaty (łącznie około 20):

- `CB_GETCOUNT` : zwraca liczbę elementów
- `CB_FINDSTRING` : szuka tekstu wśród elementów listy
- `CB_GETITEMDATA`, `CB_SETITEMDATA` : zwraca lub ustawia wartość związaną z elementem listy
- `CB_GETTOPINDEX`, `CB_SETTOPINDEX` : zwraca lub ustawia indeks pierwszego widocznego elementu listy
- ...

Do `ListView` można wysyłać komunikaty (łącznie około 30):

- `LVM_DELETECOLUMN`
- `LVM_ENSUREVISIBLE`
- `LVM_GETCOLUMNWIDTH`, `LVM_SETCOLUMNWIDTH`
- `LVM_GETITEM`, `LVM_SETITEM`
- `LVM_SORTITEMS`
- ...

Znając identyfikator okna potomnego można łatwo uzyskać jego uchwyt i odwrotnie - znając uchwyt można łatwo uzyskać identyfikator.

```
id = GetDlgCtrlID (hwndChild) ;
hwndChild = GetDlgItem (hwndParent, id) ;
```

Przykład użycia comboboxa:

```
// Przygotuj kombo
hwndChild = CreateWindow ( "COMBOBOX",
    "",
    WS_CHILD | WS_VISIBLE | CBS_DROPDOWNLIST,
    posX, posY,
    width, height,
    hwnd, (HMENU) (1),
    ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;
SendMessage( hwndChild, CB_ADDSTRING, 0, "Item1" );
SendMessage( hwndChild, CB_ADDSTRING, 0, "Item2" );
```



Rysunek 2.3: Rozwijalny combobox z dwoma elementami

### 2.2.4 Subclasowanie okien potomnych

W poprzednich przykładach widzieliśmy, że okna potomne informują o zdarzeniach, które zaszły w ich obszarze roboczym za pomocą powiadomień. Niestety, ilość możliwych powiadomień przysyłanych przez okna potomne jest śmiesznie mała w porównaniu z możliwościami jakie dawałoby samodzielne oprogramowanie pętli komunikatów okna potomnego.

Problem w tym, że okna potomne są egzemplarzami klas już opisanych, w związku z czym mają już swoje funkcje obsługi. Czy jest możliwe samodzielne obsługiwanie komunikatów okna potomnego, dzięki czemu możnaby na przykład dowiedzieć się o dwukliku w jego obszar roboczy?

Okazuje się, że taka możliwość istnieje i nosi nazwę *subclasowania*<sup>8</sup> okna. Programista może określić własną funkcję obsługi okna za pomocą funkcji:

```
LONG GetWindowLong(
```

```
    HWND hWnd,
    int nIndex
);
```

```
LONG SetWindowLong(
```

```
    HWND hWnd,
    int nIndex,
    LONG dwNewLong
);
```

odczytując i zapamiętując najpierw wskaźnik na już istniejącą funkcję obsługi komunikatów, a następnie podając wskaźnik na nową. Należy pamiętać o tym, aby nowa funkcja obsługi komunikatów, po obsłużeniu przekazywała wszystkie komunikaty do starej funkcji (chyba że taka sytuacja jest niepożądana). Chodzi o to, aby okno nie straciło dotychczasowej funkcjonalności,

<sup>8</sup>Nie znam sensownego polskiego odpowiednika. Słyszałem już różne propozycje, na przykład mylnie kojarzące się z obiektowością "przeciążanie", czy przegadane "przeciążanie funkcji obsługi okna". Termin subclassowanie jest zwięzły i precyzyjny, z pewnością będzie jednak razili purystów językowych.

a nowa funkcja obsługi komunikatów tylko ją rozszerzała. Dysponując wskaźnikiem na starą funkcję obsługi komunikatów, należy skorzystać z funkcji **CallWindowProc** aby wywołać ją z odpowiednimi parametrami.

```
/*
 *
 * Subclassing
 *
 */
#include <windows.h>
#include <string.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
WNDPROC lpEditOldWndProc = NULL;
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK EditWindowProcedure(HWND hwnd, UINT message,
                                     WPARAM wParam, LPARAM lParam);

/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;        /* Komunikaty okna */
    WNDCLASSEX wincl;    /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName  = szClassName;
    wincl.lpfnWndProc    = WindowProcedure;    // wskaźnik na funkcję obsługi okna
    wincl.style          = CS_DBLCLKS;
    wincl.cbSize         = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName   = NULL;
    wincl.cbClsExtra     = 0;
    wincl.cbWndExtra     = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground  = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0,
        szClassName,
        "PRZYKLAD",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        HWND_DESKTOP,
        NULL,
        hThisInstance,
        NULL
    );

    ShowWindow(hwnd, nFunsterStil);
    /* Pętla obsługi komunikatów */
    while(GetMessage(&messages, NULL, 0, 0))
    {
        /* Tłumacz kody rozszerzone */
        TranslateMessage(&messages);
    }
}
```



```

        /* Obsługa komunikat */
        DispatchMessage(&messages);
    }

    /* Zwróć parametr podany w PostQuitMessage( ) */
    return messages.wParam;
}

int xSize, ySize;

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                  WPARAM wParam, LPARAM lParam)
{
    static HWND hwndEdit;
    static int  cxChar, cyChar;
    static RECT r;
    HDC        hdc;
    int         i;
    PAINTSTRUCT ps;

    TCHAR       szFormat[] = TEXT ("%-16s Akcja: %04X, ID:%04X, hWnd:%08X");
    TCHAR       szBuffer[80];

    switch (message)
    {
        case WM_CREATE :
            cxChar = LOWORD (GetDialogBaseUnits ());
            cyChar = HIWORD (GetDialogBaseUnits ());

            hwndEdit = CreateWindow ( "EDIT",
                                     "TEXTBOX",
                                     WS_CHILD | WS_VISIBLE | WS_BORDER | ES_MULTILINE,
                                     cxChar, cyChar,
                                     20 * cxChar, 7 * cyChar,
                                     hwnd, (HMENU)1,
                                     ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;

            // zapamiętaj starą i ustal nową funkcję
            // obsługi komunikatów
            lpEditOldWndProc = GetWindowLong( hwndEdit, GWL_WNDPROC );
            SetWindowLong( hwndEdit, GWL_WNDPROC, EditWindowProcedure );

            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_SIZE:
            xSize = LOWORD(lParam);
            ySize = HIWORD(lParam);

            r.left   = 24 * cxChar ;
            r.top    = 2 * cyChar ;
            r.right  = LOWORD (lParam) ;
            r.bottom = HIWORD (lParam) ;

            break;
        case WM_COMMAND:
            hdc = GetDC (hwnd);

            ScrollWindow (hwnd, 0, -cyChar, &r, &r) ;
            SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

            SetBkMode (hdc, TRANSPARENT) ;
            TextOut (hdc, 24 * cxChar, cyChar * (r.bottom / cyChar - 1),
                    szBuffer,
                    wsprintf (szBuffer, szFormat,
                              "WM_COMMAND",
                              HIWORD (wParam), LOWORD (wParam), lParam ));
    }
}

```

```

        ReleaseDC( hwnd, hdc );
        return DefWindowProc(hwnd, message, wParam, lParam);
    case WM_PAINT:
        hdc = BeginPaint( hwnd, &ps );
        EndPaint( hwnd, &ps );
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
}
return 0;
}

LRESULT CALLBACK EditWindowProcedure(HWND hwnd, UINT message,
                                     WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_RBUTTONDOWN :
            SetWindowText( hwnd, "NOWYTEXT" );
            break;
        case WM_LBUTTONDBLCLK :
            MessageBox( 0, "DoubleClick", "", 0 );
            break;
    }
    return CallWindowProc( lpEditOldWndProc, hwnd, message, wParam, lParam );
}

```

## 2.2.5 Obsługa grafiki za pomocą GDI

### 2.2.5.1 Podstawy GDI

Podsystem GDI odpowiada za rysowanie elementów graficznych w specjalnie utworzonych kontekstach urządzeń (*DC, Device Contexts*). Kontekst urządzenia może być skojarzony nie tylko z oknem, ale także na przykład z wirtualnym obrazem strony tworzonej na drukarce. Dzięki takiemu podejściu programista może użyć dokładnie tych samych mechanizmów do tworzenia obrazu na w oknie i na drukarce.

GDI jest jednym z najlepszych przykładów na to, że z perspektywy programisty nie tylko każda odmiana systemu Windows zachowuje się tak samo, ale również każdy model PCta, choć przecież zbudowany z innych podzespołów, identycznie reaguje na polecenia programisty. Nie ważne, czy w komputerze mam najnowszy model karty graficznej, czy zwykłą kartę VGA, Windows na polecenie narysowania linii na ekranie zareaguje tak samo.

Dzieje się tak dlatego, że między wywołaniem funkcji przez programistę, a pojawieniem się jej efektów, system operacyjny wykonuje mnóstwo pracy, o której nawet programista nie ma pojęcia. W przypadku GDI, Windows wysyła odpowiednie polecenia do **sterownika ekranu**, który, co nie powinno dziwić, również ma swój interfejs programowania, służący do porozumiewania się sterownika z systemem, tyle że ukryty przed programistą pracującym z Win32API.

Zobaczmy przykład użycia GDI:

```

/*
 *
 * Tworzenie grafiki za pomocą GDI
 *
 */
#include <windows.h>
#include <string.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

```

```

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;        /* Komunikaty okna */
    WNDCLASSEX wincl;    /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName  = szClassName;
    wincl.lpfnWndProc     = WindowProcedure; // wskaźnik na funkcję obsługi okna
    wincl.style          = CS_DBLCLKS;
    wincl.cbSize         = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName    = NULL;
    wincl.cbClsExtra     = 0;
    wincl.cbWndExtra     = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground  = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0,
        szClassName,
        "PRZYKŁAD",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        512,
        512,
        HWND_DESKTOP,
        NULL,
        hThisInstance,
        NULL
    );

    ShowWindow(hwnd, nFunsterStil);
    /* Pętla obsługi komunikatów */
    while(GetMessage(&messages, NULL, 0, 0))
    {
        /* Tłumacz kody rozszerzone */
        TranslateMessage(&messages);
        /* Obsłuż komunikat */
        DispatchMessage(&messages);
    }

    /* Zwróć parametr podany w PostQuitMessage( ) */
    return messages.wParam;
}

int xSize, ySize;

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    char sText[] = "Przykład 1, witam";
    HDC      hdc ; // kontekst urządzenia
    int      i ;
    PAINTSTRUCT ps ;
    RECT r;

    HPEN     hPen;

```

```

HBRUSH hBrush;

switch (message)
{
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    case WM_SIZE:
        xSize = LOWORD(lParam);
        ySize = HIWORD(lParam);

        GetClientRect( hwnd, &r );
        InvalidateRect( hwnd, &r, 1 );

        break;
    case WM_PAINT:
        hdc = BeginPaint( hwnd, &ps );

        // linie
        hPen = CreatePen( PS_SOLID, 3, RGB(255, 0, 0) );
        SelectObject( hdc, hPen );
        for ( i=0; i<xSize; i+=xSize/40 )
        {
            MoveToEx( hdc, xSize/2, 0, NULL );
            LineTo( hdc, i, ySize );
        }
        DeleteObject( hPen );

        // kształty
        SetBkColor( hdc, RGB(0, 255, 0) );
        hBrush = CreateHatchBrush( HS_CROSS, RGB(0, 0, 255) );
        SelectObject( hdc, hBrush );
        r.left   = 10;
        r.top    = 10;
        r.right  = 50;
        r.bottom = 50;
        FillRect( hdc, &r, hBrush );
        DeleteObject( hBrush );

        // tekst
        if ( xSize > 0 && ySize > 0 )
        {
            SetTextAlign( hdc, TA_CENTER | VTA_CENTER );
            SetBkMode( hdc, TRANSPARENT );
            TextOut( hdc, xSize / 2, 20, sText, strlen( sText ) );
        }

        EndPaint(hwnd, &ps);
        break;

    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
}
return 0;
}

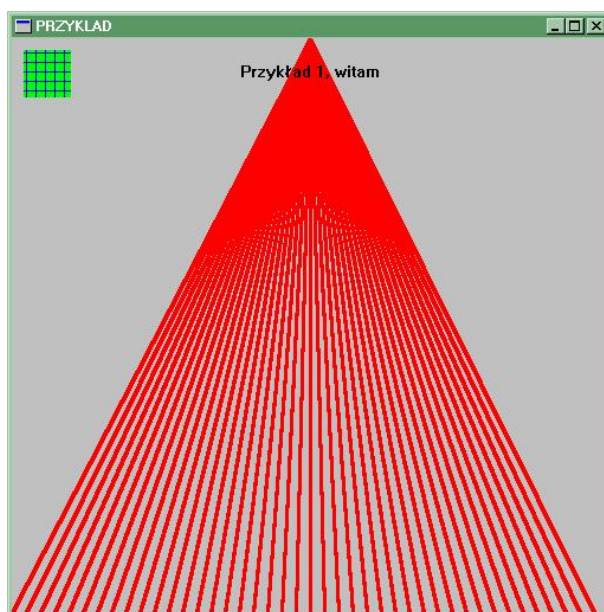
```

Jak widać obiektów GDI używa się w sposób dość prosty. Obiekt jest najpierw tworzony za pomocą odpowiedniej funkcji (na przykład **CreatePen**), następnie jest ustawiany jako bieżący (za pomocą funkcji **SelectObject**), zaś po użyciu jest niszczone (**DeleteObject**).

### 2.2.5.2 Uchwyty do kontekstów urządzeń

Wszystkie funkcje GDI, które odpowiadają za tworzenie obrazu, przyjmują jako pierwszy parametr uchwyt do kontekstu urządzenia. Dzięki temu system wie do jakiego obiektu (okna, drukarki) odnosi się aktualna funkcja.

W przypadku rysowania w oknach, kontekst urządzenia można uzyskać na dwa sposoby.



Rysunek 2.4: Obsługa grafiki okna za pomocą GDI

**Wewnątrz WM\_PAINT** W kodzie obsługującym komunikat WM\_PAINT uchwyt kontekstu można pobrać i zwolnić za pomocą funkcji

```
HDC BeginPaint(
    HWND hwnd,
    LPPAINTSTRUCT lpPaint
);

BOOL EndPaint(
    HWND hwnd,
    CONST PAINTSTRUCT *lpPaint
);
```

**Poza WM\_PAINT** Poza kodem obsługującym komunikat WM\_PAINT uchwyt kontekstu można pobrać i zwolnić za pomocą funkcji

```
HDC GetDC(
    HWND hwnd
);

HDC GetWindowDC(
    HWND hwnd
);

int ReleaseDC(
    HWND hwnd,
    HDC hDC
);
```

Skąd system Windows wie, kiedy do okna przesłać komunikat WM\_PAINT oznaczający konieczność odświeżenia zawartości okna? Otóż z każdym oknem system kojarzy informację o tym, czy jego zawartość jest ważna, czy nie.

Po zakończeniu rysowania i wywołaniu funkcji **EndPaint**, zawartość okna jest ważna. Kiedy okno zostanie na przykład przykryte innym oknem, a następnie odsłonięte z powrotem lub na przykład zminimalizowane a następnie przywołane z powrotem, Windows automatycznie wysyła do okna komunikat WM\_PAINT, uznając powierzchnię okna za nieważną.

Bardzo często okazuje się, że programista chce powierzchnię okna unieważniać częściej niż gdyby miało działać się to automatycznie. Na przykład wtedy, kiedy zawartość okna musi być odświeżana regularnie, ponieważ zawiera jakieś chwilowe, ulotne informacje. W takim przypadku obszar okna może być unieważniany bądź zatwierdzany za pomocą funkcji:

```

BOOL InvalidateRect(

    HWND hWnd,
    CONST RECT *lpRect,
    BOOL bErase
);

BOOL ValidateRect(

    HWND hWnd,
    CONST RECT *lpRect
);

```

Pierwsza z tych funkcji powoduje natychmiastowe wysłanie do okna komunikatu WM\_PAINT, druga zaś powoduje zatwierdzenie obszaru okna. System traktuje komunikat WM\_PAINT w sposób trochę szczególny, bowiem wysyłanie tego komunikatu częściej niż jest on obsługiwany nie ma żadnego efektu - w kolejce komunikatów do okna może znajdować się w danej chwili tylko jeden komunikat WM\_PAINT.

### 2.2.5.3 Własne kroje pisma

Własne kroje pisma można tworzyć za pomocą funkcji

```

HFONT CreateFont(

    int nHeight,
    int nWidth,
    int nEscapement,
    int nOrientation,
    int fnWeight,
    DWORD fdwItalic,
    DWORD fdwUnderline,
    DWORD fdwStrikeOut,
    DWORD fdwCharSet,
    DWORD fdwOutputPrecision,
    DWORD fdwClipPrecision,
    DWORD fdwQuality,
    DWORD fdwPitchAndFamily,
    LPCTSTR lpzFace
);

```

Aby utworzona czcionka stała się aktywna należy oczywiście wybrać ją w jakimś kontekście graficznym za pomocą funkcji **SelectObject**.

### 2.2.6 Tworzenie menu

Do tworzenia menu przeznaczone są funkcje **CreateMenu**, **AppendMenu** i **SetMenu**.

```

#include <windows.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
void CreateMyMenu( HWND hwnd );

```

```

/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;        /* Komunikaty okna */
    WNDCLASSEX wincl;    /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc    = WindowProcedure; // wskaźnik na funkcję obsługi okna
    wincl.style          = CS_DBLCLKS;
    wincl.cbSize         = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName   = NULL;
    wincl.cbClsExtra     = 0;
    wincl.cbWndExtra     = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0,
        szClassName,
        "Przykład",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        512,
        512,
        HWND_DESKTOP,
        NULL,
        hThisInstance,
        NULL
    );

    CreateMyMenu( hwnd );
    ShowWindow(hwnd, nFunsterStil);
    /* Pętla obsługi komunikatów */
    while(GetMessage(&messages, NULL, 0, 0))
    {
        /* Tłumacz kody rozszerzone */
        TranslateMessage(&messages);
        /* Obsłuż komunikat */
        DispatchMessage(&messages);
    }

    /* Zwróć parametr podany w PostQuitMessage( ) */
    return messages.wParam;
}

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
    }
}

```

```

        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case 101 : SendMessage( hwnd, WM_CLOSE, 0, 0 );break;
            }
            default:
                return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

void CreateMyMenu( HWND hwnd )
{
    HMENU hMenu;
    HMENU hSubMenu;

    hMenu = CreateMenu ( ) ;

    hSubMenu = CreateMenu ( ) ;
    AppendMenu (hSubMenu, MF_STRING , 100, "&Nowy") ;
    AppendMenu (hSubMenu, MF_SEPARATOR, 0 , NULL) ;
    AppendMenu (hSubMenu, MF_STRING , 101, "&Koniec") ;
    AppendMenu (hMenu, MF_POPUP, hSubMenu, "&Plik") ;

    hSubMenu = CreateMenu ( ) ;
    AppendMenu (hSubMenu, MF_STRING, 102, "&Undo") ;
    AppendMenu (hSubMenu, MF_SEPARATOR, 0, NULL) ;
    AppendMenu (hSubMenu, MF_STRING, 103, "Re&do") ;
    AppendMenu (hMenu, MF_POPUP, hSubMenu, "&Edycja") ;

    SetMenu( hwnd, hMenu );
}

```

Menu utworzone w taki sposób może być również wykorzystywane jak menu kontekstowe:

```

case WM_RBUTTONDOWN:
    point.x = LOWORD (lParam) ;
    point.y = HIWORD (lParam) ;
    ClientToScreen (hwnd, &point) ;

    TrackPopupMenu (hMenu, TPM_RIGHTBUTTON, point.x, point.y,
        0, hwnd, NULL) ;

return 0 ;

```

## 2.3 Procesy, wątki, synchronizacja

### 2.3.1 Tworzenie wątków i procesów

Zadaniem systemu operacyjnego jest wykonywanie programów, przechowywanych najczęściej na różnego rodzaju nośnikach. Z punktu widzenia systemu operacyjnego, **program** to więc nic więcej niż plik, w którym przechowywany jest obraz kodu wynikowego programu.

Program uaktywnia się w wyniku jawnego utworzenia przez system operacyjny **procesu**, który odpowiada obrazowi programu. W systemie Windows do tworzenia procesu służy funkcja:

```

BOOL CreateProcess(

    LPCTSTR lpApplicationName, // nazwa modułu wykonywalnego
    LPCTSTR lpCommandLine, // linia poleceń
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // atrybuty bezpieczeństwa procesu
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // atrybuty bezpieczeństwa wątku
    BOOL bInheritHandles, // dziedziczenie uchwytów
    DWORD dwCreationFlags, // dodatkowe flagi, np. priorytet
    LPVOID lpEnvironment, // środowisko
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo, // właściwości startowe okna
    LPPROCESS_INFORMATION lpProcessInformation // zwraca informacje o procesie i wątku
);

```



Proces po załadowaniu do systemu nie wykonuje kodu, dostarcza jedynie przestrzeni adresowej *wątkom*. To wątki są jednostkami, którym system przydziela czas procesora. Każdy proces w systemie ma niejawnie utworzony jeden wątek wykonujący kod programu. Każdy następny wątek w obrębie jednego procesu musi być utworzony *explicite*.

Tworzenie wielu wątków w obrębie jednego procesu jest czasami bardzo przydatne. Wątki mogą na przykład przejmować na siebie długotrwałe obliczenia nie powodując "zamierania" całego procesu. Ponieważ wątki współdzielą zmienne globalne procesu, możliwa jest niedoczesna praca wielu wątków na jakimś zbiorze danych procesu.

Podsumujmy związek pomiędzy procesami a wątkami:

- Proces nie wykonuje kodu, proces jest obiektem dostarczającym wątkowi przestrzeni adresowej,
- Kod zawarty w przestrzeni adresowej procesu jest wykonywany przez wątek,
- Pierwszy wątek procesu tworzony jest *implicite* przez system operacyjny, każdy następny musi być utworzony *explicite*,
- Wszystkie wątki tego samego procesu dzielą wirtualną przestrzeń adresową i mają dostęp do tych samych zmiennych globalnych i zasobów systemowych.

Do tworzenia dodatkowych wątków w obrębie jednego procesu służy funkcja:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // atrybuty bezpieczeństwa wątku
    DWORD dwStackSize, // rozmiar stosu (0 - domyślny)
    LPTHREAD_START_ROUTINE lpStartAddress, // wskaźnik na funkcję wątku
    LPVOID lpParameter, // wskaźnik na argument
    DWORD dwCreationFlags, // dodatkowe flagi
    LPDWORD lpThreadId // zwraca identyfikator wątku
);
```

Po utworzeniu nowy wątek jest wykonywany równolegle z pozostałymi wątkami w systemie.

```
/*
 * Tworzenie wątków
 */
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI ThreadProc(LPVOID* theArg);

int main(int argc, char *argv[])
{
    DWORD threadID;
    DWORD thread_arg = 4;

    HANDLE hThread = CreateThread( NULL, 0,
                                   (LPTHREAD_START_ROUTINE)ThreadProc,
                                   &thread_arg, 0, &threadID );

    WaitForSingleObject( hThread, INFINITE );

    return 0;
}

DWORD ThreadProc(LPVOID* theArg)
{
    DWORD timestoprint = (DWORD)*theArg;
    for (int i = 0; i<timestoprint; i++)
        printf("Witam %d \n", i);
    return TRUE;
}
```

Programista ma do dyspozycji kilka dodatkowych funkcji do manipulacji tworzonymi wątkami i procesami, m.in.:

- Ustalanie priorytetu wątku.

```
BOOL SetThreadPriority(
    HANDLE hThread, // handle to the thread
    int nPriority    // thread priority level
);
```

- Ustalanie procesora na którym wykonuje się wątek w systemach wieloprocessorowych.

```
DWORD SetThreadAffinityMask (
    HANDLE hThread, // handle to the thread of interest
    DWORD dwThreadAffinityMask // a thread affinity mask
);
```

### 2.3.2 Synchronizacja wątków

Sytuacja w której wiele wątków jednocześnie operuje na danych globalnych procesu może rodzić problemy. Wyobraźmy sobie bowiem dwa wątki mające dostęp do zmiennej globalnej. Niech pierwszy z wątków wykonuje następujący kod :

```
i = 0;
// !
if ( i == 0 )
{
    ...
}
```

a drugi:

```
i = 1;
// *
if ( i == 1 )
{
    ...
}
```

Jeśli któryś z wątków zostanie przez system operacyjny wywłaszczony w miejscu oznaczonym w kodzie znakiem "\*", drugi wątek stanie się aktywnym i wykona swoją instrukcję przypisania, po czym ponownie pierwszy wątek stanie się aktywny, to operacja porównania zakończy się niepowodzeniem, najprawdopodobniej wbrew intencjom programisty. To czego potrzeba, aby unikać tego typu problemów, to jakaś forma kontroli nad przełączaniem wątków przez system.

Win32API udostępnia 5 sposoby synchronizacji wątków. Są to:

- zdarzenia,
- mutexy,
- semaforey,
- sekcje krytyczne,
- zegary oczekujące

Mechanizm sekcji krytycznej możliwy jest do wykorzystania tylko w obrębie jednego procesu (do synchronizacji wątków), jednak jest to metoda najszybsza i najwydajniejsza. Pozostałe metody mogą być stosowane również dla wielu procesów.

### 2.3.2.1 Zdarzenia

Win32API umożliwia definiowanie własnych zdarzeń za pomocą funkcji

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, // atrybuty bezpieczeństwa
    BOOL bManualReset, // flaga ręcznego resetowania
    BOOL bInitialState, // flaga początkowego stanu
    LPCTSTR lpName // nazwa
);
```

Każdy oczekujący wątek widzi zdarzenie jako pewną dwustanową flagę: zdarzenie jest zgłoszone albo odwołane. Za pomocą funkcji

```
BOOL SetEvent(
    HANDLE hEvent // uchwyt zdarzenia
);
```

informujemy system o zaistnieniu zdarzenia. Od tej pory zdarzenie jest zgłoszone i wszystkie wątki oczekujące do tej pory na jego zgłoszenie mogą wznowić działanie. Zdarzenie zostaje odwołane, kiedy zostanie wywołana funkcja

```
BOOL ResetEvent(
    HANDLE hEvent // uchwyt zdarzenia
);
```

Na zaistnienie wydarzenia w systemie wątki oczekują za pomocą funkcji

```
DWORD WaitForSingleObject(
    HANDLE hHandle, // uchwyt obiektu synchronizacji
    DWORD dwMilliseconds // czas oczekiwania (INFINITE czeka aż do
                        // zajścia zdarzenia)
);
```

Zdarzenie utworzone z ustawioną flagą ręcznego odwoływania (`CreateEvent(...,TRUE,...,...)`) wymaga odwołania explicite (przez `ResetEvent()`), natomiast zdarzenie utworzone z flagą automatycznego odwoływania (`CreateEvent(...,FALSE,...,...)`) zostaje odwołane automatycznie po przepuszczeniu jednego wątku przez funkcję oczekującą.

Warto również omówić działanie funkcji

```
BOOL PulseEvent(
    HANDLE hEvent // uchwyt zdarzenia
);
```

Otóż powoduje ona zgłoszenie zdarzenia, po czym natychmiastowe jego odwołanie. Działanie oczekujących wątków zależy od tego, czy zdarzenie jest odwoływane automatycznie czy ręcznie (patrz paragraf wyżej): jeśli zdarzenie odwoływane jest ręcznie, to funkcja `PulseEvent()` przepuszcza wszystkie wątki oczekujące w danej chwili na zdarzenie, po czym odwołuje zdarzenie, jeśli zaś zdarzenie odwoływane jest automatycznie, to funkcja `PulseEvent()` przepuszcza tylko jeden wątek z puli oczekujących w danej chwili wątków, po czym odwołuje zdarzenie.

```
/*
 * Wykorzystanie zdarzeń do synchronizacji wątków
 */
void main(void)
{
```

```

HANDLE hThread[2];
DWORD threadID1, threadID2;
char szFileName="c:\\myfolder\\myfile.txt";

hEvent=CreateEvent(NULL, TRUE, FALSE, "FILE_EXISTS");

// tworzymy dwa wątki które czekają na utworzenie pliku
hThread[0]=CreateThread(NULL, 0, ThreadProc1, szFileName, 0, &threadID1);
hThread[1]=CreateThread(NULL, 0, ThreadProc2, szFileName, 0, &threadID1);

HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . . .);

// kod wypełniający plik danymi np. WriteFile(...)

// sygnalizacja wątkom tego, że dane są gotowe
// wątki od ich utworzenia tylko na to czekały
SetEvent(hEvent);
WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

CloseHandle(hEvent);
CloseHandle(hFile);
CloseHandle(hThread[0]);
CloseHandle(hThread[1]);
}

DWORD ThreadProc1(LPVOID* arg)
{
    char szFileName = (char*)arg;
    // tutaj wątek otwiera zdarzenie określone w module głównym
    HANDLE hEvent = OpenEvent(SYNCHRONIZE, FALSE, "FILE_EXISTS");
    // czeka na jego pojawienie się
    WaitForSingleObject(hEvent, INFINITE);
    // i czyta dane zapisane do pliku
    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
                                     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    // przetwarzaj dane
    // ...
    return TRUE;
}

DWORD ThreadProc2(LPVOID* arg)
{
    char szFileName = (char*)arg;
    // tutaj wątek otwiera zdarzenie określone w module głównym
    HANDLE hEvent = OpenEvent(SYNCHRONIZE, FALSE, "FILE_EXISTS");
    // czeka na jego pojawienie się
    WaitForSingleObject(hEvent, INFINITE);
    // i czyta dane zapisane do pliku
    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
                                     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    // przetwarzaj dane
    // ...
    return TRUE;
}

```

### 2.3.2.2 Mutexy

Nazwa mutex pochodzi od angielskiego terminu mutually exclusive (wzajemnie wykluczający się). Mutex jest obiektem służącym do synchronizacji. Jego stan jest ustawiony jako "sygnalizowany", kiedy żaden wątek nie sprawuje nad nim kontroli oraz "niesygnalizowany" kiedy jakiś wątek sprawuje nad nim kontrolę. Synchronizację za pomocą mutexów realizuje się tak, że każdy wątek czeka na objęcie mutexu w posiadanie, zaś po zakończeniu operacji wymagającej wyłączności, wątek uwalnia mutex.

W celu stworzenia mutexu, wątek woła funkcję

```
HANDLE CreateMutex(

    LPSECURITY_ATTRIBUTES lpMutexAttributes, // atrybuty bezpieczeństwa
    BOOL bInitialOwner, // flaga własności przy tworzeniu
    LPCTSTR lpName // nazwa
);
```

W chwili tworzenia wątek może zażądać natychmiastowego prawa własności do mutexa. Inne wątki (nawet innych procesów) uzyskują uchwyt mutexa za pomocą funkcji

```
HANDLE OpenMutex(

    DWORD dwDesiredAccess, // flaga dostępu
    BOOL bInheritHandle, // uchwyt dziedziczony na procesy tworzone
                          // przez CreateProcess
    LPCTSTR lpName // nazwa
);
```

Następnie czekają na objęcie mutexa w posiadanie (za pomocą WaitForSingleObject()). Do uwalniania mutexów służy funkcja

```
BOOL ReleaseMutex(

    HANDLE hMutex // handle of mutex object
);
```

Jeśli wątek kończy się bez uwalniania mutexów, które posiadał, takie mutexy uważa się za porzucone. Każdy czekający wątek może objąć takie mutexy w posiadanie, zaś funkcja czekająca na przydział mutexu (WaitForSingleObject(), jak widać bardzo uniwersalna funkcja) zwraca wartość WAIT\_ABANDONED. W takiej sytuacji warto zastanowić się, czy gdzieś nie wystąpił jakiś błąd (skoro wątek, który był w posiadaniu mutexu nie oddał go explicite przez ReleaseMutex(), to najprawdopodobniej został zakończony w jakiś nieprzewidziany sposób). Mutexy są w działaniu bardzo podobne do semaforów.

```
/*
 * Wykorzystanie mutexów do synchronizacji wątków
 */
void main(void)
{
    HANDLE hThread[2];
    DWORD threadID1, threadID2;

    char szFileName="c:\\myfolder\\myfile.txt";

    hMutex=CreateMutex(NULL, TRUE, "FILE_EXISTS");

    // tworzymy dwa wątki które czekają na utworzenie pliku

    hThread[0]=CreateThread(NULL, 0, ThreadProc1, &hMutex, 0, &threadID1);
    hThread[1]=CreateThread(NULL, 0, ThreadProc2, &hMutex, 0, &threadID1);

    HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . . .);

    // kod wypełniający plik danymi np. WriteFile(...)

    // sygnalizacja wątkom tego, że dane są gotowe
    // wątki od ich utworzenia tylko na to czekały

    ReleaseMutex(hMutex);
    WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

    CloseHandle(hMutex);
    CloseHandle(hFile);
    CloseHandle(hThread[0]);
```

```

        CloseHandle(hThread[1]);
    }

DWORD ThreadProc1(LPVOID* arg)
{
    HANDLE hMutex = (HANDLE)(*arg);
    WaitForSingleObject(hMutex, INFINITE);

    // i czyta dane zapisane do pliku
    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL, ...);

    // przetwarzaj dane
    ReleaseMutex(hMutex);
    return TRUE;
}

DWORD ThreadProc2(LPVOID* arg)
{
    HANDLE hMutex = (HANDLE)(*arg);
    WaitForSingleObject(hMutex, INFINITE);

    // i czyta dane zapisane do pliku
    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL, ...);

    // przetwarzaj dane
    ReleaseMutex(hMutex);
    return TRUE;
}

```

### 2.3.2.3 Semaforey

Semaforey mogą być wykorzystywane tam, gdzie zasób dzielony jest na ograniczoną ilość użytkowników. Semafor działa jak furtka kontrolująca ilość wątków wykonujących jakiś fragment kodu. Za pomocą semaforów aplikacja może kontrolować na przykład maksymalną ilość otwartych plików, czy utworzonych okien. Semaforey są w działaniu bardzo podobne do mutexów.

Nowy semafor tworzony jest w funkcji

```

HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount, // początkowa wartość licznika
    LONG lMaximumCount, // maksymalna wartość licznika
    LPCTSTR lpName // nazwa
);

```

Wątek tworzący semafor specyfikuje wartość wstępną i maksymalną licznika. Inne wątki uzyskują dostęp do semafora za pomocą funkcji

```

HANDLE OpenSemaphore(
    DWORD dwDesiredAccess, // dostęp
    BOOL bInheritHandle, // dziedziczenie
    LPCTSTR lpName // nazwa
);

```

i czekają na wejście za pomocą funkcji ... (to już powinno być jasne jakiej).

Po zakończeniu pracy w sekcji krytycznej wątek uwalnia semafor za pomocą funkcji

```

BOOL ReleaseSemaphore(
    HANDLE hSemaphore, // uchwyt
    LONG lReleaseCount, // wartość dodawana do licznika
    LPLONG lpPreviousCount // otrzymuje poprzednią wartość licznika
);

```

Wątki nie wchodzi w posiadanie semaforów! W przypadku mutexów, jeśli wątek zażąda po raz kolejny dostępu do tego mutexu, którego jest już właścicielem, dostęp taki zostaje mu przyznany natychmiast.

Jeśli wątek nagle rozpocznie czekanie na ten sam semafor, to semafor zachowuje się tak, jakby wejścia zażądał każdy inny wątek. Inaczej wygląda także sprawa uwalniania semaforów i mutexów: mutex może być uwolniony tylko przez wątek, który jest jego właścicielem, licznik semafora może być zwiększony przez dowolny wątek, który z tego semafora korzysta.

```
/*
 * Wykorzystanie semaforów do synchronizacji wątków
 */
void main(void)
{
    HANDLE hThread[2];
    DWORD threadID1, threadID2;
    char szFileName="c:\\myfolder\\myfile.txt";

    hSemaphore=CreateSemaphore(NULL, 0, 1, "FILE_EXISTS");

    // tworzymy dwa wątki które czekają na utworzenie pliku
    hThread[0]=CreateThread(NULL, 0, ThreadProc1, &hSemaphore, 0, &threadID1);
    hThread[1]=CreateThread(NULL, 0, ThreadProc2, &hSemaphore, 0, &threadID1);

    HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . . .);

    // kod wypełniający plik danymi np. WriteFile(...)

    // sygnalizacja wątkom tego, że dane są gotowe
    // wątki od ich utworzenia tylko na to czekały

    ReleaseSemaphore(hSemaphore, 1, NULL);
    WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

    CloseHandle(hSemaphore);
    CloseHandle(hFile);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
}

DWORD ThreadProc1(LPVOID* arg)
{
    HANDLE hSem = OpenSemaphore( SEMAPHORE_ALL_ACCESS, "FILE_EXISTS");
    WaitForSingleObject(hSem, INFINITE);

    // i czyta dane zapisane do pliku

    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL, ...);

    // przetwarzaj dane

    ReleaseSemaphore(hSem, 1, NULL);
    return TRUE;
}

DWORD ThreadProc2(LPVOID* arg)
{
    HANDLE hSem = OpenSemaphore( SEMAPHORE_ALL_ACCESS, "FILE_EXISTS");
    WaitForSingleObject(hSem, INFINITE);

    // i czyta dane zapisane do pliku

    HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL, ...);

    // przetwarzaj dane

    ReleaseSemaphore(hSem, 1, NULL);
    return TRUE;
}
```

### 2.3.2.4 Sekcja krytyczna

Interfejs programowania Win32API udostępnia typ danych `CRITICAL_SECTION`, który wraz z odpowiednim zestawem funkcji może być wykorzystany do implementacji sekcji krytycznej.

```

/*
 * Wykorzystanie sekcji krytycznej do synchronizacji wątków
 */
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAXTRY 3

CRITICAL_SECTION cs;          // dzielona na wszystkie wątki

// główny wątek programu
void ThreadMain(char *name)
{
    int i;

    for (i=0; i<MAXTRY; i++)
    {
        EnterCriticalSection(&cs);

        /* proszę spróbować też zamiast
        powyższej linii napisać

        while ( TryEnterCriticalSection(&cs)==FALSE )
        {
            printf("\%s, czekam na wejście\n", name);
            Sleep(5);
        }

        uwaga! - tylko na WinNT
        */

        printf("\%s, jestem w sekcji krytycznej!\n", name);
        Sleep(5);
        LeaveCriticalSection(&cs);

        printf("\%s, wyszedłem z sekcji krytycznej!\n", name);
    }
}

// tworzy wątek potomny
HANDLE CreateChild(char* name)
{
    HANDLE hThread; DWORD dwId;
    hThread = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)ThreadMain,
        (LPVOID)name, 0, &dwId);

    assert(hThread!=NULL); return hThread;
}

int main(void)
{
    HANDLE hT[4];

    InitializeCriticalSection(&cs);

    hT[0]=CreateChild("\Jurek");
    hT[1]=CreateChild("\Ogórek");
    hT[2]=CreateChild("\Kiełbasa");
    hT[3]=CreateChild("\Sznurek");

```



```

WaitForMultipleObjects(4, hT, TRUE, INFINITE);

CloseHandle(hT[0]);CloseHandle(hT[1]);
CloseHandle(hT[2]);CloseHandle(hT[3]);

DeleteCriticalSection(&cs);

return 0;
}

```

## 2.4 Komunikacja między procesami

Zajmiemy się dokładniej komunikacją za pomocą tzw. gniazd. Sama idea gniazd została opracowana na Uniwersytecie Kalifornijskim w Berkley w celu zapewnienia możliwości komunikacyjnych w systemie Unix. Opracowany tam interfejs programowania nosi nazwę "Berkley socket interface" i jest stopniowo z mniejszymi lub większymi zmianami przejmowany przez kolejne systemy operacyjne.

Największa zaleta jaką daje korzystanie z gniazd to w miarę prosty i przejrzysty interfejs niezależny od warstwy komunikacyjnej (oczywiście stosunkowo najmniej wygodnie korzysta się z interfejsu gniazd w czystym C, w Javie czy C# jest jeszcze prościej).

### 2.4.1 Charakterystyka protokołów sieciowych

Skoro gniazda są tylko interfejsem programowania służącym do oprogramowania transmisji sieciowych, to zajmijmy się przez chwilę samymi protokołami<sup>9</sup> i ich charakterystykami.

Bez wchodzenia w szczegóły, możemy podzielić protokoły na

- oparte o wiadomości (ang. message-oriented). Takim mianem określamy protokoły, które przesyłają dane w paczkach o skończonej pojemności. Nadawca wielokrotnie wysyła małe paczki informacji, odbiorca zaś musi wielokrotnie te paczki odczytywać. Wyobraźmy sobie, że nadawca wysyła 10 paczek po 64 znaki. Mimo że w warstwie transmisji protokół może zdecydować o połączeniu tych wiadomości w jedną paczkę, po dotarciu do odbiorcy wiadomości mogą być rozdzielone i przekazane w takich samych paczkach, w jakich były nadawane (choć niekoniecznie).
- strumieniowe (ang. stream-based). Tutaj proces nadawania trwa nieustannie, zaś odbiorca w danej chwili dostaje tylko informacji, ile w danej chwili do niego dotarło.

Inna ważna linia dzieli protokoły na

- wykorzystujące bezpośrednie połączenie (ang. connection-oriented). Te protokoły przed wysłaniem czy odebraniem jakichkolwiek informacji nawiązują bezpośrednie połączenie między nadawcą i odbiorcą. Chodzi o zagwarantowanie tego, że oba zainteresowane podmioty istnieją i są połączone ścieżką gotową do transmisji danych.
- bezpołączeniowe (ang. connectionless). Nadawca nie ma tutaj nawet gwarancji, że odbiorca istnieje. Trochę przypomina to usługi poczty - nadawca adresuje wiadomość i wysyła ją, nie wie jednak czy odbiorca na tę wiadomość czeka ani czy wiadomość do odbiorcy dotrze.

Każdy protokół może charakteryzować się posiadaniem lub brakiem następujących cech:

- wiarygodność (ang. reliability). Wiarygodny protokół musi zapewniać dotarcie każdego bajtu informacji od nadawcy do odbiorcy,

<sup>9</sup>Mowa na razie o protokołach *fizycznych*, czyli sposobach transmisji danych między różnymi maszynami w sieci. Jeśli przesyłane dane są w jakiś sposób interpretowane, to mamy do czynienia z protokołem *logicznym*.

- zachowywanie porządku (ang. ordering). Protokół który zachowuje porządek, dba o to by informacje docierały do odbiorcy w takiej samej kolejności w jakiej były nadawane,
- łagodne zakończenie (ang. graceful close). Kiedy któraś ze stron zamierza zamknąć połączenie, protokół może dać obu stronom szansę na doczytanie informacji, które są jeszcze w drodze.
- otwarta transmisja (ang. broadcast data). Protokół może transmitować dane tak, aby docierały do wszystkich maszyn w sieci. Wada takich protokołów polega na tym, że wszystkie maszyny w sieci muszą tracić czas na analizę przesyłanych informacji, mimo że niekoniecznie muszą być nimi zainteresowane.
- niezależność od warstwy komunikacyjnej (ang. routability). Transmisja w takich protokołach nie zależy od tego ile i jakie maszyny stoją pomiędzy nadawcą i odbiorcą. Możliwe jest nawet, by kolejne informacje docierały do odbiorcy inną drogą.

### 2.4.2 Podstawy biblioteki Winsock

Gniazda zaadaptowano do systemu Windows pod nazwą Winsock. Za pomocą tej biblioteki można tworzyć aplikacje korzystające z wielu protokołów sieciowych. Dzięki prostemu interfejsowi, z perspektywy programisty korzystanie z różnych protokołów wygląda niemal identycznie.

Przy nawiązywaniu połączeń jedna strona jest "serwerem", który oczekuje połączenia, druga strona jest "klientem", który nawiązuje połączenie z "serwerem". Protokół fizyczny określa sposób transmisji danych między klientem a serwerem (sposób w jaki dane są przesyłane). Protokół fizyczny to jednak nie wszystko - gwarancja, że dane zostały przesłane nie oznacza, że przekaz został zrozumiany. Wyobraźmy sobie na przykład, że klient i serwer komunikują się za pomocą protokołu fizycznego TCP/IP i że klient wysłał do serwera strumień danych "QWERTY". Co serwer ma zrobić z takim przekazem? Otóż za interpretację przesyłanych danych odpowiadają coś co moglibyśmy nazwać protokołem logicznym, czyli pewien umówiony zestaw komunikatów rozumianych przez obie strony transmisji. Kilka powszechnie znanych przykładów protokołów logicznych to HTTP, FTP, TELNET.

Jeden fizyczny komputer może pełnić rolę serwera sieciowego dla wielu różnych protokołów logicznych. W przypadku protokołu TCP/IP istnieje możliwość zdefiniowania aż 65535 rozłącznych serwerów wirtualnych, oczekujących na połączenia z klientami. Numer takiego wirtualnego serwera zwykle nazywa się *portem*<sup>10</sup>.

Aby uporządkować możliwy bałagan jaki daje dowolność wyboru portu dla serwera nasłuchującego połączeń, usługi typowe mają ściśle przyporządkowane numery portów. Na przykład łącząc się z jakąś maszyną na port o numerze 80 możemy być niemal pewni, że skomunikujemy się z serwerem HTTP, zaś na porcie 21 oczekuje TELNET, a na porcie 25 FTP<sup>11</sup>. Zobaczmy więc najpierw jak przeglądać porty lokalnej maszyny w poszukiwaniu serwerów oczekujących na połączenia.

```
#include <stdio.h>
#include <winsock2.h>

int ssocket, new_socket;
WSADATA wsd;
struct sockaddr_in addr;
```

<sup>10</sup>Słowo *port* określa tu więc tylko numer za pomocą którego klienci mogą rozróżniać wirtualne serwery oczekujące na połączenia na tej samej maszynie. Nie ma to nic wspólnego z żadnymi fizycznymi portami komputera.

<sup>11</sup>Jest to jednak tylko umowa. Nie ma przeszkód w uruchomieniu serwera HTTP na porcie powiedzmy 3333. Przeglądarki Internetowe domyślnie łączą się właśnie do portu 80 maszyny docelowej, jednak istnieje możliwość wymuszenia innego numeru portu, na przykład <http://www.qwe.com:3333>.

```

int      sourceport;

int main()
{
    printf( "TCP/IP port status:\n" );

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Błąd ładowania Winsock 2.2!\n");
        return 1;
    }

    for ( sourceport=0; sourceport<65535; sourceport++ )
    {
        ssocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        addr.sin_family=AF_INET;
        addr.sin_addr.s_addr=htonl(INADDR_ANY);
        addr.sin_port=htons( (unsigned short)sourceport);

        if (bind(ssocket,(struct sockaddr*)&addr,sizeof(addr)))
        {
            printf("Otwarty port %d\n", sourceport);
        }
        shutdown(ssocket, SD_BOTH);
        closesocket(ssocket);
    }

    return 0;
}

```

Jeśli próba połączenia gniazda z adresem o ustalonym porcie kończy się niepowodzeniem, to przyjmujemy, że port jest zajęty<sup>12</sup>, jednak nie ma możliwości określenia jakiego protokołu logicznego spodziewa się serwer nasłuchujący na określonym porcie. Jak zobaczymy bowiem w poniższym przykładzie, serwer może w ogóle nie posługiwać się żadnym protokołem logicznym.

Kod programu serwera:

```

// prosty moduł serwera
// komunikacji za pomocą Winsock
// użycie: server.exe

#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT 5000
#define DEFAULT_BUFFER 4096

// tylko Visual C++
#pragma comment(lib, "ws2_32.lib")

// funkcja: wątek do komunikacji z klientem
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET sock = (SOCKET)lpParam;
    char szBuf[DEFAULT_BUFFER];
    int ret,
        nLeft,
        idx;

    // serwer będzie oczekiwał na informacje od klienta
    while(1)
    {
        // najpierw odbierz dane
        ret = recv(sock, szBuf, DEFAULT_BUFFER, 0);
        if (ret == 0)
            break;
    }
}

```

---

<sup>12</sup>Tak naprawdę mogą być inne przyczyny błędu funkcji bind(), jednak możemy je pominąć.

```

else if (ret == SOCKET_ERROR)
{
    printf("błąd funkcji recv(): %d\n", WSAGetLastError());
    break;
}

szBuf[ret] = '\0';
printf("RCV: '%s'\n", szBuf);

// następnie odeślij te dane, poproś jeśli trzeba
// (niestety send() może nie wysłać wszystkiego)
nLeft = ret;
idx = 0;
while(nLeft > 0)
{
    ret = send(sock, &szBuf[idx], nLeft, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        printf("błąd funkcji send(): %d\n", WSAGetLastError());
        break;
    }
    nLeft -= ret;
    idx += ret;
}
}
return 0;
}

int main(int argc, char *argv[])
{
    WSADATA wsd;
    SOCKET sListen,
    sClient;
    int iAddrSize;
    HANDLE hThread;
    DWORD dwThreadID;
    struct sockaddr_in local, client;
    struct hostent *host = NULL;

    // inicjuj Winsock 2.2
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Błąd ładowania Winsock 2.2!\n");
        return 1;
    }

    // twórz gniazdo do nasłuchu połączeń klientów
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sListen == SOCKET_ERROR)
    {
        printf("Błąd funkcji socket(): %d\n", WSAGetLastError());
        return 1;
    }

    // wybierz interfejs (warstwę komunikacyjną)
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(DEFAULT_PORT);
    if (bind(sListen, (struct sockaddr *)&local, sizeof(local)) == SOCKET_ERROR)
    {
        printf("Błąd funkcji bind(): %d\n", WSAGetLastError());
        return 1;
    }

    // nasłuch
    host = gethostbyname("localhost");
    if (host == NULL)
    {

```

```

    printf("Nie udało się wydobyć nazwy serwera\n");
    return 1;
}

listen(sListen, 8);
printf("Serwer nasłuchuje.\n");
printf("Adres: %s, port: %d\n", host->h_name, DEFAULT_PORT);

// akceptuj nadchodzące połączenia
while (1)
{
    iAddrSize = sizeof(client);
    sClient = accept(sListen, (struct sockaddr *)&client, &iAddrSize);
    if (sClient == INVALID_SOCKET)
    {
        printf("Błąd funkcji accept(): %d\n", WSAGetLastError());
        return 1;
    }
    printf("Zaakceptowano połączenie: serwer %s, port %d\n",
        inet_ntoa(client.sin_addr), ntohs(client.sin_port));
    hThread = CreateThread(NULL, 0, ClientThread,
        (LPVOID)sClient, 0, &dwThreadID);
    if (hThread == NULL)
    {
        printf("Błąd funkcji CreateThread(): %d\n", WSAGetLastError());
        return 1;
    }
    CloseHandle(hThread);
}
closesocket(sListen);

WSACleanup();
return 0;
}

```

Kod programu klienta:

```

// prosty moduł klienta
// komunikacji za pomocą Winsock
// użycie: klient.exe -s:IP

#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_COUNT 5
#define DEFAULT_PORT 5000
#define DEFAULT_BUFFER 4096
#define DEFAULT_MESSAGE "Wiadomość testowa"

// tylko Visual C++
#pragma comment(lib, "ws2_32.lib")

char szServer[128], szMessage[1024];

// funkcja sposob_uzycia
void sposob_uzycia()
{
    printf("Klient.exe -s:IP\n");
    ExitProcess(1);
}

void WalidacjaLiniiPolecen(int argc, char **argv)
{
    int i;

    if (argc < 2)
    {
        sposob_uzycia();
    }
}

```

```

    for (i=1; i<argc; i++)
    {
        if (argv[i][0] == '-')
        {
            switch (tolower(argv[i][1]))
            {
                case 's':
                    if (strlen(argv[i]) > 3)
                        strcpy(szServer, &argv[i][3]);
                    break;
                default:
                    sposob_uzycia();
                    break;
            }
        }
    }
}

int main(int argc, char *argv[])
{
    WSADATA wsd;
    SOCKET sClient;
    char szBuffer[DEFAULT_BUFFER];
    int ret, i;
    struct sockaddr_in server;
    struct hostent *host = NULL;

    // linia poleceń
    WalidacjaLiniiPolecen(argc, argv);

    // inicjuj Winsock 2.2
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Błąd ładowania Winsock 2.2!\n");
        return 1;
    }

    strcpy(szMessage, DEFAULT_MESSAGE);

    // twórz gniazdo do nasłuchu połączeń klientów
    sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sClient == INVALID_SOCKET)
    {
        printf("Błąd funkcji socket(): %d\n", WSAGetLastError());
        return 1;
    }

    // wybierz interfejs
    server.sin_addr.s_addr = inet_addr(szServer);
    server.sin_family = AF_INET;
    server.sin_port = htons(DEFAULT_PORT);

    // jeśli adres nie był w postaci xxx.yyy.zzz.ttt
    // to spróbuj go wydobyć z postaci słownej
    if (server.sin_addr.s_addr == INADDR_NONE)
    {
        host = gethostbyname(szServer);
        if (host == NULL)
        {
            printf("Nie udało się wydobyć nazwy serwera: %s\n", szServer);
            return 1;
        }
        CopyMemory(&server.sin_addr, host->h_addr_list[0], host->h_length);
    }

    if (connect(sClient, (struct sockaddr *)&server, sizeof(server)) == SOCKET_ERROR)
    {

```

```

    printf("Błąd funkcji connect(): %d\n", WSAGetLastError());
    return 1;
}

// wysyłaj i odbieraj dane

for (i=0; i<DEFAULT_COUNT; i++)
{
    ret = send(sClient, szMessage, strlen(szMessage), 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        printf("Błąd funkcji send(): %d\n", WSAGetLastError());
        return 1;
    }
    printf("Wysłano %d bajtów\n", ret);

    ret = recv(sClient, szBuffer, DEFAULT_BUFFER, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        printf("Błąd funkcji recv(): %d\n", WSAGetLastError());
        return 1;
    }
    szBuffer[ret] = '\0';
    printf("RECV [%d bajtów]: '%s'\n", ret, szBuffer);
}
closesocket(sClient);

WSACleanup();
return 0;
}

```

Serwer w pętli oczekuje na klientów i po nawiązaniu połączenia tworzy nowy wątek, który zajmuje się odbieraniem komunikatów i odsyłaniem ich z powrotem (to tylko przykład!). Klient wymaga oczywiście parametru, którym jest nazwa serwera. Klient nawiązuje połączenie i wysyła do serwera komunikat, po czym czeka na jego echo.

Struktura obu programów jest dość podobna. Oba tworzą odpowiednie gniazda, przy czym:

- serwer najpierw przypisuje gniazdu adres (za pomocą funkcji `bind()`), a następnie wchodzi w tryb nasłuchu (`listen()`), gdzie zatrzymuje się czekając na połączenia klientów (`accept()`).
- klient nawiązuje połączenie z serwerem za pomocą funkcji `connect()`.

Oba programy wymieniają się krótkimi informacjami za pomocą funkcji `send()` i `recv()`. Serwer nasłuchuje na zadanym porcie, zaś po nawiązaniu połączenia z klientem tworzy nowe gniazdo i nowy wątek, po czym komunikuje się przez nowo utworzone gniazdo w nowym wątku. Dzięki temu serwer może obsługiwać wielu klientów jednocześnie, bez względu na to jak długo trwałoby obsługiwanie pojedynczego klienta. Podsumowanie schematów działania serwera i klienta znajdują się w tabelach 2.2 i 2.3.

Jak w związku z tym napisać własną przeglądarkę Internetową? Otóż kod programu klienta musiałby łączyć się do wybranego serwera do portu 80, a następnie wysyłać polecenia protokołu HTTP, na przykład **GET index.html**, po którym serwer odpowiedziałby przesyłając strumień danych, będący kodem strony **index.html**. Kod tej strony należałoby następnie sparsować i udostępnić użytkownikowi, wyławiając przy okazji hyperlinki, dzięki którym użytkownik mógłby wydawać programowi kolejne polecenia.

A jak napisać własny serwer WWW? Otóż kod programu serwera musiałby oczekiwać na połączenia klientów i reagować na dobrze sformułowane polecenia protokołu HTTP. Na przykład,

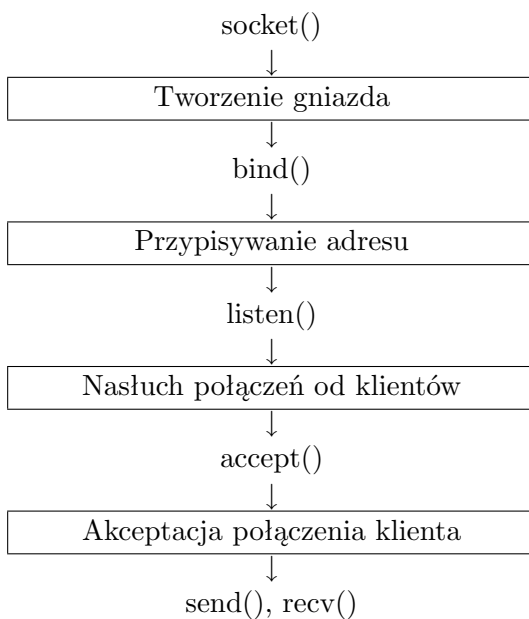


Tabela 2.2: Schemat serwera TCP korzystającego z gniazd

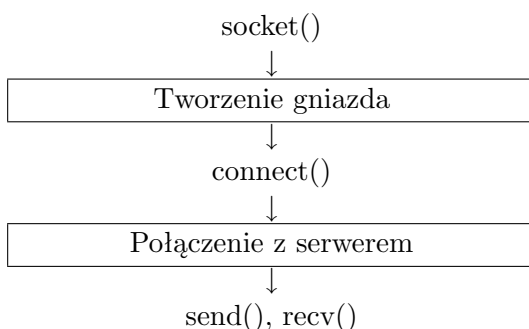


Tabela 2.3: Schemat klienta TCP korzystającego z gniazd

gdyby polecenie od klienta brzmiało **GET index.html**, serwer musiałby z dysku odczytać zawartość pliku **index.html** i wysłać go do połączonego klienta.

W praktyce napisanie dobrej przeglądarki Internetowej czy dobrego serwera WWW mogłoby być dość żmudne, jednak bardzo ograniczone możliwości można uzyskać niewielkim nakładem pracy.

#### 2.4.2.1 Gniazda asynchroniczne

Wywołania funkcji operujących na gniazdach są najczęściej *blokujące*, to znaczy że wykonanie kodu zatrzymuje się na wywołaniu funkcji na tak długo, aż działanie takiej funkcji zakończy się. Może to powodować mnóstwo kłopotów przy tworzeniu programów okienkowych - okno przestanie reagować na komunikaty, ponieważ kod zatrzyma się na wykonaniu funkcji obsługującej gniazda.

Aby poradzić sobie z tym problemem można zażądać od WinSock *asynchronicznej* obsługi gniazd, to znaczy informowania o zdarzeniach związanych z gniazdami za pomocą komunikatów. Asynchroniczny tryb pracy WinSock ustala się za pomocą funkcji



```
int WSAAsyncSelect (
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
);
```

## 2.5 Inne ważne elementy Win32API

### 2.5.1 Biblioteki ładowane dynamicznie

Oprócz statycznego łączenia bibliotek podczas linkowania programu, w Windows istnieje możliwość ładowania kodu w trakcie działania aplikacji. Wiele z dotychczas wykorzystywanych funkcji znajduje się w takich właśnie modułach: KERNEL32.DLL, GDI32.DLL, czy USER32.DLL. Tworzenie bibliotek funkcji zasadne jest tam, gdzie pewna funkcjonalność może być współdzielona przez wiele różnych modułów. Zastosowanie bibliotek oznacza zmniejszenie zapotrzebowania pamięci, ponieważ system potrafi zoptymalizować przydział pamięci dla biblioteki dołączanej dynamicznie.

Przykład biblioteki:

```
/* Wiktor Zychla 2003 */
#include <windows.h>

#ifdef __cplusplus
#define EXPORT extern "C" __declspec (dllexport)
#else
#define EXPORT __declspec (dllexport)
#endif

EXPORT int MojaFunkcja();

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwreason, LPVOID lpvReserved)
{
    return 1;
}

int MojaFunkcja()
{
    return 1;
}
```

Program, który korzysta z biblioteki:

```
/* Wiktor Zychla 2003 */
#include <windows.h>
#include <stdio.h>

const char libName[] = "abcLib.dll";

int main()
{
    typedef int(*pfPInt)();
    pfPInt myFunc;

    HMODULE hLibrary;

    hLibrary = LoadLibrary(libName);

    if (hLibrary == NULL)
    {
        MessageBox(NULL, "Bład ładowania biblioteki", "", MB_OK);
        return 1;
    }
}
```

```

myFunc = (pfPInt)GetProcAddress(hLibrary, "MojaFunkcja");

if (myFunc == NULL)
{
    MessageBox(NULL, "Bład ładowania funkcji", "", MB_OK);
    return 1;
}

char buf[80];

int res = myFunc();
sprintf( buf, "Wynik : %d", res );
MessageBox(NULL, buf, "", MB_OK);

FreeLibrary(hLibrary);

return 0;
}

```

## 2.5.2 Różne przydatne funkcje Win32API

- Typowe okno informacyjne tworzy się za pomocą funkcji

```

int MessageBox(

    HWND hWnd,          // uchwyt okna-właściciela
    LPCTSTR lpText,      // treść komunikatu
    LPCTSTR lpCaption,   // treść opisu komunikatu
    UINT uType           // styl komunikatu (dodatkowe ikony, przyciski)
);

```

- Tekst pokazywany na belce okien macierzystych oraz tekst pokazywany wewnątrz okien potomnych można odczytywać i ustawiać za pomocą funkcji

```

int GetWindowText(

    HWND hWnd, // uchwyt okna
    LPTSTR lpString, // adres bufora, który przyjmie tekst
    int nMaxCount // maksymalna ilość znaków do skopiowania
);

BOOL SetWindowText(

    HWND hWnd, // uchwyt okna
    LPCTSTR lpString // nowy tekst
);

```

Taki sam efekt można uzyskać wysyłając do okna komunikaty WM\_GETTEXT i WM\_SETTEXT.

- Stan klawiatury i myszy można odczytać w każdym momencie pracy programu, nie tylko obsługując odpowiednie komunikaty. Jest to wyjątkowo przydatne w programach interaktywnych.

```

BOOL GetKeyboardState(

    PBYTE lpKeyState // adres 256 bajtowej tablicy, która
                    // otrzyma informację o stanie klawiatury
);

BOOL GetCursorPos(

    LPPOINT lpPoint // pozycja kursora myszy
);

```

- Niektóre funkcje API korzystają ze współrzędnych punktu odniesionych do punktu w lewym górnym rogu okna, inne ze współrzędnych ekranu. Przeliczanie współrzędnych między tymi układami odniesienia jest łatwe dzięki funkcjom

```

BOOL ClientToScreen(

    HWND hWnd,          // uchwyt okna
    LPPOINT lpPoint      // punkt we współrzędnych obszaru klienckiego
);

BOOL ScreenToClient(

    HWND hWnd,          // uchwyt okna
    LPPOINT lpPoint      // punkt we współrzędnych ekranu
);

```

### 2.5.3 Zegary

Aplikacje DOSowe mogły polegać jedynie na przerwaniu BIOSu, które niezależnie od prędkości procesora pojawiało się regularnie 18.2 raza na sekundę. Windows udostępnia mechanizm zegarów systemowych, które zajmują się informowaniem aplikacji o upływie jakiegoś okresu czasu. Aplikacja tworzy zegar za pomocą funkcji

```

UINT SetTimer(

    HWND hWnd, // uchwyt okna, które będzie otrzymywać komunikaty zegara
    UINT nIDEvent, // identyfikator zegara
    UINT uElapse, // interwał czasowy
    TIMERPROC lpTimerFunc // adres funkcji obsługi zdarzenia zegara
);

```

Istnieją dwie możliwości użycia tej funkcji:

1. Podanie pustego wskaźnika na funkcję obsługi, na przykład `SetTimer(hWnd, 1, 1000, NULL)`, spowoduje że okno aplikacji będzie otrzymywało komunikaty `WM_TIMER` w ustalonych odstępach czasu. System nie przysyła aplikacji komunikatów `WM_TIMER` częściej niż aplikacja jest w stanie je obsłużyć (18 razy na sekundę w przypadku Windows 98 i około 100 razy na sekundę w Windows NT).
2. Podanie wskaźnika na istniejącą w kodzie programu funkcję postaci

```

VOID CALLBACK TimerProc (HWND hWnd, UINT message, UINT iTimerID, DWORD dwTime)
{
    [obsłuż komunikat WM_TIMER]
}

```

spowoduje wysyłanie komunikatów `WM_TIMER` do tej funkcji. Parametr *iTimerID* odpowiada za identyfikator zegara, zaś *dwTime* jest równy bieżącej wartości funkcji `GetTickCount()`.

Dobłą praktyką jest *explicite* niszczenie zegara przy zakończeniu aplikacji za pomocą funkcji

```

BOOL KillTimer(

    HWND hWnd, // uchwyt okna które instalowało zegar
    UINT uIDEvent // identyfikator zegara
);

```

Poniższy przykład, oprócz zegara, pokazuje również sposób użycia kilku nieomawianych do tej pory funkcji GDI.



Rysunek 2.5: Zegar elektroniczny, bardzo łatwo byłoby zrobić z niego budzik

```

/*
 *
 * Zegar
 *
 */
#include <windows.h>

#define ID_TIMER    1

int xSize, ySize;

/* Deklaracje wyprzedzające */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
void PaintCurrentTime( HDC hdc );

/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwyt okna */
    MSG messages;        /* Komunikaty okna */
    WNDCLASSEX wincl;    /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance     = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc   = WindowProcedure;    // wskaźnik na funkcję obsługi okna
    wincl.style         = CS_DBLCLKS;
    wincl.cbSize        = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName  = NULL;
    wincl.cbClsExtra    = 0;
    wincl.cbWndExtra    = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */

```

```

hwnd = CreateWindowEx(
    0,
    szClassName,
    "Przykład",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    512,
    256,
    HWND_DESKTOP,
    NULL,
    hThisInstance,
    NULL
);

ShowWindow(hwnd, nFunsterStil);
/* Pętla obsługi komunikatów */
while(GetMessage(&messages, NULL, 0, 0))
{
    /* Tłumacz kody rozszerzone */
    TranslateMessage(&messages);
    /* Obsłuż komunikat */
    DispatchMessage(&messages);
}

/* Zwróć parametr podany w PostQuitMessage( ) */
return messages.wParam;
}

/* Tę funkcję woła DispatchMessage( ) */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    RECT r;
    HDC hdc;

    switch (message)
    {
        case WM_CREATE:
            SetTimer(hwnd, ID_TIMER, 500, NULL);
            break;
        case WM_TIMER:
            GetClientRect( hwnd, &r );
            InvalidateRect( hwnd, &r, 1 );
            break;
        case WM_PAINT:
            hdc = BeginPaint( hwnd, &ps );
            PaintCurrentTime( hdc );
            EndPaint( hwnd, &ps );
            break;
        case WM_SIZE:
            xSize = LOWORD(lParam);
            ySize = HIWORD(lParam);
            break;
        case WM_DESTROY:
            KillTimer(hwnd, ID_TIMER);
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

/* Maluj aktualny czas na ekranie */
void PaintCurrentTime( HDC hdc )
{
    char sTime[256];
    SYSTEMTIME time;

```

```

HFONT      hFont;
SIZE       size;

// pobierz aktualny czas systemowy
// i konwertuj go na zadany format
GetSystemTime( &time );
GetTimeFormat( LOCALE_SYSTEM_DEFAULT, 0, &time,
               "HH':'mm':'ss", sTime, 256 );

// twórz font logiczny
hFont = CreateFont( 112, 0, 0, 0,
                   FW_NORMAL, 0, 0, 0,
                   DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
                   CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                   DEFAULT_PITCH, "LED" );
SelectObject( hdc, hFont );

// licz rozmiar tekstu
GetTextExtentPoint32( hdc, sTime, lstrlen( sTime ), &size );

// rozpocznij ścieżkę graficzną
BeginPath( hdc );

SetBkMode( hdc, TRANSPARENT );
TextOut( hdc, (xSize-size.cx)/2, (ySize-size.cy)/2,
         sTime, strlen( sTime ) );

// zakończ ścieżkę
EndPath( hdc );

// rysuj ścieżkę odpowiednim pędzlem
SelectObject( hdc, CreateHatchBrush( HS_DIAGCROSS, RGB( 0, 0, 255 ) ) );
SetBkColor( hdc, RGB( 255, 0, 0 ) );
SetBkMode( hdc, OPAQUE );
StrokeAndFillPath( hdc );

DeleteObject( SelectObject( hdc, GetStockObject( WHITE_BRUSH ) ) );
SelectObject( hdc, GetStockObject( SYSTEM_FONT ) );
DeleteObject( hFont );
}

```

### 2.5.4 Okna dialogowe

Aplikacja złożona z jednego okna dialogowego jest oczywiście rzadkością. Zwykle zaprojektowanie interfejsu użytkownika odpowiadającego modelowanemu problemowi wymaga od kilku do nawet kilkuset różnych okien dialogowych. Programista tworzy nowe okno dialogowe za pomocą jednej z funkcji:

```

int DialogBox(

    HINSTANCE hInstance, // instancja aplikacji
    LPCTSTR lpTemplate, // szablon okna
    HWND hWndParent, // okno macierzyste
    DLGPROC lpDialogFunc // funkcja obsługi okna
);

HWND CreateDialog(

    HINSTANCE hInstance, // instancja aplikacji
    LPCTSTR lpTemplate, // szablon okna
    HWND hWndParent, // okno macierzyste
    DLGPROC lpDialogFunc // funkcja obsługi okna
);

```

Funkcja `DialogBox()` tworzy tzw. *modalne* okno dialogowe, tzn. takie, które nie pozwala użytkownikowi na uaktywnienie żadnego innego okna aplikacji do czasu zamknięcia okna dialogowego.

Funkcja `CreateDialog()` tworzy niemodalne okno dialogowe, tzn. okno z własną, niezależną pętlą obsługi komunikatów.

Obie z tych funkcji oczekują wskazania odpowiedniego szablonu okna. Szablon taki dodaje się do zasobów aplikacji (zwykle ma rozszerzenie `*.rc`). Szablony okien dialogowych mają swoją specjalną składnię i choć można wykonstruować okno bez dodawania szablonu do zasobów (szablon tworzy się dynamicznie, a następnie korzysta się z funkcji `CreateDialogIndirect()` lub `DialogBoxIndirect()`, zaś okna potomne dodaje się przy obsłudze komunikatu `WM_INITDIALOG`), to korzystanie z nich znacznie ułatwia cały proces.

```
/* EX.C */
/*
 *
 * Okna dialogowe
 *
 */
#include <windows.h>

/* Deklaracja wyprzedzająca: funkcja obsługi okna */
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);
BOOL DialogBoxWindowProcedure(HWND, UINT, WPARAM, LPARAM);
void CreateMyMenu( HWND hwnd );
/* Nazwa klasy okna */
char szClassName[] = "PRZYKLAD";

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;           /* Uchwył okna */
    MSG messages;         /* Komunikaty okna */
    WNDCLASSEX wincl;     /* Struktura klasy okna */

    /* Klasa okna */
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName  = szClassName;
    wincl.lpfnWndProc    = WindowProcedure;  // wskaźnik na funkcję obsługi okna
    wincl.style          = CS_DBLCLKS;
    wincl.cbSize         = sizeof(WNDCLASSEX);

    /* Domyślna ikona i wskaźnik myszy */
    wincl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    wincl.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wincl.lpszMenuName   = NULL;
    wincl.cbClsExtra     = 0;
    wincl.cbWndExtra     = 0;
    /* Jasnoszare tło */
    wincl.hbrBackground  = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

    /* Rejestruj klasę okna */
    if(!RegisterClassEx(&wincl)) return 0;

    /* Twórz okno */
    hwnd = CreateWindowEx(
        0,
        szClassName,
        "Przykład",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        512,
        512,
        HWND_DESKTOP,
        NULL,
        hThisInstance,
        NULL
    );

    CreateMyMenu( hwnd );
}
```





```

    }
    return 0;
}

/* EX.RC */
#include <windows.h>

501 DIALOG 32, 32, 180, 40
STYLE WS_VISIBLE | WS_SYSMENU | WS_POPUP | WS_CAPTION | DS_MODALFRAME
CAPTION "Okno dialogowe"
FONT 12,"Times New Roman"
BEGIN
    DEFPUSHBUTTON    "OK",IDOK,66,80,50,14
    CTEXT            "0 programie...",-1,40,12,100,8
END

```

Zauważmy, że funkcja obsługi komunikatów nie jest zwykłą funkcją obsługi komunikatów okna. Tak naprawdę obsługą komunikatów okna dialogowego zajmuje się domyślna funkcja obsługi komunikatów okien dialogowych, istniejąca w systemie

```

BOOL CALLBACK DialogProc(

    HWND hwndDlg, // handle to dialog box
    UINT uMsg, // message
    WPARAM wParam, // first message parameter
    LPARAM lParam // second message parameter
);

```

i to ona przekazuje komunikaty do funkcji obsługi komunikatów w oknie dialogowym.

Istnieją cztery zasadnicze różnice między zwykłą funkcją obsługi okna, a funkcją obsługi okna dialogowego:

- zwykła funkcja obsługi okna zwraca wartość typu LRESULT, funkcja obsługi okna dialogowego zwraca BOOL
- w przypadku nieobsługiwanego jakiegoś komunikatu zwykła funkcja obsługi okna woła domyślną funkcję obsługi okien (*DefWindowProc*), zaś funkcja obsługi okna dialogowego zwraca wartość TRUE kiedy obsługuje jakiś komunikat i FALSE jeśli go nie obsługuje
- funkcja obsługi okna dialogowego nie musi obsługiwać komunikatów WM\_PAINT i WM\_DESTROY
- funkcja obsługi okna dialogowego nie otrzymuje komunikatu WM\_CREATE, tylko WM\_INITDIALOG

Szablon okna dialogowego, oprócz opisu stylu okna i cech okien potomnych może zawierać m.in.

- wskazanie menu (MENU menu-name)
- wskazanie czcionki (FONT font)
- wskazanie klasy (CLASS "klasa")

Istnieje ponadto możliwość skorzystania z typowych okien dialogowych. Każde z tych okien po zamknięciu zwraca zestaw parametrów niezbędnych do zidentyfikowania wyboru użytkownika.

- Okna do wyboru nazwy pliku

```

BOOL GetOpenFileName(

    LPOPENFILENAME lpofn
);

```

```

BOOL GetSaveFileName(
    LPOpenFileName lpofn
);

```

- Typowe okno wyboru koloru

```

BOOL ChooseColor(
    LPCHOOSECOLOR lpcc
);

```

- Typowe okno wyboru czcionki

```

BOOL ChooseFont(
    LPCHOOSEFONT lpcf
);

```

- Typowe okno ustalania parametrów drukowania

```

BOOL PrintDlg(
    LPPRINTDLG lppd
);

```

Przykład użycia okna do wyboru koloru:

```

#include <windows.h>
#include <commdlg.h>
#include <stdio.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR szCmdLine, int iCmdShow)
{
    char buf[80];
    char *msgTpl = "Wybrano kolor o składowych: [%d, %d, %d]";

    static CHOOSECOLOR cc ;
    static COLORREF crCustColors[16] ;

    cc.lStructSize = sizeof (CHOOSECOLOR) ;
    cc.hwndOwner = NULL ;
    cc.hInstance = NULL ;
    cc.rgbResult = RGB (0x80, 0x80, 0x80) ;
    cc.lpCustColors = crCustColors ;
    cc.Flags = CC_RGBINIT | CC_FULLOPEN ;
    cc.lCustData = 0 ;
    cc.lpfnHook = NULL ;
    cc.lpTemplateName = NULL ;

    if (ChooseColor (&cc))
    {
        sprintf( buf, msgTpl,
            GetRValue( cc.rgbResult ),
            GetGValue( cc.rgbResult ),
            GetBValue( cc.rgbResult ) );
        MessageBox( 0, buf, "", 0 );
    }
}

```

#### 2.5.4.1 Powłoka systemu

Programista ma dostęp do powłoki systemu dzięki funkcji

```
HINSTANCE ShellExecute(  
    HWND hwnd, // okno macierzyste  
    LPCTSTR lpOperation, // rodzaj operacji  
    LPCTSTR lpFile, // nazwa pliku  
    LPCTSTR lpParameters, // parametry  
    LPCTSTR lpDirectory, // domyślny katalog  
    INT nShowCmd // flaga otwarcia okna  
);
```

Powłoka potrafi wykonać na zadanym pliku kilka rodzajów operacji:

- "open"
- "print"
- "explore"
- "properties"



## Rozdział 3

# .NET

### 3.1 Programowanie obiektowe

Programowanie obiektowe jest naturalnym rozwinięciem programowania strukturalnego, możliwego chociażby w języku C. Program strukturalny składa się z szeregu *struktur danych* oraz *funkcji* operujących na tych strukturach. Struktury danych reprezentują *wiedzę* (dane), funkcje reprezentują *algorytmy*.

```
struct Foo
{
    int Bar;
} foo;

int DoSomething( struct Foo f )
{
    return f.Bar + 1;
}

int main()
{
    printf( "%d", DoSomething( foo ) );
}
```

Na język obiektowy można patrzeć jak na pewną *konwencję programowania strukturalnego*, prowadzącą do specyficznego uporządkowania kodu. Ta konwencja wypływa naprzeciw pewnej obserwacji - dość często jest tak, że algorytm można przypisać jednej określonej strukturze danych na której operuje.

Takie jednoznaczne przypisanie prowadzi do powstania pojęcia *klasy*, która łączy ze sobą dane i algorytmy w taki szczególny sposób, w którym funkcja operująca na danych **nie musi** otrzymywać tych danych w postaci jawnego argumentu (`struct Foo f` w powyższym przykładzie), ponieważ ma do danych "swojego" (tego, który w wersji nieobektowej musiałby być przekazany jako jawny argument) obiektu zawsze dostęp za pomocą zarezerwowanego słowa kluczowego `this`:

```
class Foo
{
    int Bar;

    int DoSomething()
    {
        return this.Bar + 1;
    }
};

int main()
{
}
```

```

Foo foo;
printf( "%d", DoSomething( foo ) );
}

```

Skądinąd, jest to dość interesujące od strony technicznej, bowiem oznacza że metody klasy, które przynależą do wystąpień obiektów (metody niestatyczne), mają w językach obiektowych ten jeszcze jeden jeden, niejawny argument i że rzeczywista liczba argumentów metody typu

```

class Foo
{
    void DoSomething( int a, int b )
    {
    }
};

```

to nie dwa (a i b) ale trzy (**this**, a i b), co znaczy że tak naprawdę to jest to właśnie

```

struct Foo
{
}

void DoSomething( struct Foo this, int a, int b )
{
}

```

Ta konwencja ma wiele ważnych pozytywnych konsekwencji, wśród których najważniejszą wydaje się zmiana sposobu myślenia - o klasie zaczyna się myśleć jako o abstrakcie pewnego **pojęcia**, często odpowiadającego jakiemuś pojęciu ze świata rzeczywistego. Można o tych pojęciach mówić bardziej formalnie, wprowadzać elementy *analizy obiektowej* i budować całe *modele pojęciowe*, które z kolei można mniej lub bardziej formalnie analizować itd.

Istnieją jednak również oczywiście takie konsekwencje przyjęcia konwencji obiektowej, które (przynajmniej przez pewien czas) mogą wydawać się niewygodne.

Przede wszystkim - w przypadku funkcji/algoritmów operujących na danych więcej niż jednej struktury, wybór tej właśnie jednej do której algorytm zostanie przypisany, nie wydaje się możliwy do formalizacji bez odniesienia się do konkretnego przypadku. Mówiąc inaczej, mając

```

struct Foo
{
    int foo;
};

struct Bar
{
    int bar;
}

int DoSomething( struct Foo f, struct Bar b )
{
    return f.foo + b.bar;
}

```

niekoniecznie łatwo wyczuć czy funkcja **DoSomething** bardziej należy do **Foo** czy do **Bar** i do której struktury przywiązać tę funkcję dokonując konwersji kodu strukturalnego do wymogów konwencji obiektowej.

Nie wgłębiając się bardzo w ten problem, powiedzmy tylko że nie ma takich reguł projektowania obiektowego, które rozstrzygałyby tego rodzaju wątpliwości obiektywnie i deterministycznie. W takich przypadkach jak ten, przypisanie funkcji do jednej tych z kilku klas, które są argumentami funkcji, bywa zależne od "stylu" programisty i dla kodu

```

struct Ser
{
}

struct Mysz
{
}

void MyszZjadaSer( Mysz mysz, Ser ser )
{
}

```

następujące dwa podejścia są równoważne (jeśli nie ma żadnych innych dodatkowych przesłanek za którymkolwiek z nich):

```

class Ser
{
}

class Mysz
{
    void Zjada( Ser ser );
}

oraz

class Mysz
{
}

class Ser
{
    void JestZjadanyPrzez( Mysz mysz );
}

```

Kolejna oczywiste ograniczenie paradygmatu obiektowego pojawia się w językach *ściśle* obiektowych, takich w których **każda funkcja musi** być przypisana do **jakiejs** klasy. Tak jest na przykład w Javie czy C#, z kolei C++ nie nakłada takich ograniczeń. U niedoświadczonych programistów to wymaganie rodzi pewne wątpliwości, no bo jakże to taką prostą metodę przypisać do jakiejś struktury:

```

int Sum( int x, int y )
{
    return x + y;
}

```

?

Choć wydaje się, że kandydatem na bycie "właścicielem" takiego algorytmu ma szansę być struktura (klasa) `int`, w wielu językach zwyczajnie nie ma możliwości rozszerzania klas biblioteki standardowej o własne, niestandardowe metody. W takich przypadkach zwykle stosuje się regułę **Pure Fabrication**, jedną z formalnych reguł projektowania obiektowego GRASP, która mówi tyle że w przypadku algorytmów które nie przynależą w naturalny sposób do żadnej struktury należy po prostu utworzyć dla nich osobną klasę w taki sposób, żeby nie naruszyć dwóch innych zasad GRASP, Low Coupling i High Cohesion.

W tym konkretnym przypadku mogłoby to być na przykład:

```

class Math
{
    static int Sum( int x, int y )
    {
        return x + y;
    }
}

```

gdzie `static` dodatkowo podkreśla brak zależności od instancji (wystąpienia) danej klasy.

## 3.2 C# - podstawowe elementy języka

Język C# jest językiem obiektowym. Z wcześniejszych języków najbardziej przypomina Javę, jednak kilka istotnych ułomności Javy<sup>1</sup> zostało w C# skorygowanych, czyniąc C# jeszcze przyjemniejszym w zastosowaniu.

Autorzy projektu języka wyraźnie i często podkreślają, że wybranie takiej a nie innej składni podstawowych konstrukcji języka oznacza nie tylko dużą łatwość pracy dla programistów znających wcześniej Javę czy C++, ale oznacza również możliwość łatwej konwersji już istniejącego kodu.

Jak już wcześniej powiedziano, kompilator C# jest częścią środowiska uruchomieniowego .NET Framework. Początkowo do tworzenia i uruchamiania programów nadawał się każdy system Windows, od Windows 98. Również niektóre darmowe środowiska developerskie można było uruchomić na Windows 98 i wyższych.

Aktualnie, dla najnowszych wersji środowiska .NET Framework wymagany jest co najmniej Windows 7 i niewykluczone że po zakończeniu wsparcia dla Windows 7 i Windows 8, w przyszłości wspierane będą wyższe wersje systemu.

Kompilator C# tworzy kod wynikowy w języku pośrednim, zwanym IL i umieszcza go w module binarnym. Moduł może być aplikacją konsolową lub okienkową (pliki \*.exe) lub biblioteką klas (pliki \*.dll). .NET Framework traktuje każdy moduł w sposób jednakowy - podczas uruchamiania kodu zawartego w module, uruchamiany jest kompilator JIT (*Just-In-Time*), który tworzy kod natywny systemu operacyjnego, po czym uruchamia ten kod jak zwykłą aplikację w systemie operacyjnym. Dzięki temu prędkość działania aplikacji C#-owej w systemie jest porównywalna z prędkością aplikacji napisanej w C++.

### 3.2.1 Pierwszy program

Zgodnie z tradycją rozpoczniemy od najprostszego programu C#-owego i przeanalizujemy elementy jego kodu.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            Console.WriteLine( "Pierwszy program w C#" );
        }
    }
}
```

Kompilator przywołany z linii poleceń przedstawia się i kompiluje program. Kod wynikowy powyższego programu zajmuje 3072 bajty.

```
C:\Examples>csc.exe example.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
```

---

<sup>1</sup>Najbardziej jaskrawy przykład to brak w Javie typu funkcyjnego, umożliwiającego przekazywanie funkcji do innych funkcji jako argumentów oraz zwracanie funkcji z funkcji jako wartości. Brak takiego mechanizmu w zasadzie przez wiele lat wykluczał elementy programowania funkcyjnego. W Java 8 wprowadzono mechanizm, który podczas kompilacji rozwija tzw. *lambda wyrażenia* (czyli skrócone postaci funkcji anonimowych) do klas z jedną metodą, implementujących pewien ustalony interfejs. Choć w ten sposób syntaktycznie możliwe jest przekazanie funkcji do funkcji jako argument, w rzeczywistości do funkcji nie trafia bezpośrednio funkcja, ale właśnie obiekt automatycznie utworzonej klasy implementującej ten konkretny interfejs.



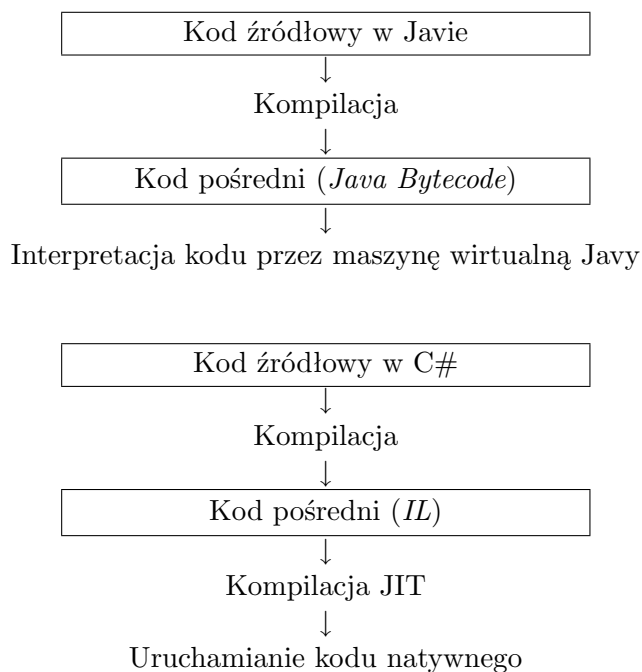


Tabela 3.1: Schematy uruchamiania kodów Javy i C# w systemie operacyjnym

Copyright (C) Microsoft Corporation 2001. All rights reserved.

Tak jak w przypadku każdego języka obiektowego, również kod programu C#-owego składa się z klas. Podobnie jak w Javie jedna z klas musi zawierać publiczną statyczną metodę *Main*, od której rozpoczyna się wykonanie programu. Możliwe jest zdefiniowanie metody *Main* w więcej niż jednej klasie, jednak wtedy należy explicite podać kompilatorowi nazwę klasy zawierającej tę metodę *Main*, która ma zostać uwzględniona jako główna metoda aplikacji.

```
C:\Examples>csc.exe example.cs /main:Example.CMain
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

Programista ma do dyspozycji kilkanaście typów prostych wspólnych dla wszystkich aplikacji platformy .NET. Deklarując zmienne można używać pełnej nazwy typu lub skrótu jego nazwy

Nazwa typu	Skrót nazwy	Opis
System.Object	object	Klasa bazowa dla wszystkich typów
System.String	string	Napis
System.Sbyte	sbyte	8-bitowa liczba całkowita ze znakiem
System.Byte	byte	8-bitowa liczba całkowita bez znaku
System.Int16	short	16-bitowa liczba całkowita ze znakiem
System.UInt16	ushort	16-bitowa liczba całkowita bez znaku
System.Int32	int	32-bitowa liczba całkowita ze znakiem
System.UInt32	uint	32-bitowa liczba całkowita bez znaku
System.Int64	long	64-bitowa liczba całkowita ze znakiem
System.UInt64	ulong	64-bitowa liczba całkowita bez znaku
System.Char	char	16-bitowy znak Unicode
System.Single	float	32-bitowa liczba zmiennoprzecinkowa
System.Double	double	64-bitowa liczba zmiennoprzecinkowa
System.Boolean	bool	wartość logiczna (true/false)
System.Decimal	decimal	128-bitowa wartość numeryczna

Oprócz typów prostych biblioteka standardowa zawiera setki typów złożonych. Poznanie tych bardziej użytecznych jest jednym z zadań jakie czeka programistę chcącego nauczyć się biegle programować aplikacje na platformie .NET.

### 3.2.2 Struktura kodu, operatory

Kod C#-owy najbardziej przypomina kod Javy. Wszystkie podstawowe konstrukcje językowe takie jak deklaracje zmiennych, operatory, instrukcje warunkowe czy pętle działają dokładnie tak jak w Javie. Dzięki temu programiści znający C, C++ czy Javę bardzo szybko odnajdą się w nowym języku.

Przykład:

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            int i, j=0, n;

            Console.Write( "Podaj liczbę naturalną: " );
            n = int.Parse( Console.ReadLine() );

            for ( i=1; i<n; i++ )
                j+=i;

            Console.WriteLine( "Suma wynosi " + j.ToString() );
        }
    }
}
```

Kompilator dość restrykcyjnie traktuje powszechnie popełniane przez programistów pomyłki, zwykle sygnalizując błąd tam, gdzie kompilator C czy C++ poprzestaje na ostrzeżeniu. Na przykład w powyższym przykładzie deklaracja

```
int i, j, n; // brak przypisania j=0
```

spowoduje błąd kompilacji

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

```
example.cs(17,9): error CS0165: Use of unassigned local variable 'j'
```

W przeciwieństwie do C, kompilator C# nie pozwala redefiniować zmiennych na kolejnych poziomach zagnieżdżenia kodu, uznano bowiem że jest to źródłem zbyt dużej ilości niezamierzonych pomyłek.

```
/* Wiktor Zychla, 2003 */
using System;
```

```
class Example
{
    public static void Main()
    {
        bool b=true;
        int i;

        while ( b )
        {
            int i;
        }
    }
}
```

```
example.cs(12,8): error CS0136: A local variable named 'i' cannot be declared in
this scope because it would give a different meaning to 'i', which is
already used in a 'parent or current' scope to denote something else
```

Niespodziewanie, lecz konsekwentnie, taka konstrukcja nie jest możliwa również wtedy, gdy deklaracja bardziej zagnieżdżona *poprzedza* deklarację mniej zagnieżdżoną.

```
/* Wiktor Zychla, 2003 */
using System;
```

```
class Example
{
    public static void Main()
    {
        bool b=true;
        while ( b )
        {
            int i;
        }
        int i;
    }
}
```

### 3.2.3 System typów, model obiektowy

W obiektowych językach programowania zwykle funkcjonują dwie rozłączne klasy bytów, na których można operować: typy proste (liczby całkowite, zmiennoprzecinkowe, napisy) oraz typy złożone (klasy). Istnienie dwóch rozłącznych światów rodzi mnóstwo problemów i niejednokrotnie zmusza programistę do pisania "brzydkiego" kodu. Na przykład w sytuacji, kiedy potrzebna jest funkcja operująca na wartości dowolnego typu, istnienie typów prostych zmusza programistę do przeciążania tej funkcji tyle razy ile różnych jej wariantów będzie potrzebował. W C++ pewnym sposobem na przezwyciężanie takich trudności są szablony, jednak nie istnieje sposób

na stosowanie szablonu do typu nieznanego kompilatorowi podczas kompilacji. Taka cecha języka sprawia, że trudno nazwać C++ językiem w pełni obiekowym. Nawet Java dzieli typy na proste i złożone, bowiem stosowanie typów prostych znacząco poprawia wydajność kodu.

Model obiekowy C# to model z pojedynczym dziedziczeniem<sup>2</sup>. Zakłada się istnienie jednej wspólnej klasy *object* dla wszystkich obiektów. Choć nadal istnieje podział na typy proste i typy złożone, to z punktu widzenia systemu typów wszystko jest obiektem, co więcej typ obiektu jest możliwy do odzyskania w trakcie działania programu.

Dzięki takiej konstrukcji systemu typów możliwe jest zdefiniowanie pewnej funkcjonalności już na poziomie klasy *object*. Ta funkcjonalność jest dziedziczona na klasy potomne. Najważniejsze dwie metody wirtualne zdefiniowane w klasie *object* to:

**string ToString()** Domyślnie ta metoda zwraca nazwę typu obiektu. Przeciążona może zwracać opis zawartości obiektu w postaci przyjaznej dla użytkownika.

**Type GetType()** Zwraca zmienną typu *Type*, która określa typ obiektu.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class Klasa1 {}
    public class Klasa2
    {
        public override string ToString()
        {
            return "Jestem obiektem klasy Klasa2";
        }
    }

    public class CMain
    {
        public static void Main()
        {
            Klasa1 k1 = new Klasa1();
            Klasa2 k2 = new Klasa2();

            Console.WriteLine( k1.ToString() );
            Console.WriteLine( k2.ToString() );

            Console.WriteLine( k1.GetType().ToString() );
            Console.WriteLine( k2.GetType().ToString() );

            Console.WriteLine( k1.GetType().GetType().ToString() );
        }
    }
}

C:\Example>example.exe
Example.Klasa1
Jestem obiektem klasy Klasa2
Example.Klasa1
Example.Klasa2
System.RuntimeType
```

Jak widać wartość wyrażenia

```
object_value.GetType()
```

sama jest wartością typu *Type*, można więc zapytać o jej typ

```
object_value.GetType().GetType()
```

---

<sup>2</sup>Sposobem na pokonanie ograniczeń pojedynczego dziedziczenia są tzw. *interfejsy*.

i poprosić o jego reprezentację

```
object_value.GetType().GetType().ToString()
```

Zalety zunifikowanego systemu typów (*CTS, Common Type System*) to jednak nie tylko elegancja języka. CTS gra główną rolę przy łączeniu przez środowisko uruchomieniowe kodu napisanego w różnych językach. Każdy kompilator musi umieć posługiwać się zdefiniowanymi w CTS typami prostymi, musi także definiować własne typy wpasowując je w określoną przez CTS hierarchię typów. Sam CTS jest częścią szerszej specyfikacji zwanej CLS (*Common Language Specification*), która dodatkowo określa inne istotne wymagania dla kompilatora (takie jak sposób zarządzania pamięcią, obsługi wyjątków).

Największą zaletą CTS jest jednak bezpieczeństwo jakie oferuje tak zaprojektowany system typów.

- Typ każdego obiektu może być jednoznacznie określony, nawet dynamicznie, czyli w trakcie działania programu.
- Nie ma możliwości oszukania systemu typów przez próbę przekonania go że jakiś obiekt ma inny typ niż jego prawdziwy typ.
- Dostęp do składowych każdego obiektu (publiczny, prywatny) jest określony na poziomie definicji klasy i nie jest możliwe działanie wbrew określonym prawom dostępu. To znaczy, że na przykład jeśli składowa klasy jest prywatna, to system typów i środowisko uruchomieniowe chronią ją przed dostępem z zewnątrz<sup>3</sup>.

### 3.2.4 Typy proste a typy referencyjne, boxing i unboxing

System typów, w którym każdy byt jest obiektem nie jest pomysłem nowym. Tak jest na przykład w SmallTalku. Niestety, SmallTalk płaci za tę cechę języka cenę efektywności - tam gdzie mamy do czynienia na przykład z liczbami całkowitymi czy zmiennoprzecinkowymi (na których procesor potrafi przecież dokonywać szybkich obliczeń) konieczność opakowywania ich w struktury obiektowe dramatycznie zmniejsza wydajność.

Projektanci języka C# rozwiązali ten problem dzieląc świat wszystkich obiektów na dwie kategorie: obiekty proste i obiekty referencyjne. Obiekty proste są tworzone na stosie i reprezentowane są przez aktualną wartość obiektu. Obiekty proste nie mogą więc mieć wartości *null*, która związana jest z pustym wskazaniem - one zawsze mają wartość. Obiekty referencyjne tworzone są na sterwie, zaś na stosie znajduje się referencja do obiektu na sterwie.



Obiektami prostymi są w C# na przykład arytmetyczne typy wbudowane, enumeracje i typy zdefiniowane jako struktury (patrz 3.2.5). Obiektami referencyjnymi są pozostałe klasy, tablice, delegaci, interfejsy.

Obiekt typu prostego jest inicjowany bezpośrednio po zadeklarowaniu. Oznacza to, że nie ma potrzeby jawnego wywołania konstruktora, na przykład:

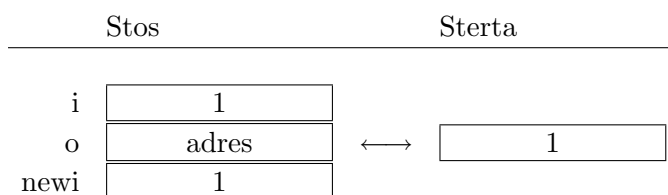
<sup>3</sup>W C++ kwalifikatory dostępu to tak naprawdę mechanizm którym programista chroni się sam przed sobą. Można bowiem zrzucać obiekt klasy z polami prywatnymi na inną klasę z polami publicznymi, w ten sposób obchodząc mechanizm kwalifikatorów.

```
int i;

for ( i=0; i<20; i++ )
    ...
```

Programista może przekształcać obiekty o typach prostych do postaci referencyjnej za pomocą tzw. opakowywania (ang. *boxing*), a następnie z powrotem do postaci prostej (odpakowywanie, *unboxing*).

```
int    i = 1;
object o = i;
int    newi = (int)o;
```



### 3.2.5 Klasy

Pojęcie klasy jest fundamentalnym pojęciem programowania obiektowego. Pojedyncza klasa opisuje cechy jakiegoś konkretnego obiektu - jego właściwości i możliwe akcje.

Najprostsza definicja klasy w C# mogłaby wyglądać tak:

```
class cOsoba
{
    public int wiek;
}
```

Taka definicja pozwala konstruować obiekty opisanego typu i odwoływać się do jedynego *pola* obiektów tej klasy:

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class cOsoba
    {
        public int wiek;
    }

    public class CMain
    {
        public static void Main()
        {
            cOsoba o = new cOsoba();
            o.wiek = 13;

            Console.WriteLine( "wiek osoby to " + o.wiek.ToString() + " lat" );
        }
    }
}
```

Ponieważ w C# nie ma znanych z C i C++ plików nagłówkowych, w których umieszczano deklaracje funkcji, cała definicja klasy musi znajdować się w jednym pliku. Nie ma również potrzeby pisania deklaracji wyprzedzających - klasa może być używana w dowolnym miejscu kodu bez względu na miejsce jej definicji, na przykład:

```
/* Wiktor Zychla, 2003 */
using System;

class Example
{
    class A
    {
        B b;
    }
    class B
    {
        A a;
    }
    public static void Main()
    {
    }
}
```

Definicja klasy składa się z definicji *elementów składowych* określających jej funkcjonalność. W C# istnieje 7 możliwych rodzajów elementów składowych:

**pol** Pole jest elementem składowym klasy, który przechowuje jakąś wartość.

**metody** Metoda jest funkcją, która najczęściej w jakiś sposób operuje na wartościach przechowywanych w polach.

**właściwości (propercje)** Propercje są metodami, które z punktu widzenia klientów klasy wyglądają jak pola.

**stałe** Stałe są polami, których wartość nie może ulegać zmianom.

**indeksery** Indeksery są konstrukcjami językowymi, które pozwalają na dostęp do danych klasy tak, jakby były one umieszczone w tablicy, choć wewnętrzna reprezentacja może być zupełnie inna.

**zdarzenia** Zdarzenia powodują wykonywanie się jakiegoś kodu. Zdarzenia mają swoje listy słuchaczy, a zaistnienie zdarzenia powoduje wykonanie wszystkich funkcji na liście słuchaczy.

**operatory** W C# istnieje możliwość przeciążania kilku standardowych operatorów.

Każdy element składowy (z drobnymi wyjątkami) może być opatrzony odpowiednim *kwalifikatorem dostępu*.

**public** Brak ograniczeń w dostępie do składowej.

**protected** Dostęp jest ograniczony do składowych danej klasy i klas potomnych.

**internal** Dostęp jest ograniczony do bieżącego modułu.

**protected internal** Dostęp jest ograniczony do składowych danej klasy i klas potomnych bieżącego modułu.

**private** Dostęp jest ograniczony do składowych danej klasy.

W przeciwieństwie do C++ każda składowa klasy musi być jawnie opatrzona odpowiednim kwalifikatorem dostępu, zaś jego brak oznacza domyślnie kwalifikator *private*, na przykład:

```

/* Wiktor Zychla, 2003 */
using System;

class Example
{
    class A
    {
        public string s;
        int i;
    }
    public static void Main()
    {
        A a = new A();
        a.s = "Ala ma kota";
        a.i = 5; // błąd
    }
}

example.cs(14,2): error CS0122: 'Example.A.i' is inaccessible due to its
    protection level

```

### 3.2.5.1 Pola

Projektując obiekty dla swojej aplikacji, programista zwykle stoi przed zadaniem zbudowania zbioru klas tak, aby jak najlepiej opisać problem, który rozwiązywać ma aplikacja. Stąd naturalne są konstrukcje, w których polami klasy opisującej osobę byłyby jej atrybuty takie jak imię, nazwisko, data urodzenia itp., klasa opisująca pozycję w bibliotece mogłaby zawierać pola opisujące rodzaj pozycji, jej tytuł, autora i datę wydania itp.

```

class COsoba
{
    public string Imie;
    public string Nazwisko;
    public DateTime data_urodzenia;
}

```

Przypomnijmy sobie, że w C++ istnieją dwie możliwości utworzenia obiektu:

```

COsoba osoba1;
COsoba* osoba2 = new COsoba();
...

```

W C# klasa opisana tak jak wyżej będzie typem referencyjnym, to znaczy że użycie obiektu będzie wymagało jego jawnego utworzenia:

```

...
COsoba osoba = new COsoba();
...

```

Sam term *osoba* funkcjonuje w CTS jako referencja do obiektu na sterzie programu. Odwołania do jego składowych odbywają się za pomocą operatora ".", na przykład

```

...
osoba.Imie = "Xawery";
...

```

Programista może w klasie zdefiniować również pola *statyczne*. Mają one własność przynależenia do klasy, a nie do żadnego konkretnego obiektu klasy. Intuicyjnie można więc rozumieć pola statyczne jako odpowiednik zmiennych globalnych, występujących w innych językach programowania.



```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class COsoba
    {
        public static int IloscOsob;
    }

    public class CMain
    {
        public static void Main()
        {
            COsoba.IloscOsob = 17;
            Console.WriteLine( COsoba.IloscOsob.ToString() );
        }
    }
}

```

### 3.2.5.2 Metody

Metody zawierają w sobie kod wykonywany podczas działania programu. Na przykład:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class COsoba
    {
        public string Imie;
        public string Nazwisko;

        public string ImieNazwisko()
        {
            return Imie+" "+Nazwisko;
        }
    }

    public class CMain
    {
        public static void Main()
        {
            COsoba o  = new COsoba();

            o.Imie      = "Xawery";
            o.Nazwisko  = "Xawerowski";

            Console.WriteLine( o.ImieNazwisko() );
        }
    }
}

```

Specjalne znaczenie mają metody statyczne, które podobnie jak pola statyczne nie są przypisane do konkretnej instancji obiektu danej klasy, tylko do klasy jako takiej. Podobnie jak pola, metody statyczne są intuicyjnymi odpowiednikami funkcji globalnych z C czy C++.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class CInfo
    {
        public static string GetInfo()
        {
            return "Info";
        }
    }
}

```

```

    }
}

public class CMain
{
    public static void Main()
    {
        Console.WriteLine( CInfo.GetInfo() );
    }
}
}

```

W standardowy sposób *przeciąża się* metody, tak aby akceptowały różne listy wywołania:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        static void Metoda( int i, string s )
        {
            Console.WriteLine( String.Format( "Liczba '{0}', napis '{1}' ", i, s ) );
        }
        static void Metoda( int i )
        {
            Metoda( i, "jakis napis" );
        }
        static void Metoda( string s )
        {
            Metoda( 17, s );
        }

        public static void Main()
        {
            Metoda( 5, "Ala ma kota" );
            Metoda( 13 );
            Metoda( "kot ma Ale" );
        }
    }
}

C:\Example>example.exe
Liczba '5', napis 'Ala ma kota'
Liczba '13', napis 'jakis napis'
Liczba '17', napis 'kot ma Ale'

```

Istnieje również możliwość poinformowania kompilatora o tym, że metoda może być wołana z nieznaną w czasie kompilacji liczbą parametrów:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        static void VariableParList( params int[] iInfo )
        {
            Console.Write( "parametry: " );
            for ( int i=0; i<iInfo.GetLength(0); i++ )
                Console.Write( iInfo[i].ToString()+" ", " );
            Console.WriteLine();
        }

        public static void Main()
        {
            VariableParList( 1 );
        }
    }
}

```

```

        VariableParList( 1, 2 );
        VariableParList( 1, 2, 3, 4, 5 );
    }
}

```

Kompilator nie pozwoli jednak na skompilowanie kodu, w którym ze względu na przeciążenie funkcji intencje programisty są niejasne.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class CMain
    {
        static void A( int a, params int[] tab )
        {
            Console.WriteLine( "A1" );
        }

        static void A( int a, int b, params int[] tab )
        {
            Console.WriteLine( "A2" );
        }

        public static void Main()
        {
            A( 1, 2, 3 );
        }
    }
}

```

Czy wołając funkcję **A** programista miał na myśli wersję pierwszą, z jednym parametrem jawnym, czy drugą, z dwoma parametrami jawnymi? Cokolwiek myślał programista, kompilator ma własne zdanie na temat takiego kodu:

```

example.cs(19,8): error CS0121: The call is ambiguous between the following methods
or properties: 'Example.CMain.A(int, int, params int[])' and
'Example.CMain.A(int, params int[])'

```

Specjalną rolę wśród metod w klasie pełni metoda `Main()`, która określa punkt startowy aplikacji<sup>4</sup>. Parametry startowe programu są przekazane jako tablica do metody `Main()`:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main( string[] args )
        {
            Console.WriteLine( "Oto kolejne argumenty wywołania programu: " );
            foreach ( string s in args )
                Console.WriteLine( s );
        }
    }
}

C:\Example>example.exe 17 "napis napis"
Oto kolejne argumenty wywołania programu:
17
napis napis

```

---

<sup>4</sup>O możliwości umieszczenia wielu alternatywnych metod `Main()` w kodzie aplikacji napisano więcej na stronie 73.

### 3.2.5.3 Przekazywanie parametrów do metod

Sposób przekazania parametru do metody zależy od tego, czy zmienna jest typu prostego czy typu referencyjnego. Jeśli zmienna jest typu prostego, jak *int*, to do metody zostanie przekazana wartość, jeśli zmienna jest typu referencyjnego, to do metody zostanie przekazana referencja.

Oznacza to, żewołana metoda nie ma możliwości zmiany, w przypadku typów prostych - wartości zmiennej, w przypadku typów referencyjnych - referencji do zmiennej.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Zmien( int i, string s )
        {
            i = 1;
            s = "Ala ma kota";
        }
        public static void Main()
        {
            int    i = 0;
            string s = "Kot ma Ale";

            Console.WriteLine( "{0}, {1}", i, s );
            Zmien( i, s );
            Console.WriteLine( "{0}, {1}", i, s );
        }
    }
}

C:\Example>example.exe
0, Kot ma Ale
0, Kot ma Ale
```

To, że referencja do przekazywanego obiektu nie może ulegać zmianie nie oznacza, że wartość obiektu referencyjnego nie może być zmodyfikowana - wprost przeciwnie, metoda ma możliwość zmiany właściwości obiektu przekazanego przez referencję.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        public static void Zmien( ArrayList a )
        {
            a.Add( 0 );
        }
        public static void Main()
        {
            ArrayList a = new ArrayList();
            Console.WriteLine( "elementow na liscie {0}", a.Count );
            Zmien( a );
            Console.WriteLine( "elementow na liscie {0}", a.Count );
        }
    }
}

C:\Example>example.exe
elementow na liscie 0
elementow na liscie 1
```

Jeśli intencją programisty jest zmiana wartości przekazywanej do funkcji, może zażądać przekazania do funkcji referencji do obiektu (w przypadku typu referencyjnego będzie to referencja do referencji) za pomocą słowa kluczowego *ref*. Jest to dosłowny odpowiednik przekazywania parametrów do funkcji przez referencje, znany z C++.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Zmien( ref int i, ref string s )
        {
            i = 1;
            s = "Ala ma kota";
        }
        public static void Main()
        {
            int i = 0;
            string s = "Kot ma Ale";

            Console.WriteLine( "{0}, {1}", i, s );
            Zmien( ref i, ref s );
            Console.WriteLine( "{0}, {1}", i, s );
        }
    }
}

C:\Example>example.exe
0, Kot ma Ale
1, Ala ma kota
```

Wydawać by się mogło, że w taki sposób należy również przekazywać parametry, które miałyby służyć do przekazywania wyników do funkcji. Naiwnie możnaby więc spróbować napisać coś takiego:

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Oblicz( ref int wynik )
        {
            wynik = 1;
        }
        public static void Main()
        {
            int wynik;

            Oblicz( ref wynik );
            Console.WriteLine( "wynik: {0}", wynik );
        }
    }
}
```

jednak kompilator takiej konstrukcji nie przyjmie

```
example.cs(16,19): error CS0165: Use of unassigned local variable 'wynik'
```

Istnieją dwa możliwe rozwiązania takiego problemu:

- Przed wykonaniem obliczeń przypisać jakąś wartość zmiennej *wynik*. Nie jest to rozwiązanie eleganckie, skoro *wynik* ma dopiero otrzymać wartość w wyniku obliczeń.

- Zadeklarować parametr funkcji jako *out int wynik* zamiast *ref int wynik*. Słowo kluczowe *out* jest równoważne *ref*, przy czym zmienna nie musi otrzymać wartości przed wykorzystaniem jej jako parametru do funkcji.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Oblicz( out int wynik )
        {
            wynik = 1;
        }
        public static void Main()
        {
            int    wynik;

            Oblicz( out wynik );
            Console.WriteLine( "wynik: {0}", wynik );
        }
    }
}
```

### 3.2.5.4 Konstruktory

Konstruktory są w pewnym sensie specjalnymi metodami, które zawierają kod inicjujący obiekty. Konstruktory definiuje się dokładnie tak samo jak w C++ czy w Javie: przez utworzenie kodu pseudo-metody o nazwie takiej jak nazwa klasy. Podstawowa różnica między C# a na przykład C++ jest taka, że, podobnie jak w Javie, środowisko uruchomieniowe za pomocą *odśmiecacza* zajmuje się oczyszczaniem pamięci z nieużywanych już obiektów<sup>5</sup>.

Konstruktory można oczywiście przeładowywać, można również korzystać z konstruktorów klas bazowych lub z innych konstruktorów już określonych w klasie za pomocą wyrażeń inicjujących **base(...)** oraz **this(...)**, na przykład:

```
/* Wiktor Zychla, 2003 */
using System;
using System.Threading;

namespace Example
{
    public class CExample
    {
        DateTime d;
        int i=0, j=0;

        public CExample()
        {
            d = DateTime.Now;
        }

        public CExample( int I, int J ) : this()
        {
            this.i = I;
            this.j = J;
        }

        public override string ToString()
        {
            return String.Format( "[{0},{1}], utworzone {2}", i, j, d );
        }
    }
}
```

---

<sup>5</sup>Istnieje specjalny interfejs *IDisposable* przygotowany na użytek klas, które potrzebują jawnie wykonać akcje przy niszczeniu obiektu przez odśmiecacz.

```

    }
    public class CMain
    {
        public static void Main()
        {
            CExample e1 = new CExample();
            Thread.Sleep( 1000 );
            CExample e2 = new CExample( 13, 17 );

            Console.WriteLine( e1 );
            Console.WriteLine( e2 );
        }
    }
}

```

```

C:\Example>example.exe
[0,0], utworzone 2003-03-18 21:48:08
[13,17], utworzone 2003-03-18 21:48:10

```

C# pozwala zdefiniować konstruktor statyczny, który będzie wywołany przed skonstruowaniem pierwszego obiektu klasy. Statyczny konstruktor może być tylko jeden, bez żadnego kwalifikatora dostępu i z pustą listą parametrów.

```

class CExample
{
    static CExample()
    {
        ...
    }
}

```

### 3.2.5.5 Właściwości (propercje)

Właściwości pozwalają ukryć implementację metody tak, aby z punktu widzenia klienta klasy wyglądała ona jak pole. Właściwości stosuje się tam, gdzie istnieje konieczność nadania lub pobrania wartości pola, a przy tym wykonać jakieś dodatkowe operacje. Za pomocą właściwości można także ograniczyć dostęp do jakiegoś pola, czyniąc je tylko do odczytu lub tylko do zapisu.

Technicznie - implementacja właściwości ma jeden lub dwa tzw. *akcesory* dostępu: akcesor **get** i/lub akcesor **set**, z których pierwszy służy do pobierania wartości, drugi do jej nadawania.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class COsoba
    {
        private int      m_wiek;
        private DateTime m_dataUrodzenia;

        public int wiek
        {
            get { return m_wiek; }
        }

        public DateTime dataUrodzenia
        {
            get { return m_dataUrodzenia; }
            set
            {
                m_dataUrodzenia = value;
                m_wiek = DateTime.Now.Year-m_dataUrodzenia.Year;
            }
        }
    }
}

```

```

public class CMain
{
    public static void Main()
    {
        COsoba o = new COsoba();
        o.dataUrodzenia = new DateTime( 1950, 3, 8 );

        Console.WriteLine( "Osoba:\r\nUrodzona\t{0:d}\r\nWiek\t\t{1}",
                           o.dataUrodzenia, o.wiek );
    }
}

```

```

C:\Example>example.exe
Osoba:
Urodzona      1950-03-08
Wiek          53

```

### 3.2.5.6 Stałe

Stałe można zadeklarować w klasie przez opatrzenie deklaracji pola kwalifikatorem *const*.

```
public const string sKraj = "Polska";
```

Co jednak zrobić, gdy wartość stałej jest znana dopiero **po** uruchomieniu programu? W C# taki problem rozwiązuje kwalifikator *readonly*, który oznacza pole zawierające stałą, przy czym wartość takiego pola można modyfikować **tylko** w konstruktorze klasy.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class COsoba
    {
        public readonly DateTime dataI;
        public COsoba()
        {
            dataI = DateTime.Now;
        }
    }
    public class CMain
    {
        public static void Main()
        {
            COsoba o = new COsoba();
            Console.WriteLine( o.dataI );
        }
    }
}

```

### 3.2.5.7 Indeksery

Indeksery pozwalają klientom klasy traktować obiekt tak, jakby był on tablicą, bez względu na reprezentację pól obiektu. Indeksery podobne są trochę do propekcji - podobnie jak propekcje indeksery mogą pobierać wartość *get* lub sposób ustalać wartość *set*.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class COsoba
    {

```



```

    public int this[int i, int j]
    {
        get { return i+j; }
    }
}
public class CMain
{
    public static void Main()
    {
        COsoba o = new COsoba();
        Console.WriteLine( o[5,17] );
    }
}

```

Idea indeksatorów narodziła się z chęci ułatwienia programistom dostępu do składowych obiektu, który w jakiś sposób opisuje strukturę tablicopodobną. Na przykład klasy opisujące okna potomne, takie jak ComboBox czy ListView, z pewnością jako jedno z pól będą zawierać jakąś tablicę elementów przechowywanych w wewnętrznej liście obiektu. Indeksor umożliwia w takim przypadku dostęp do takiej listy bezpośrednio przez indeksowanie obiektu, a nie jego pola, tzn. na przykład zamiast

```

ComboBox comboBox = new ComboBox();
...
comboBox.Items[5] = ...

```

moglibyśmy pisać (mając zdefiniowany odpowiedni indeksor)

```

ComboBox comboBox = new ComboBox();
...
comboBox[5] = ...

```

### 3.2.5.8 Przeciążanie operatorów

Przeciążanie operatorów nie wnosi do języków programowania nic, poza czystością i elegancją kodu. Z technicznego punktu widzenia przeciążone operatory są jakimiś metodami, które biorą określoną ilość parametrów i zwracają wyniki.

C# pozwala przeciążać operatory za pomocą składni

```

public static retval operatorop (object1 [, object2])

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CVec
    {
        public int x;
        public int y;

        public CVec( int X, int Y )
        {
            x = X; y = Y;
        }

        public static CVec operator+( CVec v1, CVec v2 )
        {
            return new CVec( v1.x+v2.x, v1.y+v2.y );
        }

        public override string ToString()
        {
            return String.Format( "[{0},{1}]", x, y );
        }
    }
}

```

```

    }
}
public class CMain
{
    public static void Main()
    {
        CVec u = new CVec( 2, 3 );
        CVec v = new CVec( 1, 1 );
        Console.WriteLine( "{0}+{1}={2}", u, v, u+v );
    }
}
}

C:\Example>example.exe
[2,3]+[1,1]=[3,4]

```

Obowiązują następujące zasady przy przeciążaniu operatorów w C#:

- można przeciążać operatory unarne `+`, `-`, `!`, `++`, `-`, `true` i `false` oraz binarne `+`, `-`, `*`, `/`,
- nie można przeciążać operatora `[]`, można jednak zdefiniować indeksy, który pozwala traktować obiekt jak tablicę
- nie można przeciążać operatora `()`, z wyjątkiem definiowania własnych jawnych konwersji
- operatory warunkowe (`&&`, `-`, i `?:`) nie mogą być przeciążane
- operatory nie występujące w C# nie mogą być przeciążane
- operatory zdefiniowane przez Framework (`.`, `=`, `new`) nie mogą być przeciążane
- operatory (`==` i `!=`) mogą być przeciążane, wymaga to jednak przeciążenia metod *Equals* i *GetHashCode*
- przeładowanie niektórych operatorów binarnych (na przykład `+`) powoduje automatyczne przeładowanie pewnych innych operatorów (w tym przypadku `+=`)
- operatory `<` i `>` muszą być przeładowywane jednocześnie

### 3.2.6 Struktury

Typy proste w C# deklaruje się tak samo jak typy referencyjne, zastępując słowo kluczowe *class* słowem kluczowym *struct*. Tak jak typy referencyjne nazywamy klasami, tak typy proste nazywamy strukturami.

Deklaracja struktury może zawierać dowolną ilość konstruktorów, z wyjątkiem konstruktora bezparametrowego, który jest tworzony domyślnie i powoduje wyzerowanie (nadanie wartości domyślnych) wartości wszystkich pól struktury. W przypadku typów prostych, które nie zawierają pól, korzystanie ze zmiennych możliwe jest więc bez wywołania konstruktora (jak na przykład w przypadku typu *int*).

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    struct RGB
    {
        public int r;
        public int g;
        public int b;
    }
}

```

```

    public RGB( int R, int G, int B )
    {
        r = R; g = G; b = B;
    }

    public override string ToString()
    {
        return String.Format( "[R:{0}, G:{1}, B:{2}]", r, g, b );
    }
}
public class CMain
{
    public static void Main()
    {
        RGB rgb = new RGB();
        RGB rgb2 = new RGB( 1, 2, 3 );
        Console.WriteLine( rgb );
        Console.WriteLine( rgb2 );
    }
}

C:\Example>example.exe
[R:0, G:0, B:0]
[R:1, G:2, B:3]

```

### 3.2.7 Dziedziczenie

Model obiektowy C# udostępnia pojedyncze dziedziczenie. Oznacza to, że każda klasa może mieć co najwyżej jedną klasę bazową. O relacjach między klasami programista informuje kompilator za pomocą składni

```

class <klasaPotomna> : <klasaBazowa>

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class A
    {
        public int ID;
    }
    class B : A
    {
        public int ID2;
    }
    public class CMain
    {
        public static void Main()
        {
            A a = new A();
            B b = new B();

            a.ID = 1;
            b.ID = 2;
            b.ID2 = 3;
        }
    }
}

```

Jeśli jakaś metoda występuje w klasie potomnej i w klasie bazowej, to kompilator zakłada, że metoda z klasy potomnej *przykrywa* definicję metody z klasy bazowej (ominięcie obowiązkowego w takim przypadku kwalifikatora *new* zostanie przez kompilator wykryte i programista zostanie ostrzeżony o jego braku).

Co się jednak stanie, jeśli obiekt klasy potomnej zostanie najpierw zrzucony na obiekt klasy macierzystej, a następnie zostanie wywołana metoda?

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class A
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    class B : A
    {
        new public void DajGlos()
        {
            Console.WriteLine( "B" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            A a = new A();
            B b = new B();

            a.DajGlos();
            b.DajGlos();
            ((A)b).DajGlos();
        }
    }
}

C:\Example>example.exe
A
B
A

```

Jak widać, kompilator *statycznie* wyznaczył typ obiektu i wymusił zawołanie funkcji właściwej dla wyznaczonego typu.

Programista może jednak zażądać *polimorficznego* traktowania przeciążonych w klasach potomnych metod, to znaczy *dynamicznego wyznaczania* typu obiektu i wołania odpowiedniej funkcji. Służą do tego kwalifikatory *virtual* i *override* umieszczone przy odpowiednich deklaracjach.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class A
    {
        virtual public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    class B : A
    {
        override public void DajGlos()
        {
            Console.WriteLine( "B" );
        }
    }
    public class CMain
    {
        public static void Main()
        {

```

```

    A a = new A();
    B b = new B();

    a.DajGlos();
    b.DajGlos();
    ((A)b).DajGlos();
}
}
}

C:\Example>example
A
B
B

```

Programista może zabezpieczyć klasę przed dziedziczeniem z niej za pomocą kwalifikatora *sealed*, na przykład

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    sealed class A
    {
    }
    class B : A
    {
    }
    public class CMain
    {
        public static void Main()
        {
        }
    }
}

example.cs(9,9): error CS0509: 'Example.B' : cannot inherit from sealed class
'Example.A'

```

### 3.2.8 Niszczanie obiektów

Niszczaniem obiektów w C# zajmuje się odśmieczacz. Co pewien czas odśmieczacz przegląda stertę w poszukiwaniu obiektów do których brak już referencji i zwalnia przydzieloną im pamięć. Programista nie ma więc kontroli nad niszczeniem obiektów takiej jak na przykład w C++. Istnieją jednak dwa aspekty niszczenia obiektów, których programista musi być świadomy.

#### 3.2.8.1 Destruktory

Zadziwiające, ale destruktory istnieją w C#. Ich działanie zdecydowanie różni się od działania destruktorów w C++, bardziej zaś przypomina działanie metod typu *Finalize* z Javy.

Chodzi o to, że programista nie ma żadnej kontroli nad tym, kiedy destruktor zostanie wywołany. Odśmieczacz wykona kod zawarty w destruktorze tuż przed usunięciem obiektu, jednak sam moment usuwania obiektu jest z punktu widzenia programisty nie możliwy do określenia.

Poniższy przykład pokazuje, że destruktory są wywoływane tuż przed zakończeniem programu. Okazuje się jednak, że przerwanie działania programu za pomocą CTRL+C spowoduje, że destruktory nie wykonają się!

```

/* Wiktor Zychla, 2003 */
using System;
using System.Threading;

```

```

namespace Example
{
    public class CExample
    {
        public int numer;

        public CExample( int Numer )
        {
            numer = Numer;
            Console.WriteLine( "Utworzono obiekt {0}", numer );
        }

        ~CExample()
        {
            Console.WriteLine( "Zniszczono obiekt {0}", numer );
        }

        public static void Main(string[] args)
        {
            int objectNo = 10;

            CExample[] examples = new CExample[ objectNo ];
            for ( int i=0; i<10; i++ )
                examples[i] = new CExample( i );

            Thread.Sleep( 5000 );
        }
    }
}

```

```

C:\Example>example.exe
Utworzono obiekt 0
Utworzono obiekt 1
Utworzono obiekt 2
Utworzono obiekt 3
Utworzono obiekt 4
Utworzono obiekt 5
Utworzono obiekt 6
Utworzono obiekt 7
Utworzono obiekt 8
Utworzono obiekt 9
Zniszczono obiekt 9
Zniszczono obiekt 8
Zniszczono obiekt 7
Zniszczono obiekt 6
Zniszczono obiekt 5
Zniszczono obiekt 4
Zniszczono obiekt 3
Zniszczono obiekt 2
Zniszczono obiekt 1
Zniszczono obiekt 0

```

### 3.2.8.2 Interfejs *IDisposable*

Z innego rodzaju problemem mamy do czynienia, kiedy obiekt C# inicjuje zasoby innego rodzaju niż pamięć, na przykład obiekty systemowe takie jak gniazda, połączenia do baz danych, obiekty GDI. Brak kontroli nad zwalnianiem tych zasobów (na przykład uchwytów GDI), może wręcz zakłócić pracę systemu! Aby uniknąć takich problemów, należy zaimplementować w klasie interfejs *IDisposable*, który ma jedną metodę: *Dispose()*, w której programista może jawnie zniszczyć zasoby przyznane obiektowi.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Threading;

namespace Example
{

```

```

public class CExample : IDisposable
{
    public int numer;

    public CExample( int Numer )
    {
        numer = Numer;
        Console.WriteLine( "Utworzono obiekt {0}", numer );
    }

    public void Dispose()
    {
        Console.WriteLine( "Zniszczono obiekt {0}", numer );
    }

    public static void Main(string[] args)
    {
        int objectNo = 10;

        CExample[] examples = new CExample[ objectNo ];
        for ( int i=0; i<10; i++ )
            examples[i] = new CExample( i );

        Thread.Sleep( 5000 );

        for ( int i=0; i<10; i++ )
            examples[i].Dispose();
    }
}

```

Niestety, programista musi sam pamiętać o wywołaniu metody *Dispose* gdy obiekt przestaje być potrzebny. W przypadku konstrukcji pojedynczego obiektu, przydatny okazuje się *lukier syntaktyczny*, polegający na umieszczeniu konstrukcji obiektu w klauzuli *using*. W chwili zakończenia następującego po niej bloku kodu, metoda *Dispose()* wołana jest automatycznie.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Threading;

namespace Example
{
    public class CExample : IDisposable
    {
        public int numer;

        public CExample( int Numer )
        {
            numer = Numer;
            Console.WriteLine( "Utworzono obiekt {0}", numer );
        }

        public void Dispose()
        {
            Console.WriteLine( "Zniszczono obiekt {0}", numer );
        }

        public static void Main(string[] args)
        {
            using ( CExample example = new CExample( 0 ) )
            {
                Thread.Sleep( 5000 );
            }
        }
    }
}

C:\Example>example.exe
Utworzono obiekt 0

```

Zniszczono obiekt 0

### 3.2.9 Interfejsy

Kiedy mówimy o klasach, mamy na myśli pewne właściwości. Interfejsy są odpowiednikami klas, przy czym dotyczą one jakiejś określonej funkcjonalności. Interfejsy są odpowiednikami klas abstrakcyjnych, znanych z innych języków programowania - kiedy klasa *implementuje interfejs* (czasem mówimy też *dziedziczy z interfejsu*), czyli implementuje wszystkie metody interfejsu, klienci tej klasy mogą być pewni istnienia w klasie wszystkich zdefiniowanych przez interfejs metod.

Interfejsy są sposobem na przezwycięzenie ograniczenia pojedynczego dziedziczenia - o ile klasa może mieć tylko jedną klasę bazową, o tyle ta sama klasa może implementować dowolną ilość interfejsów. Interfejs nie może zawierać pól, tylko specyfikacje metod przy czym w definicji interfejsu metody nie mogą mieć żadnych kwalifikatorów dostępu, zaś w implementacji interfejsu w klasie implementowane metody muszą być publiczne (inaczej nie miałyby sensu umieszczanie ich w publicznym interfejsie, który klasa rzekomo miałaby spełniać).

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    interface I
    {
        void DajGlos();
    }
    class A : I
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            A a = new A();
            a.DajGlos();
        }
    }
}
```

Jeśli klasa nie implementuje wszystkich wymaganych funkcji, to kompilator podczas kompilacji zgłosi błąd.

Z punktu widzenia programisty interfejs zachowuje się jak klasa, to znaczy można obiekty rzutować na interfejs, statycznie i dynamicznie<sup>6</sup>. Można także definiować zmienne których typem jest konkretny interfejs.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    interface I
    {
        void DajGlos();
    }
    class A : I
    {
```

---

<sup>6</sup>O konwersji między typami można przeczytać na stronie 99.



```

        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            A a = new A();
            I i = a as I;
            i.DajGlos();
        }
    }
}

```

Programista może w czasie wykonania programu dowiedzieć się czy klasa implementuje jakiś interfejs za pomocą słowa kluczowego *is*. Jest to przydatne zwłaszcza wtedy, kiedy obiekty znajdują się w jakimś kontenerze, w którym wszystkie są rzutowane do typu bazowego *object*.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    interface I
    {
        void DajGlos();
    }
    class A : I
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    class B
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            A a = new A();
            B b = new B();

            if ( a is I ) Console.WriteLine( "A implementuje I" );
            if ( b is I ) Console.WriteLine( "B implementuje I" );
        }
    }
}

C:\Example>example.exe
A implementuje I

```

Interfejsy nie są domyślnie dziedziczone z klasy bazowej do klas potomnych. Jeśli programista rzutuje obiekt klasy potomnej na interfejs implementowany w klasie bazowej, to nawet jeśli klasa potomna zawiera odpowiednie metody, zostanie zawołana metoda z klasy bazowej.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example

```

```

{
    interface I
    {
        void DajGlos();
    }
    class A : I
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    class B : A
    {
        new public void DajGlos()
        {
            Console.WriteLine( "B" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            B b = new B();
            b.DajGlos();

            ((I)b).DajGlos();
        }
    }
}

```

C:\Example>example.exe

B

A

Jeśli klasa potomna miałaby implementować interfejs, to należy o tym poinformować kompilator.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    interface I
    {
        void DajGlos();
    }
    class A : I
    {
        public void DajGlos()
        {
            Console.WriteLine( "A" );
        }
    }
    class B : A, I
    {
        new public void DajGlos()
        {
            Console.WriteLine( "B" );
        }
    }
    public class CMain
    {
        public static void Main()
        {
            B b = new B();
            b.DajGlos();

            ((I)b).DajGlos();
        }
    }
}

```

```
    }
  }
}
```

```
C:\Example>example.exe
B
B
```

Interfejsy mogą implementować inne interfejsy.

```
interface I
{
    void I();
}
interface J
{
    void J();
}
interface IJ : I, J
{
    void IJ();
}
```

### 3.2.10 Konwersje między typami

Możliwość przypisania wprost wartości jednego typu do wartości innego typu zależy tylko od tego, czy zdefiniowano bezpośredni operator konwersji między tymi typami. Najczęściej taka próba nie powiedzie się, bowiem operatory konwersji zdefiniowano tylko dla wybranych par typów.

Zobaczmy jak kompilator reaguje na próbę wymuszenia konwersji między wartościami różnych typów gdy operator konwersji bezpośredniej istnieje tylko dla konwersji w jedną stronę.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            int i=1;
            long j=1;

            j=i;
            i=j;
        }
    }
}
```

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

```
example.cs(14,6): error CS0029: Cannot implicitly convert type 'long' to 'int'
```

Wszystko się zgadza - konwersja z wartości 32-bitowej do wartości 64-bitowej jest legalna (tak została określona) bowiem nie powoduje utraty danych. Konwersja odwrotna może prowadzić do utraty danych i choć kompilator taką konwersję dopuści, programista musi swój zamiar potwierdzić:

```
j=i;
i=(int)j;
```

### 3.2.10.1 Typy wbudowane

Wartość każdego typu można przekształcić do wartości typu *string* za pomocą metody **To-String()**. Wbudowane klasy arytmetyczne mają statyczne metody *Parse*, służące do konwersji napisów.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            string sI = "45";
            string sF = "123,5";

            int    i = int.Parse( sI );
            float  f = float.Parse( sF );

            Console.WriteLine( "{0}; {1}", i, f );
        }
    }
}
```

Bardziej ogólne podejście możliwe jest dzięki konwersjom między typami wbudowanymi, dostępnymi jako statyczne metody klasy *System.Convert*.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            string sI = "45";
            string sF = "123,5";

            int    i = Convert.ToInt32( sI );
            double f = Convert.ToDouble( sF );

            Console.WriteLine( "{0}; {1}", i, f );
        }
    }
}
```

### 3.2.10.2 Typy własne

Możliwe są dwa rodzaje konwersji wprost (przez przypisanie):

**konwersja jawna** Z konwersją jawną mamy do czynienia wtedy, kiedy wymiana wartości między dwoma typami wymaga jawnego rzutowania, na przykład:

```
double d = 1.5;
int    i = (int)d;
```

**konwersja niejawna** Z konwersją niejawną mamy do czynienia wtedy, kiedy wymiana wartości między dwoma typami nie wymaga jawnego rzutowania, na przykład:

```
int    i = 1;
double d = i;
```

W przypadku typów wbudowanych rodzaje konwersji są już zdeterminowane, jednak w przypadku własnych klas programista staje przed wyborem rodzajów konwersji między swoim typem a innymi typami. Wybór odpowiedniego typu konwersji zależy od tego czy podczas konwersji może dojść do utraty danych, czy nie. Wyobraźmy sobie sytuację, w której typ A jest bardziej pojemny informacyjnie niż typ B. Wobec tego konwersja danej typu B do danej typu A może być niejawna, ponieważ nie istnieje ryzyko utraty informacji. Konwersja w odwrotną stronę powinna być jawna, aby klient klasy był świadomy możliwej utraty informacji.

O tym, czy konwersja jest dokonywana jawnie, czy niejawnie, decyduje definicja operatora konwersji. Operatory konwersji definiuje się jako jawne przez użycie kwalifikatora *explicit* lub niejawne *implicit*.

Jako przykład rozważmy strukturę opisującą liczby rzymskie. Konwersja liczby rzymskiej do typu *int* może być niejawna, bowiem *int* jest bardziej pojemny informacyjnie niż zbiór wszystkich liczb rzymskich. Konwersja odwrotna powinna być jawna.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;

namespace Example
{
    public class CMain
    {
        struct RomanNumeral
        {
            readonly string[] literals;
            int value;

            public RomanNumeral( int value )
            {
                if ( value > 3999 ) throw new ArgumentOutOfRangeException();

                literals = new string[] { "I", "V", "X", "L",
                                          "C", "D", "M" };

                this.value = value;
            }

            public static explicit operator RomanNumeral( int value )
            {
                return new RomanNumeral( value );
            }

            public static implicit operator int(RomanNumeral roman)
            {
                return roman.value;
            }

            string BuildRomanString( int index, int v )
            {
                if ( v <= 0 ) return String.Empty;

                string s = String.Empty;
                int digit = v%10;
                int j;

                if ( digit == 4 )
                    s = literals[index]+literals[index+1]+s;
                else if ( digit == 9 )
                    s = literals[index]+literals[index+2]+s;
                else if ( digit >= 5 && digit <= 8 )
                {
                    s = literals[ index+1 ];
                    digit -= 5;
                }
                if ( digit >= 1 && digit <=3 )
                {
```

```

        for ( j=0; j<digit; j++ )
            s = s+literals[ index ];
    }

    return BuildRomanString( index+2, v/10 )+s;
}

public override string ToString()
{
    string s = String.Empty;
    int tVal = value;

    return BuildRomanString( 0, value );
}
}

public static void Main()
{
    int i = 2003;
    RomanNumeral r = (RomanNumeral)i;
    int j = r;

    Console.WriteLine( "{0}, {1}, {2}", i, r, j );
}
}
}

```

### 3.2.10.3 Dziedziczenie a konwersje

Jak w każdym obiektowym języku programowania, obiekt klasy potomnej można zawsze niejawnie rzutować na obiekt klasy macierzystej. W drugą stronę możliwa jest tylko konwersja jawna, jednak uda się ona tylko wtedy, kiedy dany obiekt jest rzeczywiście odpowiedniego typu.

Oprócz rzutowania statycznego, między typami referencyjnymi możliwe jest również rzutowanie dynamiczne za pomocą operatora *as*. Jeśli wartość rzutowania dynamicznego równa jest *null*, to znaczy że rzutowanie nie powiodło się. Rzutowanie dynamiczne pozwala uniknąć wyjątku, który byłby wyrzucony przez rzutowanie statyczne przy błędzie rzutowania, na przykład:

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        static void f1( object o )
        {
            // możliwy wyjątek!
            Hashtable h = (Hashtable)o;
        }
        static void f2( object o )
        {
            // rzutowanie może nie udać się
            // ale nie będzie wyjątku
            Hashtable h = o as Hashtable;
            if ( h == null )
            {
            }
        }
    }
    public static void Main()
    {
        ArrayList a = new ArrayList();

        f1( a );
        f2( a );
    }
}

```

```
}
```

### 3.2.11 Wyjątki

Wyjątki są mechanizmem jaki nowoczesne języki programowania wykorzystują w celu zwiększenia produktywności i czytelności kodu. Projektanci platformy .NET przyjęli, że biblioteki systemowe informują kod użytkownika o błędach za pomocą wyjątków i zachęcają użytkowników do zerwania z przyzwyczajeniami, znanymi na przykład z C, polegającymi na przekazywaniu informacji o błędach przez wartości funkcji.

Wyjątki przechwytyuje się za pomocą składni

```
try
{
    // blok w którym przechwytywane będą wyjątki
}
catch ( TypWyjątku wyjątek )
{
    // obsługa przechwyconego wyjątku
}
finally
{
    // kod wykonywany zawsze na zakończenie bloku
}
```

Wyjątki *wyrzuca* się za pomocą składni

```
throw <obiekt opisujący wyjątek>
```

Istnieją trzy podstawowe strategie dotyczące wyjątków. Wybór odpowiedniej strategii należy do programisty:

**brak obsługi wyjątków** Kod funkcji nie zawiera obsługi wyjątków

**informacja dla funkcji wołającej** Kod klauzuli catch wyrzuca przechwycony wyjątek

**pełne obsłużenie wyjątku** Kod klauzuli catch zawiera pełny kod obsługi wyjątku i nie informuje funkcji wołającej o problemach

Wyjątki są obiektami klas, które dziedziczą z klasy *Exception*. Klasa ta może być przeciążona, choć już w podstawowej wersji zawiera sposoby użytecznych informacji, takich jak informacja diagnostyczna czy ślad stosu, na przykład:

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        public static void f()
        {
            try
            {
                int i = 1-1;
                int j = 1/i;
            }
            catch ( Exception ex )
            {
                Console.WriteLine( "Informacja o błędzie:\r\n{0}\r\nŚlad stosu:\r\n{1}",
                                   ex.Message, ex.StackTrace );
            }
        }
    }
}
```

```

    }
    public static void Main()
    {
        f();
    }
}

```

```

C:\Example>csc.exe example.cs /debug
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

```

```

C:\Example>example.exe
Informacja o błędzie:
Attempted to divide by zero.
Sład stosu:
   at Example.CMain.f() in C:\000\example.cs:line 14

```

### 3.2.12 Klasa *string*

#### 3.2.12.1 Podstawowe możliwości klasy *string*

Obsługa napisów możliwa jest w C# dzięki klasie *string*. Każdy obiekt tej klasy ma dostęp do wielu przydatnych właściwości i metod.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            string s = "Ala ma kota";
            Console.WriteLine( "Length('{0}'): {1}", s, s.Length );

            Console.WriteLine( "ToLower: '{0}', ToUpper: '{1}'",
                               s.ToLower(), s.ToUpper() );

            string sS = "Ala";
            if ( s.StartsWith( sS ) )
                Console.WriteLine( "'{0}' StartsWith '{1}'", s, sS );

            string sE = "ota";
            if ( s.EndsWith( sE ) )
                Console.WriteLine( "'{0}' EndsWith '{1}'", s, sE );

            Console.WriteLine( "Remove(2,3): '{0}'", s.Remove( 2, 3 ) );

            string sI = "!++!";
            Console.WriteLine( "Insert(3, {1}): '{0}'", s.Insert( 3, sI ), sI );

            Console.WriteLine( "Substring(4, 2): '{0}'", s.Substring( 4, 2 ) );

            Console.WriteLine( "IndexOf( 'a' ): '{0}'", s.IndexOf( 'a' ) );
            Console.WriteLine( "LastIndexOf( 'a' ): '{0}'", s.LastIndexOf( 'a' ) );

            Console.WriteLine( "PadLeft( 20, '_' ): '{0}'", s.PadLeft( 20, '_' ) );

            string sT = "   qwerty   ";
            Console.WriteLine( "Trim('{0}'): '{1}'", sT, sT.Trim() );
        }
    }
}

```



```

C:\Example>example.exe
Length('Ala ma kota'): 11
ToLower: 'ala ma kota', ToUpper: 'ALA MA KOTA'
'Ala ma kota' StartsWith 'Ala'
'Ala ma kota' EndsWith 'ota'
Remove(2,3): 'Ala kota'
Insert(3, !++!): 'Ala!++! ma kota'
Substring(4, 2): 'ma'
IndexOf( 'a' ): '2'
LastIndexOf( 'a' ): '10'
PadLeft( 20, '_' ): '_____Ala ma kota'
Trim( '   qwerty   ' ): 'qwerty'

```

### 3.2.12.2 Formatowanie

C# udostępnia bardzo elegancki sposób formatowania napisów, przypominającą trochę sposób formatowania znany z C, jednak znacznie udoskonalony. Programista przygotowuje formatowany napis, przetykany wyrażeniami formatującymi, zawierającymi numery kolejnych parametrów dla formatowanego napisu wraz ze wskazaniem sposobu formatowania.

Zobaczmy najpierw prosty przykład:

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            string s = "{0}+{1}={2}\r\n{0}+{2}={3}\r\n{2}+{3}={4}";
            int i0 = 17, i1 = 23;
            Console.WriteLine( s, i0, i1, i0+i1, 2*i0+i1, 3*i0+2*i1 );
        }
    }
}

```

Wyższość takiego sposobu przekazywania parametrów dla wyrażeń formatujących nad tym dostępnym w C polega na tym, że wyrażenie formatujące zawiera w sobie numer parametru, w związku z czym ten sam parametr może w napisie występować wiele razy bez konieczności powtarzania go na liście parametrów, na przykład:

```
Console.WriteLine( "{0}, {0}, {0}, {0}", 1 );
```

Wyniki formatowania mogą być przekazane nie tylko do konsoli, ale do zmiennej typu *string*, dzięki statycznej funkcji *Format* w klasie *string*, na przykład:

```

string s = "{0}+{1}={2}\r\n{0}+{2}={3}\r\n{2}+{3}={4}";
int i0 = 17, i1 = 23;

string sResult = String.Format( s, i0, i1, i0+i1, 2*i0+i1, 3*i0+2*i1 );

```

Standardowy sposób formatowania wartości numerycznych pozwala, oprócz numeru parametru, dodać do wyrażenia formatującego także długość pola oraz sposób formatowania, na przykład:

```

Console.WriteLine(
    "{0,5} {1,5}", 123, 456);    // wyrównaj do prawej
Console.WriteLine(
    "{0,-5} {1,-5}", 123, 456);  // wyrównaj do lewej

123   456
123   456

```

Dostępne sposoby formatowania wyrażeń numerycznych:

Znak formatujący	Interpretacja
C lub c	Finansowa
D lub d	Dziesiętna
E lub e	Wykładnicza
F lub f	Ustalona ilość pozycji dziesiętnych
G lub g	Ogólna
N lub n	Numeryczna
P lub p	Procentowa
R lub r	Możliwa do ponownego sparsowania
X lub x	Heksadecymalna

Przykład:

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            int i = 123456;
            Console.WriteLine("{0:C}", i);
            Console.WriteLine("{0:D}", i);
            Console.WriteLine("{0:E}", i);
            Console.WriteLine("{0:F}", i);
            Console.WriteLine("{0:G}", i);
            Console.WriteLine("{0:N}", i);
            Console.WriteLine("{0:P}", i);
            Console.WriteLine("{0:X}", i);

            Console.WriteLine();

            double d = 123.456;
            Console.WriteLine("{0:E}", d);
            Console.WriteLine("{0:F}", d);
            Console.WriteLine("{0:G}", d);
            Console.WriteLine("{0:N}", d);
            Console.WriteLine("{0:P}", d);
            Console.WriteLine("{0:R}", d);
        }
    }
}
```

```
C:\Example>example.exe
123_456,00 zł
123456
1,234560E+005
123456,00
123456
123_456,00
12_345_600,00%
1E240

1,234560E+002
123,46
123,456
123,46
```

```
12_345,60%
123,456
```

Programista może wyposażyć własne obiekty w możliwość dowolnego zadawania parametrów formatowania. Umożliwia to interfejs *IFormattable*.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    class CFormatExample : IFormattable
    {
        public int    ie;
        public string se;

        private CFormatExample() {}
        public CFormatExample( int ie, string se )
        {
            this.ie = ie; this.se = se;
        }
        public string ToString( string format, IFormatProvider fp )
        {
            switch ( format )
            {
                case "A" : return ie.ToString();
                case "B" : return se;
                default  : return String.Format( "{0}:{1}", ie, se );
            }
        }
    }
}

public class CExample
{
    public static void Main(string[] args)
    {
        CFormatExample fe = new CFormatExample( 17, "Ala ma kota" );

        Console.WriteLine( String.Format( "{0}", fe ) );
        Console.WriteLine( String.Format( "{0:A}", fe ) );
        Console.WriteLine( String.Format( "{0:B}", fe ) );
    }
}
```

```
C:\Example>example.exe
17:Ala ma kota
17
Ala ma kota
```

### 3.2.12.3 Encoding

Klasyczny problem związany z obsługą napisów to konwersja napisu do tablicy znaków i tablicy znaków do napisu. Klasycznie do tego problemu podchodzi się traktując napis jako tablicę znaków (tak jest na przykład w C).

Zauważmy jednak, że istnieje wiele możliwych standardów kodowania znaków. Niekoniecznie kod znaku w standardzie ASCII musi być taki sam jak w standardzie UNICODE, pewne znaki mogą wręcz być niedostępne w jednych standardach a dostępne w innych.

W C# udostępniono klasę *System.Text.Encoding*, za pomocą której można konwertować napisy i tablice znaków w następujących standardach:

---

Standard kodowania    Uwagi

---

ASCII	
BigEndianUnicode	
Unicode	
UTF7	Unicode, strona kodowa 65000
UTF8	Unicode, strona kodowa 65001

Przykład:

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            byte[] ba = new byte[]
            {
                72, 101, 108, 108, 111;
            };
            string s = Encoding.ASCII.GetString(ba);
            Console.WriteLine(s);

            string sP = "Chrzęszcz chrzęści w Żyrardówku";
            byte[] unicodeB = Encoding.Unicode.GetBytes( sP );
            char[] unicodeC = Encoding.Unicode.GetChars( unicodeB );

            Console.WriteLine();
            foreach ( byte b in unicodeB )
                Console.Write( "{0:D3} ", b );
            Console.WriteLine();
            foreach ( char c in unicodeC )
                Console.Write( "{0} ", c );
        }
    }
}

C:\Example>example
Hello

067 000 104 000 114 000 122 000 005 001 115 000 122 000 099 000 122 000 032 000
099 000 104 000 114 000 122 000 025 001 091 001 099 000 105 000 032 000 119 000
032 000 123 001 121 000 114 000 097 000 114 000 100 000 243 000 119 000 107 000
117 000
C h r z ą s z c z    c h r z ę ś c i    w    Ż y r a r d ó w k u
```

### 3.2.13 Delegaty i zdarzenia

#### 3.2.13.1 Delegaty

*Delegat* jest typem referencyjnym, który w elegancki sposób przechowuje wskaźnik na funkcję. CTS za pomocą delegatów pozwala przekazywać funkcje jako parametry do innych funkcji, kontrolując jednocześnie zgodność typów - kompilator nie pozwoli na utworzenie delegata z funkcji o nieodpowiednim prototypie.

Sama deklaracja delegata przypomina deklarację typu wskaźnika na funkcję w języku C:

```
typedef int(*pfPInt)(); // definicja typu wskaźnika na funkcję w C

delegate int pfPInt(); // definicja delegata - wskaźnika na funkcję w C#
```

Podkreślmy - delegaty są obiektami, konstruuje się je więc w standardowy sposób za pomocą *new*.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;

namespace Example
{
    public class MathClass
    {
        public static int Kwadrat( int n )
        {
            return n*n;
        }

        public static int Dwukrotnosc( int n )
        {
            return n+n;
        }
    }

    public class CMain
    {
        public delegate int MathDelegate( int n );

        public static int Oblicz( int n, MathDelegate m )
        {
            return m( n );
        }

        public static void Main()
        {
            int n1 = Oblicz( 11, new MathDelegate( MathClass.Kwadrat ) );
            int n2 = Oblicz( 11, new MathDelegate( MathClass.Dwukrotnosc ) );

            Console.WriteLine( "{0}, {1}", n1, n2 );
        }
    }
}
```

Dowolna liczba delegatów może być złożona do jednego delegata za pomocą operatora `+`. Wykonanie takiego złożonego delegata powoduje wykonanie powołuje wykonanie całej sekwencji.

Jeśli delegaty zwracają wyniki, to wynik złożonego delegata jest wynikiem ostatnio wykonanego delegata.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;

namespace Example
{
    public class InfoClass
    {
        public static void WypiszKwadrat( int n )
        {
            Console.WriteLine( n*n );
        }

        public static void WypiszDwukrotnosc( int n )
        {
            Console.WriteLine( n+n );
        }
    }

    public class CMain
```

```

{
    public delegate void InfoDelegate( int n );

    public static void Oblicz( int n, InfoDelegate m )
    {
        m( n );
    }

    public static void Main()
    {
        InfoDelegate d1 = new InfoDelegate( InfoClass.WypiszKwadrat );
        InfoDelegate d2 = new InfoDelegate( InfoClass.WypiszDwukrotnosc );

        InfoDelegate d3 = d1+d2;

        Oblicz( 11, d3 );
    }
}

```

W dalszej części rozdziału zostanie pokazane w jaki sposób można obejść to ograniczenie i otrzymywać wyniki ze wszystkich kolejno wykonujących się delegatów jak również - jak wykonywać delegaty w sposób asynchroniczny.

### 3.2.13.2 Zdarzenia

Kiedy zachodzi konieczność poinformowania jakiegoś obiektu o zajściu jakiegoś zdarzenia, bardzo przydatny okazuje się mechanizm zdarzeń. Obiekt, który jest sprawcą pojawienia się zdarzenia przechowuje listę delegatów, którzy zostaną wykonani kiedy zdarzenie ma zostać ogłoszone światu. Każdy inny obiekt, który jest zainteresowany otrzymaniem powiadomienia, po prostu dopisuje swojego delegata do listy delegatów zdarzenia.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Threading;

namespace DelegaciIZdarzenia
{
    public delegate void CZdarzenieDelegate( CZdarzenieEventArgs e );

    // Argumenty zdarzenia
    public class CZdarzenieEventArgs
    {
        public int informacja1;
        public int informacja2;

        private CZdarzenieEventArgs() {}
        public CZdarzenieEventArgs( int Informacja1, int Informacja2 )
        {
            informacja1 = Informacja1;
            informacja2 = Informacja2;
        }
    }

    // Obiekt, który będzie wysyłał zdarzenie
    public class CObiekt
    {
        public event CZdarzenieDelegate Zdarzenie;

        public void ZdarzenieZaszlo( CZdarzenieEventArgs e )
        {
            this.Zdarzenie(e);
        }

        public CObiekt() {}
    }
}

```

```

class CMain
{
    // Reakcja na zdarzenie
    static void Reakcja1( CZdarzenieEventArgs e )
    {
        Console.WriteLine( String.Format( "Reakcja 1: {0},{1}",
            e.informacja1, e.informacja2 ) );
    }

    static void Reakcja2( CZdarzenieEventArgs e )
    {
        Console.WriteLine( String.Format( "Reakcja 2: {0},{1}",
            e.informacja1, e.informacja2 ) );
    }

    public static void Main()
    {
        CObiekt obiekt = new CObiekt();
        obiekt.Zdarzenie += new CZdarzenieDelegate( Reakcja1 );
        Thread.Sleep(1000);
        obiekt.ZdarzenieZaszlo( new CZdarzenieEventArgs( 1, 2 ) );
        Thread.Sleep(1000);
        obiekt.Zdarzenie += new CZdarzenieDelegate( Reakcja2 );
        obiekt.ZdarzenieZaszlo( new CZdarzenieEventArgs( 3, 4 ) );
        Thread.Sleep(1000);
        Console.WriteLine( "koniec" );
    }
}

```

```

C:\Example>example.exe
Reakcja 1: 1,2
Reakcja 1: 3,4
Reakcja 2: 3,4
koniec

```

### 3.2.13.3 Uczeń Czarnoksiężnika

Możliwości C# w zakresie delegatów i zdarzeń podsumujmy bajką o uczniu czarnoksiężnika<sup>7</sup>.

*Dawno dawno temu, za siedmioma górami i siedmioma rzekami, mieszkał potężny Czarnoksiężnik. Czarnoksiężnik miał Ucznia, który bardzo chciał kiedyś być tak mądry jak Czarnoksiężnik. Póki co, Czarnoksiężnik wymyślał swojemu Uczniowi kolejne, coraz bardziej skomplikowane zadania, a Uczeń skrupulatnie je wykonywał.*

*Taki układ trwał i trwał, aż w końcu Czarnoksiężnik uznał, że nie musi już cały czas doglądać pracy swojego Ucznia. "Uczniu!" - rzekł któregoś dnia - "Jesteś już na tyle samodzielny, że w czasie kiedy pracujesz mógłbym zająć się swoimi sprawami. Po prostu informuj mnie o tym, kiedy skończysz pracę."*

```

using System;
using System.Threading;

namespace UcenCzarnoksieznika
{
    class Ucen
    {
        public void PoradzSie( Czarnoksieznik czarnoksieznik )
        {
            _czarnoksieznik = czarnoksieznik;
        }
        public void Pracuj()
        {

```

<sup>7</sup>Oryginał, historia pracownika biurowego, dostępny jest pod adresem <http://www.sellbrothers.com/writing/default.aspx?content=delegates.htm>.

```

        Console.WriteLine( "Uczen: rozpoczynam prace." );
        if ( _czarnoksieznik != null ) _czarnoksieznik.PracaRozpoczeta();

        Console.WriteLine( "Uczen: pracuje." );
        if ( _czarnoksieznik != null ) _czarnoksieznik.PracaTrwa();

        Console.WriteLine( "Koncze prace." );
        if ( _czarnoksieznik != null )
        {
            int ocena = _czarnoksieznik.PracaZakonczona();
            Console.WriteLine( "Ocena : {0}", ocena );
        }
    }
    private Czarnoksieznik _czarnoksieznik;
}

class Czarnoksieznik
{
    public void PracaRozpoczeta() { }
    public void PracaTrwa()      { }
    public int PracaZakonczona()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 2;
    }
}

class Uniwersum
{
    public static void Main()
    {
        Uczen uczen = new Uczen();
        Czarnoksieznik czarnoksieznik = new Czarnoksieznik();

        uczen.PoradzSie( czarnoksieznik );
        uczen.Pracuj();
    }
}

```

Wydawało się, że wszystko funkcjonuje jak należy, jednak Uczeń zaczął zastanawiać się co by się stało, gdyby nie tylko Czarnoksiężnik, ale ktoś inny był również zainteresowany jego postęпами (na przykład Uczennica pewnej zaprzyjaźnionej z Czarnoksiężnikiem Wróżki). W pierwszej chwili Uczeń trochę się zmartwił, bo wyobraził sobie, że jak dużo różnych metod musiałby znać, aby o swoich postępach informować innych. W końcu każdy mógłby chcieć być informowany w trochę inny sposób. Trochę się jednak uspokoił kiedy pomyślał o rozdzieleniu listy możliwych powiadomień od implementacji tych powiadomień. Zaprojektował więc odpowiedni interfejs.

```

using System;
using System.Threading;

namespace UczenCzarnoksieznika
{
    interface IUczenPowiadomi
    {
        void PracaRozpoczeta();
        void PracaTrwa();
        int PracaZakonczona();
    }

    class Uczen
    {
        public void PoradzSie( IUczenPowiadomi uczenpowiadomi )
        {
            _uczenpowiadomi = uczenpowiadomi;
        }
        public void Pracuj()

```



```

{
    Console.WriteLine( "Uczen: rozpoczynam prace." );
    if ( _uczenpowiadomi != null ) _uczenpowiadomi.PracaRozpoczeta();

    Console.WriteLine( "Uczen: pracuje." );
    if ( _uczenpowiadomi != null ) _uczenpowiadomi.PracaTrwa();

    Console.WriteLine( "Koncze prace." );
    if ( _uczenpowiadomi != null )
    {
        int ocena = _uczenpowiadomi.PracaZakonczona();
        Console.WriteLine( "Ocena : {0}", ocena );
    }
}
private IUczenPowiadomi _uczenpowiadomi;
}

class Czarnoksieznik : IUczenPowiadomi
{
    public void PracaRozpoczeta() { }
    public void PracaTrwa()      { }
    public int PracaZakonczona()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 3;
    }
}

class Uniwersum
{
    public static void Main()
    {
        Uczen uczen = new Uczen();
        Czarnoksieznik czarnoksieznik = new Czarnoksieznik();

        uczen.PoradzSie( czarnoksieznik );
        uczen.Pracuj();
    }
}

```

Przekonanie Czarnoksiężnika do zaimplementowania interfejsu trochę trwało i choć na razie nikt inny nie był implementowaniem jego interfejsu zainteresowany, to Uczeń był z siebie bardzo zadowolony. "W końcu teraz" - pomyślał - "każdy zainteresowany będzie mógł łatwo dowiedzieć się jak sobie radzę."

Czarnoksiężnik nie był jednak zachwycony. "Uczniu!" - zagrmiał - "Dlaczego zadajesz sobie tyle trudu informując mnie o rozpoczęciu Twojej pracy i o jej trwaniu? Nie jestem tym zainteresowany. Nie dość, że zmuszasz mnie do implementowania odpowiednich metod w interfejsie, to jeszcze tracisz swój czas czekając aż zauważę Twoje poczynania. Przecież gdybym akurat gdzieś wybył, to musiałbyś bardzo długo czekać na zakończenie wykonania moich metod. Zrób coś z tym, Uczniu!"

Chcąc nie chcąc, zganiony przez swojego Mistrza, Uczeń uznał, że interfejsy są owszem użyteczne w wielu przypadkach, jednak niespecjalnie nadają się do implementowania zdarzeń. Pomyślał, że rzeczywiście byłoby właściwie informować zainteresowanych tylko o tych wydarzeniach, którymi są oni zainteresowani. Zamiast interfejsu stworzył więc delegatów do odpowiednich funkcji.

```

using System;
using System.Threading;

namespace UczenCzarnoksieznika
{
    delegate void PracaRozpoczeta();
    delegate void PracaTrwa();
}

```

```

delegate int PracaZakonczona();

class Uczeń
{
    public void Pracuj()
    {
        Console.WriteLine( "Uczeń: rozpoczynam prace." );
        if ( rozpoczynaprace != null ) rozpoczynaprace();

        Console.WriteLine( "Uczeń: pracuje." );
        if ( pracuje != null ) pracuje();

        Console.WriteLine( "Koncze prace." );
        if ( zakonczy prace != null )
        {
            int ocena = zakonczy prace();
            Console.WriteLine( "Ocena : {0}", ocena );
        }
    }
    public PracaRozpoczeta rozpoczynaprace;
    public PracaTrwa pracuje;
    public PracaZakonczona zakonczy prace;
}

class Czarnoksiężnik
{
    public int PracaZakonczona()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 4;
    }
}

class Uniwersum
{
    public static void Main()
    {
        Uczeń uczen = new Uczeń();
        Czarnoksiężnik czarnoksiężnik = new Czarnoksiężnik();

        uczen.zakonczy prace = new PracaZakonczona( czarnoksiężnik.PracaZakonczona );
        uczen.Pracuj();
    }
}

```

*W ten sposób Uczeń przestał zajmować Czarnoksiężnika zdarzeniami, którymi tamten nie był zainteresowany. W międzyczasie okazało się, że samo Uniwersum zainteresowało się poczynaniami Ucznia i chciało być informowane o rozpoczęciu i zakończeniu przez niego pracy.*

```

using System;
using System.Threading;

namespace UczeńCzarnoksiężnika
{
    delegate void PracaRozpoczeta();
    delegate void PracaTrwa();
    delegate int PracaZakonczona();

    class Uczeń
    {
        public void Pracuj()
        {
            Console.WriteLine( "Uczeń: rozpoczynam prace." );
            if ( rozpoczynaprace != null ) rozpoczynaprace();

            Console.WriteLine( "Uczeń: pracuje." );
            if ( pracuje != null ) pracuje();
        }
    }
}

```

```

        Console.WriteLine( "Koncze prace." );
        if ( zakonczy prace != null )
        {
            int ocena = zakonczy prace();
            Console.WriteLine( "Ocena : {0}", ocena );
        }
    }
    public PracaRozpoczeta rozpoczyn prace;
    public PracaTrwa      pracuje;
    public PracaZakonczona zakonczy prace;
}

class Czarnoksiężnik
{
    public int PracaZakonczona()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 4;
    }
}

class Uniwersum
{
    static void UczeńRozpoczalPrace()
    {
        Console.WriteLine( "Uniwersum zauważa, że Uczeń rozpoczął pracę." );
    }
    static int UczeńZakończyPrace()
    {
        Console.WriteLine( "Uniwersum zauważa, że Uczeń zakończył pracę." );
        return 7;
    }
    public static void Main()
    {
        Uczeń uczeń = new Uczeń();
        Czarnoksiężnik czarnoksiężnik = new Czarnoksiężnik();

        uczeń.rozpoczyn prace = new PracaRozpoczeta( Uniwersum.UczeńRozpoczalPrace );
        uczeń.zakonczy prace = new PracaZakonczona( czarnoksiężnik.PracaZakonczona );
        uczeń.zakonczy prace = new PracaZakonczona( Uniwersum.UczeńZakończyPrace );
        uczeń.Pracuj();
    }
}
}

```

*Katastrofa! Okazało się, że uczynienie delegatów publicznymi polami w swojej klasie, było błędem Ucznia. Uniwersum, w swoim uniwersalnym wymiarze, przysłoniło powiadomienie o zakończeniu pracy skierowane do Czarnoksiężnika swoim własnym.*

*Uczeń postanowił, że musi coś na to poradzić. Zdał sobie sprawę, że potrzebuje jakiegoś mechanizmu rejestrowania i wyrejestrowywania delegatów, tak aby słuchacze zdarzeń mogli dodawać i usuwać swoje funkcje do powiadomień, ale nie mogli zniszczyć całej listy funkcji powiadomień. Uczeń skorzystał więc ze zdarzeń, o których wiedział że automatycznie tworzą odpowiednie procepery związane z obsługą delegatów, tak że słuchacze mogli być dodawani i usuwani za pomocą operatorów += i -=.*

```

using System;
using System.Threading;

namespace UczeńCzarnoksiężnika
{
    delegate void PracaRozpoczeta();
    delegate void PracaTrwa();
    delegate int  PracaZakonczona();

    class Uczeń
    {

```

```

public void Pracuj()
{
    Console.WriteLine( "Uczen: rozpoczynam prace." );
    if ( rozpoczynaprace != null ) rozpoczynaprace();

    Console.WriteLine( "Uczen: pracuje." );
    if ( pracuje != null ) pracuje();

    Console.WriteLine( "Koncze prace." );
    if ( zakonczy prace != null )
    {
        int ocena = zakonczy prace();
        Console.WriteLine( "Ocena : {0}", ocena );
    }
}

public event PracaRozpoczeta rozpoczynaprace;
public event PracaTrwa      pracuje;
public event PracaZakonczone zakonczy prace;
}

class Czarnoksieznik
{
    public int PracaZakonczone()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 4;
    }
}

class Uniwersum
{
    static void UczenRozpoczalPrace()
    {
        Console.WriteLine( "Uniwersum zauwaza, ze Uczen rozpoczal prace." );
    }
    static int UczenZakonczyłPrace()
    {
        Console.WriteLine( "Uniwersum zauwaza, ze Uczen zakonczy prace." );
        return 7;
    }
    public static void Main()
    {
        Uczen uczen = new Uczen();
        Czarnoksieznik czarnoksieznik = new Czarnoksieznik();

        uczen.rozpoczynaprace += new PracaRozpoczeta( Uniwersum.UczenRozpoczalPrace );
        uczen.zakonczy prace += new PracaZakonczone( czarnoksieznik.PracaZakonczone );
        uczen.zakonczy prace += new PracaZakonczone( Uniwersum.UczenZakonczyłPrace );
        uczen.Pracuj();
    }
}

```

*Po tym wszystkim Uczeń odetchnął z ulgą. W elegancki sposób poradził sobie z zaspokojeniem potrzeb wszystkich zainteresowanych jego postępami, a sam nie musiał się specjalnie przejmować wewnętrznymi implementacjami ich metod. Zauważył jedynie, że choć zarówno Czarnoksiężnik jak i Uniwersum oceniają jego poczynania, to do niego dociera tylko jedna ocena. Uczeń chciał zaś znać oceny, które wystawiają mu wszyscy zainteresowani zakończeniem przez niego pracy. Na szczęście był w stanie przeglądać listę słuchaczy zdarzenia i zbierać wyniki od wszystkich po kolei.*

```

using System;
using System.Threading;

namespace UczenCzarnoksieznika
{
    delegate void PracaRozpoczeta();

```

```

delegate void PracaTrwa();
delegate int  PracaZakonczona();

class Uczeń
{
    public void Pracuj()
    {
        Console.WriteLine( "Uczeń: rozpoczynam prace." );
        if ( rozpoczynaprace != null ) rozpoczynaprace();

        Console.WriteLine( "Uczeń: pracuje." );
        if ( pracuje != null ) pracuje();

        Console.WriteLine( "Koncze prace." );
        if ( zakonczy prace != null )
        {
            foreach ( PracaZakonczona pz in zakonczy prace.GetInvocationList() )
            {
                int ocena = pz();
                Console.WriteLine( "Ocena : {0}", ocena );
            }
        }
    }
    public event PracaRozpoczeta rozpoczynaprace;
    public event PracaTrwa      pracuje;
    public event PracaZakonczona zakonczy prace;
}

class Czarnoksiężnik
{
    public int PracaZakonczona()
    {
        Console.WriteLine( "Ach, znakomicie!" );
        return 4;
    }
}

class Uniwersum
{
    static void UczeńRozpoczalPrace()
    {
        Console.WriteLine( "Uniwersum zauważy, że Uczeń rozpoczął prace." );
    }
    static int UczeńZakończyłPrace()
    {
        Console.WriteLine( "Uniwersum zauważy, że Uczeń zakończył prace." );
        return 7;
    }
    public static void Main()
    {
        Uczeń uczeń = new Uczeń();
        Czarnoksiężnik czarnoksiężnik = new Czarnoksiężnik();

        uczeń.rozpoczynaprace += new PracaRozpoczeta( Uniwersum.UczeńRozpoczalPrace );
        uczeń.zakonczy prace += new PracaZakonczona( czarnoksiężnik.PracaZakonczona );
        uczeń.zakonczy prace += new PracaZakonczona( Uniwersum.UczeńZakończyłPrace );
        uczeń.Pracuj();
    }
}

```

*Uczeń usiadł na chwilę zadowolony ze swoich pomysłów. Niestety, okazało się że zarówno Uniwersum jak i Czarnoksiężnik mają mnóstwo własnych zajęć i zauważenie postępów ucznia zajmuje im coraz więcej czasu.*

```

class Czarnoksiężnik
{
    public int PracaZakonczona()
    {

```

```

        Thread.Sleep( 5000 );
        Console.WriteLine( "Ach, znakomicie!" );
        return 4;
    }
}

class Uniwersum
{
    static int UczeńZakoczyłPracę()
    {
        Thread.Sleep( 7000 );
        Console.WriteLine( "Uniwersum zauważyło, że Uczeń zakończył pracę." );
        return 7;
    }
    ...
}
}

```

*Dla Ucznia oznaczało to, że zamiast pilnie pracować musi po wywołaniu słuchacza zdarzenia czekać w nieskończoność na jego zakończenie. Postanowił więc chwilowo przestać przejmować się ocenami, za to wołać odpowiednich słuchaczy asynchronicznie.*

```

public void Pracuj()
{
    ...
    Console.WriteLine( "Koniec pracy." );
    if ( zakończyłPracę != null )
    {
        foreach ( PracaZakończona pz in zakończyłPracę.GetInvocationList() )
        {
            pz.BeginInvoke( null, null );
        }
    }
}
}

```

*Dzięki temu Uczeń mógł natychmiast po wywołaniu powiadomienia zająć się z powrotem swoimi sprawami. Brakowało mu jednak tego, że ktoś docenia jego pracę. Postanowił więc nadal wołać słuchaczy asynchronicznie i co jakiś czas sprawdzać, czy jego praca jest już oceniona.*

```

class Uczeń
{
    public void Pracuj()
    {
        ...
        Console.WriteLine( "Koniec pracy." );
        if ( zakończyłPracę != null )
        {
            foreach ( PracaZakończona pz in zakończyłPracę.GetInvocationList() )
            {
                IAsyncResult res = pz.BeginInvoke( null, null );
                while ( !res.IsCompleted ) Thread.Sleep(1);
                int ocena = pz.EndInvoke(res);
                Console.WriteLine( "Ocena : {0}", ocena );
            }
        }
    }
}
}

```

*W ten sposób, niestety, Uczeń co prawda mógł kontynuować swoją pracę natychmiast po zwołaniu funkcji-słuchacza, jednak w osobnym wątku co chwila zaglądał przez ramię czy zwołany słuchacz zdarzenia już zakończył pracę. Uczeń nie był z tej konieczności zadowolony, postanowił więc zatrudnić własnego delegata który powiadamiałby go o zakończeniu pracy przez asynchronicznego delegata.*

```

using System;
using System.Threading;

```

```

namespace UcenCzarnoksieznika
{
    delegate void PracaRozpoczeta();
    delegate void PracaTrwa();
    delegate int PracaZakonczone();

    class Ucen
    {
        public void Pracuj()
        {
            Console.WriteLine( "Ucen: rozpoczynam prace." );
            if ( rozpoczynapraca != null ) rozpoczynapraca();

            Console.WriteLine( "Ucen: pracuje." );
            if ( pracuje != null ) pracuje();

            Console.WriteLine( "Koncze prace." );
            if ( zakonczypracze != null )
            {
                foreach ( PracaZakonczone pz in zakonczypracze.GetInvocationList() )
                {
                    pz.BeginInvoke( new AsyncCallback(OcenPrace), pz );
                }
            }
        }

        private void OcenPrace( IAsyncResult res )
        {
            PracaZakonczone pz = (PracaZakonczone)res.AsyncState;
            int ocena = pz.EndInvoke(res);
            Console.WriteLine( "Ocena : {0}", ocena );
        }

        public event PracaRozpoczeta rozpoczynapraca;
        public event PracaTrwa pracuje;
        public event PracaZakonczone zakonczypracze;
    }

    class Czarnoksieznik
    {
        public int PracaZakonczone()
        {
            Thread.Sleep( 5000 );
            Console.WriteLine( "Ach, znakomicie!" );
            return 4;
        }
    }

    class Uniwersum
    {
        static void UcenRozpoczalPrace()
        {
            Console.WriteLine( "Uniwersum zauwaza, ze Ucen rozpoczal prace." );
        }
        static int UcenZakoczylPrace()
        {
            Thread.Sleep( 7000 );
            Console.WriteLine( "Uniwersum zauwaza, ze Ucen zakonczy pracze." );
            return 7;
        }
        public static void Main()
        {
            Ucen uczen = new Ucen();
            Czarnoksieznik czarnoksieznik = new Czarnoksieznik();

            uczen.rozpoczynapraca += new PracaRozpoczeta( Uniwersum.UcenRozpoczalPrace );
            uczen.zakonczypracze += new PracaZakonczone( czarnoksieznik.PracaZakonczone );
            uczen.zakonczypracze += new PracaZakonczone( Uniwersum.UcenZakoczylPrace );
            uczen.Pracuj();
        }
    }
}

```

```

        Thread.Sleep( 20000 );
    }
}
}

```

*Teraz wszyscy byli zadowoleni. Czarnoksiężnik i Uniwersum byli powiadamiani o zdarzeniach, które ich interesowały. Uczeń mógł powiadamiać wszystkich zainteresowanych, a sam nie musiał czekać na zakończenie metod implementujących powiadomienia. Mógł za to asynchronicznie zbierać wyniki tych metod.*

*Zmęczony całym dniem ciężkiej pracy, Uczeń mógł w końcu iść spać...*

### 3.2.14 Zestawy

Klasy w programie C#-owym pogrupowane są w rozłącznych przestrzeniach nazw (*namespace'ach*). Dostęp do klas umieszczonych w określonej przestrzeni nazw możliwy jest dzięki konstrukcji

```
NazwaPrzestrzeniKlas.NazwaFunkcji
```

na przykład

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

bądź zadeklarowaniu na początku programu chęci dostępu do określonej przestrzeni nazw

```
using NazwaPrzestrzeniKlas
```

dzięki czemu do funkcji z tej przestrzeni nazw można odwoływać się bez poprzedzania ich nazw nazwą przestrzeni nazw. Podział programu na różne przestrzenie nazw zwykle wynika z logicznego podziału programu na tzw **zestawy** (ang *assembly*).

Zestaw zawiera skompilowany kod klas, przy czym zestaw wykonywalny (\*.exe) różni się od zestawu-biblioteki (\*.dll) tylko tym, że w jednej z klas zawiera kod startowy (tu: funkcję **Main**).

Przykład najprostszego zestawu:

```

/* Wiktor Zychla, 2003 */
using System;

namespace ExampleModule
{
    public class ExampleClass
    {
        public int Metoda()
        {
            return 17;
        }
    }
}

C:\Example>csc.exe /target:library exampleM.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

```

Przykład programu korzystającego z zestawu:

```

/* Wiktor Zychla, 2003 */
using System;
using ExampleModule;

namespace Example
{

```



```

public class CMain
{
    public static void Main()
    {
        ExampleClass e = new ExampleClass();
        Console.WriteLine( e.Metoda() );
    }
}

```

```

C:\Example>csc.exe /reference:exampleM.dll example.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

```

Od tej pory zestaw wykonywalny i biblioteka mogą być kompilowane niezależnie, zaś prawidłowe wykonanie kodu zestawu głównego możliwe będzie tylko wtedy, kiedy biblioteka będzie dostępna dla środowiska uruchomieniowego w czasie wykonania programu, czyli na przykład znajdzie się w tym samym folderze co moduł główny.

### 3.2.15 Refleksje

Możliwość odczytywania i analizy metadanych, czyli opisu typów z już istniejącego kodu nosi nazwę *refleksji*. Refleksje są jednym z najpotężniejszych mechanizmów platformy .NET.

Przed wszystkim każdy typ w systemie może być zidentyfikowany, ponadto można utworzyć zmienną typową podając nazwę typu.

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        static void PrintTypeInfo( Type t )
        {
            Console.WriteLine( "Definicja {0} znajduje się w module {1}.",
                               t, t.Module );
        }

        public static void Main(string[] args)
        {
            string s = String.Empty;
            PrintTypeInfo( s.GetType() );

            Type t = Type.GetType( "Example.CExample" );
            PrintTypeInfo( t );
        }
    }
}

```

```

C:\Example>example.exe
Definicja System.String znajduje się w module CommonLanguageRuntimeLibrary.
Definicja Example.CExample znajduje się w module example.exe.

```

Każdy zestaw może być przeanalizowany pod kątem zawartych w nim typów:

```

/* Wiktor Zychla, 2003 */
using System;
using System.Reflection;

namespace Example
{
    class CExample
    {

```

```

static Assembly GetAssembly(string[] args)
{
    Assembly assembly;
    if (0 == args.Length)
    {
        assembly = Assembly.GetExecutingAssembly();
    }
    else
    {
        assembly = Assembly.LoadFrom(args[0]);
    }
    return assembly;
}

public static void Main(string[] args)
{
    Assembly assembly = GetAssembly(args);
    if (null != assembly)
    {
        Console.WriteLine("Informacje o typach dla {0}", assembly);

        Type[] types = assembly.GetTypes();
        foreach (Type type in types)
        {
            Console.WriteLine("\nTyp: {0}", type);
            foreach (MemberInfo member in type.GetMembers())
            {
                Console.WriteLine("\tSkładowa: {0}", member);
            }
        }
    }
}
}

```

```

C:\Example>example.exe
Informacje o typach dla Example, Version=1.0.1176.39934, Culture=neutral, Public
KeyToken=null

```

```

Typ: Example.CExample
    Składowa: Int32 GetHashCode()
    Składowa: Boolean Equals(System.Object)
    Składowa: System.String ToString()
    Składowa: Void Main(System.String[])
    Składowa: System.Type GetType()
    Składowa: Void .ctor()

```

Za pomocą tego programu można obejrzeć listy typów w zestawach .NET, na przykład System.Windows.Forms.dll czy MSCORLIB.DLL.

Mechanizm refleksji może być wykorzystany do dynamicznego tworzenia instancji obiektów, gdy znany jest typ obiektu. Można w taki sposób zrealizować dynamiczne łączenie kodu - łączenie nie w czasie kompilacji, tylko w czasie wykonania.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Reflection;

namespace Example
{
    public class CTest
    {
        public int testVal;

        public CTest() {}

        override public string ToString()
        {
            return "CTest: " + testVal.ToString();
        }
    }
}

```

```

    }
}

public class CExample
{
    public static void DynamicObjectCreation( Type t )
    {
        int i;
        object o;

        ConstructorInfo c;
        FieldInfo fi;

        ConstructorInfo[] ci;
        ParameterInfo[] pi;
        FieldInfo[] fi;

        ci = t.GetConstructors();
        for (i=0; i<ci.Length; i++)
        {
            c = ci[i];

            pi = c.GetParameters();
            if ( pi.Length == 0 )
            {
                o = c.Invoke(null);

                fi = t.GetFields();
                for ( int j=0; j<fi.Length; j++ )
                {
                    f = fi[j];
                    if ( f.FieldType == Type.GetType("System.Int32" ) )
                        f.SetValue( o, 17 );
                }

                Console.WriteLine( "Typ {0}, ToString(): {1}", o.GetType(), o );
            }
        }

        public static void Main()
        {
            DynamicObjectCreation( Type.GetType( "Example.CTest" ) );
        }
    }
}

C:\Example>example.exe
Typ Example.CTest, ToString() CTest: 17

```

### 3.2.16 Atrybuty

Myśląc o typach i obiektach, które są instancjami odpowiednich typów, wyraźnie rozróżniamy te dwa światy. W chwili wykonania programu instancje obiektów są elementami dynamicznymi - pojawiają się i giną zależnie od woli programisty. Tworząc, modyfikując i niszcząc obiekty programista pracuje na *poziomie języka*, czyli na poziomie konkretnych wartości wypełniających szablony jakimi są typy. Dzięki mechanizmowi refleksji, programista w trakcie działania programu może również pracować na *poziomie metajęzyka*, czyli na poziomie informacji o typach: o ich składowych, o zależnościach między typami.

Mechanizm atrybutów to kolejny mechanizm z poziomu metajęzyka. Atrybuty pozwalają rozszerzyć definicje typów o dodatkowe informacje, możliwe do wydobycia dzięki refleksji. Wyobraźmy sobie pewien abstrakcyjny scenariusz, w którym każdy typ pojawiający się w programie byłby określony jako niebieski, czarny lub zielony. Uwaga - nie *instancja typu* (czyli konkretna zmienna), ale właśnie typ sam w sobie. Taka informacja mogłaby być na przykład jakąś dodatkową wskazówką dla kompilatora lub być w jakiś inny sposób wykorzystana w trakcie działania

aplikacji.

Scenariusz ten zrealizujemy właśnie dzięki atrybutom. Atrybuty są klasami dziedziczącymi z klasy **Attribute**, których instancje dzięki specjalnej składni można związać z klasami bądź ich składowymi.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class KolorKlasyAttribute : Attribute
    {
        public KolorKlasyAttribute( string kolor )
        {
            this.kolor = kolor;
        }
        public string Kolor
        {
            get { return kolor; }
        }

        string kolor;
    }

    [KolorKlasy( "zielony" )]
    public class Typ1
    {
    }

    [KolorKlasy( "niebieski" )]
    public class Typ2
    {
    }

    public class CMainForm
    {
        public static void Main()
        {
            Type type;

            // zbadaj Typ1
            type = typeof( Typ1 );
            foreach ( Attribute a in type.GetCustomAttributes( true ) )
            {
                KolorKlasyAttribute kolorKlasy = a as KolorKlasyAttribute;
                if ( kolorKlasy != null )
                    Console.WriteLine( "Typ {0} ma kolor {1}", type.Name, kolorKlasy.Kolor );
            }

            // zbadaj Typ2
            type = typeof( Typ2 );
            foreach ( Attribute a in type.GetCustomAttributes( true ) )
            {
                KolorKlasyAttribute kolorKlasy = a as KolorKlasyAttribute;
                if ( kolorKlasy != null )
                    Console.WriteLine( "Typ {0} ma kolor {1}", type.Name, kolorKlasy.Kolor );
            }
        }
    }
}

C:\Example>example.exe
Typ Typ1 ma kolor zielony
Typ Typ2 ma kolor niebieski
```

### 3.2.16.1 Predefiniowane atrybuty

W świecie .NET istnieje kilkanaście gotowych atrybutów, których można użyć do poinformowania kompilatora o specjalnych właściwościach typów lub ich składowych. W kolejnych rozdziałach zobaczymy przykłady użycia atrybutu `[Serializable]`, który służy do poinformowania kompilatora o tym, że klasa może być serializowana. Zobaczymy także przykłady użycia atrybutów `[DllImport]` i `[StructLayout]`, które umożliwiają zdefiniowanie funkcji i struktur odwołujących się do funkcji i struktur ze zwykłych, niezarządzanych bibliotek Windowsowych.

Tu wspomnimy jedynie o atrybutach, pozwalających na określenie informacji o pliku, będącym efektem kompilacji. Można je umieścić w dowolnym miejscu w kodzie, bowiem odnoszą się do całego modułu, będącego wynikiem kompilacji. Efekt ich dodania można obejrzeć na zakładce **Wersja**, w oknie właściwości pliku.

```
[assembly: AssemblyTitle("Moja Aplikacja numer 1")]
[assembly: AssemblyDescription("Opis mojej aplikacji")]
[assembly: AssemblyCompany("Moja firma")]
[assembly: AssemblyProduct("A1")]
[assembly: AssemblyCopyright("Moja firma")]
[assembly: AssemblyVersion("1.7.1.0")]
```

### 3.2.17 Kod niebezpieczny

Programiści, którzy dobrze znają C czy C++, czasami bywają w pierwszym kontakcie rozczarowani brakiem bezpośredniej kontroli nad zarządzaniem pamięcią oraz brakiem wskaźników w C#.

Brak możliwości sterowania destrukcją obiektów jest w pełni uzasadniony. Z pewnością w pewnych przypadkach możliwe jest napisanie przez programistę kodu, który samodzielnie zarządza przydzielaniem i zwalnianiem pamięci, jednak w większości przypadków automat i tak robi to lepiej. Jeśli komuś bardzo zależy na możliwości pełnej kontroli nad przydzielaniem i zwalnianiem pamięci - niech po prostu programuje w C.

Co zaś do wskaźników, to okazuje się, że istnieje w C# możliwość korzystania z nich. Po prostu fragment kodu korzystający ze wskaźników musi być oznakowany kwalifikatorem *unsafe*. Choć nazwa sugeruje, że kod taki jest w jakiś sposób niebezpieczny, tak naprawdę chodzi tu tylko o obejście dość restrykcyjnego systemu typów C#, który w żaden sposób nie potrafiłby przepuścić kodu upstrzonego wskaźnikami. Oprócz kodu *niebezpiecznego* możliwy jest również kod *niezarządzany*. Z kodem niezarządzanym mamy do czynienia przy uruchamianiu z kodu C#-owego modułów COM lub COM+, napisanych w Visual Basicu lub C++. Kod niezarządzany sam zajmuje się obsługą pamięci i nie korzysta z bibliotek platformy .NET.

Kod niebezpieczny musi być kompilowany z przełącznikiem */unsafe*.

```
/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        unsafe public static void Swap(int* pi, int* pj)
        {
            int tmp = *pi;
            *pi = *pj;
            *pj = tmp;
        }

        public static void Main(string[] args)
        {
            int i = 17;
            int j = 23;
```

```

        Console.WriteLine("Przed zamianą: i = {0}, j = {1}", i, j);
        unsafe { Swap(&i, &j); }
        Console.WriteLine("Po zamianie: i = {0}, j = {1}", i, j);
    }
}
}

```

W kodzie niebezpiecznym o wiele łatwiej popełnić niezamierzoną pomyłkę. Jednak złe odwołania do pamięci będą wylapywane przez środowisko uruchomieniowe:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        unsafe public static void Blad()
        {
            int i;
            int* pi = &i;
            pi += 10000;
            *pi = 0;
        }

        public static void Main(string[] args)
        {
            unsafe { Blad(); }
        }
    }
}

C:\Example>example.exe

Unhandled Exception: System.NullReferenceException: Object reference not set to
an instance of an object.
   at Example.CExample.Blad()
   at Example.CExample.Main(String[] args)

```

### 3.2.18 Dokumentowanie kodu

Programiści bardzo niechętnie dokumentują kod, który tworzą. Z drugiej strony nawet kilka słów komentarza bywa często bardzo cenne, zwłaszcza kiedy wraca się do kodu napisanego dawno temu. Oczywiście istnieje możliwość tworzenia komentarzy w kodzie, jednak nie da się na podstawie takich komentarzy zbudować niczego co mogłoby być jakąś formą dokumentacji całości kodu.

W C# zaproponowano pewien jednolity sposób tworzenia komentarzy w kodzie jako tagów języka XML<sup>8</sup>. Zobaczmy prosty przykład komentarzy w kodzie programu, sposób tworzenia dokumentacji i ostateczną postać dokumentacji:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    /// <summary>
    /// Klasa zawierająca kod startowy aplikacji.
    /// </summary>
    class CExample
    {
        /// <summary>
        /// Tutaj mógłby znaleźć się opis funkcji f.
    }
}

```

---

<sup>8</sup>Więcej o języku XML na stronie 185.

```

    /// Opis ten może być dowolnie długi.
    /// </summary>
    void f()
    {
    }
    /// <summary>
    /// Główna funkcja aplikacji.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
    }
}
}

```

```
C:\Example>csc.exe /doc:example.xml example.cs
```

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>example</name>
  </assembly>
  <members>
    <member name="T:Example.CExample">
      <summary>
        Klasa zawierająca kod startowy aplikacji.
      </summary>
    </member>
    <member name="M:Example.CExample.f">
      <summary>
        Tutaj mógłby znaleźć się opis funkcji f.
        Opis ten może być dowolnie długi.
      </summary>
    </member>
    <member name="M:Example.CExample.Main(System.String[])">
      <summary>
        Główna funkcja aplikacji.
      </summary>
    </member>
  </members>
</doc>

```

Tworzenie dokumentacji w ten sposób jest szczególnie łatwe w Visual Studio .NET. Edytor kodu C# potrafi automatycznie zbudować odpowiedni szablon dokumentacji po wprowadzeniu przez programistę znaku rozpoczęcia takiego komentarza, czyli `///`. Istnieje kilkanaście różnych możliwych tagów XML jakimi można opatrywać różne elementy kodu, najbardziej przydaje się jednak możliwość pełnego dokumentowania metod:

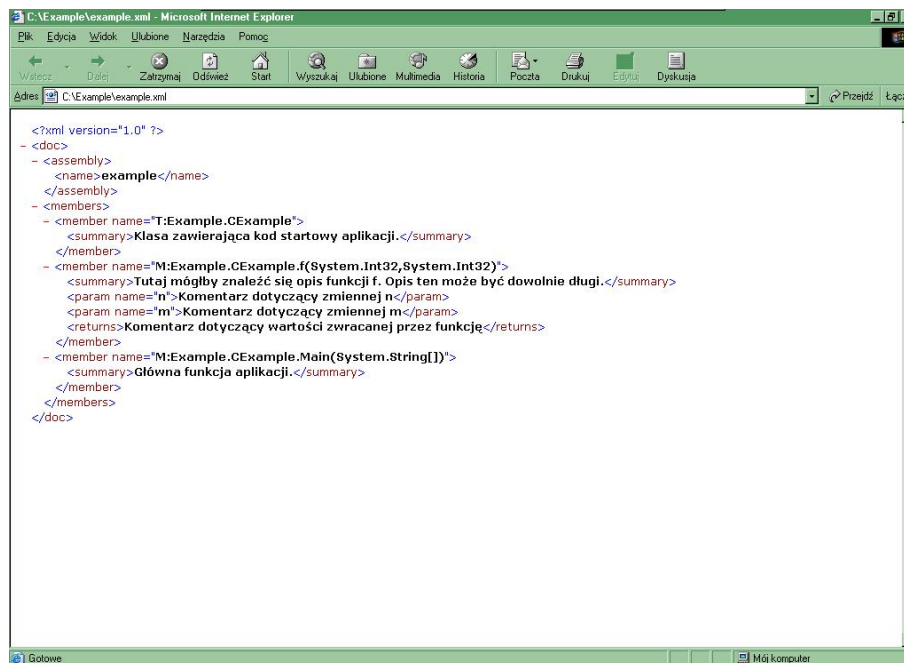
```

/// <summary>
/// Tutaj mógłby znaleźć się opis funkcji f.
/// Opis ten może być dowolnie długi.
/// </summary>
///<param name="n">Komentarz dotyczący zmiennej n</param>
///<param name="m">Komentarz dotyczący zmiennej m</param>
///<returns>Komentarz dotyczący wartości zwracanej przez funkcję</returns>
int f( int n, int m)
{
}

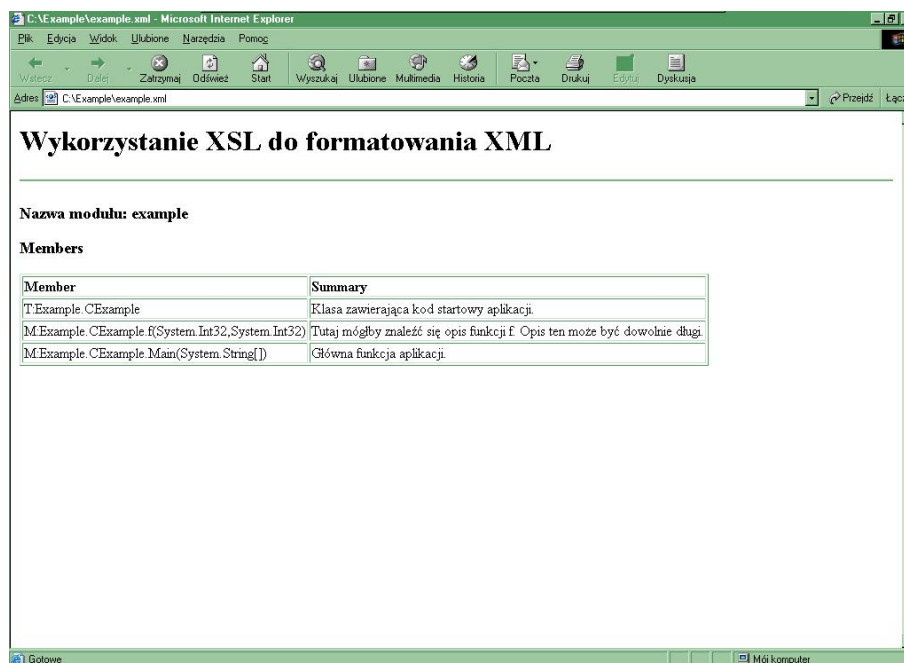
```

Utworzony plik z dokumentacją może być otworzony na przykład przez przeglądarkę Internetową, jednak można, za pomocą arkuszy stylów XSL, nadać mu własne formatowanie. Visual Studio .NET potrafi wykorzystać tę możliwość do utworzenia elegancko sformatowanych stron HTML z dokumentacją.

Wykorzystajmy dla przykładu bardzo prosty arkusz stylów:



Rysunek 3.1: Dokumentacja XML w przeglądarce Internetowej



Rysunek 3.2: Zastosowanie prostego arkusza stylów XSL, przedstawionego w tekście, do sformatowania dokumentacji XML



```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
  <html><body>
    <h1>Wykorzystanie XSL do formatowania XML</h1>
    <hr/>
    <h3>
      Nazwa modułu: <xsl:value-of select="doc/assembly/name"/>
    </h3>
    <table border="1">
      <thead><h3>Members</h3></thead>
      <tbody>
        <tr>
          <td><b>Member</b></td>
          <td><b>Summary</b></td>
        </tr>
        <xsl:for-each select="doc/members/member">
          <tr>
            <td><xsl:value-of select="@name"/></td>
            <td><xsl:value-of select="summary/text()"/></td>
          </tr>
        </xsl:for-each>
      </tbody>
    </table>
  </body></html>
</xsl:template>
</xsl:stylesheet>

```

i w pliku z dokumentacją XML dodajmy informację o arkuszu stylów:

```

<?xml-stylesheet href="example.xsl" type="text/xsl"?>
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>example</name>
  </assembly>
  <members>
    <member name="T:Example.CExample">
      <summary>
        Klasa zawierająca kod startowy aplikacji.
      </summary>
    </member>
    <member name="M:Example.CExample.f(System.Int32,System.Int32)">
      <summary>
        Tutaj mógłby znaleźć się opis funkcji f.
        Opis ten może być dowolnie długi.
      </summary>
      <param name="n">Komentarz dotyczący zmiennej n</param>
      <param name="m">Komentarz dotyczący zmiennej m</param>
      <returns>Komentarz dotyczący wartości zwracanej przez funkcję</returns>
    </member>
    <member name="M:Example.CExample.Main(System.String[])">
      <summary>
        Główna funkcja aplikacji.
      </summary>
    </member>
  </members>
</doc>

```

### 3.2.19 Dekompilacja kodu

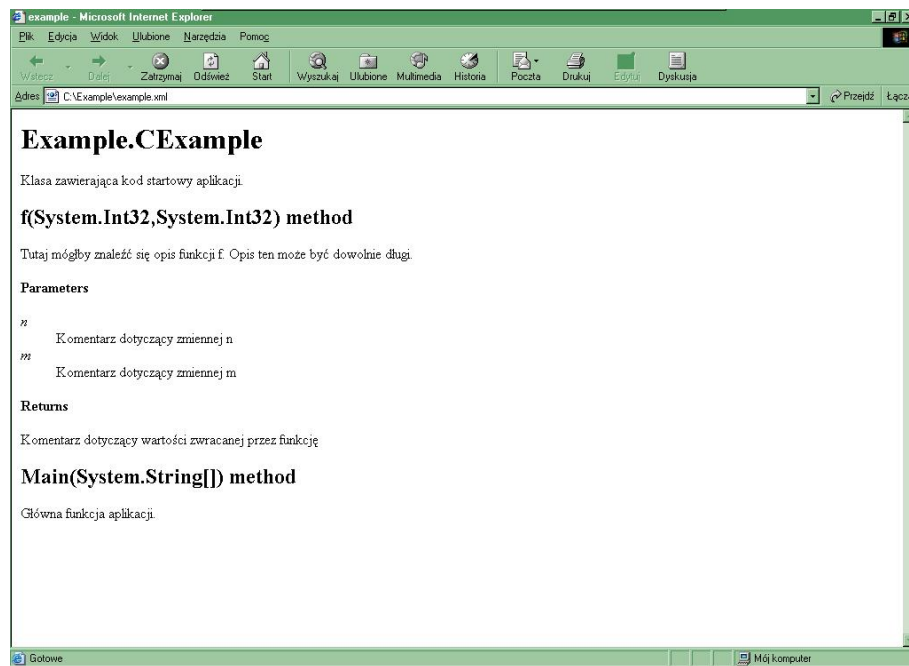
Rozważmy przykład prostego kodu:

```

/* Wiktor Zychla, 2003 */
using System;

namespace NExample
{

```



Rysunek 3.3: Zastosowanie innego arkusza stylów XSL, programista ma tutaj całkowitą dowolność

```

class CExample
{
    public int i;
    public string s;

    public int Oblicz( int n )
    {
        int k = 0;
        for ( int l=0; l<n; l++ )
            k+=1;

        return k;
    }

    public CExample()
    {
        i = 0;
        s = String.Empty;
    }
}

class CMain
{
    public static void Main()
    {
        CExample e = new CExample();
        Console.WriteLine( e.Oblicz(7).ToString() );
    }
}

```

### 3.2.19.1 Dekompilacja do języka CIL

.NET Framework SDK zawiera m.in. dekompiletor kodu, dzięki któremu można obejrzeć kod CIL-owy dowolnego modułu .NETowego<sup>9</sup>. Dekompilator uruchamia się poleceniem *ildasm.exe*.

Dekompilator pozwala obejrzeć kod dowolnego obiektu w dowolnej klasie projektu. Oznacza to, że może być nawet wykorzystany do podglądania kodu bibliotek .NET!

Wykorzystajmy więc *Ildasm* do zdekompilewania metody *Main* z powyższego przykładu:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size          27 (0x1b)
    .maxstack 2
    .locals init (class NExample.CExample V_0,
                 int32 V_1)
    IL_0000: newobj        instance void NExample.CExample::.ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: ldc.i4.7
    IL_0008: callvirt        instance int32 NExample.CExample::Oblicz(int32)
    IL_000d: stloc.1
    IL_000e: ldloc.s        V_1
    IL_0010: call        instance string [mscorlib]System.Int32::ToString()
    IL_0015: call        void [mscorlib]System.Console::WriteLine(string)
    IL_001a: ret
} // end of method CMain::Main
```

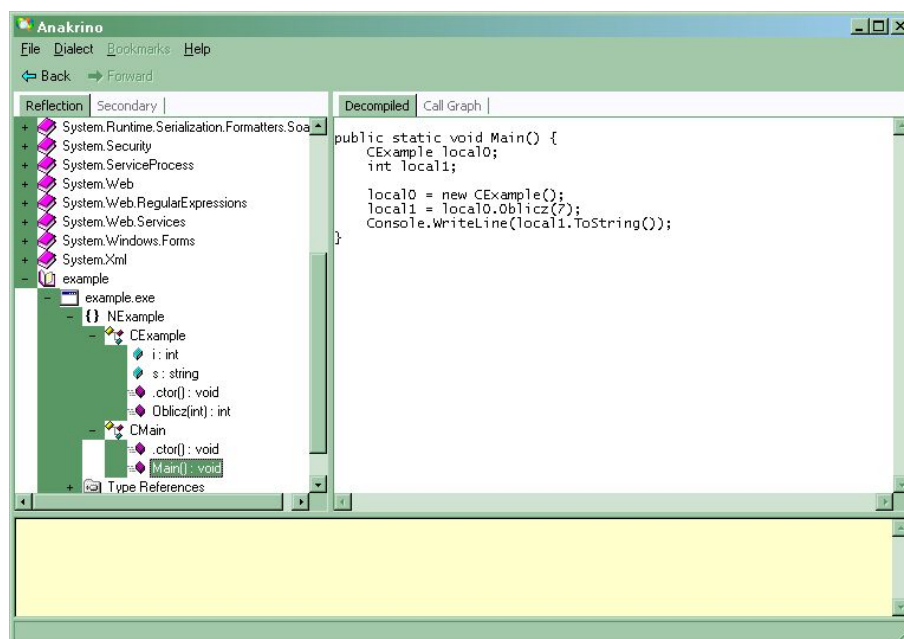
oraz do zdekompilewania metody *CExample.Oblicz*:

```
.method public hidebysig instance int32 Oblicz(int32 n) cil managed
{
    // Code size          24 (0x18)
    .maxstack 2
    .locals init (int32 V_0,
                 int32 V_1,
                 int32 V_2)
    IL_0000: ldc.i4.0
    IL_0001: stloc.0
    IL_0002: ldc.i4.0
    IL_0003: stloc.1
    IL_0004: br.s        IL_000e
    IL_0006: ldloc.0
    IL_0007: ldloc.1
    IL_0008: add
    IL_0009: stloc.0
    IL_000a: ldloc.1
    IL_000b: ldc.i4.1
    IL_000c: add
    IL_000d: stloc.1
    IL_000e: ldloc.1
    IL_000f: ldarg.1
    IL_0010: blt.s        IL_0006
    IL_0012: ldloc.0
    IL_0013: stloc.2
    IL_0014: br.s        IL_0016
    IL_0016: ldloc.2
    IL_0017: ret
} // end of method CExample::Oblicz
```

Struktura kodu pośredniego jest bardzo prosta, kompilator C# nie stosuje praktycznie żadnych optymalizacji. To reguła w świecie .NET - kompilator JIT podczas kompilacji kodu pośredniego do kodu natywnego i tak dokonuje swoich optymalizacji, dlatego kompilatory języków nie muszą tego robić na poziomie kompilacji kodu języka do kodu pośredniego.

Język CIL będzie dokładniej omówiony w rozdziale 3.10.2

<sup>9</sup>Dekompilator CIL jest częścią .NET Framework SDK, kompilator CIL jest częścią samego .NET Frameworka.



Rysunek 3.4: Dekompilacja kodu CIL do C# w Anakrino

### 3.2.19.2 Dekompilacja do C#

Czy można wyobrazić sobie narzędzie, które pozwalałoby odtwarzać kod C# z binarium zawierającego kod pośredni?

Okazuje się, że tak. Najpopularniejszym w tej chwili dekompiłatorem kodu C# jest aktualnie narzędzie **ILSpy**. Zdekompilowany kod zawiera poprawne nazwy klas, metod i parametrów, niepoprawnie natomiast odtwarzane są na przykład nazwy zmiennych lokalnych metod. Dzieje się tak, ponieważ nazwy zmiennych lokalnych nie są przechowywane w kodzie pośrednim.

Zdekompilowany kod metody *Main*:

```
public static void Main() {
    CExample local0;
    int local1;

    local0 = new CExample();
    local1 = local0.Oblicz(7);
    Console.WriteLine(local1.ToString());
}
```

oraz kod metody *CExample.Oblicz*:

```
public int Oblicz(int n) {
    int local0;
    int local1;
    int local2;

    local0 = 0;
    local1 = 0;
    while (local1 < n) {
        local0 += local1;
        local1++;
    }
    local2 = local0;
    return local2;
}
```

<i>Język źródłowy</i>	<i>Język docelowy</i>	<i>Dekompilator</i>
Dowolny język	IL	Ildasm
C#	C#	ILSpy
Dowolny inny język	C#	ILSpy (czasami)
Dowolny język	Inny niż IL lub C#	?

Tabela 3.2: Schemat dekompilacji między różnymi językami platformy .NET

Okazuje się, że struktura kodu pośredniego dla *for* i *while* jest identyczna, dlatego dekompi-lator odtwarza kod każdej pętli jako *while*.

### 3.2.19.3 Zabezpieczanie się przed dekompilacją

Możliwość dekompilacji dowolnego kodu do postaci CILa, a w niektórych wypadkach nawet do postaci kodu C# oznacza, że każdy może analizować kod napisany przez innych programistów. W pierwszej chwili może się więc wydawać że jest to poważna luka, dzięki której osoby niepowołane mogłyby wejść w posiadanie jakichś poufnych informacji.

Chwila zastanowienia wystarczy jednak by dojść do wniosku, że przecież możliwość dekom-pilacji dowolnego kodu do postaci kodu assemblerowego istniała zawsze. Dowolny moduł zawieraj-ający kod maszynowy mógł być analizowany za pomocą debuggerów lub dekompileatorów. To co do tej pory było wręcz niemożliwe, to dekompilacja kodu maszynowego do języków wysokiego poziomu. Kod maszynowy programu kompilowanego kompilatorem Visual Basica nie mógł być w żaden sensowny sposób zdekompilowany z powrotem do postaci kodu VB.

Aby zminimalizować ryzyko związane z analizą zdekompilowanego kodu przez osoby trzecie, należy zastosować narzędzia do zaciemniania kodu (ang *obfuscators*).

## 3.2.20 Kolekcje niegeneryczne - System.Collections

Przygotowując swoją aplikację do określonych zadań, programista musi zmierzyć się z dwoma czynnikami kształtującymi jej obraz: algorytmami i strukturami danych. O ile konkretne algo-rytmy są zwykle zależne od postaci problemu, który aplikacja ma rozwiązywać, o tyle te same struktury danych spotyka się niemal co chwila.

Pewną bardzo specjalną grupę struktur danych stanowią byty, które bardzo ogólnie możnaby nazwać **kontenerami**. Zadaniem kontenerów jest *grupowanie* w większe struktury obiektów, które z jakichś powodów powinny być trzymane razem. Różne języki programowania wspomagają programistów w tym zakresie w różny sposób: w C tablice są częścią języka, wszystkie inne struktury danych programista musi przygotować sam; w C++ możliwości C rozszerzono przez dodanie biblioteki szablonów STL, w której programista może znaleźć większość potrzebnych rodzajów kontenerów oraz algorytmy do operowania na nich.

Programista tworząc aplikację w C# ma do dyspozycji solidną bibliotekę kontenerów (zwa-nych tu *kolekcjami*), zgromadzone w przestrzeni nazw *System.Collections*.

### 3.2.20.1 Tablice

Tablice są najprostszymi kontenerami. W C#, podobnie jak w wielu innych językach, programi-sta po określeniu rozmiaru tablicy nie ma wprost możliwości zmiany jej rozmiaru. Z tego powodu tablice przydają się najczęściej tam, gdzie ilość elementów jest z góry znana.

```
/* Wiktor Zychla, 2003 */
using System;
```

```

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            Console.Write( "Podaj ilosc elementow tablicy: " );

            int    n = int.Parse( Console.ReadLine() );
            int[] tab = new int[n];

            for ( int i=0; i<n; i++ )
                tab[i] = 2*i+1;

            for ( int i=0; i<n; i++ )
                Console.WriteLine( "{0} element tablicy -> {1}", i, tab[i] );
        }
    }
}

```

```

C:\Example>example.exe
Podaj ilosc elementow tablicy: 7
0 element tablicy -> 1
1 element tablicy -> 3
2 element tablicy -> 5
3 element tablicy -> 7
4 element tablicy -> 9
5 element tablicy -> 11
6 element tablicy -> 13

```

Tablice mogą być inicjowane w momencie deklaracji, na przykład:

```
int[] tab = {1, 2, 3, 4, 5, 6};
```

lub równoważnie

```
int[] tab = new int[]{1, 2, 3, 4, 5, 6};
```

Inaczej niż w przypadku prostych imperatywnych języków programowania, tablice w C# są w każdej chwili swojego istnienia świadome swoich atrybutów. Oznacza, to że programista może na przykład w każdej chwili dowiedzieć się jaki jest rozmiar tablicy:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            int[] tab = new int[]{1, 2, 3, 4, 5, 6};

            Console.WriteLine( tab.Length.ToString() );
        }
    }
}

```

```

C:\Example>example.exe
6

```

### 3.2.20.2 Tablice referencji

O ile w przypadku tablic, przechowujących obiekty o typach prostych, dostęp do elementów tablicy możliwy jest natychmiast po przydzieleniu pamięci dla tablicy, o tyle w przypadku typów referencyjnych programista może być w pierwszej chwili zaskoczony:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CObiekt
    {
        public int dana;
        public CObiekt() {}
    }

    public class CExample
    {
        public static void Main(string[] args)
        {
            const int IL = 5;
            CObiekt[] tab = new CObiekt[IL];

            tab[0].dana = 1;
        }
    }
}

C:\Example>example

Unhandled Exception: System.NullReferenceException: Object reference not set to
an instance of an object.
   at Example.CExample.Main(String[] args)

```

Dlaczego próba odwołania do elementu, zainicjowanej przecież tablicy, kończy się niepowodzeniem? Otóż dzieje się tak dlatego, że w przypadku tablic przechowujących obiekty typów referencyjnych, zainicjowanie tablicy:

```

...
const int IL = 5;
CObiekt[] tab = new CObiekt[IL];

```

spowoduje utworzenie kontenera zawierającego 5 niezainicjowanych referencji. Aby odwołać się do elementów tablicy, należy więc oprócz zainicjowania samej tablicy, zainicjować jej elementy:

```

...
const int IL = 5;
CObiekt[] tab = new CObiekt[IL];

for ( int i=0; i<IL; i++ )
    tab[i] = new CObiekt();
...

```

### 3.2.20.3 Tablice wielowymiarowe

Tablice wielowymiarowe deklaruje się w C# równie łatwo jak jednowymiarowe, zaś ich obsługa również nie nastręcza żadnych trudności. W każdej chwili programista może dowiedzieć się jakie są wymiary takiej tablicy:

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            int[, ,] tab = new int[3,5,2,6];
        }
    }
}

```

```

        tab[0, 2, 1, 4] = 3;

        Console.WriteLine( "Tablica {0}-wymiarowa.", tab.Rank );
        for ( int i=0; i<tab.Rank; i++ )
            Console.WriteLine( "Długość w {0}-tym wymiarze : {1}",
                               i, tab.GetLength(i) );
    }
}

```

```

C:\Example>example
Tablica 4-wymiarowa.
Długość w 0-tym wymiarze : 3
Długość w 1-tym wymiarze : 5
Długość w 2-tym wymiarze : 2
Długość w 3-tym wymiarze : 6

```

Pewnym wariantem tablic wielowymiarowych są tzw. *tablice postrzępione* (ang. *jagged arrays*).

```

/* Wiktor Zychla, 2003 */
using System;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            int[][] tab = new int[4][];

            tab[0] = new int[6];
            tab[1] = new int[2];
            tab[2] = new int[3];
            tab[3] = new int[5];

            tab[2][2] = 5;
        }
    }
}

```

Aby zrozumieć różnicę między zwykłymi tablicami wielowymiarowymi, a tablicami postrzępionymi, wyobraźmy sobie 2-wymiarową tablicę zadeklarowaną w następujący sposób:

```

int[,] tab = new int[3,3];
tab[1,1] = 5;

```

oraz jej postrzępionego kuzyna

```

int[][] tab = new int[10][];
tab[0] = new int[3];
tab[1] = new int[2];
tab[2] = new int[1];
tab[1][1] = 5;

```

Tablica dwuwymiarowa ma dokładnie 9 elementów ułożonych w prostokątną macierz 3 na 3 elementy. W przeciwieństwie do niej, tablica postrzępiona przechowuje referencje do trzech tablic, z których pierwsza ma 3 elementy, druga 2, a trzecia tylko 1.

Zarówno w jednym jak i w drugim przypadku z punktu widzenia użytkownika są to tablice dwuwymiarowe, jednak tablica postrzępiona może optymalnie wykorzystywać zasoby pamięci, definiując w razie potrzeby krótsze lub dłuższe "podtablice". Można więc powiedzieć, że  $n$ -wymiarowa tablica jest po prostu macierzą  $n$ -wymiarową, zaś  $n$ -wymiarowa tablica postrzępiona to w istocie  $n$  niezależnych tablic o wymiarze  $n - 1$ , z których każda może mieć inne rozmiary.



### 3.2.20.4 ArrayList

Zwykle tablice zdecydowanie nie rozwiązują problemu kontenerów, bowiem tablice mają bardzo poważną wadę. Otóż ilość elementów tablicy musi być znana w momencie inicjowania tablicy. Gdyby w trakcie działania programu ilość danych uległa zmianie, programista stanąłby przed zadaniem mozolnego przekopiowania tablicy do innej, najprawdopodobniej większej tablicy.

Aby pokonać tę niedogodność należy skorzystać z kolekcji, z których najprostszą jest *ArrayList*. W przeciwieństwie do na przykład kolekcji z STL w C++, *ArrayList* i pozostałe kolekcje .NET korzystają z jednorodnego interfejsu, traktującego wszystko to co wrzucono do kontenera jako *object*.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;
```

```
namespace Example
{
    public class CExample
    {
        static void InfoOKolekcji( ArrayList a )
        {
            Console.WriteLine( "Kolekcja ma {0} elementow: ", a.Count );
            foreach ( object o in a )
                Console.WriteLine( "{0} : {1}", o.GetType(), o );
        }
        public static void Main(string[] args)
        {
            ArrayList a = new ArrayList();

            a.Add( 5 );
            a.Add( 7 );
            a.Add( 9 );
            a.Add( true );
            a.Add( "ala ma kota" );

            InfoOKolekcji( a );
        }
    }
}
```

```
C:\Example>example.exe
Kolekcja ma 5 elementow:
System.Int32 : 5
System.Int32 : 7
System.Int32 : 9
System.Boolean : True
System.String : ala ma kota
```

Oczywiście sytuacja, w której w jednym kontenerze znajdują się obiekty różnych typów jest dość rzadka. Najczęściej programista używa kontenera jak zwykłej tablicy, o której rozmiary nie chce się martwić. Wtedy iteracja przez kolejne elementy może wręcz wymuszać typ elementu

```
...
foreach ( int i in a )
    ...
```

co oczywiście spowoduje wyrzucenie wyjątku, jeśli przypadkiem któryś element kontenera nie jest takiego typu jakiego spodziewa się programista. W przypadku wątpliwości zawsze można rzutować dynamicznie

```
...
foreach ( object o in a )
    if ( o is int )
        ...
```

### 3.2.20.5 Kolekcje silnie otypowane

Programiści, którzy przychodzący ze świata C++, gdzie korzystali z kontenerów z biblioteki STL, niejednokrotnie zgłaszają pod adresem kontenerów C#-owych jedno zastrzeżenie. "Otóż" - jak mawiają - "możliwość umieszczania w kontenerze obiektów dowolnego typu, oznacza podatność takich kontenerów na przypadkowe błędy".

Rzeczywiście, jeśli z jakichś powodów programista spodziewa się w kontenerze obiektów typu `int`, z związku z czym napisze gdzieś w kodzie

```
...
foreach ( int i in a )
    ...
```

to może skończyć się to wyrzuceniem wyjątku, w przypadku omyłkowego umieszczenia w kontenerze obiektu innego typu. Być może nawet obiekt taki umieszczany jest w kontenerze statycznie:

```
...
a.Add( "Ala ma kota" );
...
```

a kompilator mimo to nie zgłasza żadnych zastrzeżeń.

Dzieje się tak dlatego, że jak już powiedziano, kolekcje przechowują referencje do obiektów promując je wcześniej do typu *object* i dopiero na wyraźne życzenie programista może dowiedzieć się jaki jest prawdziwy typ obiektu przechowywanego w kontenerze. Kiedy programista korzysta z C++ kontenera `vector<T>`, kompilator jest w stanie *statycznie* wychwycić tego rodzaju błąd.

Cóż, z perspektywy programisty kolekcje STL mają zdecydowanie poważniejsze wady (wynikające z tego, że zdefiniowane są w postaci szablonów), których nie można w żaden sposób obejść. Okazuje się za to, że w C# przez utworzenie własnej klasy dziedziczącej z **CollectionBase** można zdefiniować kontenery otypowane statycznie.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

public class IntArrayList : System.Collections.CollectionBase
{
    public virtual void Add( int i )
    {
        InnerList.Add( i );
    }
    public int this[int index]
    {
        get { return (int)InnerList[index]; }
    }
}

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            IntArrayList a = new IntArrayList();
            a.Add( 4 );
            a.Add( 7 );
            a.Add( 11 );
            a.Add( "Ala ma kota" );

            Console.WriteLine( a[2] );
        }
    }
}
```

```
}
```

```
C:\Example>csc.exe example.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

example.cs(27,7): error CS1502: The best overloaded method match for
    'IntArrayList.Add(int)' has some invalid arguments
example.cs(27,14): error CS1503: Argument '1': cannot convert from 'string' to
    'int'
```

### 3.2.20.6 Stos, kolejka

Wbudowane w *System.Collections* kontenery **Stack** i **Queue** zachowują się dokładnie tak, jak należałoby tego oczekiwać. Oprócz "zwykłych" operacji wstawiania i zdejmowania elementów, zarówno kolejka jak i stos umożliwiają "podejrzenie" aktualnie dostępnej wartości.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            Stack s = new Stack();

            s.Push( 4 );
            s.Push( "Ala ma kota" );
            s.Push( 3 );
            s.Pop();

            Console.WriteLine( s.Peek() );

            Queue q = new Queue();
            q.Enqueue( 4 );
            q.Enqueue( 5 );
            Console.WriteLine( q.Peek() );

            q.Dequeue();
            Console.WriteLine( q.Peek() );
        }
    }
}
```

### 3.2.20.7 Hashtable

**Hashtable** jest *kolekcją asocjacyjną*, to znaczy że pamięta pary w postaci klucz  $\rightarrow$  wartość. Dzięki wewnętrznej strukturze, czas dostępu do wartości skojarzonej z kluczem jest bardzo szybki i nie zależy od ilości elementów w kolekcji.

Hashtable wykorzystuje się na przykład do pamiętania odwzorowań częściowych (par  $x \rightarrow f(x)$ ) lub fragmentów tabel bazodanowych (par  $ID \rightarrow \text{rekord z tabeli}$ ).

W przeciwieństwie do innych kontenerów, element Hashtable'a jest więc parą typu **DictionaryEntry**. Programista musi o tym pamiętać podczas przeglądania kolekcji.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
```

```

public class CExample
{
    public static void Main(string[] args)
    {
        Hashtable h = new Hashtable();
        h.Add( 5, "Ala ma kota" );
        h.Add( 3, "Kot ma Ale" );
        h.Add( 18, "Ktos ma cos" );

        foreach ( DictionaryEntry de in h )
            Console.WriteLine( "Para {0} - {1}", de.Key, de.Value );
    }
}

```

```

C:\Example>example.exe
Para 18 - Ktos ma cos
Para 5 - Ala ma kota
Para 3 - Kot ma Ale

```

Innym sposobem przeglądania Hashtable'a jest przeglądanie kolekcji kluczy oraz kolekcji wartości niezależnie.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            Hashtable h = new Hashtable();
            h.Add( 5, "Ala ma kota" );
            h.Add( 3, "Kot ma Ale" );
            h.Add( 18, "Ktos ma cos" );

            // przegladaj wartosci
            foreach ( string s in h.Values )
                Console.WriteLine( "Wartosc {0}", s );

            // przegladaj klucze, skojrz wartosci
            foreach ( int key in h.Keys )
                Console.WriteLine( "Para {0} - {1}", key, h[key] );
        }
    }
}

```

```

C:\Example>example.exe
Wartosc Ktos ma cos
Wartosc Ala ma kota
Wartosc Kot ma Ale
Para 18 - Ktos ma cos
Para 5 - Ala ma kota
Para 3 - Kot ma Ale

```

Pewnym zaskoczeniem może być fakt, że elementy Hashtable'a są ujawniane kolejności *innej*, niż były umieszczane w kolekcji. Wyjaśnieniem tego zjawiska i sposobami radzenia sobie z nim zajmiemy się na stronie 149.

### 3.2.21 Enumerowalne kolekcje, interfejsy IEnumerable i IEnumerator

Istnienie wbudowanych kontenerów nie oznacza, że każdy kolejny tworzony kontener musi dziedziczyć z któregoś już zdefiniowanego. Inwencja programistów jest w końcu nieograniczona i

wewnętrzna reprezentacja jakiegos kontenera zdefiniowanego przez uzytkownika może być mocno odległa od typowej.

To czego potrzeba, aby kontener spełniał swoje zadanie, to umożliwienie klientowi korzystającemu z niego jakiegos ogólnego mechanizmu przeglądania elementów, tak aby klient nie musiał być świadomy wewnętrznej reprezentacji danych w kontenerze.

Taką możliwość daje para interfejsów **IEnumerable** oraz **IEnumerator**.

**IEnumerator** ma 3 elementy:

**bool MoveNext()** Metoda **MoveNext** służy klientowi do poinformowania interfejsu o tym, że klient chce obejrzeć kolejny element kontenera. Metoda ta powinna zwrócić wartość *true* jeśli po obejrzeniu kolejnego elementu klient może kontynuować przeglądanie oraz *false* w przeciwnym przypadku

**object Current** Właściwość **Current** powinna ujawniać bieżący element kontenera.

**void Reset()** Metoda **Reset** powinna umożliwić klientowi przywrócenie stanu wyjściowego przeglądania elementów, czyli najczęściej ustawienie bieżącego elementu jako pierwszego elementu kontenera.

**IEnumerable** ma tylko 1 element:

**IEnumerator GetEnumerator()** Metoda **GetEnumerator** służy do pobrania instancji obiektu pozwalającego przeglądać zawartość kolekcji.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class MyCol : IEnumerable
    {
        public class MyColEnumerator : IEnumerator
        {
            int index;
            MyCol myCol = null;

            public bool MoveNext()
            {
                index++;
                if ( index >= IL )
                    return false;
                else
                    return true;
            }

            public object Current
            {
                get { return myCol.t[index]; }
            }

            public void Reset()
            {
                index = -1;
            }

            public MyColEnumerator( MyCol myCol )
            {
                this.myCol = myCol;
                Reset();
            }
        }
    }
}
```

```

const int IL = 3;
int[] t = new int[IL];

public IEnumerator GetEnumerator()
{
    return new MyColEnumerator( this );
}

public MyCol()
{
    for ( int i=0; i<IL; i++ ) t[i] = 2*i;
}
}

public class CExample
{
    public static void Main(string[] args)
    {
        MyCol myCol = new MyCol();

        IEnumerator ie = myCol.GetEnumerator();
        while ( ie.MoveNext() )
            Console.WriteLine( ie.Current.ToString() );
    }
}

C:\Example>example.exe
0
2
4

```

Zaimplementowanie interfejsu **IEnumerable** umożliwia także klientom kontenera na przeglądanie go za pomocą **foreach**. **Foreach** jest **lukrem syntaktycznym**<sup>10</sup>, który w trakcie kompilacji jest tłumaczony do postaci takiej, jak w powyższym przykładzie.

```

...
MyCol myCol = new MyCol();

foreach ( int i in myCol )
    Console.WriteLine( i.ToString() );
...

```

### 3.2.21.1 Sortowanie kolekcji

Framework pozwala rozwiązać problem sortowania w dość elegancki sposób. W przypadku typów prostych kryteria sortowania są już ustalone, zaś programista musi jedynie skorzystać z odpowiednich sposobów ich użycia. W przypadku własnych typów programista może określić różne porządki sortowania przez użycie któregoś z interfejsów: **IComparer** lub **IComparable**.

Zacznijmy od najprostrzego przykładu: sortowania tablic i kolekcji zawierających obiekty typów prostych. Aby osiągnąć zamierzony cel, wystarczy skorzystać ze statycznej funkcji **Array.Sort** w celu posortowania tablicy lub metody **Sort** kolekcji typu **ArrayList**.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

```

<sup>10</sup>Lukier syntaktyczny to sympatyczne tłumaczenie angielskiego terminu *syntactic sugar*. Termin ten oznacza taki element składni języka, który z jednej strony nie wnosi niczego nowego do możliwości samego języka, z drugiej zaś strony upraszcza kod, bądź czyni go przejrzystszym. Typowym przykładem lukru syntaktycznego jest pętla **for** w rodzinie języków C-podobnych. Język nie straciłby nic, gdyby wyeliminować z niego konstrukcję **for (;);**, bowiem to samo można zawsze wyrazić przy pomocy **while**. Pętla **for** jest jednak czytelniejsza.

```

namespace Example
{
    public class CExample
    {
        static void Wypisz( IEnumerable ie )
        {
            Console.Write( "{0}: [", ie.GetType() );
            foreach ( int i in ie )
                Console.Write( "{0}, ", i );
            Console.WriteLine( "]" );
        }

        public static void Main(string[] args)
        {
            const int IL = 10;
            Random r = new Random();

            int[] tab = new int[IL];
            ArrayList atab = new ArrayList();

            for ( int i=0; i<IL; i++ )
            {
                tab[i] = r.Next()%100;
                atab.Add(r.Next()%100);
            }

            Wypisz( tab );
            Array.Sort( tab );
            Wypisz( tab );

            Wypisz( atab );
            atab.Sort();
            Wypisz( atab );
        }
    }
}

C:\Example>example.exe
System.Int32[]: [94,43,42,78,52,50,88,47,73,48,]
System.Int32[]: [42,43,47,48,50,52,73,78,88,94,]
System.Collections.ArrayList: [29,92,23,60,77,15,99,7,46,15,]
System.Collections.ArrayList: [7,15,15,23,29,46,60,77,92,99,]

```

Przy okazji tego przykładu zauważmy, że zarówno tablice jak i kolekcje implementują interfejs *IEnumerable*, zwracający domyślny enumerator do przeglądania elementów w kontenerze. Skorzystaliśmy z tego sprytnie przekazując do funkcji *Wypisz* obiekt typu *IEnumerable*, dzięki czemu jedna i ta sama funkcja służy do przeglądania elementów tablicy i kolekcji.

Powyższy przykład oczywiście nie rozwiązuje problemu, bowiem w przypadku typów użytkownika funkcje sortujące nie miałyby żadnych podstaw do określenia porządku sortowania.

W najprostrzym scenariuszu programista we własnej klasie implementuje interfejs *IComparable*, dzięki któremu instancja obiektu wie jak porównać się z inną instancją obiektu.

Załóżmy, że w klasie *CSoba* mamy pola przechowujące imię i nazwisko i chcemy skonstruować porządek, który w pierwszej kolejności porównywałby nazwisko, zaś w przypadku równych nazwisk porównywałby imiona.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CSoba : IComparable
    {
        public string imie;
        public string nazwisko;
    }
}

```

```

public int CompareTo( object o )
{
    if ( o is COsoba )
    {
        COsoba o2 = o as COsoba;

        if ( this.nazwisko == o2.nazwisko )
            return this.imie.CompareTo( o2.imie );
        else
            return this.nazwisko.CompareTo( o2.nazwisko );
    }
    else
        return -1;
}

public override string ToString()
{
    return String.Format( "{0} {1}", nazwisko, imie );
}

public COsoba( string imie, string nazwisko )
{
    this.imie = imie; this.nazwisko = nazwisko;
}
}

public class CExample
{
    static void Wypisz( IEnumerable ie )
    {
        foreach ( object o in ie )
            Console.WriteLine( "{0}", o );
    }

    public static void Main(string[] args)
    {
        ArrayList atab = new ArrayList();

        atab.Add( new COsoba( "Jan", "Kowalski" ) );
        atab.Add( new COsoba( "Zdzisław", "Kowalski" ) );
        atab.Add( new COsoba( "Jan", "Malinowski" ) );
        atab.Add( new COsoba( "Tomasz", "Abacki" ) );

        Wypisz( atab );
        atab.Sort();
        Wypisz( atab );
    }
}

```

```

C:\Example>example.exe
Kowalski Jan,
Kowalski Zdzisław,
Malinowski Jan,
Abacki Tomasz,

```

```

Abacki Tomasz,
Kowalski Jan,
Kowalski Zdzisław,
Malinowski Jan,

```

Zwróćmy uwagę w jaki sposób odbywa się ustalenie sortowania według 2 pól obiektu: otóż najpierw odbywa się porównanie nazwisk, a następnie, w przypadku równości nazwisk, porównanie imion. Porównanie odbywa się za pomocą tego samego mechanizmu, który jest właśnie oprogramowywany, czyli za pomocą interfejsu *Comparable*, tyle że tym razem metoda pochodzi z klasy *string*.

```

if ( this.nazwisko == o2.nazwisko )

```



```

        return this.imie.CompareTo( o2.imie );
    else
        return this.nazwisko.CompareTo( o2.nazwisko );

```

W taki sam sposób można ustalać dowolne kryteria sortowania według dowolnej ilości pól. Co jednak zrobić w przypadku, kiedy dla jednego rodzaju obiektów chciałoby się kilka różnych porządków sortowania? Załóżmy, że w klasie *COsoba* dołożymy nowe pole określające wiek osoby i chcielibyśmy aby istniał inny porządek sortowania niż alfabetyczny - na przykład porządek chronologiczny (a może jeszcze jakieś inne)? Jak rozwiązać taki problem, skoro zaimplementowanie interfejsu *Comparable* pozwala określić tylko jeden porządek sortowania?

Otóż aby określić więcej niż jeden porządek sortowania należy utworzyć jakąś pomocniczą klasę, która będzie implementować interfejs *IComparer*. Interfejs ten ma tylko jedną metodę *Compare*, która służy do porównywania dwóch obiektów. W celu użycia wybranego interfejsu do uporządkowania obiektów w kontenerze, należy użyć przeciążonej wersji metody *Sort*, która oczekuje jako parametru właśnie obiektu typu *IComparer*.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class COsoba : IComparable
    {
        public class COsobaSortByDataUr : IComparer
        {
            public int Compare( object obj1, object obj2 )
            {
                COsoba o1 = obj1 as COsoba;
                COsoba o2 = obj2 as COsoba;

                return o1.dataUr.CompareTo( o2.dataUr );
            }
            public COsobaSortByDataUr() {}
        }

        public string imie;
        public string nazwisko;
        public DateTime dataUr;

        public int CompareTo( object o )
        {
            if ( o is COsoba )
            {
                COsoba o2 = o as COsoba;

                if ( this.nazwisko == o2.nazwisko )
                    return this.imie.CompareTo( o2.imie );
                else
                    return this.nazwisko.CompareTo( o2.nazwisko );
            }
            else
                return -1;
        }

        public override string ToString()
        {
            return String.Format( "{0} {1}, ur. {2:d}", nazwisko, imie, dataUr );
        }

        public COsoba( string imie, string nazwisko, string dataUr )
        {
            this.imie = imie; this.nazwisko = nazwisko;
            this.dataUr = DateTime.Parse( dataUr );
        }
    }
}

```

```

public class CExample
{
    static void Wypisz( IEnumerable ie )
    {
        Console.WriteLine();
        foreach ( object o in ie )
            Console.WriteLine( "{0}", o );
    }

    public static void Main(string[] args)
    {
        ArrayList atab = new ArrayList();

        atab.Add( new COsoba( "Jan",      "Kowalski", "1994-03-01" ) );
        atab.Add( new COsoba( "Zdzisław", "Kowalski", "1992-11-29" ) );
        atab.Add( new COsoba( "Jan",     "Malinowski", "1990-02-16" ) );
        atab.Add( new COsoba( "Tomasz",  "Abacki",   "1991-01-12" ) );

        Wypisz( atab );
        atab.Sort();
        Wypisz( atab );
        atab.Sort( new COsoba.COsobaSortByDataUr() );
        Wypisz( atab );

    }
}

```

C:\Example>example.exe

```

Kowalski Jan, ur. 1994-03-01,
Kowalski Zdzisław, ur. 1992-11-29,
Malinowski Jan, ur. 1990-02-16,
Abacki Tomasz, ur. 1991-01-12,

```

```

Abacki Tomasz, ur. 1991-01-12,
Kowalski Jan, ur. 1994-03-01,
Kowalski Zdzisław, ur. 1992-11-29,
Malinowski Jan, ur. 1990-02-16,

```

```

Malinowski Jan, ur. 1990-02-16,
Abacki Tomasz, ur. 1991-01-12,
Kowalski Zdzisław, ur. 1992-11-29,
Kowalski Jan, ur. 1994-03-01,

```

Zauważmy, że tam gdzie jawnie nie podano odpowiedniego kryterium sortowania, zostanie użyte sortowanie określone przez interfejs *IComparable* implementowany przez obiekt. Każde inne kryterium musi być użyte jawnie.

W powyższym przykładzie klasa udostępniająca interfejs do sortowania została umieszczona wewnątrz klasy głównej, aby programista korzystający z niej miał świadomość jej przeznaczenia. Mimo to sposób wywołania sortowania nie jest zbyt elegancki:

```

atab.Sort( new COsoba.COsobaSortByDataUr() );

```

Można uczynić kod nieco przejrzystszy przez dołożenie do klasy *COsoba* publicznej statycznej propeccji zwracającej odpowiedni obiekt:

```

...
public class COsoba : IComparable
{
    class COsobaSortByDataUr : IComparer
    {
        public int Compare( object obj1, object obj2 )
        {
            COsoba o1 = obj1 as COsoba;

```

```

        COsoba o2 = obj2 as COsoba;

        return o1.dataUr.CompareTo( o2.dataUr );
    }
    public COsobaSortByDataUr() {}
}

public string    imie;
public string    nazwisko;
public DateTime dataUr;

public static IComparer SortByDataUr
{
    get
    {
        return (IComparer)(new COsobaSortByDataUr());
    }
}
...

```

Klasa implementująca sortowanie nie musi już być publiczna, zaś sortowanie z użyciem odpowiedniego kryterium jest już proste:

```
atab.Sort( COsoba.SortByDataUr );
```

### 3.2.21.2 Opakowywanie enumeratorów

Intensywne korzystanie z kolekcji znacząco wpływa na wydajność pracy programisty. Kod tworzony jest szybciej, jest w nim mniej pomyłek i jest znacznie czytelniejszy.

Załóżmy, że aplikacja rozwija się pomyślnie i w pewnym momencie pojawiają się dodatkowe okoliczności. Elementy jakiegoś kontenera powinny być przeglądnięte, a część z nich, spełniająca jakieś kryteria, usunięta. Naiwnie napisalibyśmy coś, co nieoczekiwanie kończy się katastrofą!

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            ArrayList atab = new ArrayList();

            atab.Add( 5 );
            atab.Add( 10 );
            atab.Add( 3 );

            foreach ( int i in atab )
                if ( i < 4 )
                    atab.Remove( i );
        }
    }
}

```

```
C:\Example>example.exe
```

```

Unhandled Exception: System.InvalidOperationException: Collection was modified;
enumeration operation may not execute.
   at System.Collections.ArrayListEnumeratorSimple.MoveNext()
   at Example.CExample.Main(String[] args)

```

Natknęliśmy się na dość spory problem - w trakcie enumeracji nie wolno modyfikować zawartości kontenera, bowiem enumeracja traci sens, jeśli - obrazowo mówiąc - usuwa się jej grunt spod nóg.

Ten problem można rozwiązać na kilka sposobów, na przykład w czasie enumeracji tworząc pomocniczą listę referencji do obiektów, które należy usunąć, a potem usuwać je w kolejnej pętli, lub korzystając z innych pętli niż *foreach*, gdzie istnieje możliwość wyspecyfikowania bardziej subtelnych warunków zakończenia iteracji, uwzględniających możliwe zmiany w zawartości kontenera.

Okazuje się, że można postąpić jeszcze inaczej, definiując enumerator opakowujący<sup>11</sup>.

Enumerator taki będzie inicjowany dowolnym obiektem, który implementuje interfejs *IEnumerable*, a następnie będzie tworzył kopię referencji do obiektów w źródłowym kontenerze. Jeżeli programista zechce obejrzeć opakowane elementy, dostanie do ręki listę tych właśnie kopii referencji do obiektów z oryginalnej kolekcji.

Aby skorzystać z iteratora opakowującego, zamiast

```
foreach ( int i in atab )
    if ( i < 4 )
        atab.Remove( i );
```

programista napisze

```
foreach ( int i in new IterIsolate( atab ) )
    if ( i < 4 )
        atab.Remove( i );
```

Wadą takiego rozwiązania jest konieczność tworzenia listy z duplikatami referencji do obiektów z oryginalnej kolekcji. Zaletą - niezwykła elegancja kodu.

```
/* Wiktor Zychla, 2003. IterIsolate: Eric Gunnerson */
using System;
using System.Collections;

namespace Example
{
    public class IterIsolate: IEnumerable
    {
        internal class IterIsolateEnumerator: IEnumerator
        {
            protected ArrayList items;
            protected int currentItem;

            internal IterIsolateEnumerator(IEnumerator enumerator)
            {
                IterIsolateEnumerator chainedEnumerator =
                    enumerator as IterIsolateEnumerator;

                if (chainedEnumerator != null)
                {
                    items = chainedEnumerator.items;
                }
                else
                {
                    items = new ArrayList();
                    while (enumerator.MoveNext() != false)
                    {
                        items.Add(enumerator.Current);
                    }
                    IDisposable disposable = enumerator as IDisposable;
                    if (disposable != null)
                    {
                        disposable.Dispose();
                    }
                }
                currentItem = -1;
            }
        }
    }
}
```

<sup>11</sup>Autorem pomysłu jest współtwórca C#, Eric Gunnerson, którego artykuł na ten temat można znaleźć na stronach MSDN.

```

    }

    public void Reset()
    {
        currentItem = -1;
    }

    public bool MoveNext()
    {
        currentItem++;
        if (currentItem == items.Count)
            return false;

        return true;
    }

    public object Current
    {
        get
        {
            return items[currentItem];
        }
    }
}

public IterIsolate(IEnumerable enumerable)
{
    this.enumerable = enumerable;
}

public IEnumerator GetEnumerator()
{
    return new IterIsolateEnumerator(enumerable.GetEnumerator());
}

protected IEnumerable enumerable;
}

public class CExample
{
    public static void Main(string[] args)
    {
        ArrayList atab = new ArrayList();

        atab.Add( 5 );
        atab.Add( 10 );
        atab.Add( 3 );

        foreach ( int i in new IterIsolate( atab ) )
            if ( i < 4 )
                atab.Remove( i );
    }
}

```

Bardzo podobnego pomysłu można użyć do rozwiązania problemu przeglądania elementów kolekcji *Hashtable* w ustalonej przez programistę kolejności. W tym celu utworzymy nowy enumerator opakowujący, który utworzy kopie referencji i posortuje je w ustalonej kolejności.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class IterIsolate: IEnumerable
    {
        ... jak wyżej ...
    }
}

```

```

public class IterSort: IterIsolate, IEnumerable
{
    internal class IterSortEnumerator: IterIsolateEnumerator, IEnumerator
    {
        internal IterSortEnumerator(IEnumerator enumerator, IComparer comparer): base(enumerator)
        {
            if (comparer != null)
            {
                items.Sort(comparer);
            }
            else
            {
                items.Sort();
            }
        }
    }

    public IterSort(IEnumerable enumerable): base(enumerable) {}

    public IterSort(IEnumerable enumerable, IComparer comparer): base(enumerable)
    {
        this.comparer = comparer;
    }

    public new IEnumerator GetEnumerator()
    {
        return new IterSortEnumerator(enumerable.GetEnumerator(), comparer);
    }
    IComparer comparer;
}

public class COsoba : IComparable
{
    ... jak wyżej ...
}

public class CExample
{
    public static void Main(string[] args)
    {
        Hashtable hOsoby = new Hashtable();

        // w kolekcji pamiętamy pary : ID -> Osoba
        hOsoby.Add( 7, new COsoba( "Jan",      "Kowalski", "1994-03-01" ) );
        hOsoby.Add( 10, new COsoba( "Zdzisław", "Kowalski", "1992-11-29" ) );
        hOsoby.Add( 3,  new COsoba( "Jan",     "Malinowski", "1990-02-16" ) );
        hOsoby.Add( 17, new COsoba( "Tomasz",  "Abacki"   , "1991-01-12" ) );

        // przeglądaj kolekcję
        foreach ( COsoba o in hOsoby.Values )
            Console.WriteLine( o.ToString() );

        Console.WriteLine();

        // przeglądaj posortowaną kolekcję
        foreach ( COsoba o in new IterSort( hOsoby.Values, COsoba.SortByDataUr ) )
            Console.WriteLine( o.ToString() );
    }
}

C:\Example>example.exe
Kowalski Zdzisław, ur. 1992-11-29
Kowalski Jan, ur. 1994-03-01
Abacki Tomasz, ur. 1991-01-12
Malinowski Jan, ur. 1990-02-16

Malinowski Jan, ur. 1990-02-16
Abacki Tomasz, ur. 1991-01-12

```

Kowalski Zdzisław, ur. 1992-11-29  
 Kowalski Jan, ur. 1994-03-01

W podobny sposób można utworzyć inne enumeratory opakowujące, na przykład `IterSelect`, który jako parametr przyjąłby predykat przyjmujący jako parametr obiekt z kolekcji i zwracający wartości *true* lub *false*. Podczas przeglądania kolekcji taki enumerator udostępniłby tylko te obiekty z kolekcji, dla których wartość predykatu byłaby równa *true*.

```
public delegate bool IterSelectDelegate(object o);

public class IterSelect: IterIsolate, IEnumerable
{
    internal class IterSelectEnumerator: IterIsolateEnumerator, IEnumerator
    {
        internal IterSelectEnumerator(IEnumerator enumerator,
            IterSelectDelegate selector): base(enumerator)
        {
            for (int index = items.Count - 1; index >= 0; index--)
            {
                if (!selector(items[index]))
                    items.RemoveAt(index);
            }

            currentItem = items.Count;
        }

        public new void Reset()
        {
            currentItem = items.Count;
        }

        public new bool MoveNext()
        {
            currentItem--;
            if (currentItem < 0)
                return false;

            return true;
        }
    }

    public IterSelect(IEnumerable enumerable, IterSelectDelegate selector): base(enumerable)
    {
        this.selector = selector;
    }

    public new IEnumerator GetEnumerator()
    {
        return new IterSelectEnumerator(enumerable.GetEnumerator(), selector);
    }

    IterSelectDelegate selector;
}
```

Dzięki możliwości składania takich enumeratorów, programista mógłby więc napisać:

```
foreach ( COsoba o in
    new IterSelect(
        new IterSort( hOsoby, COsoba.SortByDataUr ),
        COsoba.RokUrodzenia( 1990 ) ) )
{
    // posortowane obiekty COsoba z kolekcji hOsoby
    // ale tylko te urodzone w 1990 roku
}
```

## 3.3 C# 2.0

### 3.3.1 Typy generyczne

#### 3.3.1.1 Motywacja

Rozważmy przykład klasy wektora dwuwymiarowego o składowych całkowitoliczbowych:

```
public class Vector
{
    public Vector()
    {
    }

    public Vector( int x, int y )
    {
        this.X = x;
        this.Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }
}

...
Vector v1 = new Vector(1, 0);
Vector v2 = new Vector(2, 1);
```

*Przykład ten wykorzystuje skrócony sposób definiowania właściwości, o których wcześniej była mowa w podrozdziale 3.2.5.5. Ten nowy skrócony sposób zostanie dokładniej omówiony w rozdziale 3.4.*

Taką konstrukcję można powtórzyć dla co kilku innych typów (`double`, `string`) otrzymując wektory liczb zmiennoprzecinkowych czy wektory słów. Niestety, powtórzenie oznacza, że pojawiają się kolejne typy, niezależne od poprzednich, nie bardzo też jest jak zastosować tu dziedziczenie w celu osiągnięcia reużywalności kodu - wszak składowe wektora miałyby różne typy w różnych wariantach tej klasy.

Pierwszym, oczywistym pomysłem na uogólnienie tego typu konstrukcji, jest użycie najbardziej ogólnego typu do reprezentowania składowych, w tym przypadku typu `object`:

```
public class Vector
{
    public Vector()
    {
    }

    public Vector(object x, object y)
    {
        this.X = x;
        this.Y = y;
    }

    public object X { get; set; }
    public object Y { get; set; }
}

...
Vector v1 = new Vector(1, 0);
Vector v2 = new Vector(2.0, 1.1);
Vector v3 = new Vector("foo", "bar");
```

Niestety, ceną jaką przychodzi zapłacić za takie otwarcie jest utrata **statycznej** kontroli typów. W szczególności kompilator nie zaprotestuje w żaden sposób przeciwko

```
Vector v = new Vector(1, "foo");
```



co być może miewa sens, ale w typowym przypadku wygląda na dość poważny błąd, zwykle możliwy do wykrycia dopiero w trakcie działania aplikacji a nie jej kompilacji.

Cóż, pierwotna wersja, w której typy składowych pojawiały się w definicji klasy jawnie, nie miała takiej wady - tam istniała statyczna kontrola typów podczas kompilacji.

Programowanie generyczne przy pomocy **typów generycznych** pozwala w elegancki sposób **uogólnić** definicję klasy, pozostawiając w niej definicje niektórych składowych w postaci *otwartej*, której *ukonkretnienie* odbywa się dopiero w miejscu użycia. Takie podejście rozwiązuje zaobserwowany wcześniej problem utraty kontroli nad statycznym typowaniem.

```
public class Vector<T>
{
    public Vector()
    {
    }

    public Vector( T x, T y )
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }
}

...
Vector<int>    v1 = new Vector<int>(1, 0);
Vector<double> v2 = new Vector<double>(2.0, 1.1);
Vector<string> v3 = new Vector<string>("foo", "bar");
```

Tu próba obejścia statycznej kontroli typów nie udaje się:

```
Vector<int> v = new Vector<int>(1, "foo");
...
Error CS1503 Argument 2: cannot convert from 'string' to 'int'
```

Z punktu widzenia programisty sprawa jest więc prosta - jedna i ta sama klasa może występować w wielu postaciach, za każdym razem parametr typowy ukonkretnia się tak, jak wynika to z kontekstu użycia. Typy generyczne mogą być argumentami klas

```
public class Foo<T>
{
    ...
}
```

metod w klasach

```
public class Foo
{
    public void Bar<T>( T t )
    {
        ...
    }
}
```

lub interfejsów

```
public interface IFoo<T>
{
    void Bar(T t);
}

public interface Qux
{
    void Baz<T>(T t);
}
```

Nie ma również ograniczeń na liczbę parametrów typowych, może ich być wiele, w tym - część może być na poziomie typu (klasy lub interfejsu) a część na poziomie metody

```
public class Foo<U,V>
{
    public void Bar<S,T>(U u, V v, S s, T t)
    {

    }
}
...
Foo<int, string> foo = new Foo<int, string>();
foo.Bar<double, char>(1, "foo", 2.1, 'c');
```

W tym ostatnim przypadku nie ma nawet potrzeby ukonkretniania parametrów typowych przy wywołaniu metody, ponieważ kompilator może odtworzyć typy automatycznie z kontekstu użycia

```
// to
foo.Bar(1, "foo", 2.1, 'c');

// jest równoważne temu
foo.Bar<double, char>(1, "foo", 2.1, 'c');
```

### 3.3.1.2 Ograniczenia na typy generyczne

Może się wydawać, że mechanizm typów generycznych to całkiem sporo nowych możliwości. Czar pryska w chwili kiedy oprócz danych, klasa musi zawierać również metody.

Klasa wektora generycznego z poprzedniego podrozdziału jest bardzo prosta - zawiera wyłącznie dane. Próba implementacji jakiegokolwiek prostego algorytmu jest ilustracją wspomnianej trudności. Niech tym prostym algorytmem będzie dodawanie wektorów; do bieżącego wektora (**this**) będzie dodawany wektor przesunięcia, a wynik ma być sumą obu wektorów:

```
public class Vector<T>
{
    public Vector()
    {
    }

    public Vector( T x, T y )
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }

    public Vector<T> Add( Vector<T> v )
    {
        return new Vector<T>(this.X + v.X, this.Y + v.Y); // <- czy tak można?
    }
}
...
Error CS0019 Operator '+' cannot be applied to operands of type 'T' and 'T'
```

Sytuacja jest szczególnie zaskakująca dla programistów mających doświadczenie z klasami szablonowymi w C++, gdzie analogiczna konstrukcja przechodzi bez problemów:

```
template <class T>
class Vector
{
public:
```

```

    Vector()
    {

    }
    Vector(T x, T y)
    {
        this->X = x;
        this->Y = y;
    }

    T X;
    T Y;

    Vector Add(Vector const& v)
    {
        return Vector(this->X + v.X, this->Y + v.Y);
    }
};

int main()
{
    Vector<int> v1(1, 0);
    Vector<int> v2(2, 1);

    // w C++ ok
    Vector<int> v3 = v1.Add(v2);
}

```

Zaobserwowana różnica ilustruje **fundamentalną różnicę** w sposobie implementacji typów generycznych między C# a C++.

W języku C++ klasa szablonowa jest traktowana podobnie jak **makrodefinicja** i w trakcie kompilacji samej klasy generycznej kompilator nie próbuje rozstrzygać czy napotkany w wyrażeniu arytmetycznym operator `+` ma sens czy nie w ogólnym przypadku.

Dopiero w trakcie ukonkretnienia szablonu typem, w tym przypadku `int`, kompilator *rozwija definicję klasy szablonowej* i napotyka na wyrażenie w którym po obu stronach `+` występują wartości typu `int`.

W C++ problemem byłoby dopiero

```

class Person
{
    Person()
    {

    }
};

Vector<Person> p1;
Vector<Person> p2;

Vector<Person> p3 = p1.Add(p2);
...
Error C2676 binary '+': 'Person' does not define this operator or a conversion
        to a type acceptable to the predefined operator
Error C2088 '+': illegal for class

```

i słusznie - dla klasy `Person` operacja `+` z pewnością nie ma żadnej sensownej interpretacji, a żadna nie została dostarczona. Rozwiązanie w tym przypadku polega właśnie na dostarczeniu takiej implementacji operatora `+` który nada sens wyrażeniu arytmetycznemu `Person + Person`:

```

class Person
{
public:
    Person()
    {

```

```

    }
    Person operator+(const Person &p)
    {
        // cokolwiek byleby zdefiniować operator
        return Person();
    }
};

```

W C# typy generyczne nie są makrodefinicjami, są w pełni kompilowalnymi klasami, które muszą mieć sens bez względu na to czy i jak będą ukonkretniane. Oznacza to że kompilator musi umieć nadać sens operatorowi `+` oraz każdemu innemu wyrażeniu w którym występuje zmienna otwartego typu bez względu na to czy kiedyś, gdzieś, klasa generyczna będzie ukonkretniana typem `int`, `Person` czy jakimkolwiek innym.

Zauważmy, że główna trudność w C# polega więc na tym, że skoro w klasie generycznej zmienna typowa może oznaczać *dowolny* typ, to problem nie dotyczy tylko operatora `+`, jak w przykładzie klasy `Vector`. Skala problemu jest szersza - o type nie wiadomo niczego, nie wiadomo więc czy można do niego zastosować operator `new` albo czy można na zmiennej tego typu wywoływać jakiegokolwiek metody:

```

public class Foo<T>
{
    public void Bar(T t)
    {
        T _new = new T(); // czy na pewno wolno new T()?

        t.Qux();           // czy T ma metodę Qux()?
    }
}

```

Żadna z przykładowych problematycznych linii w powyższym przykładzie nie powinna się skompilować. Kompilator nie może zezwolić na użycie operatora `new`, ponieważ w miejscu ukonkretnienia typu `Foo` może pojawić się ukonkretnienie typem, do którego `new` nie ma zastosowania, na przykład `int`. Z tego samego powodu kompilator nie może dopuścić do skompilowania kodu, w którym na zmiennej nieznanego typu wywołana jest jakakolwiek metoda.

Język C# z tymi problemami radzi sobie proponując tzw. **ograniczenia typowe**. Są to dodatkowe warunki narzucane statycznie na typy ukonkretniające typ generyczny, które są rodzajem kontraktu między klasą która używa typu a klasą która będzie go ukonkretniać. Dzięki ograniczeniu typowemu klasa używająca zmiennej typowej dostanie dostęp do większej liczby operacji które zostaną uznane za dozwolone, z kolei w miejscu ukonkretnienia kompilator odrzuci typy które nie spełniają ograniczeń.

Dozwolone są następujące ograniczenia typowe:

- **struct** - ograniczenie spełniają tylko typy proste (struktury) (`int` itp.)
- **class** - ograniczenie spełniają tylko typy referencyjne
- **new()** - ograniczenie spełniają tylko typy posiadające konstruktor bezargumentowy (i być może inne konstruktory)
- **: I** - ograniczenie spełniają tylko typy dziedziczące z `I` gdzie `I` jest jakimś konkretnym interfejsem znanym w trakcie kompilacji
- **: C** - ograniczenie spełniają tylko typy dziedziczące z `C` gdzie `C` jest jakimś konkretnym typem znanym w trakcie kompilacji

Nałożenie ograniczenia wymaga dopisania do definicji klasy generycznej słowa kluczowego **where** i wskazaniu jednego (lub wielu, w sytuacji kiedy nie są ze sobą sprzeczne) ograniczenia. Nałożenie ograniczenia daje możliwość użycia funkcjonalności narzuconej ograniczeniem.

```
public interface IQux
{
    void Qux();
}

public class Foo<T>
    where T : IQux, new()
{
    public void Bar(T t)
    {
        T _new = new T(); // czy na pewno wolno new T()?
        // tak, wolno, bo T ma ograniczenie new()

        t.Qux();           // czy T ma metodę Qux()?
        // tak, ma, bo T ma ograniczenie : IQux
    }
}
```

Próba ukonkretnienia klasy generycznej typem niespełniającym ograniczeń nie uda się na etapie kompilacji:

```
public class C1 : IQux
{
    public void Qux() { }
}

public class C2 : IQux
{
    public C2(int n) { }
    public void Qux() { }
}

public class C3
{
}

...

Foo<C1> c1 = new Foo<C1>();
Foo<C2> c2 = new Foo<C2>();
Foo<C3> c3 = new Foo<C3>();

...
Error CS0310 'C2' must be a non-abstract type with a public parameterless
        constructor in order to use it as parameter 'T' in the generic type
        or method 'Foo<T>'

Error CS0311 The type 'C3' cannot be used as type parameter 'T'
        in the generic type or method 'Foo<T>'. There is no implicit reference
        conversion from 'C3' to 'IQux'.
```

W powyższym przykładzie, tylko **C1** spełnia oba ograniczenia typowe - ma bezargumentowy konstruktor i implementuje oczekiwany interfejs. Klasa **C2** nie ma bezargumentowanego konstruktora, a **C3** nie implementuje wskazanego interfejsu.

Typy generyczne dobrze poddają się refleksji, dla dowolnego typu można za pomocą refleksji sprawdzić czy jest to typ generyczny, czy jest otwarty (możliwy do ukonkretnienia) czy ukonkretniony, można również konwertować typy z otwartych do ukonkretnionych i w drugą stronę.

```
public class Foo<T>
{
}
}
```

```

class Program
{
    static void Main(string[] args)
    {
        ReflectionDemo1();
        ReflectionDemo2();
        Console.ReadLine();
    }

    /// <summary>
    /// Pobranie otwartego typu generycznego przez refleksję
    /// i ukonkretnienie go typem int
    /// </summary>
    static void ReflectionDemo1()
    {
        // typ generyczny
        Type foo_t = typeof(Foo<>);
        foreach (Type type in foo_t.GetGenericArguments())
        {
            Console.WriteLine("Argument typowy: " + type.Name);
        }

        // ukonkretnienie
        Type foo_int = foo_t.MakeGenericType(typeof(int));

        object ofi = Activator.CreateInstance(foo_int);

        Console.WriteLine(ofi.GetType().ToString());
    }

    /// <summary>
    /// "Otwarcie" ukonkretnionego typu generycznego
    /// </summary>
    static void ReflectionDemo2()
    {
        Type foo_int = typeof(Foo<int>);

        Type foo_t = foo_int.GetGenericTypeDefinition();

        Console.WriteLine(foo_t.ToString());
    }
}

...
Argument typowy: T
ConsoleApplication18.Foo'1[System.Int32]
ConsoleApplication18.Foo'1[T]

```

### 3.3.1.3 Ograniczenia typowe w praktyce

Z bagażem wiedzy na temat ograniczeń typowych warto wrócić do rzeczywistego przykładu, który był punktem wyjścia do dyskusji, do przykładu klasy `Vector<T>` i przekonać się na ile mechanizm ograniczeń typowych rzeczywiście pomaga poradzić sobie w takim przypadku.

Możliwość zdefiniowania dla klasy przeciążonego operatora `+`, która rozwiązuje przypadek dla własnej klasy `Person` dla klasy szablonowej w C++, tu w języku C# w ogóle nie ma zastosowania - nie ma bowiem takiego ograniczenia typowego, które pozwoliłoby wymagać dla klasy zdefiniowanego operatora `+`. Szczególnym przypadkiem są operatory porównania (`<`), ponieważ w bibliotece standardowej istnieje odpowiadający im interfejs (`IComparable`), który w dodatku jest implementowany przez szereg klas z biblioteki standardowej.

Gdyby więc w klasie `Vector` pojawiła się hipotetyczna metoda w której implementacji użyty był operator porównania

```

public class Vector<T>
{

```

```

...
public bool ShorterOnX( Vector<T> v )
{
    return this.X < v.X;
}
}

```

(należy zwrócić uwagę że w tej hipotetycznej metodzie nie porównuje się długości wektorów wyliczanych zgodnie z regułami algebry, bo do takiego wyliczenia znów potrzebne byłyby operatory matematyczne)

to taki szczególny kod zależny od operatora < można zamienić na poprawny

```

public class Vector<T>
    where T : IComparable
{
    ...
    public bool ShorterOnX( Vector<T> v )
    {
        return this.X.CompareTo( v.X ) < 0;
    }
}

```

wykorzystujący zakładaną semantykę działania operacji **CompareTo**. W takim przypadku klasa generyczna dawałaby się ukonkretnić zarówno typami wbudowanymi (np. **int** - ponieważ **int** implementuje interfejs **IComparable**) jak i własnymi (pod warunkiem zaimplementowania w nich interfejsu **IComparable**).

Ale co z operatorem + w wyrażeniu **this.X + v.X**? Jak już powiedziano, operator + nie posiada odpowiadającego mu interfejsu w bibliotece standardowej.

#### 3.3.1.4 Abstrakcja operacji jako wymagania danego typu

Jedno z możliwych rozwiązań szłoby więc w stronę zdefiniowania własnego interfejsu na wzór **IComparable**, który reprezentowałby abstrakcję operacji dodawania

```

public interface IAddable<T>
{
    T AddTo( T item );
}

```

Refaktoryzacja klasy **Vector** uwzględniająca nowy interfejs i poprawnie implementująca dodawanie wektorów wyglądałaby wtedy tak

```

public class Vector<T>
    where T : IAddable<T>
{
    public Vector()
    {
    }

    public Vector( T x, T y )
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }

    public Vector<T> Add( Vector<T> v )
    {
        return new Vector<T>(this.X.AddTo(v.X), this.Y.AddTo(v.Y));
    }
}

```

To rozwiązanie, aczkolwiek poprawnie od strony technicznej, jest rozczarowująco niepraktyczne. Nie ma bowiem żadnej, najmniejszej możliwości na przekonanie wbudowanego typu `int` żeby zechciał implementować nowo zdefiniowany interfejs `IAddable` - klasy biblioteki standardowej są bowiem zamknięte na takie modyfikacje. Rozwiązanie polegające na wyniesieniu operacji `+` do osobnego interfejsu na typie `T` nie da się więc zastosować w żadnym innym przypadku niż do implementacji wektora własnych typów. A ponieważ typy proste takie jak `int` są zamknięte również na dziedziczenie po nich, to rozwiązanie można potraktować wyłącznie jako ciekawostkę.

### 3.3.1.5 Abstrakcja operacji jako zobowiązanie typu pomocniczego

Skoro więc nie można zmusić istniejącego typu do implementowania abstrakcji operacji dodawania, można tę operację wynieść do całkowicie osobnej abstrakcji. Inaczej mówiąc, można zrobić z interfejsem `IAddable` czymś dla interfejsu `Comparable` jest `Comparer` - wynieść operację poza typ do którego ona się odnosi.

```
public interface IAdder<T>
{
    T Add(T t1, T t2);
}
```

Klasa `Vector` przestaje sama implementować interfejs `IAddable`, zamiast tego wymaga ona **zewnętrznej** implementacji interfejsu dodającego wartości. Technicznie nie ma znaczenia jak ten zewnętrzny obiekt będzie dostarczony, może być wymagany argumentem konstruktora, może być dostarczony jako argument metody.

```
public class Vector<T>
{
    public Vector()
    {
    }

    public Vector(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }

    public Vector<T> Add(Vector<T> v, IAdder<T> adder)
    {
        return new Vector<T>(adder.Add(this.X, v.X), adder.Add(this.Y, v.Y));
    }
}
```

W takiej wersji klasa generycznego wektora poradzi sobie z dodawaniem wektorów dowolnego typu, o ile tylko zostanie z zewnątrz zasilona implementacją "dodawacza":

```
public class IntAdder : IAdder<int>
{
    public int Add( int t1, int t2 )
    {
        return t1 + t2;
    }
}

...
Vector<int> v1 = new Vector<int>(1, 0);
Vector<int> v2 = new Vector<int>(2, 1);

Vector<int> v3 = v1.Add(v2, new IntAdder());
```



To dużo lepsze rozwiązanie niż poprzednie, pozostawia jednak pewien niedosyt "stylistyczny" - to nie wygląda zbyt elegancko kiedy wydawałoby się prosta klasa wektora nie realizuje sama z siebie tak prostej rzeczy jak dodawanie, tylko musi operację dodawania delegować do zewnętrznego obiektu, który z kolei musi być jakoś dostarczony. Mówiąc inaczej, chciałoby się mimo wszystko móc pisać

```
Vector<int> v3 = v1.Add(v2);
```

raczej niż

```
Vector<int> v3 = v1.Add(v2, new IntAdder());
```

Istnieją co najmniej dwa rozwiązania.

### 3.3.1.6 Programowanie dynamiczne

Pierwsze rozwiązanie polega na przerzuceniu ciężaru wykonania operacji + na czas **wykonania** programu, a nie na czas kompilacji. Szerzej ta możliwość zostanie przedstawiona w kolejnych rozdziałach, tu ograniczymy się do jej zademonstrowania.

```
public class Vector<T>
{
    public Vector()
    {
    }

    public Vector(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }

    public Vector<T> Add(Vector<T> v)
    {
        dynamic _x = this.X;
        _x += v.X;

        dynamic _y = this.Y;
        _y += v.Y;

        return new Vector<T>((T)_x, (T)_y);
    }
}
```

Cały ciężar rozwiązania spoczywa teraz na środowisku uruchomieniowym - dla kodu w którym zmienna jest typu **dynamic** kompilator wygeneruje kod, który każde wyrażenie zawierające tę zmienną będzie próbował "wiązać" w trakcie działania programu (przez "wiązać" rozumiemy tu: wyszukiwać czy aktualny typ zmiennej ma metody które wyrażenie próbuje wywołać lub czy implementuje inne wymagania, takie jak wymaganie istnienia operatora+).

Taki kod ma dwie wady:

- jest wolniejszy, dynamiczne wiązanie spowalnia kod od kilku do kilkudziesięciu razy
- może powodować wyjątki dopiero w trakcie działania, przez błędy niewykrywalne w trakcie kompilacji

Istotnie, tak zaimplementowany wektor tym razem bezbłędnie zadziała dla liczb całkowitych

```
Vector<int> v1 = new Vector<int>(1, 0);
Vector<int> v2 = new Vector<int>(2, 1);
```

```
Vector<int> v3 = v1.Add(v2);
```

ale nie zadziała dla klasy niedostarczającej implementacji operacji dodawania, co więcej, wyjątek pojawia się dopiero w trakcie działania programu

```
public class Person
{
    public string Name;

    public Person() { }
    public Person(string Name)
    {
        this.Name = Name;
    }
}
```

Tu próba uruchomienia kodu

```
Vector<Person> p1 = new Vector<Person>(new Person("Jan"), new Person("Kowalski"));
Vector<Person> p2 = new Vector<Person>(new Person("Zofia"), new Person("Malinowska"));

Vector<Person> p3 = p1.Add(p2);
```

spowoduje zgłoszenie wyjątku w trakcie działania programu

```
'Microsoft.CSharp.RuntimeBinder.RuntimeBinderException' occurred in System.Core.dll
```

```
Additional information: Nie można zastosować operatora „+” do argumentów operacji
typu „Person” lub „Person”.
```

Szczęśliwie, remedium polega na dodaniu implementacji przeciążonego operatora dla typu

```
public class Person
{
    public string Name;

    public Person() { }
    public Person(string Name)
    {
        this.Name = Name;
    }

    public static Person operator+(Person p1, Person p2)
    {
        return new Person(p1.Name + p2.Name);
    }
}
```

po którym to dodaniu, wynik dodawania wektorów jest już poprawny.

### 3.3.1.7 Lokalna fabryka "dodawaczy"

Drugie rozwiązanie polega na dostarczeniu klasy gromadzącej wiedzę na temat wszystkich możliwych "dodawaczy" dla wszystkich możliwych typów. Klasa wektora nie będzie już wymagała zasilenia jej zewnętrzną informacją, bo sama będzie wiedziała gdzie tej informacji szukać. Fabryka zostanie wyposażona w podstawową wiedzę o dodawaniu kilku wbudowanych typów i udostępni interfejs do rejestrowania kolejnych. Zastosowany tu wzorzec projektowy w inżynierii oprogramowania nosi nazwę **Factory**.

Jako że rozwiązanie wykorzystuje kilka mechanizmów, które do tej pory nie były prezentowane, zachęca się Czytelnika do powrotu do tej części podręcznika po lekturze kolejnych rozdziałów, w których nowe elementy zostaną szczegółowo omówione.

```

public class AdderFactory
{
    private static Dictionary<Type, object> _adders =
        new Dictionary<Type, object>(); // 0

    static AdderFactory()
    {
        _adders.Add(typeof(int), new IntAdder()); // 1
    }

    public static void RegisterAdder<T>( IAdder<T> adder )
    {
        _adders.Add(typeof(T), adder); // 2
    }

    public IAdder<T> CreateAdder<T>()
    {
        foreach ( Type t in _adders.Keys ) // 3
        {
            if ( t == typeof(T) ) // 4
                return (IAdder<T>)_adders[t];
        }

        throw new ArgumentException(
            string.Format("Brak obsługi logiki dodawania dla typu {0}",
                typeof(T).Name) );
    }
}

public class Vector<T>
{
    public Vector()
    {
    }

    public Vector(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }

    public Vector<T> Add(Vector<T> v)
    {
        IAdder<T> adder = new AdderFactory().CreateAdder<T>(); // 5

        return new Vector<T>(adder.Add( this.X, v.X), adder.Add( this.Y, v.Y ));
    }
}

```

Klasa fabryki przechowuje statyczną mapę, przyporządkowującą typom klasy realizujące algorytmy dodawania dla nich (// 0). Wykorzystano tu nieomawianą do tej pory biblioteczną kolekcję generyczną `Dictionary`, w której kluczami są typy, a wartościami - klasy implementujące interfejs `IAdder`. Ponieważ ograniczenie typowe nie może zmieniać się dla każdego elementu słownika, jako typ wartości musi być tu wybrany typ `object`.

Dalej, klasa fabryki rejestruje znane jej implementacje (tu: tylko jedną ale łatwo to rozszerzyć, // 1) oraz przygotowuje możliwość rejestrowania kolejnych (// 2).

Najistotniejsza jest tu metoda `CreateAdder` tworzenia instancji obiektu `IAdder<T>` dla danego `T`. Metoda ta iteruje po zarejestrowanych implementacjach, szuka takiej której klucz odpowiada żadanemu typowi (// 3) i zwraca zarejestrowany obiekt rzutując go na oczekiwany interfejs (// 4).

Sama klasa wektora używa fabryki do wydobycia instancji obiektu dodającego wartości i używa jej (5).

Takie rozwiązanie jest najbardziej eleganckie ze wszystkich do tej pory przedstawionych, choć nadal ma pewne wady.

Zadziała natychmiast dla wektorów liczb:

```
Vector<int> v1 = new Vector<int>(1, 0);
Vector<int> v2 = new Vector<int>(2, 1);

Vector<int> v3 = v1.Add(v2);
```

ale dla wektorów wartości własnego typu zgłosi wyjątek w czasie działania programu

```
Vector<Person> p1 = new Vector<Person>(new Person("Jan"), new Person("Kowalski"));
Vector<Person> p2 = new Vector<Person>(new Person("Zofia"), new Person("Malinowska"));

Vector<Person> p3 = p1.Add(p2);
...
An unhandled exception of type 'System.ArgumentException' occurred in ConsoleApplication.exe

Additional information: Brak obsługi logiki dodawania dla typu Person
```

Tym razem należy fabrykę nauczyć tego na czym polega dodawanie obiektów typu `Person` przez jednokrotne zarejestrowanie odpowiedniej logiki w fabryce

```
public class PersonAdder : IAdder<Person>
{
    public Person Add( Person p1, Person p2 )
    {
        return new Person(p1.Name + p2.Name);
    }
}
...
AdderFactory.RegisterAdder(new PersonAdder());

Vector<Person> p1 = new Vector<Person>(new Person("Jan"), new Person("Kowalski"));
Vector<Person> p2 = new Vector<Person>(new Person("Zofia"), new Person("Malinowska"));

Vector<Person> p3 = p1.Add(p2);
```

### 3.3.1.8 Wykonywanie kodu generycznego

Nawet jeśli typy generyczne są w pełni kompilowane do CIL, z punktu widzenia środowiska uruchomieniowego w którym kod jest wykonywany, typ generyczny nie może występować w postaci otwartej. Powstaje więc pytanie w jaki sposób kompilator JIT obsługuje typy generyczne.

Okazuje się, że z punktu widzenia uruchamianego kodu, ważne są wyłącznie ukonkretnienia:

- dla każdego ukonkretnienia typem prostym środowisko tworzy nowy, odrębny typ uruchomieniowy. Przykładowo, w przypadku `Vector<T>` oznacza to że dwa różne wystąpienia, `Vector<int>` i `Vector<double>` spowodują powstanie dwóch różnych typów uruchomieniowych, w których `T` jest zastąpione odpowiednim typem prostym
- wszystkie ukonkretnienia typami referencyjnymi współdzielą jedną i tę samą instancję typu uruchomieniowego. Przykładowo, dla wystąpień `Vector<Person>` i `Vector<Account>` będą po skompilowaniu CIL do kodu natywnego przez kompilator JIT obsługiwane jednym i tym samym typem uruchomieniowym

### 3.3.2 Kolekcje generyczne - `System.Collections.Generic`

Wraz z wyposażeniem języka w mechanizm typów generycznych, omówiony wcześniej podsystem kolekcji niegenerycznych, `System.Collections`, został rozszerzony o wsparcie dla kolekcji generycznych. W ramach prezentacji podsystemu zwrócimy uwagę na kolekcje, które są generycznymi odpowiednikami tych kolekcji niegenerycznych, które już omówiono:

Kolekcja generyczna	Odpowiadająca kolekcja niegeneryczna
<code>List &lt; T &gt;</code>	<code>ArrayList</code>
<code>Dictionary &lt; T &gt;</code>	<code>Hashtable</code>
<code>Stack &lt; T &gt;</code>	<code>Stack</code>
<code>Queue &lt; T &gt;</code>	<code>Queue</code>

Biblioteka kolekcji generycznych to bardzo oczywiste zastosowanie mechanizmu typów generycznych. Dzięki konieczności ukonkretnienia kolekcji typem elementu, pomyłka w wyniku której do kolekcji generycznej trafiłby element niewłaściwego typu jest wykrywana w trakcie kompilacji, w odróżnieniu od kolekcji niegenerycznej gdzie ewentualne odkrycie takiego problemu możliwe jest dopiero w trakcie działania programu i to dopiero w miejscu, w którym następuje odczyt danych z kolekcji.

Porównajmy kod który powoduje wyjątek w trakcie działania

```
class Program
{
    static void Main(string[] args)
    {
        ArrayList alist = new ArrayList();
        alist.Add( 1 );
        alist.Add( "2" ); // <- czy na pewno?

        foreach ( int i in alist ) // <- wartości muszą być int
        {
            Console.WriteLine(i);
        }
    }
}
...
An unhandled exception of type 'System.InvalidCastException' occurred in ConsoleApplication.exe
```

z kodem który się nawet nie skompiluje

```
class Program
{
    static void Main(string[] args)
    {
        List<int> alist = new List<int>();
        alist.Add(1);
        alist.Add("2");

        foreach ( int i in alist )
        {
            Console.WriteLine(i);
        }
    }
}
...
Error CS1503 Argument 1: cannot convert from 'string' to 'int'
```

Odpowiednikiem kolekcji nie ograniczającej typu elementu podczas kompilacji byłby `List<object>` aczkolwiek w praktyce bardzo rzadko zachodzi potrzeba takiego luźnego ukonkretniania kolekcji, może to wręcz świadczyć o jakimś problemie projektowym. W sytuacji, w której typy obiektów w kolekcji generycznej nie są bezpośrednio powiązane w hierarchii dziedziczenia, zawsze istnieje możliwość ukonkretniania kolekcji generycznej interfejsem, zamiast konkretnym typem:

```
class Program
{
    static void Main(string[] args)
    {
```

```

        List<I> alist = new List<I>();
        alist.Add(new A());
        alist.Add(new B());

        foreach ( I i in alist )
        {
            // ...
        }
    }

    public interface I { }

    public class A : I { }

    public class B : I { }

```

Podobnie zachowuje się `Dictionary<T, K>` który jest kolekcją asocjacyjną

```

static void Main(string[] args)
{
    Dictionary<int, string> dict = new Dictionary<int, string>();
    dict.Add(1, "foo");
    dict.Add(2, "bar");

    foreach ( KeyValuePair<int, string> i in dict )
    {
        // ...
    }

    foreach ( int i in dict.Keys )
    {
        // ...
    }

    foreach ( string s in dict.Values )
    {
        // ...
    }
}

```

Z uwagi na omówioną wcześniej charakterystykę typów generycznych, we współczesnym programowaniu w C# w zasadzie sporadycznie korzysta się z kolekcji niegenerycznych. Jest to najbardziej jaskrawe w przypadku prostych kolekcji ukonkretnianych typami prostymi (`List<int>`), które przez sposób translacji kodu generycznego do kodu natywnego przez kompilator JIT działają zwyczajnie szybciej niż ich niegeneryczne odpowiedniki (`ArrayList`), w których każda wartość typu prostego musi być pakowana i odpakowywana aby stać się wartością przekazywaną przez referencję (`object`).

### 3.3.2.1 Typy funkcyjne

Wprowadzenie generyczności pozwoliło na wprowadzenie do interfejsu biblioteki standardowej interesujących typów pozwalających na programowanie w stylu funkcyjnym. Rozszerzenia te mają związek z możliwością nadania nazw typom funkcyjnym (omówiony już mechanizm `delegate`) w wersji generycznej.

Ponieważ najwięcej interesujących rozszerzeń otrzymała biblioteczna klasa `List<T>` i to na jej przykładzie warto te rozszerzenia omówić.

**3.3.2.1.1 Funkcje anonimowe** Przed omówieniem rozszerzeń funkcyjnych dla języka, warto również zwrócić uwagę na wprowadzoną do języka możliwość definiowania funkcji w sposób anonimowy, czyli bez potrzeby nadawania im jawnej nazwy.

Rozważmy przykład sygnatury funkcyjnej i metody przyjmującej funkcję jako argument:

```

public delegate int TheDelegate(int n);

...

public void HighOrderFunction( TheDelegate f )
{
    // f jest funkcją, można ją więc wywołać
    // ale jak ją przekazać jako argument?
    int result = f( 1 );
}

...

HighOrderFunction( ?? );

```

We wcześniejszych wersjach języka, przekazanie funkcji do funkcji jako argumentu wiązało się z koniecznością zdefiniowania takiej funkcji w sposób jawny:

```

public delegate int TheDelegate(int n);

...

public void HighOrderFunction( TheDelegate f )
{
    // f jest funkcją, można ją więc wywołać
    int result = f( 1 );
}

public int DoubleArgument( int n )
{
    return n+n;
}

...

// wywołanie funkcji z funkcją jako argumentem
HighOrderFunction(new TheDelegate(Double));

```

Od wersji 2.0 języka można nie tylko pominąć **new** na typie funkcyjnym (typie delegata), które kompilator sam sobie doda

```
HighOrderFunction(DoubleArgument);
```

ale również - funkcji w ogóle nie trzeba definiować na głównym poziomie klasy. Zamiast tego można przekazać ciało funkcji anonimowej wprost, w miejscu w którym spodziewany jest argument

```

HighOrderFunction( delegate( int n ) {
    return n + n;
});

```

Warto zwrócić uwagę na tę składnię. Kolejną rewolucję wnosi tu dopiero C#3.0, który pozwala na definiowanie funkcji anonimowych za pomocą tzw. *lambda wyrażień*. Ta składnia zostanie omówiona w rozdziale 3.4.

**3.3.2.1.2 Akcja - Action<T>** Typ funkcyjny Action<T> oznaczają funkcję która wykonuje jakąś akcję na jednym elemencie określonego typu i nie zwraca żadnych wyników:

```

Action<int> action = delegate (int n)
{
    Console.WriteLine(n);
};

action(1);

```

**3.3.2.1.3 Wyszukiwanie - Predicate<T>** Typ funkcyjny `Predicate<T>` to predykat, czyli funkcja przyjmująca obiekt danego typu jako argument i zwracająca wartość typu `prawda/falsz`:

```
Predicate<int> predicate = delegate (int n)
{
    return n < 5;
};

Console.WriteLine(predicate(1));
Console.WriteLine(predicate(6));
```

**3.3.2.1.4 Porównanie - Comparison<T>** Typ funkcyjny `Comparison<T>` to funkcja przyjmująca dwa argumenty (nazwijmy je *lewy* i *prawy*) i zwracająca wynik:

- -1 jeśli lewy parametr jest **mniejszy** niż prawy
- 0 jeśli lewy i prawy są równe
- 1 jeśli lewy jest **większy** niż prawy

```
Comparison<int> comparison = delegate (int x, int y)
{
    if (x < y) return -1;
    if (x == y) return 0;

    return 1;
};

Console.WriteLine(comparison(1, 2));
Console.WriteLine(comparison(1, 1));
Console.WriteLine(comparison(2, 1));
```

### 3.3.2.2 Wykorzystanie typów funkcyjnych

Wróćmy do wspomnianego wyżej w 3.3.2.1 typu `List<T>` i zobaczmy jak nowe typy funkcyjne wprowadzono do interfejsu tej klasy.

### 3.3.3 Łatwiejsze enumerowanie z yield

W podrozdziale 3.2.21 mieliśmy możliwość zobaczyć w jaki sposób za pomocą pary interfejsów `IEnumerable` i `IEnumerator` można zdefiniować obiekty o enumerowalnej zawartości. Szczególnie wygodna jest możliwość stosowania lukru syntaktycznego `foreach (...)` dzięki której kod enumerowania jest zwizły i jednorodny (niezależny od szczegółów wewnętrznej implementacji enumerowanej klasy).

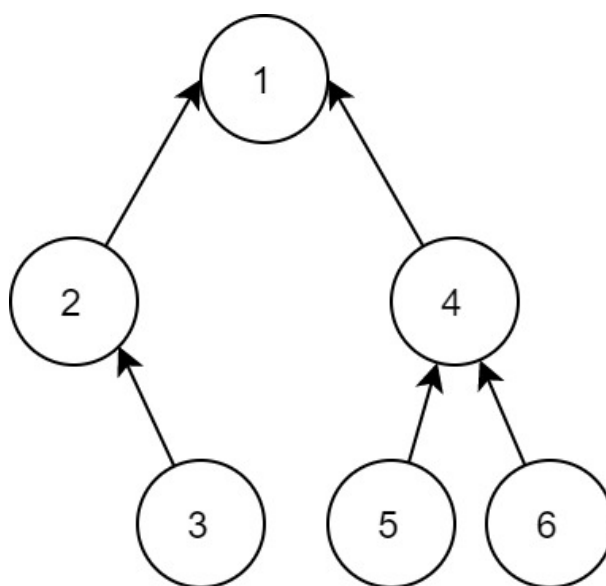
Największą słabością tego podejścia jest jej uciążliwość, jest to szczególnie widoczne wtedy kiedy enumerowany obiekt nie ma prostej, liniowej struktury. Rozważmy przykład drzewa binarnego z rysunku 3.5.

Implementacja struktury mogłaby wyglądać tak:

```
public class Tree
{
    public Tree Left { get; set; }
    public Tree Right { get; set; }
    public int Value { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Tree root = new Tree();
    }
}
```





Rysunek 3.5: Proste drzewo binarne

```

root.Value = 1;

Tree left = new Tree();
left.Value = 2;

Tree subleft = new Tree();
subleft.Value = 3;
left.Right = subleft;

Tree right = new Tree();
right.Value = 4;

Tree subright1 = new Tree();
subright1.Value = 5;
right.Left = subright1;
Tree subright2 = new Tree();
subright2.Value = 6;
right.Right = subright2;

root.Left = left;
root.Right = right;
}

```

To bardzo dobra struktura danych do napisania

```

foreach ( var elem in root )
{
}

// zwraca 1, 2, 3, 4, 5, 6

```

ale oczywiście brakuje tu enumeratora.

Jego implementacja według specyfikacji `IEnumerable` jest zaskakująco nieoczywista. Problem polega na tym że wywołanie metody `MoveNext`, które przesuwa enumerator na "kolejny element" tu musi przesuwać go po strukturze rekurencyjnej w taki sprytny sposób, żeby odwiedzając węzeł jakoś (jak?) wiedzieć, czy to jest to wywołanie w którym enumerator przesunie się

”w głąb” struktury, do węzłów-dzieci, czy też może węzły-dzieci już zostały odwiedzone więc należy wrócić do węzła-ojca i szukać innej ścieżki, do kolejnych, sąsiednich gałęzi.

Poprawny sposób odwiedzania węzłów drzewa binarnego wynika dopiero z wiedzy o strukturach danych. W tym przypadku algorytm wykorzysta dodatkową, pomocniczą strukturę danych do zapamiętania odwiedzonych węzłów. Tą dodatkową pomocniczą strukturą danych będzie albo stos - dla algorytmu odwiedzania drzewa włąb albo kolejka - dla algorytmu odwiedzania odwiedzania drzewa wszerek.

Okazuje się, że metoda `MoveNext` implementowana przy pomocy dodatkowej, pomocniczej struktury danych, może być całkiem prosta:

1. do pomocniczej struktury włóż korzeń drzewa
2. powtarzaj
  - (a) wyjmij element z pomocniczej struktury, zapamiętaj go jako wartość zwracaną z `Current`
  - (b) do pomocniczej struktury włóż dzieci zdjętego węzła (jeśli istnieją)
  - (c) jeśli struktura jest niepusta zwróć `true`, w przeciwnym razie zwróć `false`

Czytelnikowi jako ćwiczenie pozostawia się sprawdzenie że ten prosty algorytm zachowuje się faktycznie podobnie dla struktury stosu i kolejki, w obu tych przypadkach zapewniając inną strategię odwiedzania węzłów drzewa.

Kod w języku C# odpowiadający temu algorytmowi dla struktury drzewa wyglądać mógłby tak:

```
public class Tree : IEnumerable
{
    public class TreeEnumerator : IEnumerator
    {
        private Stack stack = new Stack();

        private Tree root; // zawsze korzeń
        private Tree curr; // bieżący element

        public TreeEnumerator( Tree tree )
        {
            this.root = tree;

            this.Reset();
        }

        public object Current
        {
            get
            {
                if (curr != null)
                {
                    return curr.Value;
                }
                else
                {
                    throw new ArgumentException("Brak elementu do zwrócenia wartości");
                }
            }
        }

        public bool MoveNext()
        {
            if ( stack.Count > 0 )
            {
                // zdejmij element ze stosu
```

```

        curr = (Tree)stack.Pop();

        // do stosu dodaj jego poddrzewo
        if (curr.Right != null) stack.Push(curr.Right);
        if (curr.Left != null) stack.Push(curr.Left);

        // enumeruj dalej
        return true;
    }
    else
    {
        return false;
    }
}

public void Reset()
{
    stack.Clear();
    stack.Push(this.root);
    this.curr = this.root;
}

}

public Tree Left { get; set; }
public Tree Right { get; set; }
public int Value { get; set; }

public IEnumerator GetEnumerator()
{
    return new TreeEnumerator(this);
}
}

```

Czytelnik jest proszony o sprawdzenie że taka implementacja istotnie pozwala enumerować zawartość drzewa włąb za pomocą `foreach` i że zamiana stosu na kolejkę (oraz jeszcze jeden szczegół) pozwala enumerować zawartość wszere.

To co warto zauważyć, to to że bez pomocniczej struktury implementacja mogłaby być jeszcze bardziej skomplikowana (jak bardzo?) ale nawet to nie pomaga poradzić sobie z technicznymi uciążliwościami wynikającymi z rozdzielenia implementacji `MoveNext` i `Current`, która wymusza zapamiętywanie "bieżącego elementu" po to żeby można go było zwrócić.

Okazuje się, że implementacja enumeratora może być znacznie prostsza przy wykorzystaniu dodatkowego, bardzo nietrywialnego lukru syntaktycznego - `yield`. Lukier ten pozwala na połączenie metod `MoveNext` i `Current` w jedną, co jest dość zaskakujące bowiem oznacza że metoda musi równocześnie **zwrócić wartość** (`Current`) i **kontynuować obliczenia** (`MoveNext`).

Zobaczmy prosty przykład

```

public class ExampleEnum : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return 1;
        yield return 2;

        for ( int i=3; i<=5; i++ )
        {
            yield return i;
        }
    }
}

...

ExampleEnum example = new ExampleEnum();
foreach (var elem in example)
{

```

```

        Console.WriteLine(elem);
    }

    // wynik na konsoli: 1 2 3 4 5

```

Okazuje się, że dzięki `yield` metoda zwracająca `IEnumerator` nie musi zwracać faktycznego obiektu implementującego ten interfejs (z metodami `MoveNext`, `Reset` i `Current`) ponieważ kompilator będzie potrafił wygenerować sobie taką implementację sam, na podstawie implementacji pojedynczej metody używającej `yield`.

Kolejna interesująca właściwość - wygenerowana automatycznie przez kompilator metoda `MoveNext` będzie aż 5 razy zwracała wartość `true`, a `Current`, odpowiadający kolejnym wywołaniom `MoveNext`, będzie w tym czasie zwracał kolejno wartości od 1 do 5.

Jak to możliwe? Przecież w implementacji metody widać wyraźnie nietrywialne `for (...)`, które musiałyby jakimś cudem zostać "przerwane" po każdej iteracji (i dla każdego takiego "przerwanego" `for` metoda `MoveNext` musiałaby zwracać `true`) po czym przy kolejnym wywołaniu `MoveNext` działanie tej metody musiałoby zostać **wznowione** dokładnie w tym stanie iteracji pętli (formalnie: z taką wartością zmiennej indeksowej, tu: `i`) z jaką zakończyło się poprzednie wywołanie!

Cóż, właśnie tak się dzieje. Kompilator przepisuje kod z `yield` na kod takiego `MoveNext`, którym wewnętrznie używa maszyny **stanowej**, w której poszczególne stany odpowiadają kolejnym wywołaniom `yield`, a pomiędzy kolejnymi wywołaniami stan jest zapamiętywany tak żeby móc wznowić działanie metody `MoveNext` dokładnie w miejscu jej poprzedniego zakończenia<sup>12</sup>.

Wracając do przykładu drzewa binarnego - to automatyczne zapamiętanie stanu obliczeń przy kolejnych wywołaniach `MoveNext` pozwala enumerator rekurencyjnego drzewa zapisać ... rekurencyjnie co akurat w przypadku odwiedzania drzewa w głąb wygląda wyjątkowo zwięźle i elegancko:

```

public class Tree : IEnumerable
{
    public Tree Left { get; set; }
    public Tree Right { get; set; }
    public int Value { get; set; }

    public IEnumerator GetEnumerator()
    {
        yield return this.Value;

        // rekursja
        if (this.Left != null)
            foreach (var elem in this.Left)
                yield return elem;
        if (this.Right != null)
            foreach (var elem in this.Right)
                yield return elem;
    }
}

```

Być może nie rekursji nie widać tu wprost, ale jest ona ukryta pod `foreach`, które - przypomnijmy - jest lukrem syntaktycznym na `GetEnumerator` i pętlę. W implementacji enumeratora dla korzenia sprytnie wykorzystuje się więc rekursywne wywołanie tegoż właśnie, dopiero co implementowanego, enumeratora dla węzłów-dzieci.

### 3.3.4 Typy proste dopuszczające wartości null

Pojawienie się w języku typów generycznych pozwoliło na implementację

<sup>12</sup>Czytelnikowi zainteresowanemu szczegółami technicznymi sugeruje się użycie wyszukiwarki do wyszukania frazy *C# how yield return works* i przestudiowania artykułów z wyników wyszukiwania

## 3.4 C# 3.0

## 3.5 Biblioteka standardowa platformy .NET

Biblioteka standardowa platformy .NET obejmuje wiele ważnych podsystemów. Za ich pomocą programista może oprogramować system plików, mechanizmy tworzenia wątków i procesów, komunikację z siecią i bazami danych, mechanizmy kryptograficzne, obsługę XML itd.

Co ważne, funkcje biblioteki standardowej .NET są dostępne w **każdym** języku programowania kompilowanym na platformie .NET. Oznacza to na przykład, że z funkcji do obsługi plików z `System.IO` korzysta się tak samo w C#, VB.NET, CIL i każdym innym języku platformy .NET.

Jest to jednak równie ważne autorów kompilatorów nowych języków na platformie .NET, bowiem nie muszą oni przygotowywać własnych implementacji typowych podsystemów, a zamiast tego mogą w swoim języku udostępnić mechanizmy wołania gotowych funkcji z biblioteki standardowej .NET.

### 3.5.1 Biblioteka funkcji matematycznych

Funkcje matematyczne zostały w C# umieszczone jako statyczne w klasie `System.Math`. Programista znajdzie tam takie funkcje, jak m.in.: `Abs`, `Asin`, `Atan`, `Cos`, `Cosh`, `E` (stała), `Exp`, `Floor`, `Log`, `Max`, `Min`, `PI` (stała), `Pow`, `Sign`, `Sin`, `Sinh`, `Tan`, `Sqrt`.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            Console.WriteLine( "E = {0}", Math.E );
            Console.WriteLine( "Pi = {0}", Math.PI );

            Console.WriteLine( "E^PI = {0}", Math.Pow( Math.E, Math.PI ) );
            Console.WriteLine( "Pi^E = {0}", Math.Pow( Math.PI, Math.E ) );
        }
    }
}

C:\Example>example.exe
E = 2,71828182845905
Pi = 3,14159265358979
E^PI = 23,1406926327793
Pi^E = 22,459157718361
```

Podobnie łatwo dzięki klasie `Random` można uzyskać liczby losowe.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            Random r = new Random();

            Console.WriteLine( "Sekwencja losowych liczb całkowitych: " );
            for ( int i=0; i<10; i++ )
```

```

        Console.WriteLine( r.Next() );

        Console.WriteLine( "Sekwencja losowych liczb zmiennoprzecinkowych: " );
        for ( int i=0; i<10; i++ )
            Console.WriteLine( r.NextDouble() );
    }
}

C:\Example>example.exe
Sekwencja losowych liczb całkowitych:
1537211907
1960381545
1107103792
1638000156
206550390
4349299
1902247774
493693260
357003656
1461388247
Sekwencja losowych liczb zmiennoprzecinkowych:
0,359765542372952
0,615490057792277
0,185565924358352
0,0885926261956769
0,67311383582331
0,495275298829784
0,700976026105218
0,496796116464211
0,58237144610955
0,495158822506274

```

### 3.5.2 Biblioteki wejścia/wyjścia

Tyle ile różnych języków - tyle różnych podejść do zagadnienia obsługi wejścia / wyjścia. Projektanci języka stają przed trudnym zadaniem zaprojektowania przystępnego interfejsu programowania do obsługi różnego rodzaju obiektów (pliki, konsola, połączenia sieciowe itd.) i różnego rodzaju rodzajów przekazywania danych (tekstowy, binarny, buforowany, dostęp sekwencyjny i swobodny itd.).

#### 3.5.2.1 Struktura systemu plików

Zanim przejdziemy do przekazywania danych z i do strumieni reprezentujących obiekty wejścia / wyjścia, zajmiemy się operacjami na strukturze systemu plików. Biblioteka udostępnia tu 3 klasy: **File**, **Directory** i **Path**. Żadna z tych klas nie pozwala stworzyć swojej instancji, udostępniają one tylko statyczne metody, z których korzysta programista.

Klasa **Directory** służy do bezpośrednich operacji na plikach i katalogach. Udostępnia m.in. następujące metody:

**CreateDirectory** Tworzenie katalogu.

**Delete** Usuwanie katalogu.

**Exists** Sprawdzanie czy katalog istnieje.

**GetCurrentDirectory** Zwraca bieżący katalog.

**GetFiles** Zwraca listę nazw plików w katalogu.

**GetDirectories** Zwraca listę podkatalogów w katalogu.

**GetFileSystemEntries** Zwraca listę wszystkich elementów w katalogu.

**GetParent** Zwraca nazwę katalogu poziom wyżej niż wskazany.

**GetLogicalDrives** Zwraca listę dysków logicznych w systemie.

**Move** Przesuwa katalog w systemie plików.

**SetCurrentDirectory** Ustawia bieżący katalog.

Klasa **File** udostępnia metody do operacji na plikach, m.in.:

**Copy** Kopiowanie plików.

**Create** Tworzenie nowych plików.

**Delete** Usuwanie plików.

**Exists** Sprawdzanie czy plik istnieje.

**GetAttributes** Zwraca atrybuty pliku.

**Open** Otwiera plik.

**SetAttributes** Ustawia atrybuty pliku.

Klasa **Path** udostępnia metody do obsługi nazw plików w systemie plików, m.in.:

**ChangeExtension** Zmiana rozszerzenia nazwy pliku.

**GetDirectoryName** Część określająca nazwę katalogu w ścieżce.

**GetExtension** Rozszerzenie pliku.

**GetFileName** Nazwa pliku (bez ścieżki).

**GetFileNameWithoutExtension** Nazwa pliku (bez ścieżki i rozszerzenia).

**GetFullPath** Pełna nazwa pliku.

**GetTempName** Nazwa tymczasowego pliku.

**DirectorySeparatorChar** Separator katalogów w nazwach plików (w Windows ””).

**PathSeparator** Separator ścieżek w nazwach plików (w Windows ”;”).

**VolumeSeparatorChar** Separator woluminu w nazwach plików (w Windows ”:”).

Dodatkową, usługową funkcję spełniają dwie klasy, **FileInfo** i **DirectoryInfo**. Za pomocą obiektów tych klas można uzyskać szczegółowe informacje na temat plików i folderów.

Przykład prostego programu:

```
/* Wiktor Zychla, 2003 */
using System;
using System.IO;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
```

```

{
    string[] fileNames =
        Directory.GetFiles( Directory.GetCurrentDirectory(), "*.exe" );
    foreach ( string s in fileNames )
    {
        FileInfo fi = new FileInfo( s );

        Console.WriteLine( fi.FullName );
        Console.WriteLine( " rozmiar\t{0}", fi.Length );
        Console.WriteLine( " utworzony\t{0}", fi.CreationTime );
        Console.WriteLine( " atrybuty\t{0}", fi.Attributes );
    }
}
}
}

```

```

C:\Example>example.exe
C:\Example\example.exe
rozmiar      3584
utworzony    2003-03-22 19:47:55
atrybuty     Archive

```

### 3.5.2.2 Obsługa danych w strumieniach

Interfejsy nowoczesnych języków programowania zwykle używają abstrakcyjnej reprezentacji danych przesyłanych do i z urządzeń wejścia / wyjścia w postaci *strumieni*. Udaną próbę zbudowania jednolitego interfejsu strumieni podjęto przy projektowaniu C++.

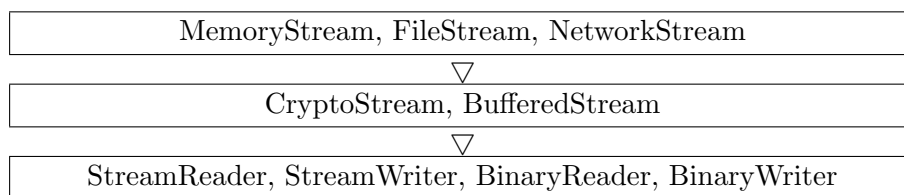


Tabela 3.3: Składanie różnych funkcji strumieni

W C# istnieje klasa *Stream*, która, oprócz udostępniania kilku prostych metod, spełnia funkcję klasy bazowej dla specjalizowanych klas do obsługi różnych strumieni:

**MemoryStream** dostarcza mechanizmów do operacji na danych w pamięci

**FileStream** dostarcza mechanizmów do operacji na plikach

**IsolatedStorageFileStream** wirtualny system plików z kontrolą dostępu

**NetworkStream** dostarcza mechanizmów do operacji sieciowych

Dodatkowe strumienie mogą stanowić warstwę pośrednią w komunikacji z wyżej wymienionymi strumieniami:

**CryptoStream** pozwala szyfrować i deszyfrować dane przesyłane z i do strumienia

**BufferedStream** pozwala przyspieszyć dostęp do strumienia przez wysyłanie większych porcji danych

W zależności od tego jakiego rodzaju dostępu do strumienia oczekuje programista, może on wybierać między:



**StreamReader**, **StreamWriter** pozwala na dostęp do strumieni traktowanych jako napisy

**BinaryReader**, **BinaryWriter** pozwala na dostęp do strumieni traktowanych jak bajty

Te trzy rodzaje funkcji tworzą niejako trzy niezależne warstwy obsługi strumieni, zaś programista może dowolnie składać funkcje z kolejnych warstw. Oznacza to, że tak naprawdę istnieje kilkadziesiąt różnych możliwości ich składania (tabela 3.3).

Warstwa pierwsza udostępnia najprostszy interfejs, w którym do strumienia można kierować i czytać tylko pojedyncze bajty. Warstwa druga umożliwia nałożenie szyfrowania lub buforowania na strumień. Warstwa trzecia pozwala na wysyłanie do strumienia całych napisów, liczb i innych obiektów.

Spróbujmy więc na przykład utworzyć strumień plikowy, na niego nałożyć funkcję zapisu tekstu w Unicode i zapisać do pliku jakiś tekst.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;
using System.IO;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            FileStream fs = new FileStream( "plik.txt", FileMode.Create );
            StreamWriter sw = new StreamWriter( fs, Encoding.Unicode );

            sw.WriteLine( "Chrzasz brzmi w Żyrardówku" );

            sw.Close();
            fs.Close();
        }
    }
}
```

### 3.5.2.3 Szyfrowanie strumieni w locie

Biblioteka wejścia / wyjścia .NET pozwala na umieszczenie strumienia szyfrującego między strumieniem, a obiektem pozwalającym czytać bądź pisać dane tego strumienia. Jest to naprawdę proste i wygodne - z perspektywy programisty zachowanie się strumienia jest nadal takie samo, mimo to dane trafiają do strumienia po przejściu przez warstwę szyfrującą.

Typ	Nazwa
Symetryczny	DES
Symetryczny	TripleDES
Symetryczny	RC2
Symetryczny	Rijndael
Asymetryczny	DSA
Asymetryczny	RSA

Udostępniono kilka znanych protokołów kryptograficznych: *symetryczne* używają tego samego klucza do szyfrowania i deszyfrowania, podczas gdy *asymetryczne* szyfrują za pomocą kluczy publicznych, zaś do odszyfrowania potrzebują kluczy prywatnych. Biblioteka kryptograficzna udostępnia metody do wspomagania tworzenia kluczy dla obu typów protokołów.

W przykładzie najpierw utworzymy strumień do zapisu danych z pośrednim strumieniem szyfrującym, a następnie zdekodujemy tekst z pliku. Gdyby podczas próby dekodowania użyto niepoprawnego hasła, to oczywiście operacja nie powiodłaby się. Oczywiście zawartość pliku z zaszyfrowaną informacją w żaden sposób nie nadaje się do odczytania bez zdeszyfrowania.

```

/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Example
{
    public class CExample
    {

        static string CzytajHaslo()
        {
            Console.Write( "podaj haslo do szyfrowania: " );
            string passwd = Console.ReadLine();
            if ( passwd.Length != 8 )
            {
                Console.WriteLine( "Haslo musi miec 8 znakow" );
                Environment.Exit(0);
            }
            return passwd;
        }

        public static void Main(string[] args)
        {
            string password      = CzytajHaslo();
            UnicodeEncoding UE = new UnicodeEncoding();
            byte[] key           = UE.GetBytes(password);

            // zapis zaszyfrowanych danych
            // cs jest strumieniem pośrednim
            FileStream fs = new FileStream( "plik.txt", FileMode.Create );
            RijndaelManaged RMCrypto = new RijndaelManaged();
            CryptoStream cs = new CryptoStream(fs,
                                                RMCrypto.CreateEncryptor(key, key),
                                                CryptoStreamMode.Write);
            StreamWriter sw = new StreamWriter( cs, Encoding.Unicode );

            sw.WriteLine( "Chrzasz brzmi w Żyrardówku" );
            sw.Close();

            // odczyt zaszyfrowanych danych
            // gs jest strumieniem pośrednim
            FileStream gs = new FileStream( "plik.txt", FileMode.Open );
            RijndaelManaged RMCryptp = new RijndaelManaged();
            CryptoStream ds = new CryptoStream(gs,
                                                RMCryptp.CreateDecryptor(key, key),
                                                CryptoStreamMode.Read);
            StreamReader sq = new StreamReader( ds, Encoding.Unicode );

            Console.WriteLine( sq.ReadLine() );
            sq.Close();
        }
    }
}

C:\Example>example.exe
podaj haslo do szyfrowania: qwertyui
Chrzasz brzmi w Żyrardówku

```

### 3.5.2.4 Strumienie konsoli

Obiekt reprezentujący konsolę dysponuje informacją o strumieniach wejścia, wyjścia i błędów. Obiekty te (`Console.In`, `Console.Out`, `Console.Error`) są strumieniami typów **TextReader** i **TextWriter** (klasy bazowe dla odpowiednio `StreamReader`, `StringReader` i `StreamWriter`, `StringWriter`). Oznacza to, że strumieni tych można użyć w każdym kontekście, w którym używa się strumieni pochodnych.

Strumienie te mogą być przekierowane za pomocą metod **SetIn**, **SetOut** i **SetError**.

### 3.5.3 Dynamiczne tworzenie kodu

Jedną z najciekawszych możliwości biblioteki .NET jest dynamiczne tworzenie kodu w czasie działania aplikacji. Programista może zażyczyć sobie utworzenia instancji obiektu kompilatora, skompilować fragment kodu na dysk lub do pamięci a nawet dynamicznie dołączyć taki kod do swojej aplikacji.

Najpierw zobaczmy w jaki sposób utworzyć dynamicznie obiekt kompilatora, skompilować kod do postaci wykonywalnej, a następnie uruchomić skompilowany kod jako nowy proces w systemie. Chcielibyśmy ponadto, aby tak utworzony obiekt kompilatora przechwytywał i raportował błędy kompilacji.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Diagnostics;
using System.IO;
using System.CodeDom;
using System.CodeDom.Compiler;

using Microsoft.CSharp;

namespace Example
{
    public class CExample
    {
        public static void Main(string[] args)
        {
            string sFileName;
            string sOutFileName;

            Console.WriteLine( "Podaj nazwe pliku do skompilowania: " );
            sFileName = Console.ReadLine();
            sOutFileName =
                Path.GetFileNameWithoutExtension( sFileName ) + ".exe";

            if ( File.Exists( sFileName ) )
            {
                CSharpCodeProvider codeProvider = new CSharpCodeProvider();
                ICodeCompiler icc = codeProvider.CreateCompiler();

                CompilerParameters parameters = new CompilerParameters();
                parameters.GenerateExecutable = true;
                parameters.OutputAssembly = sOutFileName;

                CompilerResults results = icc.CompileAssemblyFromFile( parameters, sFileName );

                if ( results.Errors.Count > 0 )
                {
                    foreach( CompilerError CompErr in results.Errors )
                    {
                        Console.WriteLine( "Linia: " + CompErr.Line +
                                           ", Numer: " + CompErr.ErrorNumber );
                        Console.WriteLine( CompErr.ErrorText );
                    }
                }
            }
            else
            {
            }
        }
    }
}
```

```

        Process.Start( sOutFileName );
    }
}
}
}

C:\Example>example.exe
Podaj nazwe pliku do skompilowania: test.cs
Linia: 5, Numer: CS1514
{ expected

```

Dynamiczne kompilowanie kodu w sposób pokazany w powyższym przykładzie ma jednak kilka wad:

- podczas kompilacji tworzony jest plik wykonywalny ze skompilowanym kodem
- kompilowany kod musi być w pełni samodzielny, w szczególności musi zawierać funkcję **Main**
- skompilowany proces podczas uruchamiania tworzy nowe okno konsoli

Aby poradzić sobie z tymi problemami, po pierwsze zażyczymy sobie tworzenia kodu do pamięci zamiast na dysk. Po drugie, skorzystamy z mechanizmu refleksji, dzięki któremu będziemy mogli obejrzeć składniki skompilowanego do pamięci kodu. Po trzecie, wykorzystamy mechanizm pozwalający na tworzenie delegatów z obiektów typu **MethodInfo**, dzięki czemu będziemy mogli wybrać z kompilowanego kodu tylko te metody, które są interesujące.

Przygotujmy najpierw testowy plik z przykładowymi funkcjami:

```

/*
    test.cs
    plik z przykładowymi funkcjami, który będzie dynamicznie kompilowany
*/
using System;

namespace NSpace
{
    public class CMain
    {
        public static int A( int n )
        {
            return n+n;
        }

        public static int B( int n )
        {
            return n*n;
        }
    }
}

```

A oto zmodyfikowany przykład dynamicznego tworzenia kodu:

```

/* Wiktor Zychla, 2003 */
using System;
using System.Diagnostics;
using System.IO;
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Reflection;

using Microsoft.CSharp;

namespace Example
{
    public class CExample

```

```

{
    public delegate int DF( int n );
    public static DF DummyDF = new DF( FDummy );
    public static int FDummy( int n )
    {
        return 0;
    }

    public static void Main(string[] args)
    {
        string sFileName;
        string sOutFileName;

        Console.Write( "Podaj nazwe pliku do skompilowania: " );
        sFileName = Console.ReadLine();
        sOutFileName =
            Path.GetFileNameWithoutExtension( sFileName ) + ".exe";

        if ( File.Exists( sFileName ) )
        {
            CSharpCodeProvider codeProvider = new CSharpCodeProvider();
            ICodeCompiler icc = codeProvider.CreateCompiler();

            CompilerParameters parameters = new CompilerParameters();
            parameters.GenerateExecutable = false;
            parameters.OutputAssembly = sOutFileName;

            CompilerResults results =
                icc.CompileAssemblyFromFile( parameters, sFileName );

            if (results.Errors.Count > 0)
            {
                foreach(CompilerError CompErr in results.Errors)
                {
                    Console.WriteLine( "Linia: " + CompErr.Line +
                                         ", Numer: " + CompErr.ErrorNumber );
                    Console.WriteLine( CompErr.ErrorText );
                }
            }
            else
            {
                try
                {
                    Assembly assembly = results.CompiledAssembly;

                    Console.Write( "Podaj nazwę typu: " );
                    Type t = assembly.GetType( Console.ReadLine() );

                    Console.Write( "Podaj nazwę funkcji o prototypie int F(int): " );
                    MethodInfo me = t.GetMethod( Console.ReadLine() );

                    DF df = (DF)DF.CreateDelegate( DummyDF.GetType(), me );
                    Console.Write( "Podaj wartość parametru (int): " );

                    int result = df( int.Parse( Console.ReadLine() ) );
                    Console.WriteLine( result );
                }
                catch ( Exception ex )
                {
                    Console.WriteLine( ex.Message );
                }
            }
        }
    }
}

```

```

C:\Example>example.exe
Podaj nazwe pliku do skompilowania: test.cs
Podaj nazwę typu: NSpace.CMain

```

Podaj nazwę funkcji o prototypie `int F(int): A`  
 Podaj wartość parametru (`int`): 24  
 48

`C:\Example>example.exe`  
 Podaj nazwę pliku do skompilowania: `test.cs`  
 Podaj nazwę typu: `Nspace.CMain`  
 Podaj nazwę funkcji o prototypie `int F(int): B`  
 Podaj wartość parametru (`int`): 25  
 625

Cała siła tego kodu opiera się na linii

```
DF df = (DF)DF.CreateDelegate( DummyDF.GetType(), me );
```

Tworzony jest tutaj delegat typu *DF* za pomocą statycznej funkcji *CreateDelegate*, która w tej (jednej z 4) wersji spodziewa się parametru określającego typ tworzonego delegata (tu: domyślnego delegata typu *DF* utworzonego w kodzie) oraz informacji o metodzie pobranej przez mechanizm refleksji.

Bardzo prosto napisać funkcję, która będzie mogła ewaluować wyrażenie dowolnego typu:

```
using System;
using System.CodeDom;
using System.CodeDom.Compiler;
using Microsoft.CSharp;
using System.Text;
using System.Reflection;

namespace Example
{
    public class Evaluator
    {
        public static object Evaluate( Type type, string expression )
        {
            ICodeCompiler comp = (new CSharpCodeProvider()).CreateCompiler();
            CompilerParameters cp = new CompilerParameters();
            cp.ReferencedAssemblies.Add("system.dll");
            cp.ReferencedAssemblies.Add("system.data.dll");
            cp.ReferencedAssemblies.Add("system.xml.dll");
            cp.GenerateExecutable = false;
            cp.GenerateInMemory = true;

            StringBuilder code = new StringBuilder();
            code.Append("using System; \n");
            code.Append("using System.Data; \n");
            code.Append("using System.Data.SqlClient; \n");
            code.Append("using System.Data.OleDb; \n");
            code.Append("using System.Xml; \n");
            code.Append("namespace _Evaluator { \n");
            code.Append("    public class _Evaluator { \n");
            code.AppendFormat("        public {0} Foo() ", type.Name );
            code.Append("{ ");
            code.AppendFormat("            return ({0}); ", expression);
            code.Append("}\n");
            code.Append("} }");

            CompilerResults cr =
                comp.CompileAssemblyFromSource(cp, code.ToString());

            if (cr.Errors.HasErrors)
            {
                StringBuilder error = new StringBuilder();
                error.Append("Error Compiling Expression: ");
                foreach (CompilerError err in cr.Errors)
                {
                    error.AppendFormat("{0}\n", err.ErrorText);
                }
            }
        }
    }
}
```

```

        throw new Exception("Error Compiling Expression: " +
                             error.ToString());
    }
    Assembly    a = cr.CompiledAssembly;
    object      c = a.CreateInstance("_Evaluator._Evaluator");
    MethodInfo mi = c.GetType().GetMethod("Foo");
    return mi.Invoke( c, null );
}
}

public class CMain
{
    public static void Main()
    {
        Console.Write( "Wpisz wyrażenie arytmetyczne: " );
        Console.WriteLine( (int)Evaluator.Evaluate( typeof(int),
                                                    Console.ReadLine() ) );
    }
}
}

C:\Example>example
Wpisz wyrażenie arytmetyczne: (8*(4+6))-12
68

C:\Example>example
Wpisz wyrażenie arytmetyczne: 5+(

Unhandled Exception: System.Exception: Error Compiling Expression: Error Compiling Expression: Invalid expression term ')'
) expected

at Example.Evaluator.Evaluate(Type type, String expression)
at Example.CMain.Main()

```

Zastosowania takiej metody tworzenia kodu mogą być bardzo szerokie. Można na przykład wyposażyć aplikację w moduł skryptowy, który pozwoli użytkownikowi podkładać własne funkcje w miejsce dostarczanych z aplikacją. Można zaprojektować całkowicie własny język skryptowy, dopisać prosty kompilator między tym językiem a C#, następnie kompilować kod w języku skryptowym najpierw do C#, a kod C# dynamicznie dołączać do własnej aplikacji w czasie jej działania.

Można również wyobrazić sobie, że pewne ważne fragmenty aplikacji dostarczone są w postaci zaszyfrowanego kodu źródłowego, do którego kod odszyfrowujący zna tylko użytkownik. Kod taki mógłby być odszyfrowywany i kompilowany w czasie działania aplikacji, zaś użytkownik miałby pewność, że w przypadku kradzieży aplikacja byłaby dla ewentualnego złodzieja bezużyteczna, gdyby nie znał hasła odszyfrowującego.

### 3.5.4 Procesy, wątki

#### 3.5.4.1 Procesy

Dzięki bibliotece *System.Diagnostics* programista może kontrolować procesy działające w systemie.

```

using System;
using System.Diagnostics;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            Process[] pt = Process.GetProcesses();

```

```

foreach ( Process p in pt )
{
    Console.WriteLine( p.ProcessName.ToString() );
    Console.WriteLine( "\t"+p.PriorityClass.ToString() );
    Console.WriteLine( "\t"+p.MainModule.ModuleName );
    Console.WriteLine( "\t"+p.MainModule.ModuleMemorySize );
}
}
}
}

```

Procesy można nie tylko przeglądać, ale również tworzyć, zabijać, czekać na ich zakończenie oraz korzystać z możliwości powłoki. Poniższy przykład jest C#-owym odpowiednikiem przykładu ze strony 66.

```

using System;
using System.Diagnostics;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            Process p = new Process();

            p.StartInfo.UseShellExecute = true;
            p.StartInfo.Verb           = "print";
            p.StartInfo.FileName       = "plik.doc";

            p.Start();
            p.WaitForExit();
            p.Dispose();

            Console.WriteLine( "zakończono drukowanie" );
        }
    }
}

```

### 3.5.4.2 Wątki

Biblioteka *System.Threading* udostępnia funkcje do tworzenia i synchronizacji wątków. Przekazywanie parametrów do wątków możliwe jest dzięki opakowywaniu ich w pomocnicze klasy:

```

using System;
using System.Threading;

namespace Example
{
    public class CMyThread
    {
        string nazwa;
        int k;

        public void ThreadFunc()
        {
            for ( int i=0; i<k; i++ )
            {
                Console.WriteLine( nazwa );
                Thread.Sleep( 1 );
            }
        }

        public CMyThread( string nazwa, int k )
        {
            this.nazwa = nazwa;
            this.k      = k;
        }
    }
}

```



```

    }
}

public class CMain
{
    public static void Main()
    {
        string[] n = { "Jurek", "Ogórek", "Kiełbasa", "Sznurek" };
        int[] m = { 4, 5, 2, 7 };

        for ( int i=0; i<n.Length; i++ )
        {
            CMyThread mt = new CMyThread( n[i], m[i] );
            Thread t = new Thread( new ThreadStart( mt.ThreadFunc ) );
            t.Start();
        }
    }
}

```

```
C:\Example>example
```

```

Ogórek
Kiełbasa
Kiełbasa
Ogórek
Ogórek
Sznurek
Jurek
Sznurek
Ogórek
Jurek
Ogórek
Sznurek
Sznurek
Sznurek
Jurek
Sznurek
Jurek
Sznurek

```

Najprostszy wariant synchronizacji wątków możliwy jest dzięki słowu kluczowemu C# `lock`. Objęcie jakiejś zmiennej tą klauzulą powoduje zablokowanie dostępu do tej zmiennej pozostałym wątkom na tak długo, aż bieżący wątek opuści blok `lock`, na przykład:

```

JakisObiekt o;
...
lock ( o )
{
    ...
}

```

Do synchronizacji wątków można również wykorzystać m.in.:

- `AutoResetEvent`
- `ManualResetEvent`
- `Monitor`
- `Mutex`

### 3.5.5 XML

Tam, gdzie zachodzi konieczność wymiany danych, tam programiści muszą ustalić jakiś sposób ich przekazywania. Wyobraźmy sobie scenariusz, w którym aplikacja A powinna udostępniać jakiś zbiór danych aplikacji B.

Najbardziej naiwnym podejściem byłoby dodanie do aplikacji B modułu pozwalającego na wczytywanie danych bezpośrednio w formie, w jakiej składa je aplikacja A. Takie rozwiązanie nie sprawdza się jednak wtedy, gdy format danych aplikacji A ulegnie z jakiegoś powodu zmianie (najczęściej rozszerzeniu).

Wydawałoby się więc, że rozwiązaniem byłoby zaprojektowanie jakiegoś formatu pliku pozwalającego na przekazywanie danych między aplikacjami. Takie rozwiązanie mogłoby być niezależne od wewnętrznych formatów danych aplikacji A i B. Gdyby jednak przekazywać dane w formie binarnej, to oczywiście w razie chęci rozszerzenia zakresu przekazywanych danych cała zabawa zaczyna się od początku.

Dość nieoczekiwanie rozwiązaniem tego i podobnych problemów okazało się zaprojektowanie standardu XML (ang *eXtended Markup Language*), czyli pewnego standardu budowania plików tekstowych do przesyłania dowolnego typu danych. XML przyjął się głównie dzięki temu, że uprościł przesyłanie danych w sieci, gdzie informacja w formie tekstowej okazuje się być często jedynym wspólnym mianownikiem dla różnego rodzaju platform biorących udział w komunikacji.

Sam w sobie XML jest rozszerzeniem idei HTML - o ile jednak HTML narzuca ściśle zbiór możliwych nazw węzłów jest ściśle ustalony, o tyle XML pozwala ustalać je dowolnie. Tak jak pliki HTML, tak i pliki XML mogą być budowane nawet w zwykłym edytorze tekstów. Przeglądarki internetowe potrafią najczęściej przeglądać takie pliki (była już o tym mowa na stronie 126).

Biblioteka *System.Xml* pozwala na manipulowanie plikami XML na 3 sposoby (a właściwie 2 i pół, bo ostatni jest tylko rozszerzeniem poprzedniego):

- za pomocą obiektów *System.Xml.XmlTextReader* i *System.Xml.XmlTextWriter*
- za pomocą obiektów DOM (klasa *System.Xml.XmlDocument*)
- za pomocą obiektów z *System.Xml.XPath*

Pierwsze dwie możliwości pozwalają na programowe czytanie i tworzenie plików XML, zaś obiekty *XPath* wspomagają tylko odczyt struktury XML.

Przygotujmy najpierw prosty plik z danymi:

```
<?xml version="1.0" encoding="windows-1250"?>
<ListaOsób nazwa="znajomi">
  <Osoba>
    <Imię>Jan</Imię>
    <Nazwisko>Kowalski</Nazwisko>
    <DataUr>1950-02-07</DataUr>
  </Osoba>
  <Osoba>
    <Imię>Tomasz</Imię>
    <Nazwisko>Malinowski</Nazwisko>
    <DataUr>1976-10-04</DataUr>
  </Osoba>
  <Osoba>
    <Imię>Adam</Imię>
    <Nazwisko>Nowak</Nazwisko>
    <DataUr>1984-02-17</DataUr>
  </Osoba>
</ListaOsób>
```

### 3.5.5.1 *XmlTextReader* i *XmlTextWriter*

Z perspektywy obiektów *XmlTextReader* i *XmlTextWriter* pliki XML są strumieniami danych. Odczyt zawartości pliku XML za pomocą obiektu *XmlTextReader* może wyglądać następująco:

```
/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Xml;
```

```

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            FileStream fs = new FileStream( "osoby.xml", FileMode.Open );
            XmlTextReader xr = new XmlTextReader( fs );

            while ( xr.Read() )
            {
                XmlNodeType t = xr.NodeType;
                Console.WriteLine( "Węzeł typu {0}", t );

                if ( t == XmlNodeType.Element )
                {
                    Console.WriteLine( "\t<{0}>", xr.Name );
                    if ( xr.HasAttributes )
                        while ( xr.MoveToNextAttribute() )
                            Console.WriteLine( "\t\t[{0}]{1}", xr.Name, xr.Value );
                }
                if ( t == XmlNodeType.Text )
                    Console.WriteLine( "Tekst: {0}", xr.Value );
            }

            xr.Close();
        }
    }
}

```

```

C:\Example>example.exe
Węzeł typu XmlDeclaration
Węzeł typu Whitespace
Węzeł typu Element
<ListaOsób>
[nazwa]znajomi
Węzeł typu Whitespace
Węzeł typu Element
<Osoba>
Węzeł typu Whitespace
Węzeł typu Element
<Imię>
Węzeł typu Text
Tekst: Jan
Węzeł typu EndElement
Węzeł typu Whitespace
Węzeł typu Element
<Nazwisko>
Węzeł typu Text
Tekst: Kowalski
Węzeł typu EndElement
Węzeł typu Whitespace
Węzeł typu Element
<DataUr>
Węzeł typu Text
Tekst: 1950-02-07
...

```

### 3.5.5.2 Obiekty DOM

Obiekty DOM najpierw wczytują zawartość XMLa do pamięci, a następnie budują drzewo składowe, które programista może przeglądać rekursywnie zgodnie z jego strukturą:

```

/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Xml;

```

```

namespace Example
{
    public class CMain
    {
        public static void InfoOWezle( XmlNode node )
        {
            Console.WriteLine( "Węzeł typu {0}", node.NodeType );

            if ( node.NodeType == XmlNodeType.Element )
            {
                Console.WriteLine( "\t<{0}>", node.Name );
                foreach ( XmlAttribute a in node.Attributes )
                    Console.WriteLine( "\t\t[{0}]{1}", a.Name, a.Value );
            }
            if ( node.NodeType == XmlNodeType.Text )
                Console.WriteLine( "Tekst: {0}", node.Value );

            foreach ( XmlNode child in node.ChildNodes )
                InfoOWezle( child );
        }

        public static void Main()
        {
            XmlDocument xml = new XmlDocument();
            xml.Load( "osoby.xml" );

            InfoOWezle( xml );
        }
    }
}

```

```

C:\Example>example.exe
Węzeł typu Document
Węzeł typu XmlDeclaration
Węzeł typu Element
<ListaOsób>
[nazwa]znajomi
Węzeł typu Element
<Osoba>
Węzeł typu Element
<Imię>
Węzeł typu Text
Tekst: Jan
Węzeł typu Element
<Nazwisko>
Węzeł typu Text
Tekst: Kowalski
Węzeł typu Element
<DataUr>
Węzeł typu Text
Tekst: 1950-02-07
...

```

### 3.5.5.3 XPath

Jeszcze jeden krok dalej idą obiekty z klasy *XPath*, za pomocą których można łatwiej nawigować po strukturze obiektu *XmlDocument*. Obiekt *XPathNavigator* umożliwia wybór konkretnego węzła, zaś *XPathNodeIterator* pozwala na nawigację po sąsiednich węzłach.

Aby uprościć przykład, przesuniemy informacje o osobach z podwęzłów do atrybutów węzłów.

```

<?xml version="1.0" encoding="windows-1250"?>
<ListaOsób nazwa="znajomi">
  <Osoba Imię="Jan" Nazwisko="Kowalski" DataUr="1950-02-07"/>
  <Osoba Imię="Adam" Nazwisko="Nowak" DataUr="1992-12-23"/>
  <Osoba Imię="Tomasz" Nazwisko="Malinowski" DataUr="1979-10-01"/>
</ListaOsób>

```

Dzięki możliwości nawigacji po strukturze można łatwiej odczytywać tylko interesujące obiekty.

```
/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Xml;
using System.Xml.XPath;

namespace Example
{
    public class CMain
    {
        public static void Info00sobie( XPathNavigator node )
        {
            Console.WriteLine( "Osoba" );

            Console.WriteLine( "Imię:\t\t" + node.GetAttribute( "Imię", "" ) );
            Console.WriteLine( "Nazwisko:\t" + node.GetAttribute( "Nazwisko", "" ) );
            Console.WriteLine( "Data ur.:\t" + node.GetAttribute( "DataUr", "" ) );
        }

        public static void Main()
        {
            XmlDocument xml = new XmlDocument();
            xml.Load( "osoby.xml" );

            XPathNavigator n = xml.CreateNavigator();
            XPathNodeIterator i = n.Select( "//ListaOsób/Osoba" );

            while ( i.MoveNext() )
                Info00sobie( i.Current );
        }
    }
}
```

```
C:\Example>example
Osoba
Imię:           Jan
Nazwisko:       Kowalski
Data ur.:       1950-02-07
Osoba
Imię:           Adam
Nazwisko:       Nowak
Data ur.:       1992-12-23
Osoba
Imię:           Tomasz
Nazwisko:       Malinowski
Data ur.:       1979-10-01
```

#### 3.5.5.4 Automatyczne tworzenie kodu do ładowania XML

Pliki XML przechowujące dane mają najczęściej prostą strukturę. Czy nie można wykorzystać tego spostrzeżenia do zautomatyzowania tworzenia kodu parsującego plik XML?

Okazuje się, że jest to możliwe. W skład programów narzędziowych, udostępnianych z **.NET Framework SDK** znajduje się niepozorne narzędzie o nazwie **xsd.exe**. To ono właśnie pozwala znacznie uprościć proces tworzenia kodu dla danych XML.

Punktem wyjścia będą dane. Zaczynamy od zaprojektowania struktury o dowolnej pojemności informacyjnej. Na przykład takiej (nazwijmy te dane **dane.xml**):

```
<?xml version="1.0" encoding="windows-1250"?>
<ListaOsob xmlns='MojeDane'>
  <Osoba obyw_polskie="T">
    <Imie>Jan</Imie>
    <Nazwisko>Kowalski</Nazwisko>
    <DataUr>1950-02-07</DataUr>
```

```

</Osoba>
<Osoba obywatel="T">
  <Imie>Tomasz</Imie>
  <Nazwisko>Malinowski</Nazwisko>
  <DataUrodzenia>1976-10-04</DataUrodzenia>
</Osoba>
<Osoba obywatel="T">
  <Imie>Adam</Imie>
  <Nazwisko>Nowak</Nazwisko>
  <DataUrodzenia>1984-02-17</DataUrodzenia>
</Osoba>
</ListaOsob>

```

Nowym elementem w tej definicji jest atrybut **xmlns** w głównym węźle danych, który wszystkie dane przypisuje do przestrzeni nazw **MojeDane**. Będzie to ważne wtedy, kiedy będziemy weryfikowali poprawność otrzymanych danych.

Następny krok polega na uruchomieniu **xsd.exe** i wskazaniu pliku XML jako parametru. Efektem będzie plik XSD, zawierający w sobie informację o strukturze podanego pliku XML.

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="ListaOsob" targetNamespace="MojeDane"
xmlns:mstns="MojeDane" xmlns="MojeDane" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="ListaOsob" msdata:IsDataSet="true" msdata:Locale="pl-PL">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Osoba">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Imie" type="xs:string" minOccurs="0" msdata:Ordinal="0" />
              <xs:element name="Nazwisko" type="xs:string" minOccurs="0" msdata:Ordinal="1" />
              <xs:element name="DataUrodzenia" type="xs:string" minOccurs="0" msdata:Ordinal="2" />
            </xs:sequence>
            <xs:attribute name="obywatel" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Tak przygotowany opis struktury danych może posłużyć do automatycznego wygenerowania kodu, który będzie potrafił wczytywać pliki XML. Wystarczy ponownie uruchomić **xsd.exe**, tym razem z przełącznikiem **classes** i jako parametr podać plik XSD. Efektem będzie plik zawierający w sobie kod C#.

```

//-----
// <autogenerated>
//   This code was generated by a tool.
//   Runtime Version: 1.0.3705.0
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </autogenerated>
//-----

//
// This source code was auto-generated by xsd, Version=1.0.3705.0.
//
using System.Xml.Serialization;

/// <remarks>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="MojeDane")]
[System.Xml.Serialization.XmlRootAttribute("ListaOsob",
    Namespace="MojeDane", IsNullable=false)]

```

```

public class ListaOsob {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("Osoba")]
    public ListaOsobOsoba[] Items;
}

/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="MojeDane")]
public class ListaOsobOsoba {

    /// <remarks/>
    public string Imie;

    /// <remarks/>
    public string Nazwisko;

    /// <remarks/>
    public string DataUr;

    /// <remarks/>
    [System.Xml.Serialization.XmlAttributeAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public string obyw_polskie;
}

```

Jak widać kod ten zawiera klasy **ListaOsob** i **ListaOsobOsoba**, których składowe odpowiadają relacjom między odpowiednimi węzłami opisanymi w strukturze XML. Wystarczy jedynie dopisać kod do deserializacji obiektu<sup>13</sup>:

```

using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class CMain
{
    public static void Main()
    {
        XmlSerializer xS = new XmlSerializer(typeof(ListaOsob));

        FileStream fs = new FileStream( "dane.xml", FileMode.Open );
        ListaOsob ds = (ListaOsob)xS.Deserialize( fs );

        foreach ( ListaOsobOsoba o in ds.Items )
        {
            Console.WriteLine( String.Format( "{0}, {1}, {2}",
                o.Imie, o.Nazwisko, o.DataUr ) );
        }
    }
}

```

Całość można już skompilować i uruchomić:

```

C:\Example>csc.exe example.cs dane.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

```

```

C:\Example>example
Jan, Kowalski, 1950-02-07
Tomasz, Malinowski, 1976-10-04
Adam, Nowak, 1984-02-17

```

<sup>13</sup>Mechanizm serializacji danych opisano dokładniej w rozdziale 3.5.8

Podsumujmy: dysponując danymi w postaci XML mogliśmy automatycznie wygenerować opis struktury danych w postaci pliku XSD, zaś ten posłużył do automatycznego wygenerowania kodu C#. Po dopisaniu kilku linijek kodu deserializującego obiekt, otrzymaliśmy możliwość łatwego ładowania pliku XML i przenoszenia jego zawartości do programu.

### 3.5.5.5 Weryfikacja poprawności danych XML

Okazuje się, że schemat XSD nadaje się nie tylko do automatycznego generowania kodu odpowiednich klas do przechowywania danych w programie, ale może być wykorzystany do dynamicznej walidacji poprawności danych XML.

Do tego celu wykorzystamy obiekt typu **XmlValidatingReader**. Podczas ładowania dokumentu sprawdza on poprawność danych, a ściśle - zgodność ze specyfikacją zawartą w strukturze XSD. Ewentualne błędy lub ostrzeżenia są zgłaszane za pomocą zdarzenia **ValidationEventHandler**.

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Schema;

namespace Example
{
    public class CMain
    {
        public static void Main()
        {
            XmlDocument xml = new XmlDocument();

            XmlTextReader tr = new XmlTextReader( "dane.xml" );
            XmlValidatingReader reader = new XmlValidatingReader(tr);

            reader.ValidationType = ValidationType.Schema;
            reader.ValidationEventHandler += new ValidationEventHandler (ValidationHandler);

            xml.Load( reader );

            Console.WriteLine( "Wczytano plik." );
        }

        public static void ValidationHandler(object sender, ValidationEventArgs args)
        {
            Console.WriteLine("*Błąd walidacji*");
            Console.WriteLine("\tWaznosc: {0}", args.Severity);
            Console.WriteLine("\tInfo: {0}", args.Message);
        }
    }
}
```

Efekt uruchomienia powyższego programu może być w pierwszej chwili dość zaskakujący: każda linia pliku z danymi powoduje zgłoszenie ostrzeżenia o braku schematu do walidacji. Dzieje się tak dlatego, że nigdzie nie skojarzyliśmy pliku z danymi (dane.xml) z plikiem zawierającym schemat struktury (dane.xsd). Poza tym, że fizycznie mogą znajdować się w jednym folderze, nic tych plików nie łączy.

Aby plik XML był walidowany przy użyciu schematu XSD, należy odpowiednią informację zaszyć we wnętrzu pliku XML.

```
<?xml version="1.0" encoding="windows-1250"?>
<ListaOsob xmlns='MojeDane'
            xmlns:MojWalidator='http://www.w3.org/2001/XMLSchema-instance'
            MojWalidator:schemaLocation='MojeDane dane.xsd'>
    ...
</ListaOsob>
```



Tak osygnowany plik XML będzie już poprawnie walidowany i każde odstępstwo od schematu będzie wyłapywane jako błąd.

Dokładne zapoznanie się z możliwościami schematów XSD umożliwia bardzo szczegółowe zapanowanie nad regułami walidacji dokumentów XML. Jest to bardzo przydatne w sytuacji, kiedy producent danych nie jest znany, nie jest wiarygodny bądź po prostu struktura danych ulega modyfikacji w czasie życia aplikacji.

### 3.5.6 Komunikacja między procesami

Biblioteka *System.Net* udostępnia komplet funkcji wspomagających komunikację za pomocą mechanizmów sieciowych. Programista znajdzie tu nie tylko obiektowe interfejsy dla gniazd, ale również wyspecjalizowane funkcje do bezpośredniej komunikacji z wybranymi usługami sieciowymi.

```
/* Wiktor Zychla, 2003 */
using System;
using System.IO
using System.Net;

namespace SimpleHttpReader
{
    public class CMain
    {
        public static void Main()
        {
            Uri uri = new Uri( "http://www.ii.uni.wroc.pl" );
            WebRequest req = WebRequest.Create( uri );
            WebResponse resp = req.GetResponse();
            Stream stream = resp.GetResponseStream();
            StreamReader sr = new StreamReader( stream );

            string s = sr.ReadToEnd();
            Console.Write( s );
        }
    }
}
```

Aby przekonać się jak łatwo oprogramować gniazda, napiszmy .NETowe odpowiedniki serwera i klienta z rozdziału 2.4.

Kod serwera:

```
/* Wiktor Zychla, 2003 */
// prosty moduł serwera
// użycie: server.exe
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace TcpSever
{
    public class CClientThread
    {
        const string serverMsg = "Tu serwer. Odpowiadam.";
        Socket clientSocket;

        public CClientThread( Socket clientSocket )
        {
            this.clientSocket = clientSocket;
        }

        public void ClientThreadFunc()
        {

```

```

NetworkStream nS = new NetworkStream( clientSocket );
StreamReader sR = new StreamReader( nS );
StreamWriter sW = new StreamWriter( nS );

while( true )
{
    // odbierz wiadomość
    string clientMessage = sR.ReadLine();
    if ( clientMessage != null )
        Console.WriteLine( "Otrzymano wiadomość: {0}", clientMessage );
    else
        break;

    // odeślij odpowiedź
    sW.WriteLine( serverMsg );
    sW.Flush();
}

sR.Close();
sW.Close();
}
}

public class CServer
{
    public const int DEFAULT_PORT = 5000;

    public static void Main()
    {
        TcpListener tcpListener = new TcpListener( DEFAULT_PORT );
        tcpListener.Start();

        Console.WriteLine( "Serwer nasłuchuje." );
        Console.WriteLine( "Adres: {0}, port: {1}", Dns.GetHostName(), DEFAULT_PORT );

        while ( true )
        {
            Thread.Sleep(1);
            if ( tcpListener.Pending() )
            {
                Socket socketForClient = tcpListener.AcceptSocket();
                string clientName =
                    String.Format( "{0} [{1}]",
                        Dns.GetHostByAddress( ((EndPoint)socketForClient.RemoteEndPoint).Address ).HostName,
                        ((EndPoint)socketForClient.RemoteEndPoint).Address.ToString()
                    );
                Console.WriteLine( "Zaakceptowano połączenie: serwer {0}", clientName );

                CClientThread ct = new CClientThread( socketForClient );
                Thread t = new Thread( new ThreadStart( ct.ClientThreadFunc ) );
                t.Start();
            }
        }
    }
}
}

```

Kod klienta:

```

/* Wiktor Zychla, 2003 */
// prosty moduł klienta
// użycie: klient.exe -s:IP
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace TcpClientNS
{

```

```

public class CClient
{
    const int DEFAULT_COUNT=5;
    const int DEFAULT_PORT =5000;
    const string DEFAULT_MESSAGE="Tu klient. Witam.";

    static string szServer;

    static void sposob_uzycia()
    {
        Console.WriteLine( "client.exe -s:IP" );
        Environment.Exit(1);
    }

    static void WalidacjaLiniiPolecen( string[] args )
    {
        int i;

        if ( args.Length < 1 )
        {
            sposob_uzycia();
        }

        for ( i=0; i<args.Length; i++ )
        {
            if ( args[i][0] == '-' )
            {
                switch( args[i][1].ToString().ToLower() )
                {
                    case "s" : if ( args[i].Length > 3 )
                        {
                            szServer = args[i].Substring(3);
                        };
                        break;
                    default : sposob_uzycia(); break;
                }
            }
        }
    }

    public static void Main( string[] args )
    {
        WalidacjaLiniiPolecen( args );

        TcpClient socketForServer = new TcpClient( szServer, DEFAULT_PORT );

        NetworkStream nS = socketForServer.GetStream();
        StreamWriter sW = new StreamWriter( nS );
        StreamReader sR = new StreamReader( nS );

        for ( int i=0; i<DEFAULT_COUNT; i++ )
        {
            sW.WriteLine( DEFAULT_MESSAGE );
            sW.Flush();
            string message = sR.ReadLine();
            if ( message != null )
                Console.WriteLine( "Serwer odpowiada: {0}", message );
        }

        sR.Close();
        sW.Close();
    }
}

```

Jak widać, opakowanie gniazd w obiektowe klasy *TcpListener* i *TcpClient* (przeznaczone do komunikacji przez TCP/IP), znacznie upraszcza kod odpowiedzialny za wysyłanie i odbieranie danych.

### 3.5.7 Wyrażenia regularne

Biblioteki do obsługi wyrażeń regularnych są standardowo dołączane do współczesnych języków programowania. Szczycą się nimi zwłaszcza języki skryptowe, ale jak zobaczymy w kolejnych przykładach, wszystko zależy od dobrej biblioteki.

#### 3.5.7.1 Czym są *wyrażenia regularne*

Pojęcie "*wyrażenia regularne*" związane jest z przetwarzaniem tekstu. Wyrażenia regularne tworzą pewien szczególny język, niezależny od żadnego języka programowania, za pomocą którego definiujemy wzorce, wykorzystywane następnie do wyszukiwania, usuwania, czy zamieniania fragmentów tekstu. Wyrażenia regularne znakomicie upraszczają proces parsowania na przykład stron HTML, plików XML, logów systemowych itd.

#### 3.5.7.2 Język wyrażeń regularnych

Poniższa tabela podsumowuje wybrane zasady budowania wyrażeń regularnych.

Wyrażenie	Opis
.	Każdy znak za wyjątkiem <code>\n</code>
[znaki]	Pojedyncze znaki z listy
[znakA-znakB]	Znaki z podanego zakresu
<code>\ w</code>	Znak tworzący słowo, równoważnie <code>[a-zA-Z_0-9]</code>
<code>\ W</code>	Znak nie tworzący słowa
<code>\ s</code>	Znak biały, równoważnie <code>[\n\r\t\f]</code>
<code>\ S</code>	Znak inny niż biały
<code>\ d</code>	Cyfra, równoważnie <code>[0-9]</code>
<code>\ D</code>	Nie-cyfra
<code>\ b</code>	Na granicy słowa
<code>\ B</code>	Nie na granicy słowa
*	Zero lub więcej
+	Jeden lub więcej
?	Zero lub jeden
{ <i>n</i> }	Dokładnie <i>n</i> -razy
{ <i>n</i> , }	Co najmniej <i>n</i> -razy
{ <i>n</i> , <i>m</i> }	Co najmniej <i>n</i> ale nie więcej niż <i>m</i>
( )	Podwyrażenie
(? <i>&lt;nazwa&gt;</i> )	Podwyrażenie jako nazwa
	Alternatywa

Tak naprawdę zaprojektowanie odpowiedniego często nie jest łatwe i wymaga po prostu trochę wprawy.

#### 3.5.7.3 Dzielenie tekstu

Pierwszym przykładem zastosowania wyrażeń regularnych jest dzielenie tekstu. Tekst jest dzielony w miejscach, które dopasowują się do zadanego wyrażenia regularnego.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;
```

```
using System.Text.RegularExpressions;

class CExample
{
    public static void Main()
    {
        string sSplit = "Ala ma kota a kot ma Ale";
        Regex r = new Regex( "[ ]+" );

        foreach( string s in r.Split( sSplit ) )
            Console.Write( s+", " );
    }
}

C:\example>example
Ala,ma,kota,a,kot,ma,Ale,
```

### 3.5.7.4 Wyszukiwanie wzorca

Wyszukiwanie zadanego wyrażeniem regularnym wzorca w zadanym tekście możliwe jest dzięki obiektom **Match** i **MatchCollection**.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;
using System.Text.RegularExpressions;

class CExample
{
    public static void Main()
    {
        string sFind = "Dobrze jest dojsc do domu radosnie i wydobrzec do rana";
        Regex r = new Regex( "(do)|(a)" );

        for ( Match m = r.Match( sFind ); m.Success; m = m.NextMatch() )
            Console.Write( "'{0}' na pozycji {1}\n", m.Value, m.Index );
    }
}

C:\example>example
'do' na pozycji 12
'do' na pozycji 18
'do' na pozycji 21
'a' na pozycji 27
'do' na pozycji 28
'do' na pozycji 39
'do' na pozycji 47
'a' na pozycji 51
'a' na pozycji 53
```

### 3.5.7.5 Edycja, usuwanie tekstu

Dzięki metodzie **Replace** wyrażień regularnych można użyć do zastępowania tekstu.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Text;
using System.Text.RegularExpressions;

class CExample
{
    public static void Main()
    {
        string sFind = "Dobrze jest dojsc do domu radosnie i wydobrzec do rana";
        Regex r = new Regex( "(do)|(a)" );

        Console.Write( r.Replace( sFind, "" ) );
    }
}
```

```
    }
}
```

```
C:\example>example
Dobrze jest jsc mu rsnie i wybrzec rn
```

### 3.5.8 Serializacja

O *serializacji* mówimy wtedy, gdy instancja obiektu jest składowana na nośniku zewnętrznym. Mechanizm ten wykorzystywany jest również do transferu zawartości obiektów między odległymi środowiskami. Oczywiście łatwo wyobrazić sobie mechanizm zapisu zawartości obiektu przygotowany przez programistę, ale serializacja jest mechanizmem niezależnym od postaci obiektu i od tego, czy programista przewidział możliwość zapisu zawartości obiektu czy nie.

#### 3.5.8.1 Serializacja binarna

Aby zawartość obiektu mogła być składowana w postaci binarnej, klasa musi spełniać kilka warunków:

- Musi być oznakowana atrybutem *Serializable*
- Musi implementować interfejs *ISerializable*
- Musi mieć specjalny konstruktor do deserializacji

```
/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

namespace NExample
{
    [Serializable()]
    public class CObiekt : ISerializable
    {
        int    v;
        DateTime d;
        string  s;

        public CObiekt( int v, DateTime d, string s )
        {
            this.v = v; this.d = d; this.s = s;
        }

        // konstruktor do deserializacji
        public CObiekt(SerializationInfo info, StreamingContext context)
        {
            v = (int)info.GetValue("v", typeof(int));
            d = (DateTime)info.GetValue("d", typeof(DateTime));
            s = (string)info.GetValue("s", typeof(string));
        }

        // serializacja
        public void GetObjectData(SerializationInfo info,
                                   StreamingContext context)
        {
            info.AddValue("v", v);
            info.AddValue("d", d);
            info.AddValue("s", s);
        }

        public override string ToString()
    }
}
```

```

    {
        return String.Format( "{0}, {1:d}, {2}", v, d, s );
    }
}

public class CMain
{
    static void SerializujBinarnie()
    {
        Console.WriteLine( "Serializacja binarna" );

        CObiekt o = new CObiekt( 5, DateTime.Now, "Ala ma kota" );
        Console.WriteLine( o );

        // serializuj
        Stream s = File.Create( "binary.dat" );
        BinaryFormatter b = new BinaryFormatter();
        b.Serialize( s, o );
        s.Close();

        // deserializuj
        Stream t = File.Open( "binary.dat", FileMode.Open );
        BinaryFormatter c = new BinaryFormatter();
        CObiekt p = (CObiekt)c.Deserialize( t );
        t.Close();

        Console.WriteLine( "Po deserializacji: " + p.ToString() );
    }

    public static void Main()
    {
        SerializujBinarnie();
    }
}

c:\Example>example.exe
Serializacja binarna
5, 2003-04-24, Ala ma kota
Po deserializacji: 5, 2003-04-24, Ala ma kota

```

### 3.5.8.2 Serializacja SOAP

Serializacja binarna ma jak widać wady (wymaga specjalnie przygotowanej klasy), ma również zalety (jest szybka, plik wynikowy zajmuje niewiele miejsca).

Alternatywne podejście możliwe jest dzięki mechanizmom SOAP (*Simple Object Access Protocol*). SOAP jest protokołem do wymiany danych, opartym o nośnik XML, niezależny od systemu operacyjnego. Serializacja SOAP jest wolniejsza niż serializacja binarna, wynik zajmuje więcej miejsca (w końcu to plik XML), jednak w ten sposób można serializować dowolne obiekty.

```

/* Wiktor Zychla, 2003 */
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

namespace NExample
{
    [Serializable()]
    public class CObiekt
    {
        int v;
        DateTime d;
        string s;

        public CObiekt( int v, DateTime d, string s )
    }
}

```

```

    {
        this.v = v; this.d = d; this.s = s;
    }

    public override string ToString()
    {
        return String.Format( "{0}, {1:d}, {2}", v, d, s );
    }
}

public class CMain
{
    static void SerializujSOAP()
    {
        Console.WriteLine( "Serializacja SOAP" );

        CObiekt o = new CObiekt( 5, DateTime.Now, "Ala ma kota" );
        Console.WriteLine( o );

        // serializuj
        Stream s = File.Create( "binary.soap" );
        SoapFormatter b = new SoapFormatter();
        b.Serialize( s, o );
        s.Close();

        // deserializuj
        Stream t = File.Open( "binary.soap", FileMode.Open );
        SoapFormatter c = new SoapFormatter();
        CObiekt p = (CObiekt)c.Deserialize( t );
        t.Close();

        Console.WriteLine( "Po deserializacji: " + p.ToString() );
    }

    public static void Main()
    {
        SerializujSOAP();
    }
}

c:\Example>example.exe
Serializacja binarna
5, 2003-04-24, Ala ma kota
Po deserializacji: 5, 2003-04-24, Ala ma kota

```

### 3.5.9 Wołanie kodu niezarządzanego

Współpraca z już istniejącymi bibliotekami jest bardzo ważnym elementem platformy .NET. Programista może nie tylko wołać funkcje z natywnych bibliotek, ale również korzystać z bibliotek obiektowych COM.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Runtime.InteropServices;

namespace NExample
{
    public class CMain
    {
        [DllImport("user32.dll", EntryPoint="MessageBox")]
        public static extern int MsgBox(int hWnd, String text,
                                         String caption, uint type);

        public static void Main()
        {
            MsgBox( 0, "Witam", "", 0 );
        }
    }
}

```



```

    }
}
}

```

Wygląda to dość prosto, jednak w rzeczywistości wymaga starannego przekazania parametrów do funkcji napisanej najczęściej w C, a następnie odebrania wyników. Każdy typ w świecie .NET ma domyślnie swojego odpowiednika w kodzie niezarządzanym, który będzie używany w komunikacji między oboma światami. Na przykład domyślny sposób przekazywana zmiennej zadeklarowanej jako *string* to LPSTR (wskaźnik na tablicę znaków). Programista może dość szczegółowo zapanować nad domyślnymi konwencjami dzięki atrybutowi *MarshalAs*.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Runtime.InteropServices;

namespace NExample
{
    public class CMain
    {
        [DllImport("user32.dll", EntryPoint="MessageBox")]
        public static extern int MsgBox(int hWnd,
                                       [MarshalAs(UnmanagedType.LPStr)]
                                       String text,
                                       String caption, uint type);

        public static void Main()
        {
            MsgBox( 0, "Witam", "", 0 );
        }
    }
}

```

Aby ustalić w ten sposób typ wartości zwracanej z funkcji należałoby napisać:

```

...
[DllImport("user32.dll", EntryPoint="MessageBox")]
[return: MarshalAs(UnmanagedType.I4)]
public static extern int MsgBox(int hWnd, ...
...

```

Możliwość tak dokładnego wpływania na postać parametrów jest szczególnie przydatna w typowym przypadku przekazywania jakiejś struktury do jakiejś funkcji, na przykład z Win32API.

Przykładowa struktura z Win32API

```

typedef struct tagLOGFONT
{
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;

```

powinna być przetłumaczona tak, aby zachować kolejność ułożenia pól oraz ograniczoną długość napisu.

```
[StructLayout(LayoutKind.Sequential)]
public class LOGFONT
{
    public const int LF_FACESIZE = 32;
    public int lfHeight;
    public int lfWidth;
    public int lfEscapement;
    public int lfOrientation;
    public int lfWeight;
    public byte lfItalic;
    public byte lfUnderline;
    public byte lfStrikeOut;
    public byte lfCharSet;
    public byte lfOutPrecision;
    public byte lfClipPrecision;
    public byte lfQuality;
    public byte lfPitchAndFamily;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=LF_FACESIZE)]
    public string lfFaceName;
}
```

Czasami nawet konieczne jest dokładne wyznaczenie położenia wszystkich pól struktury.

```
[StructLayout(LayoutKind.Explicit, Size=16, CharSet=CharSet.Ansi)]
public class MySystemTime
{
    [FieldOffset(0)]public ushort wYear;
    [FieldOffset(2)]public ushort wMonth;
    [FieldOffset(4)]public ushort wDayOfWeek;
    [FieldOffset(6)]public ushort wDay;
    [FieldOffset(8)]public ushort wHour;
    [FieldOffset(10)]public ushort wMinute;
    [FieldOffset(12)]public ushort wSecond;
    [FieldOffset(14)]public ushort wMilliseconds;
}
```

### 3.5.9.1 Funkcje zwrotne

Funkcje Win32Api, które zwracają więcej niż jeden element, najczęściej korzystają z mechanizmu funkcji zwrotnych. Programista przekazuje wskaźnik na funkcję zwrotną, która jest wywoływana dla każdego elementu na liście wyników (tak działa na przykład **EnumWindows**, czy **EnumDesktops**).

```
BOOL EnumDesktops (
    HWINSTA hinsta,
    DESKTOPENUMPROC lpEnumFunc,
    LPARAM lParam
)
```

Parametr typu HWINSTA można przekazać jako IntPtr, zaś LPARAM jako int. Wskaźnik na funkcję

```
BOOL CALLBACK EnumDesktopProc (
    LPTSTR lpszDesktop, LPARAM lParam
)
```

należy zamienić na delegata

```
delegate bool EnumDesktopProc(
    [MarshalAs(UnmanagedType.LPTStr)]
    string desktopName, int lParam
)
```

Definicja funkcji **EnumDesktops** będzie więc wyglądać tak:

```
[DllImport("user32.dll"), CharSet = CharSet.Auto]
static extern bool EnumDesktops (
    IntPtr windowStation,
    EnumDesktopProc callback,
    int lParam
)
```

### 3.5.10 Odśmiecacz

Mechanizm odśmiecania funkcjonuje samodzielnie, bez kontroli programisty. W szczególnych sytuacjach odśmiecanie może być wymuszone przez wywołanie metody obiektu odśmiecacza:

```
GC.Collect();
```

Należy pamiętać o tym, że destruktory obiektów są wykonywane w osobnym wątku, dlatego zakończenie metody **Collect** nie oznacza, że wszystkie destruktory są już zakończone. Można oczywiście wymusić oczekiwanie na zakończenie się wszystkich czekających destruktorków:

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

Działanie odśmiecacza jest dość proste. W momencie, w którym aplikacji brakuje pamięci, odśmiecacz rozpoczyna przeglądanie wszystkich referencji od zmiennych statycznych, globalnych i lokalnych, oznaczając kolejne obiekty jako używane. Wszystkie obiekty, które nie zostaną oznaczone, mogą zostać usunięte, bowiem żaden aktualnie aktywny obiekt z nich nie korzysta.

Taki sposób postępowania, mimo że poprawny, byłby dość powolny. Dlatego w rzeczywistości wykorzystuje się dodatkowo pojęcie tzw. *generacji*. Chodzi o to, że obiekt tuż po wykreowaniu należy do zerowej generacji obiektów, czyli obiektów "najmłodszych". Po "przeżyciu" odśmiecania, obiektom inkrementuje się numery generacji. Kiedy odśmiecacz zabiera się za przeglądanie obiektów, zaczyna od obiektów najmłodszych, dopiero jeśli okaże się, że pamięci nadal jest zbyt mało, usuwa obiekty coraz starsze.

Idea ta ma proste uzasadnienie - obiektami najmłodszymi najczęściej będą na przykład zmienne lokalne funkcji czy bloków kodu. Te zmienne powinny być usuwane najszybciej. Zmienne statyczne, kilkakrotnie wykorzystane w czasie działania programu, będą usuwane najpóźniej.

```
using System;

public class CObiekt
{
    private string name;
    public CObiekt(string name) { this.name = name; }
    override public string ToString() { return name; }
}

namespace Example
{
    public class CMainForm
    {
        const int IL = 3;

        public static void Main()
        {
            Console.WriteLine( "Maksymalna generacja odsmiecacza " + GC.MaxGeneration );

            CObiekt[] t = new CObiekt[IL];

            Console.WriteLine( "Tworzenie obiektow." );
            for ( int i=0; i<IL; i++ )
            {
                t[i] = new CObiekt( "obiekt " + i );
                Console.WriteLine( "{0}, generacja {1}", t[i], GC.GetGeneration( t[i] ) );
            }
        }
    }
}
```

```

    }

    // spróbuj usuwać nieużywane obiekty
    GC.Collect();
    GC.WaitForPendingFinalizers();

    Console.WriteLine( "Usuwanie obiektów." );
    for ( int i=0; i<IL; i++ )
    {
        Console.WriteLine( "{0}, generacja {1}", t[i], GC.GetGeneration( t[i] ) );
        t[i] = null;

        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
}
}

C:\Example>example
Maksymalna generacja odsmiecacza 2
Tworzenie obiektów.
obiekt 0, generacja 0
obiekt 1, generacja 0
obiekt 2, generacja 0
Usuwanie obiektów.
obiekt 0, generacja 1
obiekt 1, generacja 2
obiekt 2, generacja 2

```

### 3.6 Aplikacje okienkowe - System.Windows.Forms

Jak widzieliśmy w poprzednich rozdziałach, programowanie Windows nie polega wyłącznie na tworzeniu okien, jednak z pewnością to właśnie temu zagadnieniu programista poświęca zwykle sporo czasu. Od dobrej biblioteki wspierającej tworzenie okien i własnych komponentów należałoby oczekiwać prostoty i spójności. Z perspektywy historycznej można powiedzieć, że Win32Api jest interfejsem spójnym jednak dość żmudnym. W kolejnych latach powstawały więc kolejne wyspecjalizowane biblioteki, wspierające tworzenie aplikacji okienkowych. Dużą popularność zdobył sobie również Visual Basic, w którym projektowanie interfejsu użytkownika było wyjątkowo proste, jednak interfejs programowania był bardzo niespójny - często podobne czynności w różnych kontekstach realizowane były za pomocą zupełnie różnych mechanizmów<sup>14</sup>.

Biblioteka **System.Windows.Forms**, która umożliwia tworzenie aplikacji okienkowych w świecie .NET jest zarówno prosta jak i spójna. Nie sprawi kłopotu ani nowicjuszowi, który chciałby nauczyć się tworzyć okna jak najszybciej, ani profesjonalście, który znając ułomności innych interfejsów szybko nauczy się korzystać z zaawansowanych mechanizmów biblioteki *System.Windows.Forms*. Tworzenie interfejsu użytkownika jest równie proste jak w "starym" Visual Basicu, zaś kontrola, jaką programista ma nad oprogramowywanym interfejsem, dorównuje tej, jaką daje Win32Api.

**System.Windows.Forms**, jako interfejs w pełni obiektowy, najbardziej przypomina biblioteki okienkowe Javy. Dla programisty najważniejsze jest to, że cały opis komponentu (okna, kontrolki) jest częścią kodu, dzięki czemu kod nie jest w żaden sposób związany z jakimś środowiskiem developerskim. Przy odrobinie wprawy można z powodzeniem pisać programy okienkowe używając dowolnego edytora tekstu, nawet zwykłego Notatnika.

Interfejs obiekty oznacza również, że funkcjonalność każdego komponentu można bardzo łatwo rozszerzyć tworząc w razie potrzeby klasę potomną dziedziczącą z niego. Programista

<sup>14</sup>Visual Basic z czasów przed VB.NET jest również dość słaby jako język programowania, ponieważ ma ubogi i nieprzemysłany model obiektowy.

może również w łatwy sposób tworzyć własne komponenty wizualne (kontrolki), do których może zaprojektować własne zdarzenia.

### 3.6.1 Tworzenie okien

Przypomnijmy sobie najprostszy program ze strony 18, który tworzył zwykłe, proste okno na pulpicie. Jego odpowiednik w świecie .NET wygląda tak:

```
/* Wiktor Zychla, 2003 */
using System;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm()
        {
            this.Text = "Okna w świecie .NET";
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

Różnica w przejrzystości programu jest kolosalna! Interfejs biblioteki **System.Windows.Forms** jest w pełni obiektowy. Utworzenie okna polega po prostu na utworzeniu klasy dziedziczącej z klasy **Form**. Klasa ta zamyka w sobie całą funkcjonalność jakiej potrzeba aby obsłużyć proste okno: obiekt który utworzyliśmy w przykładowym programie powyżej ma kilkadziesiąt właściwości i metod oraz obsługuje kilkadziesiąt zdarzeń.

Powyższy kod może w pierwszej chwili wydawać się dość zaskakujący, bowiem nie ma tu nigdzie pętli obsługi komunikatów. Okazuje się, że pętla obsługi komunikatów jest ukryta w funkcji **Run** klasy **Application**. Dodatkowym, opcjonalnym parametrem metody **Run** jest obiekt, będący *głównym oknem aplikacji*. Aplikacja automatycznie zakończy się, kiedy główne okno aplikacji zostanie zniszczone.

Oczywiście taka konstrukcja utrudnia nieco sterowanie aplikacją wtedy, gdy powinna ona zajmować się czymś oprócz przetwarzania komunikatów. Na stronie 25 widzieliśmy jak radzić sobie z takim problemem w Win32Api (zamiast **GetMessage** użyliśmy **PeekMessage**), zaś na stronie ?? pokazano jak wygląda analogiczna konstrukcja w świecie .NET.

Widać również, że programista w przeciwieństwie do Win32API nie musi samodzielnie rejestrować klasy okna w systemie. Właściwości klasy okna opisuje definicja klasy, zaś sama operacja rejestrowania klasy okna w systemie odbywa się bez udziału programisty<sup>15</sup>.

### 3.6.2 Okna potomne

W obiektowym świecie **System.Windows.Forms**, każdy obiekt dziedziczący z klasy **Control** (klasa **Form** dziedziczy z klasy **Control** i jest od niej odległa o 4 pokolenia) ma właściwość **Controls**, która zwraca kolekcję okien potomnych względem tego obiektu. Oznacza to, że okna

<sup>15</sup>W świecie .NET definicja okna jest klasą. Aby takie okno mogło pojawić się w systemie, w systemie rejestrowana jest oczywiście klasa okna. Nie należy jednak mylić tych dwóch pojęć i dlatego wprowadzimy dwa różne określenia: *klasą okna* będziemy nazywać obiekt systemowy, opisujący właściwości okna i rejestrowany w systemie za pomocą funkcji **RegisterClass**, zaś *klasą opisującą okno*, będziemy nazywać definicję klasy dziedziczącej z klasy **Form**, opisującą właściwości okna w C#.



Rysunek 3.6: Proste okno w świecie .NET

potomne mogą być łatwo tworzone w czasie działania programu. Te okna potomne, które powinny być widoczne od razu po utworzeniu okna można utworzyć po prostu w konstruktorze okna macierzystego.

Najwygodniej jest uczynić okna potomne polami składowymi klasy opisującej okno macierzyste. Wtedy wszystkie inne składowe klasy okna macierzystego mają dostęp do okien potomnych. Okna potomne są również obiektami, podlegają więc dokładnie takim samym prawom jak wszystkie obiekty - muszą być jawnie skonstruowane, są odśmiecanie itd.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Button btOK;
        TextBox pTxt;

        public CMainForm()
        {
            btOK = new Button();
            btOK.Location = new Point( 25, 20 );
            btOK.Size = new Size( 150, 25 );
            btOK.Text = "Naciśnij mnie";

            pTxt = new TextBox();
            pTxt.Location = new Point( 25, 60 );
            pTxt.Size = new Size( 150, 40 );
            pTxt.Multiline = true;
            pTxt.Text = "Pole tekstowe";

            this.Controls.AddRange( new Control[] { btOK, pTxt } );

            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

Tworząc okna można zejść aż na poziom równy funkcji **CreateWindow**, na którym można

utworzyć okno podając nazwę klasy okna, jego styl i rozmiary.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        const int WS_VISIBLE = 0x10000000;
        const int WS_CHILD   = 0x40000000;

        public CMainForm()
        {
            CreateParams cp = new CreateParams();
            cp.ClassName   = "EDIT";
            cp.Style        = WS_CHILD | WS_VISIBLE;
            cp.Parent       = this.Handle;
            cp.Width        = 150;
            cp.Height       = 25;
            cp.X            = 20;
            cp.Y            = 20;

            NativeWindow t = new NativeWindow();
            t.CreateHandle( cp );

            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

### 3.6.3 Zdarzenia

W rozdziale 3.2.13 na stronie 108 widzieliśmy w jaki sposób C# rozwiązuje problem zdarzeń. Dzięki temu, że zdarzenie jest tak naprawdę listą odpowiednich delegatów, zajście zdarzenia może śledzić dowolna ilość słuchaczy.

Taki model doskonale sprawdza się w aplikacjach okienkowych, gdzie tak naprawdę istotne są właśnie reakcje na zdarzenia zgłaszane do okien aplikacji. Aby okna reagowały na działania użytkownika, wystarczy więc pod odpowiednie zdarzenia ”przypiąć” ich słuchaczy. Zdarzenia udostępniane przez komponenty wizualne dość dobrze odpowiadają komunikatom, jakie komponenty te mogłyby obsługiwać.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Button  btOK;
        TextBox pTxt;

        public CMainForm()
        {
            btOK          = new Button();
            btOK.Location  = new Point( 25, 20 );
        }
    }
}
```

```

        btOK.Size      = new Size( 150, 25 );
        btOK.Text      = "Naciśnij mnie";

        pTxt           = new TextBox();
        pTxt.Location   = new Point( 25, 60 );
        pTxt.Size       = new Size( 150, 40 );
        pTxt.Multiline  = true;
        pTxt.Text       = "Pole tekstowe";

        // dodaj zdarzenia
        btOK.Click      += new EventHandler( btOk_Click );
        pTxt.KeyPress   += new KeyPressEventHandler( pTxt_KeyPress );

        this.Controls.AddRange( new Control[] { btOK, pTxt } );

        this.Text = "Okno";
        this.Size = new Size( 200, 200 );
    }

    void btOk_Click( object sender, EventArgs e )
    {
        MessageBox.Show( "Kliknięto przycisk" );
    }

    void pTxt_KeyPress( object sender, KeyPressEventArgs e )
    {
        this.Text = e.KeyChar.ToString();
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

Wśród ważniejszych zdarzeń warto wymienić:

- Click
- DoubleClick
- Enter
- KeyDown
- KeyPress
- KeyUp
- Leave
- MouseDown
- MouseHover
- MouseUp
- Move
- Paint
- Resize
- Validating
- Validated



### 3.6.3.1 Parametry zdarzeń

Przy tak dużej ilości zdarzeń pojawiają się różne problemy. Na przykład - różne zdarzenia mogą mieć różne ilości parametrów. Informacja o naciśnięciu klawisza powinna nieść ze sobą informację o tym klawiszu, zaś informacja o naciśnięciu przycisku myszy powinna mówić który przycisk został naciśnięty i jaka jest pozycja wskaźnika w oknie. Ponadto, gdyby jedna i ta sama funkcja została przypisana do obsługi różnych zdarzeń różnych komponentów, to wewnątrz funkcji obsługującej to zdarzenie zdecydowanie powinno dać się określić ten komponent, który spowodował powstanie zdarzenia.

Na szczęście te problemy rozwiązano dość elegancko. Przyjęto konwencję, wedle której obiekt będący źródłem zdarzenia przekazuje się zawsze jako pierwszy parametr do delegata reagującego na zajście zdarzenia, zaś parametry zdarzenia przekazuje się w obiektach klas dziedziczących z klasy **EventArgs** jako drugi parametr tych delegatów. Na przykład zdarzenie naciśnięcia klawisza przekazuje swoje parametry w obiekcie typu **KeyPressEventArgs**, zaś zdarzenia myszy w obiektach typu **MouseEventArgs**.

Oznacza to, że wszyscy delegaci będący słuchaczami zdarzeń związanych z obsługą komponentów wizualnych mają bardzo podobną postać. Nazwy tych delegatów i nazwy ich parametrów odpowiadają nazwom odpowiednich zdarzeń.

```
delegate void EventHandler( object sender, EventArgs e );
delegate void KeyPressEventHandler( object sender, KeyPressEventArgs e );
...
```

### 3.6.3.2 Pokrywanie funkcji obsługi zdarzeń

Oprócz możliwości przypinania słuchaczy do odpowiednich zdarzeń, istnieje możliwość przeciążenia funkcji wirtualnych dziedziczonych z klasy **Control**, będących reakcjami na zdarzenia. W tym przypadku funkcja ma już tylko jeden parametr, określający parametry zdarzenia.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMyButton : Button
    {
        protected override void OnClick( EventArgs e )
        {
            base.OnClick( e );
            MessageBox.Show( "Kliknięto mnie!" );
        }
    }

    public class CMainForm : Form
    {
        public CMainForm()
        {
            CMyButton btOK;

            btOK          = new CMyButton();
            btOK.Location  = new Point( 25, 20 );
            btOK.Size      = new Size( 150, 25 );
            btOK.Text      = "Naciśnij mnie";

            this.Controls.Add( btOK );

            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }
    }
}
```

```

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

Nic nie stoi na przeszkodzie, aby równolegle dodać funkcje reagujące na to samo zdarzenie do odpowiedniej listy delegatów (przy okazji zwróćmy uwagę na to w jakiej kolejności wywołają się oba zdarzenia. Od czego to zależy?):

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMyButton : Button
    {
        protected override void OnClick( EventArgs e )
        {
            base.OnClick( e );
            MessageBox.Show( "Kliknięto mnie!" );
        }
    }

    public class CMainForm : Form
    {
        public CMainForm()
        {
            CMyButton btOK;

            btOK          = new CMyButton();
            btOK.Location  = new Point( 25, 20 );
            btOK.Size      = new Size( 150, 25 );
            btOK.Text       = "Naciśnij mnie";

            btOK.Click += new EventHandler( btOK_Click );

            this.Controls.Add( btOK );

            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }

        public void btOK_Click( object sender, EventArgs e )
        {
            MessageBox.Show( "I znów mnie kliknięto!" );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

Powstaje więc pytanie: gdzie w takim razie określać reakcje na zdarzenia, czy przeciążając odpowiednią funkcję czy dokładając delegata do listy słuchaczy zdarzenia?

Odpowiedź wbrew pozorom jest dość prosta: przeciążanie funkcji obsługi zdarzenia powinno stosować się tylko tam, gdzie reakcja na zdarzenie powinna być taka sama dla *wszystkich* instancji tworzonego komponentu i w dodatku powinna być jego trwałą właściwością. Takiej funkcji nie można już bowiem odwołać.

Jeśli zaś reakcja na zdarzenie ma być wewnętrzną sprawą jakiejś konkretnej *instancji* komponentu, tam reakcja ta powinna być delegatem na liście słuchaczy zdarzenia.

### 3.6.4 Okna dialogowe

Skoro okna w świecie .NET są instancjami odpowiednich klas, to tworzenie nowych okien jest tak proste jak wykreowanie nowych obiektów. Po wykreowaniu okno może być pokazane jako modalne za pomocą metody **ShowDialog** lub jako niemodalne za pomocą **Show**.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CSecondaryForm : Form
    {
        public CSecondaryForm()
        {
            this.Text = "Okno dialogowe";
        }
    }

    public class CMainForm : Form
    {
        public CMainForm()
        {
            Button btOK;

            btOK = new Button();
            btOK.Location = new Point( 25, 20 );
            btOK.Size = new Size( 150, 25 );
            btOK.Text = "Pokaż okno dialogowe";

            btOK.Click += new EventHandler( btOK_Click );

            this.Controls.Add( btOK );

            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }

        public void btOK_Click( object sender, EventArgs e )
        {
            CSecondaryForm f = new CSecondaryForm();
            f.ShowDialog();
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

Klasa opisująca okno może mieć dowolną ilość konstruktorów, których można użyć do przekazania parametrów nowo tworzonemu oknom.

### 3.6.5 Subclassowanie okien

Często przechwytywanie zdarzeń nie wystarcza, a programista chciałby sięgnąć głębiej, aż na poziom komunikatów.

#### 3.6.5.1 Obsługa komunikatów własnych okien

Obsługa komunikatów nie następuje żadnych kłopotów jeśli to programista tworzy klasę opisującą okno. Wystarczy po prostu przeciążyć metodę **WndProc**.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        const int WM_LBUTTONDOWNCLK = 0x0203;

        protected override void WndProc( ref Message m )
        {
            switch ( m.Msg )
            {
                case WM_LBUTTONDOWNCLK : MessageBox.Show( "Dwuklik!" ); break;
            }
            base.WndProc( ref m );
        }

        public CMainForm()
        {
            this.Text = "Okno";
            this.Size = new Size( 200, 200 );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

Struktura **Message** przechowuje w sobie wszystkie parametry komunikatu (**LParam**, **WParam**, itd.), ma również metodę **GetLParam**, która służy do rzutowania parametru przekazanego w **LParam** na wskazany typ.

Warto zwrócić uwagę na konieczność wywołania funkcji **WndProc** z klasy bazowej. Bez tego wywołania okno nie utworzy się, bowiem zabraknie mu większości potrzebnej funkcjonalności.

### 3.6.5.2 Obsługa komunikatów istniejących okien

Przedstawiona w poprzednim podrozdziale technika nie nadaje się do obsługi komunikatów w już istniejących komponentach, na przykład **TextBox** czy **Button**. Można by co prawda przeciążyć już istniejący komponent i dodać obsługę komunikatów do klasy potomnej, ale interesująca byłaby możliwość taka jak opisana na stronie 31, czyli dodawanie własnej funkcji obsługi do już istniejącego okna.

Okazuje się, że i taki scenariusz jest możliwy, wymaga jedynie utworzenia klasy dziedziczącej z klasy **NativeWindow** i skojarzenia uchwytów okien.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CSubclass : NativeWindow
    {
        const int WM_LBUTTONDOWNCLK = 0x0203;

        protected override void WndProc( ref Message m )
        {
            switch ( m.Msg )
            {

```

```

        case WM_LBUTTONDOWN : MessageBox.Show( "Dwuklik!" ); break;
    }
    base.WndProc( ref m );
}

public CSubclass() {}

public class CMainForm : Form
{
    CSubclass subclass = new CSubclass();
    TextBox t;

    public CMainForm()
    {
        t = new TextBox();
        this.Controls.Add( t );

        // subclassing okna potomnego
        subclass.AssignHandle( t.Handle );

        this.Text = "Okno";
        this.Size = new Size( 200, 200 );
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

### 3.6.6 Komponenty wizualne

Biblioteka **System.Windows.Forms** udostępnia szereg gotowych komponentów. Wszystkie właściwości obiektów są odpowiednimi składowymi klas opisujących te obiekty. Oprócz typowych składowych, przynależnych wszystkim obiektom dziedziczącym w klasy **Control**, każdy komponent ma szereg własnych, jemu tylko właściwych składowych. Na przykład pole tekstowe ma właściwość **MaxLength**, pozwalającą ustalić maksymalną długość wprowadzanego napisu, czy właściwość **AcceptsEnter**, decydującą o tym, czy naciśnięcie klawisza Enter w obrębie pola tekstowego dołączy do wprowadzanego tekstu znak przejścia do nowej linii, czy też spowoduje przejście do kolejnego komponentu w oknie.

#### 3.6.6.1 ComboBox, ListBox

Oba komponenty, **ComboBox** i **ListBox**, mają bardzo podobne zastosowanie i bardzo podobny interfejs służący do oprogramowywania ich. Udostępniają one kolekcję **Items**, która przechowuje elementy pokazywane na listach tych komponentów. W najprostszym scenariuszu napisalibyśmy po prostu:

```

...
ComboBox cbItems;
...
cbItems.Add( "napis 1" );
cbItems.Add( "napis 2" );
cbItems.Add( "napis 3" );
...

```

Pojawia się jednak pytanie: w jaki sposób aplikacja może być poinformowana o wyborze konkretnego elementu przez użytkownika? Przypomnijmy sobie, że na poziomie Win32API z każdym elementem **ComboBoxa** można skojarzyć 32-bitową wartość, która może służyć do

identyfikowania elementów (może na przykład przechowywać identyfikator bazodanowy elementu na liście)<sup>16</sup>.

W bibliotece okienkowej .NET elementami ComboBoxa i ListBoxa mogą być dowolne obiekty, nie tylko napisy. Jeśli do listy zostaje dodany obiekt innego typu niż **string**, na liście pojawia się jego reprezentacja napisowa, zaś na liście zapamiętana jest referencja do obiektu.

Aby zasymulować możliwość jaką daje Win32API, czyli umieszczanie na liście napisów i kojarzenie z każdym z nich wartości 32-bitowej, można posłużyć się pomocniczą klasą, która będzie przechowywać pary: napis i wartość 32-bitową. Zauważmy jednak, że programista nie jest ograniczony do jednej wartości skojarzonej z napisem, ponieważ jest to tylko i wyłącznie kwestią zaprojektowania odpowiedniej klasy.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Windows.Forms;

namespace Example
{
    public class MyComboBoxItem
    {
        string text;
        int id;

        public string Text
        {
            get { return text; }
        }

        public int ID
        {
            get { return id; }
        }

        public MyComboBoxItem( string text, int id )
        {
            this.text = text;
            this.id = id;
        }

        public override string ToString()
        {
            return text;
        }
    }

    public class CMainForm : Form
    {
        ComboBox cbItems;

        public CMainForm()
        {
            cbItems = new ComboBox();
            cbItems.Items.Add( new MyComboBoxItem( "ala", 17 ) );
            cbItems.Items.Add( new MyComboBoxItem( "ma", 24 ) );
            cbItems.Items.Add( new MyComboBoxItem( "kota", 19 ) );
            cbItems.Items.Add( new MyComboBoxItem( "!", 78 ) );
            cbItems.SelectedIndexChanged +=
                new EventHandler( cbItems_SelectedIndexChanged );

            this.Controls.Add( cbItems );
        }

        void cbItems_SelectedIndexChanged( object sender, EventArgs e )
        {

```

---

<sup>16</sup>Wartość tą można ustalić bądź pobrać za pomocą par komunikatów **CB\_SETITEMDATA**, **CB\_GETITEMDATA** oraz **LB\_SETITEMDATA**, **LB\_GETITEMDATA**.

```

        if ( cbItems.SelectedItem != null )
        {
            MyComboBoxItem myItem = cbItems.SelectedItem as MyComboBoxItem;
            MessageBox.Show( String.Format( "Wybrano element: {0} - {1}",
                                           myItem.Text, myItem.ID ) );
        }
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

### 3.6.6.2 ToolTip

Wyświetlaniem podpowiedzi zajmuje się obiekt typu **ToolTip**. Podpowiedzi mogą składać się z kilku linii tekstu.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Button b;

        public CMainForm()
        {
            b = new Button();
            b.Text = "Kliknij mnie";
            this.Controls.Add( b );

            ToolTip tTip = new ToolTip();
            tTip.SetToolTip( b, "Podpowiedź\r\nwielolinijkowa" );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

### 3.6.6.3 ListView

Komponent **ListView** jest bardzo przydatny i w związku z tym często wykorzystywany w aplikacjach Windowsowych. Potrafi pokazywać elementy w 4 różnych widokach<sup>17</sup>. Najczęściej korzysta się z widoku szczegółowego, w którym ListView staje się wielokolumnową listą elementów.

Inaczej niż w przypadku **ComboBoxa**, elementy ListView są typu **ListViewItem**. Jeżeli programista chce skojarzyć własną informację z elementem listy, powinien skorzystać z właściwości **Tag** elementu, która może przechować referencję na dowolny obiekt. Również inaczej niż w przypadku **ComboBoxa**, każdy element ListView może mieć własny kolor tła i tekstu.

ListView wyróżnia się spośród innych komponentów tym, że potrafi samodzielnie sortować swoje elementy. Wymaga to zdefiniowania klasy implementującej interfejs **IComparer** i przypisania obiektu tej klasy do właściwości **ListViewItemSorter** komponentu ListView.

<sup>17</sup>Cztery możliwości prezentowania elementów listy przez **ListView** najszybciej można zobaczyć w Eksplorerze Windows, który do pokazywania elementów systemu plików używa właśnie ListView i pozwala przełączać się pomiędzy wszystkimi dostępnymi widokami.





```

foreach ( string s in sHeaders )
    lstItems.Columns.Add( s, 60, HorizontalAlignment.Left );

// elementy
li = lstItems.Items.Add( "Jan" );
li.SubItems.Add( "Kowalski" );
li.SubItems.Add( "1971-05-05" );

li = lstItems.Items.Add( "Adam" );
li.SubItems.Add( "Malinowski" );
li.SubItems.Add( "1975-02-13" );

li = lstItems.Items.Add( "Zbigniew" );
li.SubItems.Add( "Abacki" );
li.SubItems.Add( "1972-05-11" );

// dopasuj szerokości kolumn
foreach ( ColumnHeader ch in lstItems.Columns )
    ch.Width = -2;
}

// po kliku w kolumnę ListView ustal sortowanie wg. tej kolumny
void LV_ColumnClick( object sender, ColumnClickEventArgs e )
{
    lstItems.ListViewItemSorter = new MyLVItemSorter( e.Column );
}

public CMainForm()
{
    lstItems      = new ListView();
    lstItems.Dock = DockStyle.Fill;
    lstItems.FullRowSelect = true;
    lstItems.GridLines = true;
    lstItems.View      = System.Windows.Forms.View.Details;
    lstItems.ColumnClick += new ColumnClickEventHandler( LV_ColumnClick );

    this.Controls.Add( lstItems );

    InitListViewElements();
}

public static void Main()
{
    Application.Run( new CMainForm() );
}
}

```

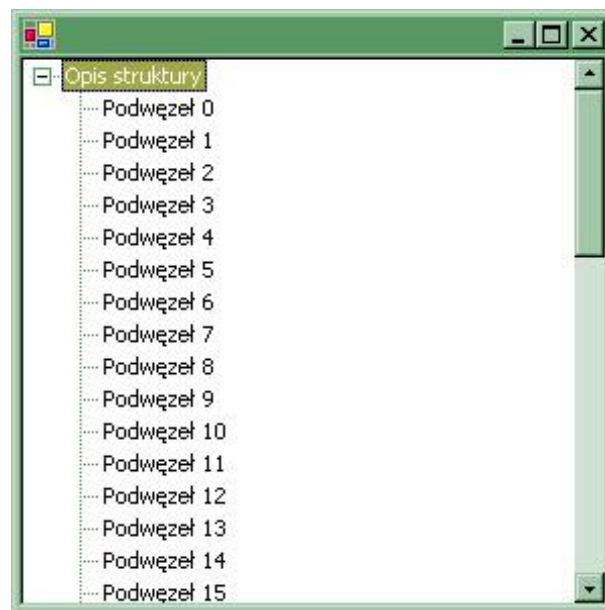
#### 3.6.6.4 TreeView

Komponent **TreeView** zyskał nowy, obiektowy interfejs, w którym każdy węzeł ma kolekcję **Nodes**, przechowującą jego podwężły.

Z komponentem tym wiąże się klasyczny problem: jak radzić sobie z wypełnianiem struktury **TreeView**, jeśli powinien on przechowywać bardzo dużo danych? Oczywiście zainicjowanie całego drzewa w konstruktorze okna macierzystego może nie wchodzić w grę, właśnie z powodu dużej ilości danych.

Problem ten rozwiązuje się zwykle tak, że inicjuje się tylko *jeden* poziom drzewa, poziom główny, dodając przy okazji tym węzłom, które mają przechowywać jakieś podwężły, tylko *jeden*, bardzo specjalny "pusty" podwężel, oznaczony w propercji **Tag** w jakiś określony sposób.

Następnie należy dodać funkcję obsługi zdarzenia **BeforeExpand**, które pojawia się, gdy użytkownik próbuje "rozwinąć" węzeł drzewa przy pomocy symbolu "+" umieszczonego przy węźle. Wewnątrz funkcji obsługi zdarzenia należy sprawdzić, czy rozwijany węzeł ma tylko jeden podwężel i to w dodatku ten specjalnie oznakowany. Jeśli tak - należy ten podwężel usunąć i dobudować kolejny poziom drzewa, znów dodając specjalne "puste" podwężły określonym



Rysunek 3.8: TreeView pozwala pokazać zależności między obiektami

węzłom.

W ten sposób drzewo budowane jest zawsze "na życzenie", przy czym dobudowywany jest zawsze tylko ten poziom drzewa, który jest akurat potrzebny.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Collections;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        TreeView tvItems;

        void InitTVElements()
        {
            TreeNode treeRoot = new TreeNode( "Opis struktury" );
            treeRoot.ForeColor = Color.Blue;

            TreeNode treeSubNode;
            for ( int i=0; i<45; i++ )
            {
                treeSubNode = new TreeNode( String.Format( "Podwzest {0}", i ) );
                treeRoot.Nodes.Add( treeSubNode );
            }

            tvItems.Nodes.Add( treeRoot );
        }

        public CMainForm()
        {
            tvItems = new TreeView();
            tvItems.Dock = DockStyle.Fill;

            this.Controls.Add( tvItems );

            InitTVElements();
        }
    }
}
```

```

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

### 3.6.7 Rozmieszczanie okien potomnych

Dobrze zaprojektowany interfejs użytkownika powinien być czytelny i przejrzysty. Jednak przy odrobinie wprawy i doświadczenia można sobie z tym poradzić. O wiele trudniej jest zaprojektować interfejs tak, aby pozostawał spójny gdy okno zmienia swoje rozmiary, na przykład gdy jest rozciągane przez użytkownika.

Istnieją dwa możliwe rozwiązania: można albo zabronić zmian rozmiaru okna (przez ustawienie właściwości **FormBorderStyle** na **FormBorderStyle.FixedDialog**) albo reagować na zmianę rozmiaru okna i dopasowywać rozmiary okien potomnych do rozmiaru okna macierzystego. Oczywiście nie zawsze można po prostu zabronić zmian rozmiaru okna. Czy w związku z tym .NET wspomaga jakoś proces rozmieszczania okien potomnych przy zmianie rozmiaru okna macierzystego? Otóż tak.

#### 3.6.7.1 Kotwice i dokowanie

Najprostszy sposób dopasowywania rozmiarów okna potomnego do rozmiarów okna macierzystego to tzw. *kotwicowanie*. Wystarczy nadać oknu potomnemu właściwość *bycia zaczepionym* któregoś z boków okna macierzystego, aby okno potomne zachowywało odległość od odpowiedniego boku podczas zmiany rozmiarów okna macierzystego.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Button b;

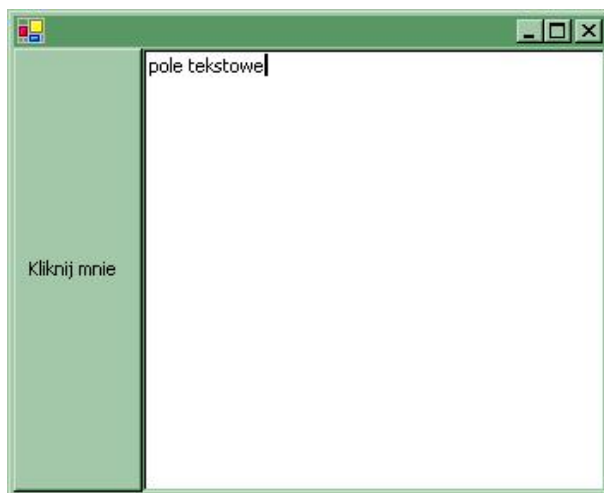
        public CMainForm()
        {
            b = new Button();
            b.Text = "Kliknij mnie";
            b.Location = new Point( 40, 40 );
            b.Anchor = AnchorStyles.Bottom |
                      AnchorStyles.Top    |
                      AnchorStyles.Left   |
                      AnchorStyles.Right;

            this.Controls.Add( b );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

Okna potomne można również *dokować*, czyli przywiązywać na stałe do któregoś z boków lub całego obszaru okna macierzystego. Dokowanie jest szczególnie przydatne w przypadku dwóch



Rysunek 3.9: Okna potomne zadokowane w obrębie okna macierzystego

okien potomnych, bowiem jedno z nich można zadokować do któregoś z boków, a drugie do całego obszaru okna. Oba okna potomne zajmą wtedy cały obszar okna macierzystego i będą poprawnie dostosowywać się do zmian jego rozmiaru.

Należy jedynie pamiętać o tym, aby to okno potomne, które powinno wypełniać obszar okna macierzystego było umieszczone "na wierzchu", czyli nad oknem zadokowanym do któregoś z boków.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Button b;
        TextBox t;

        public CMainForm()
        {
            b = new Button();
            b.Text = "Kliknij mnie";
            b.Dock = DockStyle.Left;

            t = new TextBox();
            t.Multiline = true;
            t.Dock = DockStyle.Fill;

            this.Controls.Add( t );
            this.Controls.Add( b );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

### 3.6.7.2 Panele

Sytuacja, w której okno macierzyste ma tylko dwa okna potomne jest niezwykle rzadka. Zastosowanie pokazanej powyżej metody wydaje się być więc dość ograniczone. Okazuje się jednak, że istnieje specjalny typ komponentu, **Panel**, który z jednej strony zachowuje się jak okno potomne, bowiem jest komponentem umieszczanym wewnątrz jakiegoś okna dialogowego, z drugiej strony zachowuje się jak okno macierzyste, bowiem ma swoją własną kolekcję okien potomnych, które są kotwicowane i dokowane względem obszaru Panela, a nie okna macierzystego.

Można więc używać paneli do podziału okna macierzystego na drobniejsze fragmenty, w obrębie których można dokonywać odpowiednich ustaleń rozmieszczenia okien potomnych. Panele można zagnieżdżać.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Panel p;
        Button b1;
        Button b2;
        TextBox t;

        public CMainForm()
        {
            // panel zadokowany do lewej, a w nim dwa przyciski
            p = new Panel();
            p.Dock = DockStyle.Left;

            b1 = new Button();
            b1.Text = "Kliknij mnie";
            b1.Dock = DockStyle.Top;

            b2 = new Button();
            b2.Text = "Kliknij mnie";
            b2.Dock = DockStyle.Fill;

            p.Controls.Add( b2 );
            p.Controls.Add( b1 );

            // pole tekstowe wypełnia obszar okna
            t = new TextBox();
            t.Multiline = true;
            t.Dock = DockStyle.Fill;

            this.Controls.Add( t );
            this.Controls.Add( p );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

### 3.6.7.3 Splittery

Splittery są elementami graficznymi w postaci poziomych lub pionowych ”belek”, pozwalających użytkownikowi zmienić rozmiary okien potomnych. Splitterów używa się tam, gdzie używa się dokowania.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Panel p; // zawiera b1 i b2
        Button b1;
        Button b2;
        TextBox t;

        Splitter s1; // rozdziela b1 i b2
        Splitter s2; // rozdziela p i t

        public CMainForm()
        {
            // panel zadokowany do lewej, a w nim dwa przyciski
            p = new Panel();
            p.Dock = DockStyle.Left;

            b1 = new Button();
            b1.Text = "Kliknij mnie";
            b1.Dock = DockStyle.Top;

            s1 = new Splitter();
            s1.Dock = DockStyle.Top;

            b2 = new Button();
            b2.Text = "Kliknij mnie";
            b2.Dock = DockStyle.Fill;

            p.Controls.Add( b2 );
            p.Controls.Add( s1 ); // splitter rozdziela b2 i b1
            p.Controls.Add( b1 );

            // pole tekstowe wypełnia obszar okna
            s2 = new Splitter();
            s2.Dock = DockStyle.Left;

            t = new TextBox();
            t.Multiline = true;
            t.Dock = DockStyle.Fill;

            this.Controls.Add( t );
            this.Controls.Add( s2 ); // splitter rozdziela t i p
            this.Controls.Add( p );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

### 3.6.8 GDI+

Tak jak w Win32API istnieją funkcje GDI, tak w świecie .NET istnieje biblioteka GDI+, która udostępnia obiektowy interfejs do funkcji GDI.

Używając GDI+ programista musi pamiętać o jednej bardzo ważnej rzeczy. Otóż zainicjowanie obiektu graficznego, takiego jak pędzel, szczotka, font, obiekt typu **Graphics** itd., spowoduje utworzenie odpowiedniego elementu w systemie, do którego uchwyt będzie przechowywany wewnętrznie obiektu.

Interfejs GDI skonstruowany jest jednak tak, że gdy element graficzny przestaje być potrzeb-

ny, powinien być usunięty, aby system mógł zwolnić zasoby związane z nim. W GDI+ rozwiązano ten problem tak, że wszystkie obiekty graficzne implementują interfejs **IDisposable**, zaś zasoby systemowe są zwracane w metodzie **Dispose**. Warto w tym miejscu przypomnieć sobie więc lukier syntaktyczny ze strony 95, dzięki któremu programista nie musi pamiętać o wywołaniu metody **Dispose**.

### 3.6.8.1 Obiekt Graphics

W GDI do narysowania czegokolwiek potrzebny był kontekst urządzenia. W GDI+ analogiczną rolę pełni obiekt typu **Graphics**.

Obiekt ten jest dostarczany do wszystkich funkcji obsługujących zdarzenia związane z rysowaniem w parametrze typu **PaintEventArgs** i jest to pewna analogia do obsługi zdarzenia WM\_PAINT w Win32API.

Obiekt ten może być również utworzony w dowolnej chwili działania aplikacji za pomocą statycznych funkcji **FromHdc**, **FromHwnd** czy **FromImage**.

Obiekt **Graphics** potrafi wykonać większość operacji związanych z rysowaniem (wymagają wskazania pędzla jako jednego z parametrów), m.in.:

- DrawArc
- DrawBezier
- DrawEllipse
- DrawIcon
- DrawImage
- DrawLine
- DrawPath
- DrawPie
- DrawRectangle
- DrawString

oraz kilka funkcji związanych z wypełnianiem obszarów (wymagają szczotki jako parametru), m.in:

- FillEllipse
- FillPie
- FillRectangle
- FillRegion

### 3.6.8.2 Kolory

W każdym miejscu, w którym potrzebne jest określenie koloru, należy skorzystać z obiektu **Color**. Klasa kolor ma predefiniowane około 140 nazw kolorów, dostępnych jako statyczne właściwości, na przykład **Color.Black**, **Color.AliceBlue**, czy **Color.Red**. Oprócz tego istnieje klasa **SystemColors**, która udostępnia wartości kolorów przypisanych elementom interfejsu graficznego Windows. Mamy tu więc m.in. **SystemColors.ActiveBorder**, **SystemColors.Control**, czy **SystemColors.WindowText** (w sumie około 25 predefiniowanych kolorów).

W każdej chwili programista może utworzyć własny kolor, opisując jego składowe:

```
Color c = Color.FromArgb( 40, 50, 60 );
```

### 3.6.8.3 Czcionki

Konstruktor obiektu **Font** pozwala na określenie parametrów czcionki m.in.: wielkości, stylu, zestawu znaków. Poniższy przykład jest interesujący również z innego powodu: czcionka jest tworzona i usuwana, zaś obiekt **Graphics** nie jest usuwany. Jak to wytłumaczyć?

Otóż zauważmy, że obiekt typu **Graphics** jest dostarczony jako parametr w zmiennej typu **PaintEventArgs**. Oznacza to, że jest on konstruowany gdzieś indziej. Również gdzieś indziej może być więc wykorzystywany. Usunięcie go przez **Dispose** mogłoby w szczególnym przypadku objawić się trudnym do zdiagnozowania błędem.

W przeciwieństwie do obiektu **Graphics**, czcionka jest konstruowana lokalnie i powinna być usunięta po użyciu.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm() {}

        protected override void OnPaint( PaintEventArgs e )
        {
            Graphics g = e.Graphics;

            using ( Font f = new Font( "Courier", 24, FontStyle.Italic ) )
            {
                g.DrawString( "Przykład GDI+", f, Brushes.Black, 0, 0 );
            }
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

### 3.6.8.4 Pędzle, szczotki

GDI+ dostarcza całego zestawu gotowych pędzli i szczotek w obiektach **Pens** i **Brushes**. W każdym miejscu kodu można z nich skorzystać, a jest to o tyle łatwe, że nazwano je po prostu nazwami kolorów. Mamy więc na przykład pióro czarne **Pens.Black** czy szczotkę niebieską **Brushes.Blue**.

Oprócz gotowych piór i szczotek, programista może tworzyć własne. Konstruktor pióra przyjmuje jako parametr kolor i opcjonalnie grubość pióra:





Rysunek 3.10: Gradientowe tło uzyskane dzięki odpowiedniej szczotce

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm() {}

        protected override void OnPaint( PaintEventArgs e )
        {
            Graphics g = e.Graphics;

            using ( Pen p = new Pen( Color.FromArgb( 40, 50, 130 ), 5 ) )
            {
                g.DrawLine( p, 0, 0, 50, 50 );
            }
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

W przypadku szczotek możliwości jest trochę więcej. Istnieje klasa bazowa **Brush**, z której wyprowadzono klasy umożliwiające tworzenie różnego rodzaju szczotek: **SolidBrush**, **HatchBrush**, **LinearGradientBrush**, **PathGradientBrush** czy **TextureBrush**.

Zobaczmy na przykład jak za pomocą szczotki gradientowej wyposażyć okno w automatycznie odrysowywane gradientowe tło:

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Drawing.Drawing2D;

```

```

using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm() {}

        protected override void OnPaint( PaintEventArgs e )
        {
            Graphics g = e.Graphics;

            using ( LinearGradientBrush lgb =
                new LinearGradientBrush( this.ClientRectangle, Color.Green,
                                         Color.LightGreen, -45f, false ) )
            {
                g.FillRectangle( lgb, this.ClientRectangle );
            }

            protected override void OnResize( EventArgs e )
            {
                Invalidate();
            }

            public static void Main()
            {
                Application.Run( new CMainForm() );
            }
        }
    }
}

```

### 3.6.8.5 Obrazki

Tworzenie obrazków możliwe jest dzięki dwóm klasom: **Image** i dziedziczącej z niej **Bitmap**. Obrazek może być utworzony dynamicznie bądź załadowany z pliku (**Image.FromFile**). Obrazek w pamięci można poddać różnym operacjom, można nawet utworzyć obiekt **Graphics** dzięki funkcji **Graphics.FromImage** i rysować na powierzchni obrazka za pomocą funkcji z GDI+.

Gotowy obrazek można zapisać za pomocą metody **Save**, wybierając przy okazji jeden z dostępnych formatów m.in. GIF, BMP, PNG, JPG.

Zawartość obrazka można dzięki funkcji **DrawImage** obiektu **Graphics** narysować w kontekście, na który wskazuje obiekt **Graphics** lub umieścić w komponencie typu **PictureBox**, który może być umieszczony w oknie. Komponent **PictureBox** sam dba o automatycznie odświeżanie swojej zawartości, programista nie musi więc odrysowywać zawartości obrazka gdy okno wymaga odświeżenia.

### 3.6.8.6 Podwójne buforowanie

GDI+ udostępnia możliwość automatycznego podwójnego buforowania wyświetlanej grafiki. Dzięki temu obraz rysowany jest na niewidocznej stronie graficznej i jest błyskawicznie przenoszony na powierzchnię okna. Podwójne buforowanie umożliwia całkowite wyeliminowanie efektu "migania" obrazu podczas rysowania.

Aktywowanie podwójnego buforowania wymaga jedynie 3 lini kodu w konstruktorze okna:

```

/* Wiktor Zychla, 2003 */
this.SetStyle(ControlStyles.UserPaint, true);
this.SetStyle(ControlStyles.AllPaintingInWmPaint, true);
this.SetStyle(ControlStyles.DoubleBuffer, true);

```



Rysunek 3.11: Zegarek w C# z podwójnym buforowaniem grafiki

### 3.6.9 Zegary

Oprogramowanie zegarów jest bardzo proste, ponieważ wystarczy utworzyć obiekt typu **Timer**, przypiąć funkcję do listy słuchaczy zdarzenia **Tick** i ustalić interwał czasu między kolejnymi zgłoszeniami zdarzenia.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        Timer timer;

        public CMainForm()
        {
            timer = new Timer();
            timer.Tick += new EventHandler( Timer_Tick );
            timer.Interval = 50;
            timer.Start();

            this.SetStyle(ControlStyles.UserPaint, true);
            this.SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            this.SetStyle(ControlStyles.DoubleBuffer, true);
        }

        void Timer_Tick( object sender, EventArgs e )
        {
            this.Invalidate();
        }

        protected override void OnPaint( PaintEventArgs e )
        {
            Graphics g = e.Graphics;
            using ( Font f = new Font( "LED", 48 ) )
            {
```

```

        StringFormat sf = new StringFormat();
        sf.Alignment = StringAlignment.Center;
        sf.LineAlignment = StringAlignment.Center;

        g.Clear( SystemColors.Control );
        g.DrawString( DateTime.Now.ToLongTimeString(), f, Brushes.Black,
            this.Width / 2, this.Height / 2, sf );
    }
}

public static void Main()
{
    Application.Run( new CMainForm() );
}
}
}

```

### 3.6.10 Menu

#### 3.6.10.1 Tworzenie menu

Tworzenie menu możliwe jest dzięki dwóm typom danych:

**MainMenu** który służy do tworzenia menu dla okna dialogowego

**ContextMenu** który służy do tworzenia menu kontekstowych dla okien

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        TextBox tb;

        void InitMenus()
        {
            // MainMenu
            MainMenu mainMenu = new MainMenu();

            MenuItem mPlik = new MenuItem( "Plik" );
            MenuItem mPlikZakoncz = new MenuItem( "Zakończ" );
            mPlikZakoncz.Click += new EventHandler( mPlikZakoncz_Click );

            mPlik.MenuItems.Add( mPlikZakoncz );
            mainMenu.MenuItems.Add( mPlik );

            this.Menu = mainMenu;

            // ContextMenu
            ContextMenu cMenu = new ContextMenu();
            MenuItem mZakoncz = new MenuItem( "Zakończ" );
            cMenu.MenuItems.Add( mZakoncz );

            this.tb.ContextMenu = cMenu;
        }

        void mPlikZakoncz_Click( object sender, EventArgs e )
        {
            this.Close();
        }

        public CMainForm()
        {
            tb = new TextBox();
        }
    }
}

```

```

        this.Controls.Add( tb );

        InitMenus();
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

Obiekt typu **ContextMenu**, reprezentujący menu kontekstowe, ma jeszcze jedną interesującą właściwość. Otóż ma on metodę **Show**, która po prostu pokazuje menu kontekstowe przy wskazanym oknie potomnym. Jest to bardzo wygodne wtedy, kiedy menu kontekstowe powinno zostać ujawnione w jakiejś nietypowej sytuacji.

Wyobraźmy sobie na przykład scenariusz, w którym każdemu elementowi drzewa **TreeView** powinno odpowiadać jakieś inne menu kontekstowe, zależne od tego, co zawiera wskazany element. Zamiast przywiązywać jakieś konkretne menu kontekstowe do obiektu **TreeView**, programista może po prostu przechwycić zdarzenie kliknięcia myszą w węzeł drzewa, sprawdzić czy kliknięto prawy klawisz myszy i dopiero wtedy wykorzystać metodę **Show** do pokazania odpowiedniego menu kontekstowego.

### 3.6.10.2 Własne funkcje rysowania menu

Wygląd menu można uatrakcyjnić dzięki możliwości określenia własnych funkcji odpowiedzialnych za rysowanie elementów menu. Programista musi jedynie dodać funkcje obsługi zdarzeń **DrawItem**, odpowiedzialnej za rysowanie i **MeasureItem**, odpowiedzialnej za określanie obszaru zajmowanego przez element menu<sup>18</sup>.

W poniższym przykładzie zmieniamy sposób rysowania tylko tych pozycji menu, które widoczne są na pasku menu w oknie. W analogiczny sposób można jednak określić sposób rysowania pozycji rozwijalnych, dodając na przykład możliwość rysowania ikon obok pozycji menu itp.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WinForms_Dodatki
{
    public class C_XMainMenu : System.Windows.Forms.MainMenu
    {
        private void topMenu_DrawItem(object sender,
                                     System.Windows.Forms.DrawItemEventArgs e)
        {
            Rectangle mRect =
                new Rectangle(e.Bounds.X, e.Bounds.Y, e.Bounds.Width, e.Bounds.Height);
            Rectangle mRect2 =
                new Rectangle(e.Bounds.X, e.Bounds.Y, e.Bounds.Width+1, e.Bounds.Height+1);

            if ( (e.State & DrawItemState.Selected) != 0 )
            {
                e.Graphics.FillRectangle( new SolidBrush( SystemColors.Control ), mRect );
                e.Graphics.DrawRectangle(
                    new Pen( new SolidBrush( SystemColors.ControlDark ), 1 ), mRect );
            }
            else
            {
                if ( (e.State & DrawItemState.HotLight) != 0 )
                {
                    e.Graphics.FillRectangle( new SolidBrush( SystemColors.ControlLightLight ), mRect );
                }
            }
        }
    }
}

```

<sup>18</sup>Kilku innym komponentom wizualnym również można zmieniać wygląd w taki sposób.

```

        e.Graphics.DrawRectangle(
            new Pen( new SolidBrush( SystemColors.ControlDark ), 1 ), mRect );
    }
    else
    {
        e.Graphics.FillRectangle( new SolidBrush(SystemColors.Control), mRect2 );
    }

    MenuItem    mItem    = (MenuItem)sender;
    Font         mFont    = new Font( "MS Sans Serif", 10 );
    StringFormat sFormat = new StringFormat();

    sFormat.Alignment      = StringAlignment.Center;
    sFormat.LineAlignment = StringAlignment.Center;

    e.Graphics.DrawString(mItem.Text, mFont, new SolidBrush(Color.Black), mRect, sFormat );

    mFont.Dispose();
}

private void topMenu_MeasureItem(object sender,
    System.Windows.Forms.MeasureItemEventArgs e)
{
    MenuItem    mItem = (MenuItem)sender;
    Font         mFont = new Font( "MS Sans Serif", 10 );

    SizeF sizeF = e.Graphics.MeasureString( mItem.Text, mFont );
    e.ItemWidth = (int)sizeF.Width;

    mFont.Dispose();
}

public C_XMainMenu( Menu mMenu )
{
    foreach ( MenuItem mItem in mMenu.MenuItems )
    {
        MenuItem newMenuItem = mItem.CloneMenu();

        ApplyMenuProperties ( newMenuItem );
        this.MenuItems.Add ( newMenuItem );
    }
}

private void ApplyMenuProperties( MenuItem mItem )
{
    if ( IsTopMenu( mItem ) )
    {
        mItem.OwnerDraw = true;
        mItem.DrawItem +=
new System.Windows.Forms.DrawItemEventHandler(this.topMenu_DrawItem);
        mItem.MeasureItem +=
new System.Windows.Forms.MeasureItemEventHandler(this.topMenu_MeasureItem);
    }

    foreach ( MenuItem subMenu in mItem.MenuItems )
        ApplyMenuProperties( subMenu );
}

private static bool IsTopMenu( MenuItem mItem )
{
    if ( mItem.Parent == null ) return true;
    return mItem.Parent == mItem.GetMainMenu();
}
}
}

```

Klasa **C\_XMainMenu** określona jest tak, że w kodzie inicjującym menu należy po prostu napisać:

```
MainMenu mainMenu;
```

```
...
C_XMainMenu cxMainMenu = new C_XMainMenu( mainMenu );
this.Menu = cxMainMenu;
```

### 3.6.11 Schowek

Dostęp do schowka systemowego możliwy jest dzięki obiektowi **Clipboard**. W schowku można umieścić dowolny obiekt lub sprawdzić czy znajduje się tam obiekt określonego typu.

Poniższy przykład umieszcza w schowku napis, a następnie wydobywa go stamtąd. Dane przekazane do schowka w ten sposób są dostępne dla wszystkich aplikacji w systemie, podobnie dane pobierane ze schowka mogą pochodzić z dowolnej aplikacji w systemie.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm()
        {
            // umieść dane w schowku
            Clipboard.SetDataObject( "Tekst przesyłany do schowka", true );

            // wydobądź dane ze schowka
            IDataObject ido = Clipboard.GetDataObject();
            if ( ido.GetDataPresent( typeof( string ) ) )
            {
                string s = ido.GetData( typeof( string ) ) as string;
                MessageBox.Show( s );
            }
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}
```

### 3.6.12 Drag & drop

### 3.6.13 Tworzenie własnych komponentów

Jedną z najciekawszych możliwości nowoczesnych technologii informatycznych jest możliwość definiowania własnych komponentów wizualnych. Kiedy nie było jeszcze technologii .NET własne komponenty można było tworzyć w technologii COM, używając do tego Visual Basic'a lub C++.

Możliwości .NET jeszcze bardziej ułatwiają cały ten proces. Tak naprawdę wystarczy utworzyć klasę dziedziczącą z **UserControl** i już może ona funkcjonować jako komponent wizualny. Taki własny komponent może na przykład składać się z dowolnej ilości już istniejących komponentów i automatyzować pewne zależności między nimi, może też tworzyć całkowicie nowe możliwości interfejsowe.

Do czego może przydawać się możliwość tworzenia własnych komponentów?

Wyobraźmy sobie na przykład, że standardowy komponent **ComboBox** chcielibyśmy wypożyczyć w automatyczne dopasowywanie elementu na liście do tekstu wpisywanego przez użytkownika. Zwykły ComboBox tego nie potrafi, ale można utworzyć własny komponent i dodać reakcje na odpowiednie zdarzenia, by uzyskać porządzaną funkcjonalność. Można taki problem rozwiązać bez tworzenia nowego komponentu, tyle że gdyby chcieć użyć takiego ComboBoxa więcej niż



Rysunek 3.12: Najprostszy komponent

raz, utworzenie jednego komponentu wielokrotnego użycia, po prostu ogromnie upraszcza życie programisty.

Wyobraźmy sobie również, że chcielibyśmy mieć zupełnie nowy komponent wizualny, siatkę, z możliwością dodawania wierszy i kolumn i to taką, żeby każda komórka mogła mieć inny kolor, czcionkę wyrównanie czy orientację tekstu. Takiego komponentu standardowo w bibliotece komponentów .NET nie ma. Można by jednak utworzyć własny komponent, dodać mu jakieś struktury danych do przechowywania danych, dodać jakieś właściwości, metody i zdarzenia, tak aby można było sterować takim komponentem z poziomu kodu konkretnego okna, w którym byłby on osadzony, a następnie przeciążyć całkowicie metodę **OnPaint**, dzięki czemu wizualna zawartość komponentu mogłaby być tworzona całkowicie dowolnie, bez żadnego związku z już istniejącymi komponentami.

### 3.6.13.1 Najprostszy komponent

Zacniemy od bardzo prostego przykładu komponentu, który będzie tylko wypisywał tekst w swoim obszarze. Komponent taki jest oknem leżącym gdzieś w jakimś innym oknie. Wewnątrz kodu może więc dowiedzieć się jakie są jego bieżące rozmiary za pomocą właściwości **Width** i **Height**. Również właściwości takie jak **Font**, **Text**, **BackColor** czy **ForeColor** są, jako dziedziczone z klasy **UserControl**, dostępne dla klienta komponentu.

Klient komponentu (czyli kod, który korzysta z tego komponentu) traktuje więc nowo zaprojektowany komponent jak każdy inny - może ustalać rozmiary komponentu, jego kotwiczenie czy dokowanie oraz używać wszystkich potrzebnych właściwości, zdarzeń i metod (tak prosty komponent nie ma żadnych sensownych składowych poza tymi dziedziczonymi z **UserControl**).

W środowisku wizualnym tak zaprojektowany komponent po umieszczeniu w bibliotece obiektowej (takie jest ograniczenie na przykład VisualStudio) mógłby być umieszczony na przystrojniku z komponentami i umieszczany na oknach jak każdy inny komponent z przystrojnika!

```
/* Wiktor Zychla, 2003 */  
using System;  
using System.Drawing;
```



```

using System.Windows.Forms;

namespace Example
{
    public class CKomponent : UserControl
    {
        public CKomponent() {}

        protected override void OnPaint( PaintEventArgs e )
        {
            StringFormat sf = new StringFormat();
            sf.Alignment = StringAlignment.Center;
            sf.LineAlignment = StringAlignment.Center;

            e.Graphics.Clear( this.BackColor );
            e.Graphics.DrawString( this.Text, this.Font, Brushes.Black,
                                   this.Width / 2, this.Height / 2, sf );
        }

        protected override void OnResize( EventArgs e )
        {
            Invalidate();
        }
    }

    public class CMainForm : Form
    {
        CKomponent ck1, ck2;

        public CMainForm()
        {
            ck1 = new CKomponent();
            ck1.Text = "Komponent 1";
            ck1.BackColor = Color.Red;
            ck1.Font = new Font( "Tahoma", 18 );
            ck1.Size = new Size( 50, 50 );
            ck1.Dock = DockStyle.Fill;

            ck2 = new CKomponent();
            ck2.Text = "Komponent 2";
            ck2.Font = new Font( "Courier", 32 );
            ck2.Dock = DockStyle.Top;

            this.Controls.AddRange( new Control[] { ck1, ck2 } );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

### 3.6.13.2 Komponent złożony

Komponent może zawierać w sobie dowolną ilość innych komponentów i wewnątrz swojego kodu przechwytywać ich zdarzenia.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CKomponent : UserControl
    {
        Button b1, b2;
    }
}

```

```

public CKomponent()
{
    b1      = new Button();
    b1.Text = "1";
    b1.Dock = DockStyle.Fill;
    b1.Click += new EventHandler( bt_Click );

    b2      = new Button();
    b2.Text = "2";
    b2.Size = new Size( 40, 40 );
    b2.Dock = DockStyle.Right;
    b2.Click += new EventHandler( bt_Click );

    this.Controls.AddRange( new Control[] { b1, b2 } );
}

public void bt_Click( object sender, EventArgs e )
{
    MessageBox.Show( ((Control)sender).Text );
}
}

public class CMainForm : Form
{
    CKomponent ck1, ck2;

    public CMainForm()
    {
        ck1      = new CKomponent();
        ck1.Text = "Komponent 1";
        ck1.BackColor = Color.Red;
        ck1.Font = new Font( "Tahoma", 8 );
        ck1.Size = new Size( 50, 50 );
        ck1.Dock = DockStyle.Fill;

        ck2      = new CKomponent();
        ck2.Text = "Komponent 2";
        ck2.Font = new Font( "Courier", 12 );
        ck2.Dock = DockStyle.Top;

        this.Controls.AddRange( new Control[] { ck1, ck2 } );
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

### 3.6.13.3 Definiowanie własnych zdarzeń

Zestaw zdarzeń udostępnianych przez komponenty również może być rozbudowany. Wyobraźmy sobie, że chcielibyśmy mieć komponent, który zawierałby **ComboBox** i mały przycisk z napisem „+” z boku. Chcielibyśmy, aby użytkownik mógł użyć tego przycisku do dodawania nowych elementów do **ComboBoxa**. Z punktu widzenia logiki tego komponentu zdarzenie oznaczające chęć dodania nowego elementu mogłoby jednak pojawiać się również wtedy, kiedy użytkownik wpisałby w pole tekstowe **ComboBoxa** jakiś tekst spoza tekstów dostępnych na liście. To już aż dwa przypadki, kiedy takie zdarzenie mogłoby się pojawić.

Możemy więc zaprojektować nowe zdarzenie. Nazwiemy je **AddNewText**. Być może w toku prac okazałoby się, że zdarzenie takie powinno mieć jakieś dodatkowe parametry. Zaprojektowalibyśmy wtedy nową klasę, **AddNewTextEventArgs** i zmodyfikowalibyśmy deklarację zdarzenia. Należałoby jedynie pamiętać o zachowaniu konwencji: odpowiedni delegat powinien mieć dwa parametry, z czego pierwszy powinien wskazywać na źródło zdarzenia, drugi powinien

dziedziczyć z klasy **EventArgs**, rozszerzając ją o dodatkowe parametry zdarzenia.

Zauważmy, że kod klienta korzysta z tego zdarzenia tak samo jak z każdego zdarzenia. Z punktu widzenia kodu nie ma różnicy między naszym nowym zdarzeniem, a już istniejącymi zdarzeniami.

```
/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CComboBox : UserControl
    {
        ComboBox cb;
        Button bt;

        // własne zdarzenie definiowane przez komponent
        public event EventHandler AddNewText;

        public CComboBox()
        {
            cb = new ComboBox();
            cb.Dock = DockStyle.Fill;

            bt = new Button();
            bt.FlatStyle = FlatStyle.Popup;
            bt.Text = "+";
            bt.Dock = DockStyle.Right;
            bt.Click += new EventHandler( bt_Click );

            this.Controls.AddRange( new Control[] { cb, bt } );
        }

        protected override void OnResize( EventArgs e )
        {
            bt.Size = new Size( this.Height, this.Height );
        }

        // przechwyć klik w przycisk i na zewnątrz wystaw jako AddNewText
        void bt_Click( object sender, EventArgs e )
        {
            // sprawdź czy są jacyś słuchacze
            if ( AddNewText != null )
                AddNewText( this, new EventArgs() );
        }
    }

    public class CMainForm : Form
    {
        CComboBox cb1;

        public CMainForm()
        {
            cb1 = new CComboBox();
            cb1.Size = new Size( 150, 20 );

            cb1.AddNewText += new EventHandler( cb_AddNewText );

            this.Controls.Add( cb1 );
        }

        void cb_AddNewText( object sender, EventArgs e )
        {
            MessageBox.Show( "Zgłoszono zdarzenie AddNewText!" );
        }

        public static void Main()
        {

```

```

        Application.Run( new CMainForm() );
    }
}

```

### 3.6.14 Typowe okna dialogowe

#### 3.6.14.1 Okna wyboru plików

Standardowo, programista ma do dyspozycji okno wyboru pliku do otwarcia i zamknięcia (**OpenFileDialog** i **SaveFileDialog**). Oba działają bardzo podobnie - po ustaleniu właściwości należy wywołać metodę do pokazania okna i przechwycić rezultat, wskazujący na to czy przypadkiem użytkownik nie anulował okna. Lista wybranych plików dostępna jest dzięki właściwości **FileName[s]**.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm()
        {
            OpenFileDialog of = new OpenFileDialog();
            of.InitialDirectory =
                Environment.GetFolderPath( Environment.SpecialFolder.ProgramFiles );
            of.Title = "Wybierz plik do otwarcia...";
            of.Filter = "Moje pliki (*.xyz)|*.xyz|"+
                "Wszystkie pliki (*.*)|*.*";
            of.Multiselect = true;

            DialogResult res = of.ShowDialog();
            if ( res == DialogResult.OK )
                foreach ( string fileName in of.FileNames )
                    MessageBox.Show( "Wybrano plik " + fileName );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

#### 3.6.14.2 Okno wyboru czcionki

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm()
        {
            FontDialog fd = new FontDialog();
            fd.ShowColor = true;
            fd.ShowEffects = true;

            DialogResult res = fd.ShowDialog();
            if ( res == DialogResult.OK )

```

```

        MessageBox.Show( "Wybrano czcionkę: " + fd.Font.ToString() );
    }

    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

### 3.6.14.3 Okno wyboru koloru

```

/* Wiktor Zychla, 2003 */
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Example
{
    public class CMainForm : Form
    {
        public CMainForm()
        {
            ColorDialog cd = new ColorDialog();
            cd.AllowFullOpen = true;

            DialogResult res = cd.ShowDialog();
            if ( res == DialogResult.OK )
                MessageBox.Show( "Wybrano kolor: " + cd.Color.ToString() );
        }

        public static void Main()
        {
            Application.Run( new CMainForm() );
        }
    }
}

```

## 3.7 Ciekawostki .NET

### 3.7.1 Błąd odśmiecania we wczesnych wersjach Frameworka

Odśmiecacz, jako kluczowy element środowiska uruchomieniowego, powinien radzić sobie w wielu różnych sytuacjach. Ciekawostką jest fakt, że w pierwszej wersji .NET Frameworka odśmiecacz czasami nie potrafił radzić sobie z usuwaniem niepotrzebnej pamięci.

Przykładowy program próbował w pętli rezerwować sobie coraz większy fragment pamięci, zaczynając od bloku 10MB i zwiększając wielkość bloku o 10kB w każdej iteracji. Rezerwowany wewnątrz pętli blok powinien być oznaczony jako nieużywany, tak się jednak nie działo. W konsekwencji program działał tak długo, aż skończyła się dostępna w systemie pamięć, po czym kończył się z wyjątkiem **Out of memory**.

Problem został usunięty w poprawce **SP2** do Frameworka 1.0. Nie występuje we Frameworku 1.1.

```

/* Wiktor Zychla, 2003 */
using System;

class Test
{
    public static void Main(string [] args)
    {
        for (int tries=0; tries < 200; tries++)
        {
            int iNElements = 10000000 + tries*10000;

```

```

        try
        {
            byte [] aMyMem = new byte [iNElements-1];
        }

        catch (Exception e)
        {
            Console.WriteLine (e);
            Console.WriteLine (iNElements);
            return;
        }
    }
}
}

```

### 3.7.2 Dostęp do prywatnych metod klasy

Mimo, że CLS dość mocno broni się przed dostępem do prywatnych składowych obiektów, istnieje sposób na dostęp do prywatnych metod. Okazuje się, że przydaje się tu mechanizm refleksji. Sposób ten, z oczywistych powodów, nie nadaje się do wydobywania informacji o prywatnych polach klas. Metody są bowiem składowymi statycznymi i są częścią opisu klasy. Pola zaś są składowymi dynamicznymi, unikalnymi dla każdej nowej instancji obiektu.

```

/* Wiktor Zychla, 2003 */
using System;
using System.Reflection;

namespace Example
{
    // Klasa z metodą prywatną
    public class ClassA
    {
        private void metodaPrywatna()
        {
            Console.WriteLine( "Metoda prywatna." );
        }
    }

    public class CMain
    {
        static void Main()
        {
            ClassA classA = new ClassA();
            Type type = classA.GetType();

            foreach (MethodInfo method in type.GetMethods(
                BindingFlags.Instance | BindingFlags.NonPublic))
            {
                if (method.Name == "metodaPrywatna")
                {
                    method.Invoke( classA, new object[] {} );
                }
            }
        }
    }
}

```

### 3.7.3 Informacje o systemie

#### 3.7.3.1 Informacje o konfiguracji systemu

Informacje o konfiguracji systemu dostępne są dzięki propercjom i metodom klasy **Environment**. Istnieje m.in. możliwość uzyskania informacji:

**CommandLine** - parametry linii poleceń

**CurrentDirectory** - bieżąca ścieżka

**GetEnvironmentVariables** - zmienne systemowe

**GetFolderPath** - lokalizacja specjalnych folderów systemowych

**GetLogicalDrives** - lista napędów logicznych w systemie

**MachineName** - nazwa komputera

**SystemDirectory** - folder systemowy

**Version** - wersja systemu

### 3.7.3.2 Informacje o środowisku graficznym

Dostęp do informacji o konfiguracji środowiska graficznego możliwy jest dzięki klasie **SystemInformation**. Dostępne są m.in. następujące informacje :

**ComputerName** - nazwa komputera

**MonitorCount** - liczba monitorów

**MouseButtons** - liczba przycisków myszy

**MouseWheelPresent** - dodatkowa możliwość myszy (obecność kółka)

oraz mnóstwo proporcji opisujących rozmiary obiektów graficznych: kursora, ikon, czcionki menu, rozmiaru ekranu itd.

### 3.7.3.3 Informacje o środowisku uruchomieniowym

Istnieje również możliwość zasięgnięcia informacji o środowisku uruchomieniowym. Służy do tego klasa **RuntimeEnvironment** z biblioteki **System.Runtime.InteropServices**. Udostępnia informacje:

**GetRuntimeDirectory** - ścieżka, w której zainstalowano środowisko uruchomieniowe

**GetSystemVersion** - wersja środowiska uruchomieniowego

### 3.7.4 Własny kształt kursora myszy

W szczególnych przypadkach można zmienić kształt kursora myszy. Biblioteki .NET udostępniają klasę **Cursors**, która udostępnia większość typowych cursorów systemowych. Co jednak zrobić, gdy typowe kształty nie wystarczają?

Otóż można skorzystać z możliwości przekształcenia dowolnej bitmapy na cursor myszy:

```
...
// twórz bitmapę, można użyć już istniejącej
Bitmap b = new Bitmap( 30, 15 );
Graphics g = Graphics.FromImage ( b );
g.DrawString ( "NAPIS", this.Font, Brushes.Black, 0, 0 );

// uchwyt do ikony utworzonej z bitmapy
IntPtr ptr = b.GetHicon();
```

```
// zamień na ikonę lub kursor
Icon i = Icon.FromHandle ( ptr );
Cursor c = new Cursor( ptr );

// przypisz kursor oknu
this.Cursor = c;
```

### 3.7.5 Własne kształty okien

Windows potrafi wykorzystać maski bitowe do definiowania kształtów okien. W świecie .NET nieregularne okna można definiować dzięki obiektom **GraphicsPath**, na przykład:

```
...
// elipsoidalny kształt okna
GraphicsPath gP = new GraphicsPath();

gP.AddEllipse ( 0, 0, this.Width, this.Height );
this.Region = new Region ( gP );
```

### 3.7.6 Podwójne buforowanie grafiki w GDI+

W GDI+ istnieje możliwość włączenia automatycznego podwójnego buforowania wyświetlanej grafiki. Obraz zyskuje dzięki temu na płynności, co jest szczególnie przydatne gdy obraz odświeżany jest dość często.

```
...
SetStyle(ControlStyles.UserPaint, true);
SetStyle(ControlStyles.AllPaintingInWmPaint, true);
SetStyle(ControlStyles.DoubleBuffer, true);
```

### 3.7.7 Sprawdzanie uprawnień użytkownika

Jeśli podczas pracy aplikacji zachodzi potrzeba sprawdzenia, czy użytkownik ma uprawnienia administratora można posłużyć się następującym kodem:

```
using System.Security;
using System.Security.Permissions;
using System.Security.Principal;

...
WindowsPrincipal wid = new WindowsPrincipal( WindowsIdentity.GetCurrent());
bool isAdmin = wid.IsInRole(WindowsBuiltInRole.Administrator);
```

### 3.7.8 Ikona skojarzona z plikiem

System operacyjny Windows z niektórymi rozszerzeniami plików kojarzy aplikacje, służące do ich otwierania. Menedżery plików pokazując listę plików, pokazują też ikony skojarzone z plikami, czyli ikony aplikacji służących do otwierania tych plików.

Biblioteki .NET nie udostępniają bezpośrednio funkcji do wydobywania ikon skojarzonych z plikami, należy więc sięgnąć do Win32API:

```
public class ExtractIcon
{
    [DllImport("Shell32.dll")]
    private static extern int SHGetFileInfo
    (
        string pszPath,
        uint dwFileAttributes,
        out SHFILEINFO psfi,
        uint cbFileInfo,
        SHGFI uFlags
    );
}
```



```

);

[StructLayout(LayoutKind.Sequential)]
private struct SHFILEINFO
{
    public SHFILEINFO(bool b)
    {
        hIcon=IntPtr.Zero;iIcon=0;dwAttributes=0;szDisplayName="";szTypeName="";
    }
    public IntPtr hIcon;
    public int iIcon;
    public uint dwAttributes;
    [MarshalAs(UnmanagedType.LPStr, SizeConst=260)]
    public string szDisplayName;
    [MarshalAs(UnmanagedType.LPStr, SizeConst=80)]
    public string szTypeName;
};

private ExtractIcon() {}

private enum SHGFI
{
    SmallIcon    = 0x00000001,
    LargeIcon    = 0x00000000,
    Icon         = 0x00000100,
    DisplayName  = 0x00000200,
    Typename     = 0x00000400,
    SysIconIndex = 0x00004000,
    UseFileAttributes = 0x00000010
}

public static Icon GetIcon(string strPath, bool bSmall)
{
    SHFILEINFO info = new SHFILEINFO(true);
    int cbFileInfo = Marshal.SizeOf(info);
    SHGFI flags;
    if (bSmall)
        flags = SHGFI.Icon|SHGFI.SmallIcon|SHGFI.UseFileAttributes;
    else
        flags = SHGFI.Icon|SHGFI.LargeIcon|SHGFI.UseFileAttributes;

    SHGetFileInfo(strPath, 256, out info,(uint)cbFileInfo, flags);
    return Icon.FromHandle(info.hIcon);
}
}

```

### 3.7.9 WMI

WMI (ang. *Windows Management Instrumentation*) jest mechanizmem umożliwiającym diagnozowanie stanu komputera i systemu operacyjnego w jednorodny sposób: WMI udostępnia obiekt, który interfejsem przypomina bazę danych. Aby uzyskać interesujące informacje, należy po prostu zadać odpowiednie zapytanie w języku SQL, odwołując się do odpowiedniego obiektu WMI.

Możliwości WMI są naprawdę duże. Zadając odpowiednie zapytanie można dowiedzieć się mnóstwo szczegółów takich jak np. marka producenta płyty głównej, częstotliwość taktowania procesora, numer wersji BIOSu i wiele, wiele innych.

WMI jest dostępne w Windows począwszy od Windows 2000. W Windows 98 musi być doinstalowane przez użytkownika.

```

using System;
using System.Management;
namespace WMI
{
    class CExample
    {

```

```

static void Main(string[] args)
{
    ManagementObjectSearcher query1 =
        new ManagementObjectSearcher("SELECT * FROM win32_OperatingSystem") ;
    ManagementObjectCollection queryCollection1 = query1.Get();
    foreach( ManagementObject mo in queryCollection1 )
    {
        Console.WriteLine("Name : " + mo["name"].ToString());
        Console.WriteLine("Version : " + mo["version"].ToString());
        Console.WriteLine("Manufacturer : " + mo["Manufacturer"].ToString());
        Console.WriteLine("Computer Name : " + mo["csname"].ToString());
        Console.WriteLine("Windows Directory : " + mo["WindowsDirectory"].ToString());
    }

    query1 = new ManagementObjectSearcher("SELECT * FROM win32_Processor") ;
    queryCollection1 = query1.Get();
    foreach( ManagementObject mo in queryCollection1 )
    {
        Console.WriteLine(mo["Caption"].ToString());
        Console.WriteLine(mo["CurrentClockSpeed"].ToString());
    }

    query1 = new ManagementObjectSearcher("SELECT * FROM win32_BIOS") ;
    queryCollection1 = query1.Get();
    foreach( ManagementObject mo in queryCollection1 )
    {
        Console.WriteLine(mo["version"].ToString());
    }
}
}
}

```

## 3.8 Bazy danych i ADO.NET

### 3.8.1 Interfejsy komunikacji z bazami danych

Pisząc aplikację, której zadaniem jest gromadzenie i przetwarzanie informacji, programista często staje przed wyborem sposobu gromadzenia danych. Współcześnie najczęściej wykorzystuje się do tego serwery baz danych, bowiem gwarantują one, m.in. bezpieczeństwo, integralność i nienaruszalność danych.

Z punktu widzenia programisty, serwer baz danych jest zwykłym programem, który swoje usługi dostępu do danych oferuje wielu klientom. Schemat takiej komunikacji jest identyczny jak znany nam już z poprzednich rozdziałów schemat wymiany danych między serwerem, a klientem sieciowym: jakiś protokół **fizyczny** pełni rolę nośnika informacji, zaś jakiś protokół **logiczny** określa postać wymienianych komunikatów.

Interfejsy programowania baz danych zwalniają programistę z konieczności czuwania nad szczegółami komunikacji, pozwalają zaś skupić się na wymianie danych między klientem a serwerem. Jak zobaczymy w kolejnych podrozdziałach, istnieją trzy podstawowe operacje udostępniane przez interfejs programowania baz danych, których programista musi użyć do komunikacji z serwerem:

1. Otwarcie połączenia z wybranym serwerem baz danych
2. Wykonanie operacji na otwartym połączeniu
3. Zamknięcie połączenia do bazy danych

Możliwości oferowane przez poszczególne serwery baz danych różnią się od siebie. Mogłoby się więc wydawać, że interfejs programowania serwera MySQL, Microsoft SQL Server, czy Oracle muszą się od siebie różnić. Na szczęście nie jest aż tak źle, opracowano bowiem takie

interfejsy, które są oderwane od szczegółów implementacji konkretnego serwera i (przynajmniej teoretycznie) pozwalają oprogramować komunikację z każdym dostępnym serwerem baz danych w dokładnie taki sam sposób.

### 3.8.1.1 ODBC

**Open DataBase Connectivity** jest interfejsem umożliwiającym dostęp do danych składowanych w dowolnym systemie zarządzania bazami danych (**DataBase Management System**, DBMS<sup>19</sup>). ODBC jest zbudowany w oparciu o specyfikacje X/Open i ISO/IEC.

W systemie Windows za komunikację z systemami zarządzania baz danych odpowiada biblioteka **odbc32.dll**. Zaimplementowano wiele sterowników ODBC dla różnych DBMSów. Jeśli na rynku pojawia się nowy system bazodanowy, to jest niemal pewne, że będzie on potrafił komunikować się za pomocą ODBC. System Windows potrafi komuikować się za pomocą ODBC m.in. z MS SQL Severem, Oracle, Visual Fox Pro, czy bazami MS Access. Producenci nowych rozwiązań bazodanowych najczęściej sami dostarczają odpowiednie sterowniki, tak jest na przykład w przypadku serwera MySQL.

### 3.8.1.2 OLE DB

OLE DB jest rozwinięciem idei ODBC. W teorii umożliwia dostęp do dowolnych danych, nie tylko relacyjnym DBMSów.

Idea OLE DB polega na współistnieniu trzech rodzajów obiektów. Są to:

- dostawcy danych (*data providers*), którzy przechowują i udostępniają dane
- konsumenci danych (*data consumers*, którzy mogą korzystać z danych
- składniki usługowe (*service components*), które przetwarzają dane

Obiekty z każdej z tych grup muszą po prostu udostępniać pewien ściśle określony (przez standard OLE DB) zbiór funkcji. Teoretycznie można sobie wyobrazić na przykład taki scenariusz, w którym dostawca danych przechowuje dane w pliku tekstowym, składnik usługowy te dane sortuje, zaś konsument pobiera wynik.

### 3.8.1.3 ADO i ADO.NET

Tak naprawdę programista piszący aplikacje klienckie raczej nigdy nie będzie zmuszony do pisania własnych obiektów - dostawców danych, ani własnych usług OLE DB. Obie te funkcje spełniane są przez systemy zarządzania bazami danych i istotne są z punktu widzenia projektantów tych systemów. Programista aplikacji klienckiej jest za to zainteresowany interfejsem programowania przeznaczonym do konsumowania danych udostępnianych przez systemy bazodanowe. Najpopularniejszym interfejsem programowania konsumentów danych, dodatkowo opakowanym interfejs OLE DB, jest ADO.

Interfejs ADO (**ActiveX Data Objects**) został zaprojektowany jako interfejs obiektowy, udostępniany w modelu COM. Zdobył dużą popularność wśród programistów, ponieważ szybko powstały komponenty ADO dla popularnych systemów RAD (**Visual Basic**, **Delphi**, języki skryptowe). Siłą ADO od początku była jego prostota i jednorodność - ADO umożliwia pisanie aplikacji w dużym stopniu niezależnych od sposobu składowania danych.

ADO.NET jest kolejnym krokiem w stronę uproszczenia interfejsu klienta OLE DB, dostępnego dla programistów aplikacji na platformie .NET. Dzięki ADO.NET dostęp do baz danych

<sup>19</sup>DBMSami są na przykład serwery baz danych.

wygląda niemal identycznie nie tylko w różnych językach platformy .NET, ale jest niezależny od dostawcy danych. Oznacza to, że praktycznie nie powinno mieć większego znaczenia czy składownikiem danych aplikacji jest MySQL, MS SQL Server czy Oracle, bowiem aplikacja komunikuje się z nimi wszystkimi w prawie identyczny sposób.

Dla połączeń OLEDB, interfejs ADO.NET udostępnia szereg klas, których nazwy rozpoczynają się od **OleDb...**, na przykład **OleDbConnection**. Specjalnie dla MS SQL Servera przygotowano specjalizowany zestaw klas przeznaczonych wyłącznie dla MS SQL Servera, których funkcjonalność jest identyczna, tyle że ich nazwy rozpoczynają się od **Sql...** Inni dostawcy systemów baz danych podjęli tę konwencję, przygotowując specjalizowane klasy dla komunikacji z serwerem Oracle czy MySQL.

Zaletą specjalizowanych klas jest ich szybkość. Na przykład dzięki użyciu klas **Sql...** aplikacja wymienia dane z serwerem MS SQL około 2 razy szybciej niż za pomocą klas **OleDb...**

Ich wada polega zaś na tym, że obsługują tylko jeden typ dostawcy danych, w przeciwieństwie do klas **OleDb...**, które są ogólne i jedyna różnica w dostępie do danych z różnych serwerów wynika z konieczności nieco innego zainicjowania połączenia do bazy danych.

### 3.8.2 Manualne zakładanie bazy danych

Sposoby wymiany danych między serwerem bazy danych a aplikacją omówimy na przykładzie serwera bazodanowego Microsoft SQL Server. Wymiana danych z innymi serwerami baz danych wygląda identycznie od strony interfejsu programowego. Różnice mogą pojawiać się tylko wtedy, kiedy serwer bazodanowy, z którym komunikuje się aplikacja, nie obsługuje pewnych mechanizmów, których spodziewa się aplikacja.

Przykład rozpoczniemy od założenia bazy danych. Większość serwerów baz danych wspomaga operacje administracji specjalnymi narzędziami z wygodnym interfejsem użytkownika<sup>20</sup>, jednak prawie zawsze dają możliwość korzystania bezpośrednio z poleceń języka SQL.

Skorzystamy więc z tej możliwości, aby utworzyć prostą bazę danych na serwerze MS SQL Server. Do każdej wersji serwera SQL Server (dotyczy to nawet wersji "Desktop", czyli MSDE) dołączone jest narzędzie o nazwie **osql**, które pozwala na wydawanie poleceń SQL serwerowi.

Zacznijmy od połączenia się ze wskazanym serwerem (nazwa **(local)** oznacza serwer lokalny) jako wskazany użytkownik. MS SQL Server może podczas pracy używać własnych mechanizmów uwierzytelniania, niezależnych od uwierzytelniania w systemie. W takim scenariuszu administrator serwera nazywa się **sa** i on tworzy kolejnych użytkowników i nadaje im uprawnienia do korzystania z poszczególnych baz danych.

```
C:\MSSQL7\Binn>osql -H (local) -U sa
Password:
1>
```

Program OSQL nawiązał połączenie z serwerem i oczekuje na polecenia w języku SQL. Użytkownik może podać dowolną ilość poleceń rozdzielonych znakiem ";" i zakończonych poleceniem **GO**, które spowoduje wykonanie poleceń i zwrócenie wyników do okna konsoli.

```
1> SELECT @@VERSION
2> GO
```

```
Microsoft SQL Server 7.00 - 7.00.623 (Intel X86)
Nov 27 1998 22:20:07
Copy
right (c) 1988-1998 Microsoft Corporation
MSDE on Windows 4.10 (Build 1998: )
```

```
(1 row affected)
```

<sup>20</sup>W przypadku serwera MS SQL Server, okienkowym narzędziem administracyjnym może być nawet Microsoft Access

Najpierw utworzymy nową bazę danych i uczynimy ją bieżącą:

```
CREATE DATABASE sqlTEST
GO
USE sqlTEST
GO
```

Następnie utworzymy dwie tabele z danymi, **T\_STUDENT** i **T\_UCZELNIA**, tworząc przy okazji relację jeden-do-wielu między nimi (wielu studentów może uczęszczać do jednej uczelni).

```
CREATE TABLE T_UCZELNIA
( ID_UCZELNIA INT IDENTITY(1,1) NOT NULL
  CONSTRAINT PK_UCZELNIA PRIMARY KEY NONCLUSTERED,
  Nazwa varchar(150) NOT NULL,
  Miejscowosc varchar(50) NOT NULL )

CREATE TABLE T_STUDENT
( ID_UCZEN INT IDENTITY(1,1) NOT NULL
  CONSTRAINT PK_STUDENT PRIMARY KEY NONCLUSTERED,
  ID_UCZELNIA INT NOT NULL
  CONSTRAINT FK_STUDENT_UCZELNIA REFERENCES T_UCZELNIA(ID_UCZELNIA),
  Nazwisko varchar(150) NOT NULL,
  Imie varchar(150) NOT NULL )
```

Mając przygotowane tabele, dodajmy jakieś przykładowe dane:

```
INSERT T_UCZELNIA VALUES ( 'Uniwersytet Wrocławski', 'Wrocław' )
INSERT T_UCZELNIA VALUES ( 'Uniwersytet Warszawski', 'Warszawa' )
INSERT T_STUDENT VALUES ( 1, 'Kowalski', 'Jan' )
INSERT T_STUDENT VALUES ( 1, 'Malinowski', 'Tomasz' )
INSERT T_STUDENT VALUES ( 2, 'Nowak', 'Adam' )
INSERT T_STUDENT VALUES ( 2, 'Kamińska', 'Barbara' )
```

Sprawdźmy na wszelki wypadek poprawność wpisanych danych:

```
SELECT * FROM T_STUDENT WHERE ID_UCZELNIA=1
```

### 3.8.3 Nawiązywanie połączenia z bazą danych

Naszą bazodanową aplikację rozpoczniemy od napisania szkieletu kodu - próby połączenia się z bazą danych. Aplikację tę będziemy rozwijać o kolejne elementy komunikacji z serwerem bazy danych.

Do nawiązania połączenia potrzebne jest poprawne zainicjowanie obiektu typu **SqlConnection** (w przypadku protokołu OleDb - **OleDbConnection**). Przyjęto pewną zasadę, wedle której parametry połączenia przekazuje się w postaci napisu w propepcji **ConnectionString** obiektu połączenia. Napis ten jest odpowiednio sformatowany i przechowuje informacje m.in. o:

- Rodzaju dostawcy protokołu OleDb **Provider**
- Nazwie serwera **Server**
- Nazwie bazy danych **Database**
- Nazwie użytkownika **User ID**
- Hasle użytkownika **Pwd**

```
using System;
using System.Data;
using System.Data.SqlClient;
```

```
namespace Example
{
    class CExample
```

```

{
    public static string BuildConnectionString(string serverName,
                                              string dbName,
                                              string userName,
                                              string passWd)
    {
        return String.Format(
            @"Server={0};Database={1};User ID={2};Pwd={3};Connect Timeout=15",
            serverName, dbName, userName, passWd);
    }

    static void PracaZSerwerem( SqlConnection sqlConn )
    {
        Console.WriteLine( "Połączony z serwerem!" );
    }

    public static void Main(string[] args)
    {
        SqlConnection sqlConn = new SqlConnection();

        sqlConn.ConnectionString =
            BuildConnectionString( "(local)", "sqlTEST", "sa", String.Empty );

        try
        {
            sqlConn.Open();

            PracaZSerwerem( sqlConn );

            sqlConn.Close();
        }
        catch ( Exception ex )
        {
            Console.WriteLine( ex.Message );
        }
    }
}

```

### 3.8.4 Pasywna wymiana danych

Pierwszym ze sposobów komunikacji z serwerem baz danych jaki udostępnia ADO.NET jest komunikacja pasywna. Serwer otrzyma polecenie do wykonania i ew. zwróci wyniki, jednak po zakończeniu operacji to programista będzie musiał podejmować decyzje co do dalszej pracy z serwerem. W tym scenariuszu dane mogą zostać pobrane i od tej pory serwer przestanie interesować się tym, co się z nimi stało. Jeżeli po pewnym czasie program przyśle serwerowi zestaw poleceń dotyczący na przykład aktualizacji wcześniej pobranych danych, to z punktu widzenia serwera będzie to niezależna operacja.

Do realizacji pasywnej wymiany danych potrzebny jest obiekt **SqlCommand**, który określa parametry komendy przekazywanej serwerowi. Obiekt ten może zadaną komendę wykonać, zwracając zbiór rekordów z bazy danych, wartość skalarną lub pusty zbiór wyników, w zależności od postaci komendy. Komendy specyfikuje się oczywiście w języku SQL.

Zbiór rekordów będących wynikiem działania komendy SQL zostanie zwrócony dzięki metodzie **ExecuteReader** obiektu **SqlCommand**. Ściślej, wynikiem działania tej metody będzie obiekt typu **SqlDataReader**, który pozwala na obejrzenie wszystkich wierszy wyniku. Obiekt ten, dzięki indeksowi, pozwala na obejrzenie poszczególnych kolumn z zapytania SQL.

```

...
static void PracaZSerwerem( SqlConnection sqlConn )
{
    SqlCommand sqlCmd = new SqlCommand();

    sqlCmd.Connection = sqlConn;

```

```

sqlCmd.CommandText = "SELECT Imie, Nazwisko, Nazwa FROM T_STUDENT, T_UCZELNIA "+
    "WHERE T_STUDENT.ID_UCZELNIA = T_UCZELNIA.ID_UCZELNIA";

SqlDataReader sqlReader = sqlCmd.ExecuteReader();
while ( sqlReader.Read() )
{
    Console.WriteLine( "{0,-12}{1,-12}{2,-20}",
        (string)sqlReader["Imie"],
        (string)sqlReader["Nazwisko"],
        (string)sqlReader["Nazwa"] );
}
...

```

C:\Example>example

Jan	Kowalski	Uniwersytet Wrocławski
Tomasz	Malinowski	Uniwersytet Wrocławski
Adam	Nowak	Uniwersytet Warszawski
Barbara	Kamińska	Uniwersytet Warszawski

Zwrócenie wartości skalarnej jest prostsze, bowiem wystarczy po prostu przechwycić wynik działania metody **ExecuteScalar** obiektu **SqlCommand**.

```

static void PracaZSerwerem( SqlConnection sqlConn )
{
    SqlCommand sqlCmd = new SqlCommand();

    sqlCmd.Connection = sqlConn;
    sqlCmd.CommandText = "SELECT @@VERSION";

    string version = (string)sqlCmd.ExecuteScalar();
    Console.WriteLine( version );
}

```

Wykonanie komendy nie zwracającej wyników jest najprostsze. Wystarczy wykonać metodę **ExecuteNonQuery** obiektu **SqlCommand**.

```

static void PracaZSerwerem( SqlConnection sqlConn )
{
    SqlCommand sqlCmd = new SqlCommand();

    sqlCmd.Connection = sqlConn;
    sqlCmd.CommandText = "UPDATE T_STUDENT SET Imie='Janusz' WHERE Imie='Jan'";

    sqlCmd.ExecuteNonQuery();
}

```

### 3.8.5 Lokalne struktury danych

Dane przechowywane w tabelach relacyjnych bazy danych przesyłane są do aplikacji w postaci wierszy spełniających kryteria odpowiedniego zapytania. Programista staje więc przed wyborem sposobu, w jaki aplikacja przechowuje te dane do (być może) wielokrotnego użycia.

Jest to jedno z najbardziej złożonych zagadnień związanych z programowaniem aplikacji bazodanowych. Okazuje się, że istnieje wiele możliwości, zaś każda z nich ma swoje zalety i swoje wady. Każda z nich określa pewien **lokalny model danych**, czyli:

- zakres danych, które aplikacja powinna pobierać z serwera na czas jednej sesji pracy z programem
- zbiór struktur danych, których program używa do przechowania danych pobranych z serwera

- sposób w jaki aplikacja poinformuje serwer o zmianach w danych, jakich użytkownik dokonuje podczas sesji pracy z programem
- sposób w jaki aplikacja reaguje na zmiany danych wprowadzane przez wielu użytkowników pracujących jednocześnie, czyli wsparcie dla wielodostępu do danych

Punktem wyjścia do budowania modelu struktur danych po stronie aplikacji powinien być zbiór klas odpowiadających mniej lub bardziej zbiorowi tabel w bazie danych. Jest to podejście naturalne i elastyczne. Na przykład jeśli w bazie danych istnieją tabele T\_UCZELNIA i T\_STUDENT, to po stronie aplikacji odpowiadać im będą klasy CUczelnia i CStudent.

**Zakres danych** Czy podczas startu aplikacja powinna pobrać wszystkie dane z bazy danych serwera, czy też powinna pobierać tyle danych, ile potrzeba do zbudowania bieżącego kontekstu?

- Aplikacja powinna pobierać wszystkie dane z serwera wtedy, kiedy baza danych jest relatywnie mała. Jeżeli z szacunków wynika, że w żadnej tabeli nie będzie więcej niż powiedzmy sto tysięcy rekordów, a tabel jest powiedzmy nie więcej niż pięćdziesiąt, to z powodzeniem można podczas startu aplikacji przeczytać je wszystkie. Mając komplet danych, aplikacja może sama tworzyć proste zestawienia i obliczenia na danych, nie angażując do tego procesu serwera. Aplikacja może również posiadać szybką i jednorodną warstwę pośrednią między danymi zgromadzonymi na serwerze, a danymi udostępnianymi komponentom wizualnym w oknach.
- Jeśli z szacunków wynika, że liczba danych w niektórych tabelach może być większa niż kilkaset tysięcy rekordów, to pobranie ich w całości może być kłopotliwe, z powodu ograniczeń czasowych i pamięciowych. Należy rozważyć model, w którym aplikacja pobiera tylko tyle danych, ile potrzeba do pokazania jakiegoś widoku (okna), bądź zastosować model mieszany (czyli pobierać wszystkie dane z małych tabel i aktualnie potrzebne fragmenty większych tabel).

**Struktury danych** Jakich struktur danych należy użyć, do przechowania danych pobieranych z serwera?

- Jeżeli struktura danych powinna odzwierciedlać relacje między danymi, to można na przykład rozważyć struktury drzewopodobne. Na przykład dla naszej aplikacji możnaby klasy CUczelnia i CStudent zaprojektować tak, aby elementem klasy CUczelnia była kolekcja CStudenci, zaś w samej aplikacji możnaby zadeklarować kolekcję CUczelnie. Taki projekt klas w naturalny sposób odpowiada logicznym powiązaniom istniejącym między danymi, a które wynikają z modelu obiektowego danych. Zainicjowanie komponentów wizualnych wydaje się dość proste, na przykład komponent **TreeView** możnaby zainicjować wyjątkowo łatwo.

Inne relacje między obiektami należałoby zamodelować w podobny sposób, kierując się ogólnymi zasadami modelowania obiektowego.

```
public class CUczelnia
{
    public string Nazwa;
    public string Miejscowosc;
    ...
    public ArrayList CStudenci;
}
public class CStudent
{
    public string Imie;
```



```

        public string Nazwisko;
        ...
    }
    public class CDane
    {
        ...
        public static ArrayList CUczelnie;
    }

```

- Jeżeli struktura danych powinna uwypuklać nie tyle zależności między danymi, co sposób ich składowania, to można rozważyć model, w którym dane w pamięci przechowywane są w kolekcjach, będących dokładnymi kopiami tabel bazodanowych. Logika zależności między danymi musiałaby być wtedy zawarta w pewnym dodatkowym zbiorze funkcji, z konieczności "duplikujących" pewne funkcje serwera bazodanowego. Taka struktura byłaby jednak jednorodna i ułatwiałaby komunikację zwrotną z serwerem.

```

public class CUczelnia
{
    public string Nazwa;
    public string Miejscowosc;
    ...

    public Hashtable Studenci()
    {
        Hashtable hRet = new Hashtable();

        foreach ( CStudent student in CDane.CStudenci.Values )
            if ( student.ID_UCZELNIA == this.ID )
                hRet.Add( student.ID, student );

        return hRet;
    }
}
public class CStudent
{
    public string Imie;
    public string Nazwisko;
    ...
}
public class CDane
{
    ...
    public static Hashtable ArrayList CUczelnie;
    public static Hashtable ArrayList CStudenci;
}

```

**Powiadamanie o zmianach** W jaki sposób aplikacja powinna powiadamiać serwer o zmianach w danych, jakich dokonał użytkownik? Jak zareagować, jeśli użytkownik zmodyfikował na przykład imię Jana Kowalskiego na Janusz?

- Aplikacja może śledzić zmiany w danych dokonywane przez użytkownika w kolejnych widokach. Na przykład jeśli użytkownik ogląda dane w komponencie **List View** w jakimś oknie, to po dokonaniu każdej zmiany aplikacja może zapamiętać ten fakt w jakiejś dodatkowej strukturze (na przykład w **ArrayList** zachować identyfikator zmodyfikowanej danej). Przy próbie zamykania widoku, aplikacja mogłaby zapytać użytkownika o chęć zapamiętania zmian w bazie danych. Do tego celu aplikacja wysłałaby do serwera baz danych odpowiednią ilość poleceń **UPDATE ...**
- Aplikacja może śledzić zmiany w danych dokonywane przez użytkownika w samych obiektach, na przykład obsługując pole **zmodyfikowany** w klasach. Jeżeli użytkownik chce odesłać swoje dane do serwera niezależnie od aktualnego kontekstu, to aplikacja po prostu przegląda wszystkie dane i sprawdza, które zostały zmodyfikowane,

a następnie konstruuje odpowiednie polecenie SQL (**UPDATE ...**) dla każdego zmodyfikowanego obiektu.

- Aplikacja może również zlecić śledzenie zmian danych w specjalnie zaprojektowanych do tego w ADO.NET obiektach, takich jak **DataSet** i **DataGrid**.

**Wielodostęp** Czy aplikacja powinna informować inne aplikacje korzystające z tych samych danych o wprowadzanych zmianach? A może powinna blokować dostęp do danych użytkownikowi A, jeśli w tym samym czasie dane te ogląda użytkownik B? Możliwości jest tu dużo i są w różnym stopniu wspierane przez różne serwery baz danych.

- Najbardziej restrykcyjny scenariusz zakłada, że z danych może korzystać tylko jeden użytkownik w jednej chwili. Aplikacja odpytuje serwer baz danych o już podłączonych użytkowników i jeśli takowi istnieją, to odmawia pracy.
- Bardziej liberalny model zakłada, że wielu użytkowników może korzystać z tych samych danych, jednak użytkownicy w danej chwili mogą oglądać tylko rozłączne dane. Jeżeli aplikacja konstruuje widok, w którym pokazana jest lista studentów, to ten fakt odnotowywany jest w bazie danych i żaden inny użytkownik nie ma dostępu do danych o studentach, dopóki dane te nie zostaną zwolnione.
- Jeszcze liberalniejszy model zakłada, że możliwy jest dostęp do tych samych danych przez wielu użytkowników, przy czym tylko pierwszy z nich może dane modyfikować, a pozostali mogą je tylko oglądać.
- Kolejny model zakłada, że użytkownicy mogą jednocześnie oglądać i modyfikować dane, jednak nie możliwe jest jednoczesne modyfikowanie tych samych danych.
- Jeszcze inny model (dostępny w ADO.NET) umożliwia wielu użytkownikom jednoczesny dostęp do danych. Jeżeli użytkownicy A i B pobiorą pewien zestaw danych, a użytkownik A zmodyfikuje je, to kolejna modyfikacja danych przez użytkownika B powinna zakończyć się stosownym powiadomieniem.
- Najdoskonalszy model wielodostępu zakłada natychmiastowe informowanie wszystkich użytkowników korzystających z danych o modyfikacji danych przez jednego z nich. Model ten może być zrealizowany w różny sposób, jednak najczęściej jest najbardziej pracochłonny i dlatego w praktyce używany jest rzadziej niż któryś z poprzednich.

Powyższy przegląd możliwości, jakie stają przed programistą projektującym lokalny model danych dla aplikacji bazodanowej wskazuje na wiele różnych wariantów, będących efektem składania, jak z klocków, różnych wariantów z kolejnych zagadnień projektowych. Można na przykład wyobrazić sobie aplikację, która do drzewiastych struktur danych pobiera minimalny zbiór danych, potrzebnych do budowania potrzebnego widoku, śledzi dokonywane przez użytkownika zmiany danych w bieżącym widoku i nie pozwala innym użytkownikom pracującym jednocześnie na korzystanie z danych zablokowanych przez bieżącego użytkownika.

Wybór konkretnego lokalnego modelu danych zależy od wielu czynników, wśród których warto wymienić:

- łatwość implementacji - pewne modele są bardziej wymagające, co automatycznie przekłada się na czas, jaki należy poświęcić danej aplikacji
- skalowalność - pewne modele sprawdzają się tylko dla małych danych, inne dobrze radzą sobie z dowolną ilością danych

- wsparcie ze strony mechanizmów serwera lub języka programowania - pewne modele są wspierane bądź przez mechanizmy serwera, bądź przez mechanizmy programowe.

Decyzja o wyborze któregoś z modeli powinna być dobrze przedyskutowana w gronie projektantów i programistów aplikacji, ponieważ zły wybór oznacza możliwą katastrofę, gdyby w połowie prac okazało się, że z jakichś powodów wybrany model należy zmodyfikować lub zmienić.

### 3.8.6 Programowe zakładanie bazy danych

Aplikacja podczas startu powinna umożliwić użytkownikowi utworzenie bazy danych bezpośrednio z poziomu interfejsu użytkownika. Sytuacja, w której użytkownik musiałby do konstrukcji bazy danych używać narzędzi typu **osql** jest niedopuszczalna.

Dysponujemy teraz wystarczającą ilością informacji, aby procedurę zakładania bazy danych, którą przeprowadziliśmy za pomocą **osql**, przenieść do kodu aplikacji. Sposób postępowania jest następujący:

1. Poprosić użytkownika o podanie hasła administratora serwera.
2. Nawiązać połączenie do wskazanego serwera bazy danych do bazy **master** jako administrator serwera.
3. Za pomocą obiektu **SqlCommand** wykonać komendę **CREATE DATABASE ...** aby utworzyć bazę danych.
4. W taki sam sposób zmienić kontekst bazy danych na nowo utworzoną bazę danych poleceniem **USE ...**
5. Wykonać odpowiedni zestaw poleceń **CREATE TABLE ...**

Cała procedura może działać tak, że zestaw poleceń jest wczytywany z pliku - skryptu instalacyjnego, przygotowanego "na boku". Cały zestaw poleceń można wysłać do bazy jako jedną komendę lub w razie potrzeby podzielić go na mniejsze fragmenty, po to by na przykład w trakcie zakładania bazy przez program użytkownikowi pokazać pasek postępu prac.

### 3.8.7 Transakcje

Podczas pracy z bazą danych możliwa jest sytuacja, w której w pewnej chwili aplikacja wykona więcej niż jedną operację na serwerze.

Wyobraźmy sobie na przykład, że aplikacja śledzi zmiany w danych, których dokonuje użytkownik i w pewnej chwili wysyła do serwera sekwencję poleceń SQL, powodujących odświeżenie informacji w bazie danych. Podczas wykonania takiej operacji błąd może pojawić się praktycznie w dowolnej chwili i choć zostanie wychwycony i przekazany aplikacji jako wyjątek, jego skutki mogłyby być bardzo poważne.

Gdyby na przykład aplikacja zdażyła odświeżyć tylko część informacji, zaś błąd uniemożliwiłby odświeżenie całości zmian, to przy następnym uruchomieniu użytkownik mógłby zastać dane swojego programu w postaci kompletnie nie nadającej się do dalszej pracy.

Na szczęście takiego scenariusza można uniknąć, wykorzystując mechanizm tzw. *transakcji*. Transakcja gwarantuje, że serwer albo przyjmie wszystkie polecenia będące jej częścią jako niepodzielną całość, albo wszystkie je odrzuci. Transakcje gwarantują więc niepodzielność wykonania się operacji na serwerze SQL.

W ADO.NET transakcja jest obiektem typu **SqlTransaction**, który inicjowany jest unikalną nazwą, odróżniającą transakcje od siebie. Każde polecenie wykonywane za pomocą obiektu **SqlCommand** może być wykonane jako część rozpoczętej transakcji.

```

string      T_NAME = "TRANSAKCJA";
SqlTransaction sqlT;

try
{
    // rozpocznij transakcję
    sqlT = sqlConn.BeginTransaction( T_NAME );
    ...
    SqlCommand cmd = new SqlCommand( "INSERT/UPDATE/DELETE ...",
                                     sqlConn, sqlT );

    cmd.ExecuteNonQuery();
    ...
    // zatwierdź transakcję
    sqlT.Commit();
}
catch
{
    ...
    // wycofaj transakcję
    sqlT.Rollback( T_NAME );
}

```

### 3.8.8 Typ DataSet

W poprzednich rozdziałach dyskutowaliśmy zagadnienie projektowania lokalnych struktur danych po stronie aplikacji, odpowiadających danym pobranym z serwera baz danych. Okazuje się, że ADO.NET udostępnia typ danych **DataSet**, który dość dobrze nadaje się do przechowywania danych z relacyjnych baz danych. Obiekt typu **DataSet** przechowuje dane pogrupowane w kolekcji obiektów typu **DataTable**. Każdy obiekt **DataTable** odpowiada jednemu zbiorowi danych z serwera SQL. Obiekt **DataTable** ma kolekcję obiektów typu **DataColumn**, której elementy charakteryzują kolejne kolumny danych zgromadzonych w kolekcji elementów typu **DataRow**.

Aby nabrać nieco wprawy w używaniu obiektu **DataSet**, spróbujmy zacząć od prostego przykładu, w którym obiekt ten zostanie zbudowany "od zera", niezależnie od żadnego źródła danych.

```

using System;
using System.Data;

public class CMain
{
    static void WypiszInfoDataSet( DataSet d )
    {
        Console.WriteLine( "DataSet {0} zawiera {1} tabele", d.DataSetName, d.Tables.Count );

        foreach ( DataTable t in d.Tables )
        {
            Console.WriteLine( "Tabela {0} zawiera {1} wiersze", t.TableName, t.Rows.Count );
            foreach ( DataRow r in t.Rows )
            {
                Console.Write( "-> " );
                foreach ( DataColumn c in t.Columns )
                {
                    Console.Write( "{0}={1}, ", c.ColumnName, r[c.ColumnName] );
                }
                Console.WriteLine();
            }
        }
    }

    public static void Main()
    {
        // zbiór danych
        DataSet dataSet = new DataSet( "DataSetOsoby" );

        // tabela
        DataTable dataTable = new DataTable( "Osoby" );
    }
}

```

```

dataSet.Tables.Add( dataTable );

// kolumny tabeli
DataColumn dataColumn1 = new DataColumn( "Imię", typeof(string) );
DataColumn dataColumn2 = new DataColumn( "Nazwisko", typeof(string) );
DataColumn dataColumn3 = new DataColumn( "Data urodzenia", typeof(DateTime) );

dataTable.Columns.AddRange( new DataColumn[] { dataColumn1, dataColumn2, dataColumn3 } );

// wiersze
DataRow row;

row = dataTable.NewRow();
row["Imię"] = "Adam";
row["Nazwisko"] = "Kowalski";
row["Data urodzenia"] = DateTime.Parse( "1992-05-01" );
dataTable.Rows.Add( row );

row = dataTable.NewRow();
row["Imię"] = "Tomasz";
row["Nazwisko"] = "Malinowski";
row["Data urodzenia"] = DateTime.Parse( "1997-07-12" );
dataTable.Rows.Add( row );

WypiszInfoDataSet( dataSet );
}
}

C:\Example>example.exe
DataSet DataSetOsoby zawiera 1 tabelę
Tabela Osoby zawiera 2 wiersze
-> Imię=Adam, Nazwisko=Kowalski, Data urodzenia=1992-05-01 00:00:00,
-> Imię=Tomasz, Nazwisko=Malinowski, Data urodzenia=1997-07-12 00:00:00,

```

Wiedząc już w jaki sposób działa **DataSet**, skorzystajmy z możliwości jaką daje ADO.NET, czyli wypełnienia obiektu **DataSet** danymi z serwera baz danych. Do tego celu użyjemy obiektu typu **SqlDataAdapter**.

```

using System;
using System.Data;
using System.Data.SqlClient;

public class CMain
{
    static void WypiszInfoDataSet( DataSet d )
    {
        ...
    }

    public static string BuildConnectionString( string serverName,
                                              string dbName,
                                              string userName,
                                              string passWd )
    {
        ...
    }

    public static void Main()
    {
        try
        {
            SqlConnection sqlConn = new SqlConnection();

            sqlConn.ConnectionString =
                BuildConnectionString( "(local)", "sqlTEST", "sa", String.Empty );

            sqlConn.Open();

            SqlDataAdapter adapter = new SqlDataAdapter(

```

```

        "SELECT * FROM T_UCZELNIA; SELECT * FROM T_STUDENT", sqlConn );
DataSet dataSet = new DataSet( "Dane" );

// napełnij DataSet przez IDataAdapter
adapter.Fill( dataSet );

WypiszInfoODataSet( dataSet );

// zamknij połączenie
sqlConn.Close();
}
catch ( Exception ex )
{
    Console.WriteLine( ex.Message );
}
}
}

C:\Example>example
DataSet Dane zawiera 2 tabele
Tabela Table zawiera 2 wiersze
-> ID_UCZELNIA=1, Nazwa=Uniwersytet Wrocławski, Miejscowosc=Wrocław,
-> ID_UCZELNIA=2, Nazwa=Uniwersytet Warszawski, Miejscowosc=Warszawa,
Tabela Table1 zawiera 4 wiersze
-> ID_UCZEN=1, ID_UCZELNIA=1, Nazwisko=Kowalski, Imie=Janusz,
-> ID_UCZEN=2, ID_UCZELNIA=1, Nazwisko=Malinowski, Imie=Tomasz,
-> ID_UCZEN=3, ID_UCZELNIA=2, Nazwisko=Nowak, Imie=Adam,
-> ID_UCZEN=4, ID_UCZELNIA=2, Nazwisko=Kamińska, Imie=Barbara,

```

### 3.8.9 Aktywna wymiana danych

Możliwości ADO.NET obejmują również wspomaganie typowych operacji bazodanowych, takich jak tworzenie, modyfikowanie i usuwanie danych na serwerze. Tajemnica tkwi w obiekcie **SqlDataAdapter**, który działa nie tylko jako źródło danych do obiektu **DataSet** (metoda **Fill**), ale potrafi również śledzić zmiany w danych i aktualizować je na serwerze (metoda **Update**).

Powstaje pytanie: skąd **DataAdapter** wie jakich poleceń SQL użyć do modyfikacji czy usuwania danych? Odpowiedź jest prosta: to programista sam zadaje treści tych poleceń, przypisując je pod proprecje **DeleteCommand**, **InsertCommand** i **UpdateCommand** obiektu **DataAdapter**.

W wyjątkowych przypadkach, kiedy operacje aktualizacji dotyczą jednej tylko tabeli, istnieje możliwość automatycznego wygenerowania odpowiednich poleceń przez zainicjowanie obiektu typu **SqlCommandBuilder**. W poniższym przykładzie zmodyfikujemy imię jednego ze studentów.

```

using System;
using System.Data;
using System.Data.SqlClient;

public class CMain
{
    static void WypiszInfoODataSet( DataSet d )
    {
        ...
    }

    public static string BuildConnectionString(string serverName,
                                             string dbName,
                                             string userName,
                                             string passWd)
    {
        ...
    }

    public static void Main()

```

```

{
    try
    {
        SqlConnection sqlConn = new SqlConnection();

        sqlConn.ConnectionString =
            BuildConnectionString( "(local)", "sqlTEST", "sa", String.Empty );

        sqlConn.Open();

        // inicjuj DataSet przy pomocy SqlDataAdapter
        SqlDataAdapter adapter = new SqlDataAdapter(
            "SELECT * FROM T_STUDENT", sqlConn );
        // automatycznie twórz polecenia do wstawiania, modyfikacji i usuwania danych
        new SqlCommandBuilder( adapter );

        DataSet dataSet = new DataSet( "Dane" );
        // napełnij DataSet przez IDataAdapter
        adapter.Fill( dataSet );

        WypiszInfoDataSet( dataSet );

        // modyfikuj dane
        DataRow row = dataSet.Tables[0].Rows[0];
        row.BeginEdit();
        row["Imie"] = "Jan";
        row.EndEdit();
        // aktualizuj na serwerze
        int iModyf = adapter.Update( dataSet );
        Console.WriteLine( "Zmodyfikowano {0} wierszy", iModyf );

        WypiszInfoDataSet( dataSet );

        // zamknij połączenie
        sqlConn.Close();
    }
    catch ( Exception ex )
    {
        Console.WriteLine( ex.Message );
    }
}

```

```

C:\Example>example
DataSet Dane zawiera 1 tabele
Tabela Table zawiera 4 wiersze
-> ID_UCZEN=1, ID_UCZELNIA=1, Nazwisko=Kowalski, Imie=Janusz,
-> ID_UCZEN=2, ID_UCZELNIA=1, Nazwisko=Malinowski, Imie=Tomasz,
-> ID_UCZEN=3, ID_UCZELNIA=2, Nazwisko=Nowak, Imie=Adam,
-> ID_UCZEN=4, ID_UCZELNIA=2, Nazwisko=Kamińska, Imie=Barbara,
Zmodyfikowano 1 wierszy
DataSet Dane zawiera 1 tabele
Tabela Table zawiera 4 wiersze
-> ID_UCZEN=1, ID_UCZELNIA=1, Nazwisko=Kowalski, Imie=Jan,
-> ID_UCZEN=2, ID_UCZELNIA=1, Nazwisko=Malinowski, Imie=Tomasz,
-> ID_UCZEN=3, ID_UCZELNIA=2, Nazwisko=Nowak, Imie=Adam,
-> ID_UCZEN=4, ID_UCZELNIA=2, Nazwisko=Kamińska, Imie=Barbara,

```

### 3.8.10 ADO.NET i XML

Obiekt typu **DataSet** może być składowany w postaci XML i odczytywany z plików XML za pomocą metod **WriteXml**, **WriteXmlSchema**, **ReadXml** i **ReadXmlSchema**.

Poprzedni przykład zmodyfikujmy tak, aby zawartość **DataSet** i schemat XSD pokazać w oknie konsoli (oczywiście można zapisać je do dowolnego strumienia):

```

static void WypiszInfoDataSet( DataSet d )
{
    d.WriteXml( Console.OpenStandardOutput() );
}

```

```
d.WriteXmlSchema( Console.OpenStandardOutput() );
}
```

Zarówno plik XML jak i plik XSD, które będą efektem działania tych metod mogą być przetwarzane wszystkimi dostępnymi do tej pory metodami. Można na przykład zbiór rekordów XML z serwera baz danych wysłać przez sieć jako strumień XML. Można plik z danymi XML odczytać do obiektu **DataSet**, a następnie zapisać na serwerze. Można wreszcie walidować poprawność danych za pomocą schematu XSD.

```
<Dane>
  <Table>
    <ID_UCZEN>1</ID_UCZEN>
    <ID_UCZELNIA>1</ID_UCZELNIA>
    <Nazwisko>Kowalski</Nazwisko>
    <Imie>Jan</Imie>
  </Table>
  <Table>
    <ID_UCZEN>2</ID_UCZEN>
    <ID_UCZELNIA>1</ID_UCZELNIA>
    <Nazwisko>Malinowski</Nazwisko>
    <Imie>Tomasz</Imie>
  </Table>
  <Table>
    <ID_UCZEN>3</ID_UCZEN>
    <ID_UCZELNIA>2</ID_UCZELNIA>
    <Nazwisko>Nowak</Nazwisko>
    <Imie>Adam</Imie>
  </Table>
  <Table>
    <ID_UCZEN>4</ID_UCZEN>
    <ID_UCZELNIA>2</ID_UCZELNIA>
    <Nazwisko>Kamińska</Nazwisko>
    <Imie>Barbara</Imie>
  </Table>
</Dane><?xml version="1.0"?>
<xs:schema id="Dane" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Dane" msdata:IsDataSet="true" msdata:Locale="pl-PL">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Table">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID_UCZEN" type="xs:int" minOccurs="0" />
              <xs:element name="ID_UCZELNIA" type="xs:int" minOccurs="0" />
              <xs:element name="Nazwisko" type="xs:string" minOccurs="0" />
              <xs:element name="Imie" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

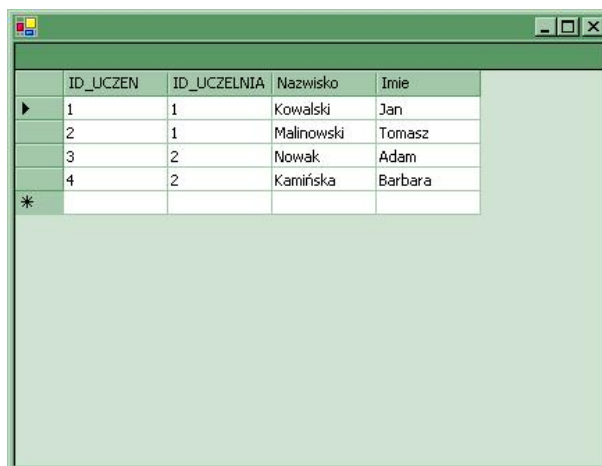
### 3.8.11 Wiązanie danych z komponentami wizualnymi

Możliwości .NET w zakresie przetwarzania danych są, jak widzieliśmy na poprzednich przykładach, duże. Niezwykle łatwo połączyć ze sobą świat serwerów baz danych i świat XML - wystarczą do tego możliwości obiektów **DataSet**.

Okazuje się, że równie łatwo zintegrować dane z obiektami wizualnymi. Służą do tego obiekty **DataBinding**, które opisują sposób wiązania kontrolki z danymi z **DataSet**.

Jednym z najciekawszych komponentów, do tej pory nieomawianym ponieważ jest on ściśle związany z ADO.NET, jest **DataGrid**. DataGrid za pomocą metody **SetDataBinding** można dynamicznie powiązać z zawartością obiektu **DataSet**.





	ID_UCZEN	ID_UCZELNIA	Nazwisko	Imie
▶	1	1	Kowalski	Jan
	2	1	Malinowski	Tomasz
	3	2	Nowak	Adam
	4	2	Kamińska	Barbara
*				

Rysunek 3.13: DataGrid związany z DataSet

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Windows.Forms;

public class CMain : Form
{
    DataGrid dataGrid;

    public static string BuildConnectionString(string serverName,
                                             string dbName,
                                             string userName,
                                             string passWd)
    {
        return String.Format(
            @"Server={0};Database={1};User ID={2};Pwd={3};Connect Timeout=15",
            serverName, dbName, userName, passWd);
    }

    public CMain()
    {
        // inicjuj DataGrid
        dataGrid = new DataGrid();
        dataGrid.Dock = DockStyle.Fill;
        this.Controls.Add( dataGrid );

        try
        {
            SqlConnection sqlConn = new SqlConnection();

            sqlConn.ConnectionString =
                BuildConnectionString( "(local)", "sqlTEST", "sa", String.Empty );

            sqlConn.Open();

            // inicjuj DataSet przy pomocy SqlDataAdapter
            SqlDataAdapter adapter = new SqlDataAdapter(
                "SELECT * FROM T_STUDENT", sqlConn );
            // automatycznie twórz polecenia do wstawiania, modyfikacji i usuwania danych
            new SqlCommandBuilder( adapter );

            DataSet dataSet = new DataSet( "Dane" );
            // napełnij DataSet przez IDataAdapter
            adapter.Fill( dataSet );

            // powiąż DataGrid i DataSet

```

```

        dataGrid.SetDataBinding( dataSet, "Table" );

        // zamknij połączenie
        sqlConn.Close();
    }
    catch ( Exception ex )
    {
        Console.WriteLine( ex.Message );
    }
}

public static void Main()
{
    Application.Run( new CMain() );
}
}

```

Możliwości komponentu **DataGrid** są naprawdę duże i szczegółowy ich opis zdecydowanie wykracza poza ramy tego skryptu. DataGrid może m.in. formatować komórki w zależności od ich zawartości czy poprawnie obsługiwać relacje między danymi z wielu tabel. W przypadku aktualizacji danych w jednej z tabel, DataGrid może reagować na to automatycznie odświeżając swoją zawartość.

## 3.9 Dynamiczne WWW i ASP.NET

### 3.9.1 Dlaczego potrzebujemy dynamicznego WWW

Gwałtowny rozwój sieci i coraz szerszy dostęp do niej sprawiają, że równie szybko rozwijają się technologie sieciowe. Zwykły protokół HTML, choć w wielu przypadkach sprawdza się doskonale, w wielu innych okazuje się niewystarczający. To czego potrzebują programiści, to możliwość tworzenia *dynamicznych* stron Internetowych, przy czym przez *dynamiczny* nie oznacza tu "animowany, żywy", tylko dostosowany do profilu konkretnego użytkownika i umożliwiający komunikację w dwie strony.

Dynamicznie budowana zawartość stron WWW najczęściej związana jest jakoś z dużymi zbiorami informacji. Wyobraźmy sobie sieciową encyklopedię, w której istnieją setki tysięcy możliwych haseł, czy system ewidencji z milionami rekordów. Nietrudno zauważyć, że sam HTML jest zbyt ubogi, aby wspomagać realizację takich przedsięwzięć (chyba, że ktoś wie jak przygotować milion stron w HTML i zbudować dla nich sensowne indeksy).

### 3.9.2 Przegląd technologii dynamicznego WWW

#### 3.9.2.1 Common Gateway Interface

CGI jest jedną z pierwszych technologii, umożliwiających tworzenie dynamicznych stron WWW. Pomysł CGI polega na tym, że serwer Internetowy uruchamia zwykły program wykonywalny (nazywany **skryptem CGI**) i wyniki działania tego programu przekazuje klientowi. Jedną z zalet CGI polega na tym, że skrypty mogą być napisane w dowolnym języku, w którym da się napisać konsolowy program, który zapisuje i odczytuje dane ze standardowych strumieni wejścia/wyjścia.

Najprostszy skrypt CGI, napisany w języku C mógłby wyglądać tak:

```

#include <stdio.h>

int main( int argc, char** argv )
{
    printf( "HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n" );
    printf( "<HTML>\r\n<HEAD>" );
}

```

```

printf( "<TITLE>Witam w CGI</TITLE></HEAD>\r\n" );
printf( "<BODY>Pierwszy skrypt w CGI</BODY>\r\n" );
printf( "</HTML>" );

return 0;
}

```

Z racji prostej idei, CGI jest bardzo popularne. Z CGI związane są jednak duże problemy z wydajnością. Po pierwsze, kiedy skrypt jest wykonywany na serwerze, jest traktowany jak każdy inny proces w systemie, a co za tym idzie musi być inicjowany jak każdy inny proces. Z punktu widzenia systemu operacyjnego, inicjowanie nowego procesu jest dość czasochłonne. Po drugie, każdy nowy skrypt CGI zajmuje pamięć wprost proporcjonalną do swojej wielkości. Przy stu użytkownikach korzystających jednocześnie z serwera jest to jeszcze możliwe. Przy kilku tysiącach jednocześnie uruchomionych procesów zasoby nawet bardzo rozbudowanego serwera najprawdopodobniej ulegną wyczerpaniu i cały system zawali się.

### 3.9.2.2 Internet Server Application Programming Interface

Aby pokonać problemy związane z wydajnością CGI, Microsoft zaprojektował alternatywną technologię dynamicznego WWW, nazwaną Internet Server Application Programming Interface (ISAPI). Główny pomysł polegał na tym, że skrypty ISAPI są bibliotekami (DLL) a nie modułami wykonywalnymi, dzięki czemu kod skryptu ładowany jest do pamięci tylko raz.

Istnieją dwa rodzaje bibliotek ISAPI: **rozszerzenia ISAPI**, które spełniają identyczną funkcję jak skrypty CGI oraz **filtry ISAPI**, które reagują na pewne zdarzenia związane z obsługą stron przez serwer.

Mimo, że technologia ISAPI jest zdecydowanie wydajniejsza od CGI, nie jest pozbawiona wad. Po pierwsze, napisanie poprawnej biblioteki ISAPI wymaga zdecydowanie więcej wiedzy niż napisanie skryptu CGI. Po drugie, jeśli biblioteka ISAPI trafi już na serwer Internetowy, to nie ma łatwego sposobu na zastąpienie jej nowszą wersją, ponieważ system operacyjny zabroni dostępu do biblioteki, która wedle jego rozeznania będzie cały czas używana. Wymiana biblioteki wymaga więc zatrzymania usługi serwera Internetowego na serwerze sieciowym.

### 3.9.2.3 ASP

Następcą ISAPI jest technologia **Active Server Pages**, która, o dziwo, jest zaimplementowana jako rozszerzenie ISAPI. W przypadku ASP nie ma jednak konieczności pisania własnych bibliotek DLL. Zamiast tego tworzy się zwykłą stronę HTML, zaś wewnątrz jej kodu można umieszczać dowolne instrukcje kodu języka skryptowego VBScript. ASP sam dba o interpretowanie kodu VBScript i odsyła do klienta wyniki tej operacji.

Oto przykład bardzo prostej strony ASP:

```

<% Option Explicit %>
<HTML>
<HEAD><TITLE>Witam w ASP</TITLE></HEAD>
<BODY>
<%
Dim n
For n = 1 to 5
    Response.Write( "<FONT size=" & n )
    Response.Write( ">Witam w ASP</FONT><br>" & vbCrLf )
Next
%>
</BODY>
</HTML>

```

Projektując strony ASP można korzystać z całej siły VBScript. Ale to właśnie siła VBScript ta okazuje się być największą słabością ASP - VBScript, jak przystało na język skryptowy,

jest bardzo słabo otypowany. Co więcej - strony są interpretowane dynamicznie. Oba te fakty oznaczają, że bardzo łatwo popełniać błędy w skryptach, które jeśli się pojawią, to wykrywane są dopiero wtedy, kiedy natrafi na nie pierwszy użytkownik.

### 3.9.3 Czym jest ASP.NET

ASP.NET jest naturalnym rozszerzeniem ASP, które integruje technologię ASP z platformą .NET. Dzięki ASP.NET możliwe jest używanie praktycznie dowolnego języka platformy .NET do tworzenia dynamicznej zawartości stron WWW. W chwili obecnej jednak ASP.NET (podobnie jak ASP) działa jedynie na serwerze WWW wbudowanym w systemy Windows począwszy od wersji 2000. Serwer ten to **Microsoft Internet Information Services (IIS)**.

### 3.9.4 Pierwszy przykład w ASP.NET

Najprostszy przykład dynamicznej strony ASP.NET ukazuje jednocześnie, że ASP.NET umożliwia użycie C# jako języka skryptowego<sup>21</sup>. Przy próbie uruchomienia kod strony będzie prekompilowany, a błędy będą statycznie raportowane użytkownikowi.

Druga ważna różnica między ASP a ASP.NET to brak możliwości bezpośredniego odwołania się do zawartości tekstu strony HTML. W ASP bardzo często używa się metody **Response.Write**, aby umieścić tekst wewnątrz strony ASP. W ASP.NET można jedynie odwoływać się do umieszczonych na stronie obiektów WWW lub kontrolek ASP.NET. W poniższym przykładzie odwołujemy się do obiektu WWW typu **SPAN**. Dostęp do obiektu możliwy jest dzięki odwołaniu się do jego nazwy (w przykładzie obiekt typu SPAN nazywa się **Message**).

```
<%@ Page Language="C#" %>
<HTML>
<HEAD><TITLE>Witam w ASP.NET</TITLE></HEAD>
<BODY>

<%
    int    i;
    string s = string.Empty;

    for ( i=1; i<=5; i++ )
    {
        s = s+String.Format(
            "<FONT size={0}>Witam w ASP.NET</FONT><br>", i );
    }
    Message.InnerHtml = s;
%>

<SPAN id="Message" runat=server/>

</BODY>
</HTML>
```

### 3.9.5 Łączenie stron ASP.NET z dowolnym kodem

Jedną z najciekawszych możliwości ASP.NET jest łączenie kodu strony z dowolnym kodem, kompilowanym przy pomocy dowolnego kompilatora platformy .NET.

Napiszmy najpierw kod prostej klasy:

```
using System;

namespace NExample
```

<sup>21</sup>Kod wewnątrz strony może być napisany w C# lub VB.NET. Tylko kod umieszczony w obiektowych bibliotekach DLL, będących dodatkowymi elementami dynamicznej strony, może zawierać skompilowany kod napisany w dowolnym języku .NET

```

{
    public class COsoba
    {
        public string Imie;
        public string Nazwisko;

        public COsoba( string Imie, string Nazwisko )
        {
            this.Imie      = Imie;
            this.Nazwisko = Nazwisko;
        }

        public override string ToString()
        {
            return String.Format( "{0} {1}", Nazwisko, Imie );
        }
    }
}

```

i skompilujemy go do postaci biblioteki:

```
C:\Example>csc.exe /target:library CExample.cs
```

Aby tak utworzona biblioteka mogła być wykorzystana w kodzie strony, plik DLL musi być umieszczony w podkatalogu **bin** katalogu wirtualnego IIS. Katalog taki może być utworzony ręcznie i nie musi mieć ustawionych żadnych specjalnych praw dostępu. Aby móc korzystać z przygotowanej klasy, w kodzie strony należy tylko dodać odwołanie do odpowiedniej przestrzeni nazw.

```

<%@ Page Language="C#" %>
<%@ import Namespace="NExample" %>
<HTML>
<HEAD><TITLE>Witam w ASP.NET</TITLE></HEAD>
<BODY>

<%
COsoba osoba = new COsoba( "Jan", "Kowalski" );
int    i;
string s = string.Empty;

for ( i=1; i<=5; i++ )
{
    s = s+String.Format(
        "<FONT size={0}>{1}</FONT><br>", i, osoba );
}
Message.InnerHtml = s;
%>

<SPAN id="Message" runat=server/>

</BODY>
</HTML>

```

Użytkownik, który ogląda naszą stronę w przeglądarce i próbuje podglądać źródło strony, widzi oczywiście tylko efekt końcowy:

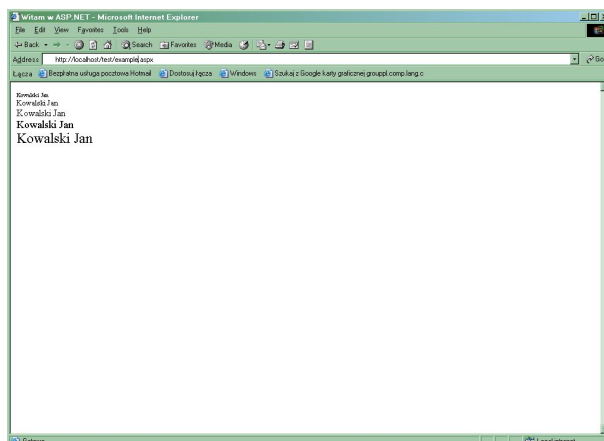
```

<HTML>
<HEAD><TITLE>Witam w ASP.NET</TITLE></HEAD>
<BODY>

<SPAN id="Message">
<FONT size=1>Kowalski Jan</FONT><br>
<FONT size=2>Kowalski Jan</FONT><br>
<FONT size=3>Kowalski Jan</FONT><br>
<FONT size=4>Kowalski Jan</FONT><br>
<FONT size=5>Kowalski Jan</FONT><br></SPAN>

</BODY>
</HTML>

```



Rysunek 3.14: Efekt końcowy w przeglądarce

### 3.9.6 Kontrolki ASP.NET

Kod strony ASP.NET może oczywiście zawierać komponenty WWW, takie, jakie można umieszczać na zwykłych stronach HTML. Oprócz tego można jednak korzystać z całej gamy komponentów właściwych dla ASP.NET. Komponenty te są obiektami pochodzącymi z biblioteki **System.Web.UI.WebControls**. Programista może oczywiście sam tworzyć własne komponenty wizualne, dziedzicząc z klasy **System.Web.UI.UserControl**.

Zbiór zdarzeń, jakie udostępniają komponenty ASP.NET różni się od zdarzeń komponentów Windows.Forms. Jest to dość oczywiste - zachowanie się komponentów w systemie operacyjnym podlega innym regułom niż zachowanie się komponentów w przeglądarce Internetowej.

Przykładowy skrypt tworzy dwa komponenty ASP.NET, etykietę i przycisk. Zauważmy, że definicje komponentów są częścią strony i wyróżniają się jedynie specjalnymi atrybutami. Wewnątrz definicji przycisku określono funkcję reagującą na przyciśnięcie przycisku. Funkcję tę umieszczono wewnątrz specjalnej sekcji strony, oznaczonej tagami **script**.

```
<%@ Page Language="C#" %>

<script runat="server">
    void Przycisk_Click(Object sender, EventArgs e) {
        Label1.Text = "Witam w ASP.NET";
    }
</script>

<html>
<head>
</head>
<body>
    <form runat="server">
        <center>
            <asp:Label id="Label1" runat="server"
                Width="193px">Etykieta</asp:Label>

            <br />
            <asp:Button id="Przycisk" onclick="Przycisk_Click"
                runat="server" Text="Twórz dane osobowe"
                Width="192px"></asp:Button>
        </center>
    </form>
</body>
</html>
```

Wśród komponentów ASP.NET znajduje się m.in. kilka rodzajów siatek, przycisków, kalendarze. Wśród nich jest na przykład **DataGrid**, który może być inicjowany w standardowy

sposób (patrz rozdział 3.8.11, strona 256).

### 3.9.7 Inne przykłady ASP.NET

#### 3.9.7.1 Identyfikacja klienta

Wewnątrz kodu strony można odwoływać się do wszystkich składowych obiektu **Page**, identyfikującego bieżącą stronę. Wśród nich przydatne są właściwości **Request** określająca parametry strony inicjującej połączenie oraz **Response** określająca parametry odpowiedzi serwera.

Właściwość **Request** może być wykorzystana na przykład do identyfikacji systemu operacyjnego i przeglądarki, której używa klient.

```
<%@ Page Language="C#" %>
<HTML>
<HEAD><TITLE>Witam w ASP.NET</TITLE></HEAD>
<BODY>

<%
Message.InnerHtml = String.Format( "Browser: {0}, platform {1}",
    Request.Browser.Browser,
    Request.Browser.Platform );
%>

<SPAN id="Message" runat=server/>

</BODY>
</HTML>
```

#### 3.9.7.2 Licznik odwiedzin strony

Przygotowanie licznika odwiedzin strony jest jednym z podstawowych zadań dynamicznego WWW. W ASP.NET sytuacja jest o tyle wygodna, że w kodzie skryptów używamy dobrze znanych bibliotek .NET.

Wartość licznika odwiedzin będzie zapisywana w pliku `counter.txt` w folderze strony WWW. Jak jednak zabezpieczyć się przed zwiększaniem tego licznika przy częstym odświeżaniu strony przez Internautę? Możemy skorzystać z *ciasteczek*, czyli informacji umieszczanych po stronie klienta, które pozwalają identyfikować go przy kolejnych odwiedzinach naszej strony. W poniższym przykładzie ciasteczko zostanie unieważnione po 2 minutach od pierwszego wejścia na stronę WWW.

```
<%@ Page Language="C#" %>
<%@ import Namespace="System.IO" %>
<%@ import Namespace="System.Drawing" %>
<%@ import Namespace="System.Drawing.Imaging" %>
<%@ import Namespace="System.Drawing.Drawing2D" %>

<script runat="server" language="C#">

string getCounter()
{
    string CookieID = "OldVisitor";

    // czytaj wartosc licznika
    string FilePath = Server.MapPath("\\") + "counter.txt";
    StreamReader sr = File.OpenText(FilePath);
    string counter = sr.ReadLine().ToString();
    sr.Close();

    // ciasteczko - czy to stary gość?
    HttpCookie Cookie;
    Cookie = Request.Cookies[CookieID];
```

```
// tak, inkrementuj licznik
if(Cookie==null)
{
    int counterInt = Convert.ToInt32(counter);
    counterInt++;
    counter = Convert.ToString(counterInt);

    FileStream fs = new FileStream(FilePath, FileMode.Open, FileAccess.Write);
    StreamWriter sw = new StreamWriter(fs);
    sw.WriteLine(counter);
    sw.Close();
    fs.Close();

    Cookie = new HttpCookie( CookieID, "true" );
    Cookie.Expires = DateTime.Now.AddSeconds( 120 );
    Response.AppendCookie( Cookie );
}

return counter;
}

</script>

<HTML>
<HEAD><TITLE>Przykładowy licznik w ASP.NET</TITLE></HEAD>
<BODY>

<% MyCounter.InnerHtml = getCounter(); %>

<center>
Gość numer: <SPAN id="MyCounter" runat=server/>
<asp:Image id="MyCounterImage"/>
</center>

</BODY>
</HTML>
```

Spróbujmy rozwinąć trochę ten przykład, tak aby licznik odwiedzin był umieszczony na stronie w postaci nie tekstu, ale dynamicznie budowanego obrazka.

W pierwszej chwili może wydać się to trudne, ale na szczęście w kodzie strony istnieje możliwość zapisania wyglądu strony w postaci strumienia danych.

```
// Plik: c.aspx
<%@ Page Language="C#" %>
<%@ import Namespace="System.IO" %>
<%@ import Namespace="System.Drawing" %>
<%@ import Namespace="System.Drawing.Imaging" %>
<%@ import Namespace="System.Drawing.Drawing2D" %>

<script runat="server" language="C#">

string getCounter()
{
    // ... to samo co poprzednio
}

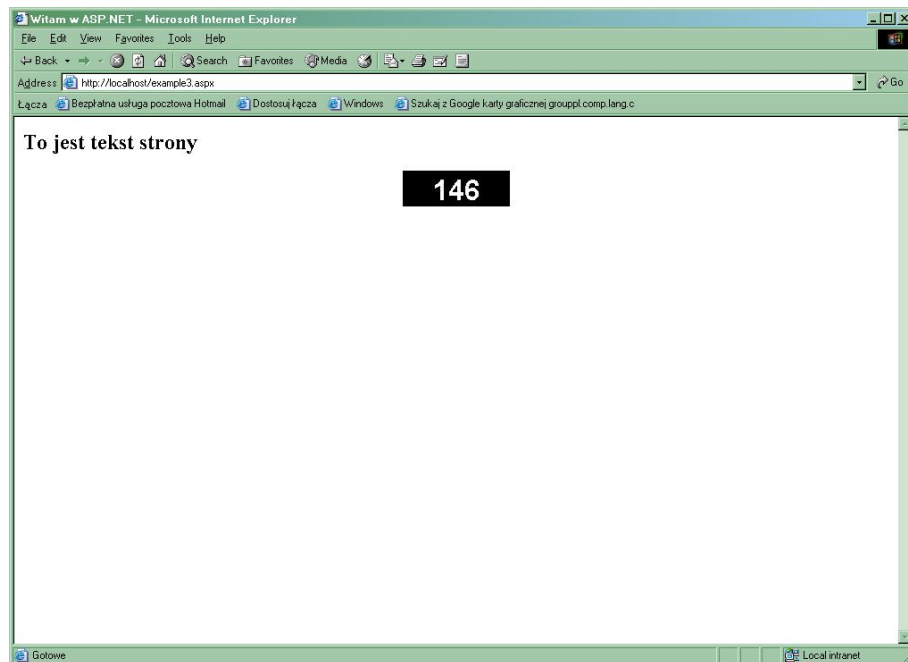
void drawCounter()
{
    int height = 40;
    int width = 120;

    Bitmap bmp = new Bitmap( width, height );
    Graphics g = Graphics.FromImage(bmp);

    string currentCounter = getCounter();
    Font counterFont = new Font( "Arial", 24, FontStyle.Bold );
    SizeF sizeF = g.MeasureString( currentCounter, counterFont );

    g.FillRectangle( Brushes.Black, 0, 0, width, height );
```





Rysunek 3.15: Dynamiczny licznik odwiedzin strony w ASP.NET

```

g.DrawString( currentCounter, counterFont,
    Brushes.White, (bmp.Width-sizeF.Width)/2, 3 );

bmp.Save(Response.OutputStream, ImageFormat.Jpeg);

g.Dispose();
bmp.Dispose();
}

private void Page_Load(object sender, System.EventArgs e)
{
    drawCounter();
}
</script>

```

Tak przygotowana strona pokazuje wartość licznika w postaci obrazka. Problem polega tylko na tym, że zapisanie strumienia danych do zawartości strony (**bmp.Save(Response.OutputStream,...)**) powoduje, że strona nie będzie zawierać żadnych innych obiektów. To nie szkodzi! Z licznika skorzystamy w kodzie każdej kolejnej strony, dynamicznie dołączając go jako obrazek:

```

<%@ Page Language="C#" %>
<HTML>
<HEAD><TITLE>Witam w ASP.NET</TITLE></HEAD>
<BODY>

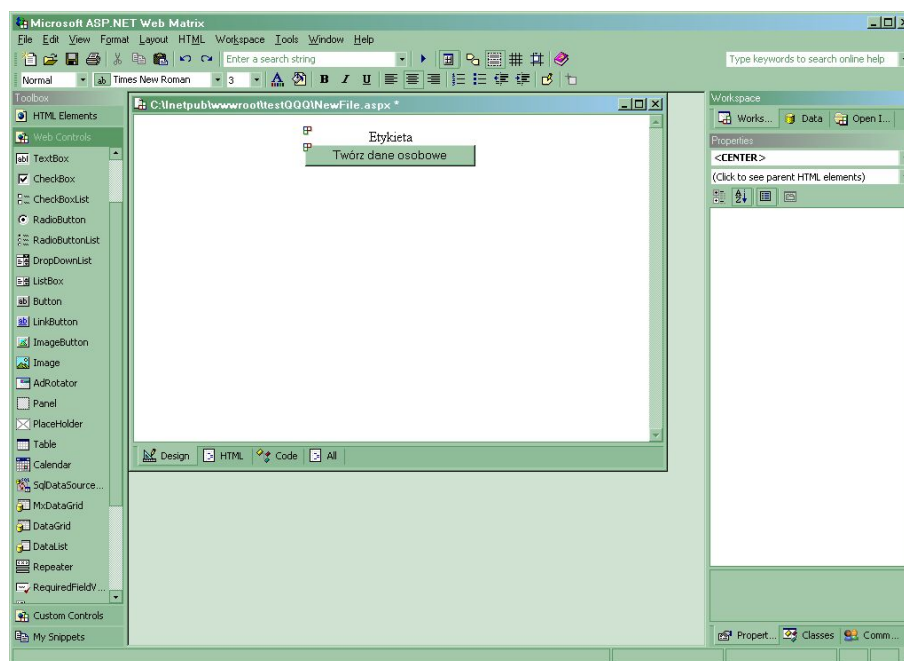
<h2>To jest tekst strony</h2>

<center>

</center>

</BODY>
</HTML>

```



Rysunek 3.16: Microsoft ASP.NET WebMatrix w akcji

### 3.9.8 Narzędzia wspomagające projektowanie stron ASP.NET

#### 3.9.8.1 Visual Studio .NET

Visual Studio .NET znakomicie radzi sobie ze wspomaganiem projektowania stron ASP.NET. Z poziomu środowiska, tworząc nowy projekt, można nawet utworzyć katalog wirtualny na serwerze IIS.

Podczas pracy nad projektem Visual Studio .NET stosuje nieco inną konwencję od przedstawionej w dotychczasowych przykładach - warstwa prezentacyjna (układ komponentów) znajduje się w osobnym pliku niż kod obsługi zdarzeń komponentów.

#### 3.9.8.2 ASP.NET WebMatrix

Microsoft ASP.NET WebMatrix jest darmowym narzędziem, wspierającym projektowanie stron ASP.NET. WebMatrix ma wizualny edytor stron, edytor kodu, palety narzędziowe. Edytor pozwala na przypisywanie zdarzeń komponentom.

WebMatrix można pobrać ze strony <http://www.asp.net>.

## 3.10 Inne języki platformy .NET

### 3.10.1 VB.NET

Visual Basic .NET jest nową odsłoną znanego i popularnego języka Visual Basic. W nowej wersji język został znacznie unowocześniony i dostosowany do możliwości platformy .NET. Kompilator VB.NET jest częścią frameworka i uruchamiany jest poleceniem **vbc.exe**.

Program w VB.NET oprócz klas może również składać się z tzw. *modułów*, które są po prostu zbiornikami kodu nie przywiązanego do żadnej klasy. Funkcje publiczne z modułów są

dostępne z każdego miejsca w kodzie, podobnie jak funkcje statyczne w klasach. Metoda **Main** może znajdować się w jakimś module, zamiast w klasie.

VB.NET, w przeciwieństwie do C#, nie rozróżnia dużych i małych liter w kodzie. Ma również znacznie liberalniejszy system typów. W VB.NET można na przykład:

- używać niezainicjowanych w kodzie zmiennych
- dokonywać niejawnych konwersji między wartościami różnych typów

Możliwości wyrazowe VB.NET odpowiadają możliwościom C#, jednak "basicopodobna" składnia jest miejscami trochę zbyt przegadana. Na przykład ograniczniki strukturalne zawsze są parami wyrażen:

```
Public Sub Metoda
    ...
End Sub

Public Function Funkcja( i As Integer, j As String ) As String
    ...
End Function
```

Z powodów historycznych zachowano m.in.

- klasyczną basicową pętlę **For**, która jest zdecydowanie słabsza od **for** z C#
- słowo kluczowe **Me**, będące odpowiednikiem **this**
- słowo kluczowe **Nothing**, będące odpowiednikiem **null**
- konieczność używania symbolu \_ do oznaczenia przeniesienia długiej linii kodu do kolejnej linii (bez tego znaku koniec linii oznacza koniec instrukcji)

W porównaniu z wersją 6.0 Visual Basic, w VB.NET znacznie poprawiono model obiektowy, pozwalając na projektowanie interfejsów i dziedziczenie takie jak w C#. Zmienne mogą być deklarowane w dowolnym miejscu kodu, podobnie jak w C#.

### 3.10.1.1 Przykładowa aplikacja w VB.NET

Pierwszy przykładowy program to basicowa wersja programu ze strony 227. Oprócz oczywistych różnic w składni, warto zwrócić uwagę na zupełnie inny sposób dodawania funkcji obsługi zdarzeń niż w C#.

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Module MainModule

    Sub Main
        Dim f As New MainForm
        f.ShowDialog()
    End Sub

    Public Class MainForm
        Inherits System.Windows.Forms.Form

        Dim timer As Timer

        Public Sub New
            MyBase.New()
```

```

timer          = new Timer
timer.Interval = 50
AddHandler timer.Tick, AddressOf Timer_Tick

timer.Start

SetStyle(ControlStyles.UserPaint, True)
SetStyle(ControlStyles.AllPaintingInWmPaint, True)
SetStyle(ControlStyles.DoubleBuffer, True)
End Sub

Sub Timer_Tick( sender As Object, e As EventArgs )
    Me.Invalidate
End Sub

Protected Overrides Sub OnPaint( e As PaintEventArgs )
    Dim g as Graphics = e.Graphics
    Dim f as Font      = new Font( "LED", 48 )

    Dim sf as StringFormat = new StringFormat()

    sf.Alignment        = StringAlignment.Center
    sf.LineAlignment    = StringAlignment.Center

    g.Clear( SystemColors.Control )
    g.DrawString( DateTime.Now.ToLongTimeString(), f, Brushes.Black, _
        Me.Width / 2, Me.Height / 2, sf )

End Sub

End Class

End Module

```

### 3.10.1.2 Dynamiczne wiązanie

Ogromna przepaść dzieli VB.NET od jego poprzednika, VB 6.0. Mimo to mało elegancka składnia sprawia, że mając do wyboru VB.NET i C#, zdecydowanie bardziej warto wybrać C#.

Okazuje się jednak, że istnieją zastosowania, w których VB.NET sprawdza się o wiele lepiej niż C#. Chodzi o współpracę z bibliotekami systemowymi w starym modelu COM. Program w VB.NET może zażądać utworzenia obiektu COM i wołać jego metody, mimo że ich prototypy *nie są znane* w trakcie kompilacji! Jest to możliwe, ponieważ VB.NET używa *poźnego* wiązania wywoływanych metod z odpowiadającym im kodem. W C#, z powodu silnego otypowania kodu, taka konstrukcja nie jest możliwa i dlatego korzystanie z obiektów COM jest trudniejsze.

Jeżeli więc aplikacja .NETowa powinna komunikować się z obiektami COM, to odpowiedni kod najwygodniej jest napisać w VB.NET, zaś całą resztę - w jakimś innym języku.

Zobaczmy prosty przykład tzw. *automatyzacji* obiektów Microsoft Office. Przykładowa aplikacja utworzy instancję obiektu Microsoft Word i otworzy w niej nowy dokument.

```

Imports System
Imports System.Drawing
Imports System.Windows.Forms

Imports Microsoft.VisualBasic

Module MainModule

    Sub Main
        Dim f As New CMainForm
        f.ShowDialog()
    End Sub

    Public Class CMainForm

```

```

Inherits System.Windows.Forms.Form

Dim b As Button

Public Sub New
    MyBase.New()

    b = new Button
    b.Text = "Utwórz obiekt MS Word"
    b.Size = new Size( 150, 25 )
    AddHandler b.Click, AddressOf b_Click

    Controls.Add( b )
End Sub

Sub b_Click( sender As Object, e As EventArgs )
    Dim o as Object

    ' twórz obiekt COM
    o = CreateObject( "Word.Application" )
    o.Visible = True
    o.Documents.Add
    o = Nothing

End Sub

End Class

End Module

```

Analogiczna operacja w C# jest nieco bardziej skomplikowana i wymaga silnego wsparcia ze strony mechanizmu refleksji.

```

...
try
{
    Type t = Type.GetTypeFromProgID( "Word.Application" );
    object w = Activator.CreateInstance( t );

    // w.Visible = true
    t.InvokeMember( "Visible", BindingFlags.SetProperty,
        null, w, new Object[] { true } );

    // w.Documents...
    object docs = t.InvokeMember( "Documents", BindingFlags.GetProperty,
        null, w, null );

    // ...Add
    t.InvokeMember( "Add", BindingFlags.InvokeMethod,
        null, docs, null );

    w = null;
}
catch( TypeLoadException ex )
{
    ...
}

```

Jak to się więc dzieje, że VB.NET potrafi wykonać kod, w którym występują odwołania do nieznanych w czasie kompilacji metod i właściwości? Dlaczego kompilator przyjmuje kod

```

o = CreateObject( "Word.Application" )
o.Visible = True
o.Documents.Add

```

skoro `o` jest typu `object`? Odpowiedź na to pytanie tkwi w kodzie ILowym powyższego modułu VB.NET. Proponuję samodzielnie zdekompilować ten moduł przy pomocy **ildasm.exe** i przekonać się jakich mechanizmów używa VB.NET do obsługi dynamicznego wiązania konkretnych metod obiektu z odwołaniami do nich w kodzie aplikacji.

### 3.10.2 ILAsm

CIL jest językiem pośrednim, do którego kompilowane są wszystkie języki platformy .NET. Wśród nich szczególną pozycję ma ILAsm (*Intermediate Language Assembler*), czyli język niskiego poziomu bezpośrednio tłumaczący się do kodu pośredniego CIL.

Oczywiście znajomość języka ILAsm nie jest niezbędna aby pisać programy dla środowiska .NET. Czasami jednak warto zdekompilować program (rozdział 3.2.19) C#owy i zobaczyć jak wygląda kod jakiegoś interesującego fragmentu.

Znajomość języka pośredniego jest oczywiście niezbędna z punktu widzenia twórcy nowego języka czy kompilatora dla platformy .NET. Kod ILAsmowy może korzystać z wyjątków i wołać funkcje z bibliotek .NET. Mimo, że znacznie ułatwia tworzenie kodu wynikowego dla języków obiektowych, nadaje się równie dobrze wszystkim typów języków. W kolejnym rozdziale zobaczymy przykłady kodu produkowanego przez kompilator SML.NET.

#### 3.10.2.1 Informacje ogólne

Z punktu widzenia kodu ILAsmowego, najistotniejszym elementem środowiska jest stos. Stos służy do przekazywania parametrów do funkcji i zbierania wyników funkcji. Na stosie można umieszczać obiekty typów prostych i referencyjnych (wtedy na stosie znajduje się referencja do obiektu, zaś wartość obiektu znajduje się na sterpie). Programista może przekazywać wartości obiektów między stosem a zmiennymi lokalnymi kodu. Do przekazania wartości na stos służą instrukcje w postaci **ld...**, zaś do pobrania wartości ze stosu i umieszczenia ich w zmiennych lokalnych instrukcje postaci **st...**.

Wykonanie funkcji w ILAsm składa się z trzech kroków:

1. Położenie na stosie parametrów wejściowych funkcji
2. Wywołanie funkcji
3. Zdjęcie ze stosu wyników funkcji

Pierwszy i ostatni krok są opcjonalne i oczywiście zależą od postaci funkcji.

IL jako język niskopoziomowy ma dość duże możliwości. Ma m.in. instrukcje do obsługi tablic, obiektów oraz wyjątków. Potrafi obsługiwać ogonowe wywołania funkcji, delegatów i zdarzenia.

#### 3.10.2.2 Pierwszy program w ILAsm

Najprostszy program w ILAsm po prostu wypisze tekst powitalny na ekranie.

```
.assembly extern mscorlib {}
.assembly Example{}
.class public CExample extends [mscorlib]System.Object
{
    .method public static void MyAppStart() il managed
    {
        .entrypoint
        .maxstack 8

        // umieść na stosie napis
        ldstr "Pierwszy program w ILAsm..."

        // wołaj funkcję Console.WriteLine( string )
        call void [mscorlib] System.Console::WriteLine
            (class System.String)

        ret
    }
}
```

Dyrektywa **.assembly extern** informuje kompilator o konieczności importu informacji o funkcjach z zewnętrznej biblioteki. W tym przypadku chodzi o bibliotekę **mscorlib**, która jest rdzeniem całego środowiska .NET i trudno wyobrazić sobie program niekorzystający z tej biblioteki.

Dyrektywa **.assembly** definiuje nowy moduł do kompilacji, w tym przypadku chodzi o program, który właśnie piszemy.

Dyrektywa **.class** definiuje nowy typ, jednak jest ona opcjonalna - ILAsm może z powodzeniem tworzyć kod nieobiektowy. W naszym przykładzie nowym typem jest typ referencyjny (klasa), dziedziczący z klasy **System.Object**.

W przeciwieństwie do C# czy VB.NET, część definicji typu w ILAsm może być umieszczona w pewnym miejscu kodu, a inna część w innym miejscu kodu. Definicja jednego i tego samego typu może nawet rozciągać się na kilka plików z kodem źródłowym.

Struktury, czyli typy proste, wyprowadza się z klasy **System.ValueType** zamiast z **System.Object**.

Dyrektywa **.method** rozpoczyna definicję kodu nowej metody. Specjalne oznaczenie **il managed** oznacza, że kod metody napisany jest w ILAsm i powinien podlegać wszelkim regułom narzucanym przez platformę .NET. Kod natywny można specyfikować za pomocą oznaczenia **native unmanaged**.

Dyrektywa **.entrypoint** określa miejsce startu aplikacji. Co ciekawe, odpowiednia funkcja może mieć dowolną nazwę, niekoniecznie **Main**.

Dyrektywa **.maxstack** określa maksymalną głębokość stosu metody.

Instrukcja **ldstr** powoduje umieszczenie na stosie napisu przekazanego jako parametr. Instrukcja **call** powoduje wywołanie funkcji ze wskazanej biblioteki i wskazanej klasy. Funkcja ta szuka na stosie odpowiednich parametrów, zdejmując je, po czym wykonuje swój kod.

Powyższy przykładowy kod może być skompilowany i uruchomiony:

```
C:\Example>ilasm example.il

Microsoft (R) .NET Framework IL Assembler. Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Assembling 'example.il' , no listing file, to EXE --> 'example.EXE'
Source file is ANSI

Assembled method CExample::MyAppStart
Creating PE file

Emitting members:
Global
Class 1 Methods: 1;
Writing PE file
Operation completed successfully

C:\Example>example
Pierwszy program w ILAsm...
```

### 3.10.2.3 Stałe i zmienne

Stałe numeryczne mogą być ładowane bezpośrednio na stos za pomocą instrukcji:

- stałe całkowitoliczbowe - **ldc.i4 (int32)**, **ldc.i4.s (int8)** (.s na końcu instrukcji zawsze oznacza "krótką" wersję instrukcji, czyli taką, która przyjmuje parametr o mniejszym zakresie danych niż "pełna" instrukcja) oraz **ldc.i8 (int64)**
- stałe całkowitoliczbowe między 0 a 8 - **ldc.i4.(0-8)**
- stałe zmiennoprzecinkowe - **ldc.r4.(float32)** oraz **ldc.r8.(float64)**

Zmienne są deklarowane za pomocą dyrektywy **.locals**. Opcjonalne słowo **init** oznacza, że zmienne mają być zainicjowane domyślnymi wartościami. Jeśli programista zdecyduje inaczej, kod będzie przez środowisko uruchomieniowe uważany za niebezpieczny.

Zmienne są numerowane kolejnymi liczbami całkowitymi i w jednej metodzie może ich być maksymalnie 65536, czyli  $2^{16}$ . Zmienne mogą być deklarowane w kilku miejscach w kodzie metody, co więcej, jeśli pewna zmienna o numerze  $k$  przestaje być potrzebna, można w jej miejsce zadeklarować nową zmienną o tym samym numerze i tym samym typie, ale o innej nazwie.

Numery zmiennych odgrywają główną rolę w kodzie ILAsmowym, bowiem przesyłanie danych ze stosu do zmiennej i ze zmiennej na stos odbywa się za pomocą instrukcji **ldloc (int32)** oraz **stloc (int32)**, które jako parametr przyjmują właśnie nazwę zmiennej.

```
.assembly extern mscorlib {}
.assembly Example{}
.class public CExample extends [mscorlib]System.Object
{
    .method public static void MyAppStart() il managed
    {
        .entrypoint
        .maxstack 8

        // int n;
        .locals init ([0] int32 n)

        // n = 100;
        ldc.i4 100
        stloc 0

        // Console.WriteLine( n );
        ldloc 0
        call void [mscorlib]System.Console::WriteLine(int32)

        ret
    }
}
```

### 3.10.2.4 Instrukcje arytmetyczne

Wśród instrukcji arytmetycznych szczególną rolę odgrywają instrukcje umożliwiające manipulację danymi na stosie:

**dup** powoduje utworzenie na stosie dodatkowej kopii już istniejącego tam obiektu

**pop** powoduje usunięcie wartości z wierzchu stosu

”Zwykle” instrukcje arytmetyczne wymagają odpowiedniej liczby wartości na stosie i zwracają wartość na stos.

**add** suma dwóch argumentów

**sub** różnica

**mul** iloczyn

**div** iloraz

**rem** reszta z dzielenia

**neg** negacja parametru z wierzchołka stosu (zmiana znaku liczby)

Operacje bitowe:



**and** iloczyn bitowy

**or** suma bitowa

**xor** różnica symetryczna

**not** negacja bitowa

**shl** przesunięcie bitowe w lewo (wymaga dwóch wartości na stosie: pierwsza określa o ile bitów przesunąć w lewo drugą)

**shr** przesunięcie bitowe w prawo

Operatory konwersji pobierają wartość ze stosu, konwertują do wskazanego typu i odkładają wynik na stos

**conv.i1** Konwertuj do **int8**

**conv.u1** Konwertuj do **unsigned int8**

**conv.i2** Konwertuj do **int16**

**conv.u2** Konwertuj do **unsigned int16**

**conv.i4** Konwertuj do **int32**

**conv.u4** Konwertuj do **unsigned int32**

**conv.i8** Konwertuj do **int64**

**conv.u8** Konwertuj do **unsigned int64**

**conv.r4** Konwertuj do **float32**

**conv.r8** Konwertuj do **float64**

### 3.10.2.5 Operacje warunkowe, skoki

Stosowe warunki logiczne sprawdzają czy zachodzi odpowiednia relacja między kolejnymi wartościami ze stosu i odkłada na stos wartość 1 jeśli warunek jest spełniony lub 0 jeśli warunek nie jest spełniony:

**ceq** Sprawdza czy dwie kolejne wartości na stosie są równe

**cgt** Sprawdza czy pierwsza wartość na stosie jest większa od drugiej.

**clt** Sprawdza czy pierwsza wartość na stosie jest mniejsza od drugiej.

Instrukcje skoku mają zwykle postać (**instrukcja**) (**numer**), gdzie (**numer**) oznacza przesunięcie (wyrażone w bajtach) instrukcji, do której należy wykonać skok. Na przykład **br 5** oznacza bezwarunkowy skok do instrukcji leżącej bajtów dalej niż bieżąca instrukcja.

Można jednak w każdym miejscu, gdzie w kodzie wynikowym pojawia się przesunięcie, umieścić etykietę, która oprócz tego powinna znajdować się gdzieś w kodzie i wyznaczać przez to pozycję jakiejś instrukcji. Podczas kompilacji odwołania do etykiet są przez kompilator zamieniane na wartości odpowiednich przesunięć, na przykład:

```
Etykieta1:
...
...
br Etykieta1
```

Unarne instrukcje warunkowe (wymagają jednego parametru na stosie):

**brfalse (int32)** Skok jeśli wartość na stosie jest równa 0

**brtrue (int32)** Skok jeśli wartość na stosie jest różna od 0

Binarne instrukcje warunkowe (wymagają dwóch parametrów na stosie):

**beq (int32)** Skok jeśli równe

**bne (int32)** Skok jeśli nierówne

**bge (int32)** Skok jeśli większe lub równe

**bgt (int32)** Skok jeśli większe

**ble (int32)** Skok jeśli mniejsze lub równe

**blt (int32)** Skok jeśli mniejsze

### 3.10.2.6 Metody i parametry

Parametry wewnątrz metod mogą być odczytywane i zapisywane za pomocą instrukcji **ldarg** i **starg**. W przykładowej aplikacji umieścimy funkcję, która oblicza kwadrat liczby przekazanej jako parametr.

```
.assembly extern mscorlib {}
.assembly Example{}
.class public CExample extends [mscorlib]System.Object
{
    .method static int32 Kwadrat( int32 n ) il managed
    {
        ldarg 0 // ładuj parametr na stos
        dup     // umieść kolejną kopię na stosie
        mul     // mnoż przez siebie
        ret     // zwróć wynik
    }

    .method public static void MyAppStart() il managed
    {
        .entrypoint
        .maxstack 8

        // int n;
        .locals init ([0] int32 n)

        // n = Kwadrat( 5 );
        ldc.i4 5
        call int32 CExample::Kwadrat(int32)
        stloc 0

        // Console.WriteLine( n )
        ldloc 0
        call void [mscorlib]System.Console::WriteLine(int32)

        ret
    }
}
```

### 3.10.2.7 Obiekty, pola, metody

IL udostępnia również mechanizmy do operacji na obiektach. Dyrektywy **.class**, **.method** i **field** pozwalają na deklarowanie odpowiednich rodzajów elementów składowych klas.

Ważniejsze instrukcje do operacji na obiektach:

**ldnull** Ładuje na stos referencję **null**

**newobj (token)** Allokuję pamięć dla nowej instancji typu referencyjnego. Wymaga na stosie odpowiedniej liczby parametrów dla konstruktora i odkłada na stos referencję do nowo utworzonego obiektu.

**ldfld (token)** Zdejmuje ze stosu referencję do obiektu i umieszcza na stosie wartość wskazanego pola.

**ldsfd (token)** Jak wyżej, tylko dotyczy pola statycznego.

**stfld (token)** Zdejmuje ze stosu wartość pola i referencję do obiektu i umieszcza wartość w odpowiednim polu obiektu.

**stsfld (token)** Jak wyżej, tylko dotyczy pola statycznego.

**castclass (token)** Zdejmuje ze stosu referencję do obiektu i rzutuje do wskazanego typu.

**box (token)** Zdejmuje ze stosu wartość typu prostego i opakowuje, zapisując na stos referencję do nowo utworzonego obiektu.

**unbox (token)** Odpakowuje wartość obiektu z zadanej referencji.

Zobaczmy przykład opakowywania. Na stos załadujemy wartość 1, opakujemy ją do obiektu i wywołamy wirtualną metodę **ToString** dla tak skonstruowanego obiektu. Wynik pokażemy w oknie konsoli.

```
.assembly extern mscorlib {}
.assembly Example{}
.class public CExample extends [mscorlib]System.Object
{
    .method public static void MyAppStart() il managed
    {
        .entrypoint
        .maxstack 8

        ldc.i4    1
        box      [mscorlib]System.Int32
        callvirt instance string [mscorlib]System.Object::ToString()
        call void [mscorlib]System.Console::WriteLine(string)

        ret
    }
}
```

### 3.10.2.8 Polimorfizm

Metody niestatyczne zdefiniowane w klasie mogą być zdefiniowane jako wirtualne lub nie za pomocą dyrektywy **virtual**. W chwiliwołania metody wirtualnej ze specjalnej struktury zwanej *tablicą metod wirtualnych* pobierana jest informacja o łańcuchu przeciążanych funkcji, z których wybierana jest odpowiednia. Oznacza to, że wiązanie nazwy metody z konkretną implementacją odbywa się tuż przed wykonaniem metody, a nie w czasie kompilacji.

Zobaczmy następujący przykład:

```

.assembly CExample{}

.class public A
{
    .method public specialname void .ctor() { ret }
    .method public void Foo()
    {
        ldstr "A::Foo"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
    .method public virtual void Bar()
    {
        ldstr "A::Bar"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
    .method public virtual void Baz()
    {
        ldstr "A::Baz"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
}

.class public B extends A
{
    .method public specialname void .ctor() { ret }
    .method public void Foo()
    {
        ldstr "B::Foo"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
    .method public virtual void Bar()
    {
        ldstr "B::Bar"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
    .method public virtual newslot void Baz()
    {
        ldstr "B::Baz"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
}

.method public static void Exec()
{
    .entrypoint
    newobj instance void B::.ctor() // new B()
    castclass class A              // rzutuj na A

    dup                          // 3 kopie na stosie
    dup                          //

    callvirt instance void A::Foo()
    callvirt instance void A::Bar()
    callvirt instance void A::Baz()
    ret
}

```

W klasie **A** zdefiniowano trzy metody, z czego dwie są metodami wirtualnymi. W klasie **B**, dziedziczącej z **A**, zdefiniowano trzy metody o takich samych sygnaturach jak metody z klasy bazowej.

- metoda **Foo** jest w obu klasach zdefiniowana jako niewirtualna

- metoda **Bar** jest w obu klasach wirtualna
- metoda **Baz** jest w obu klasach wirtualna, przy czym w klasie **B** jest opatrzona dyrektywą **newslot** (nowa pozycja w tablicy metod wirtualnych)

Efekt działania tego programu jest zgodny z oczekiwaniami: tylko kod metody **Bar** będzie pochodził z klasy **B**. Zwróćmy przy okazji uwagę, że instrukcja **callvirt** w przypadku funkcji **Foo** nie ma żadnego zastosowania, bowiem **Foo** nie jest metodą wirtualną. Podobnie, gdyby wszystkie wywołania **callvirt** zamienić na **call**, to fakt, że wywoływane metody są metodami wirtualnymi przestałby mieć znaczenie - **call** oznacza niepolimorficzne wywołanie funkcji.

```
C:\example>example.exe
A::Foo
B::Bar
A::Baz
```

### 3.10.2.9 Wyjątki

Obsługa wyjątków w ILAsm polega na użyciu dyrektyw **.try** i **.catch**. Na uwagę zasługuje fakt, że kod w obu sekcjach musi być jawnie opuszczony za pomocą instrukcji **leave**.

Poniższy przykład spowoduje wyjątek, ponieważ obiekt **FileStream** nie jest zainicjowany przed wywołaniem jego metody.

```
/*
using System;
using System.IO;

namespace NSpace
{
    class CMain
    {
        public static void Main()
        {
            try
            {
                FileStream fs = null;
                fs.Close();
            }
            catch( Exception ex )
            {
                Console.WriteLine( ex.Message );
            }
        }
    }
}
*/

.assembly extern mscorlib {}
.assembly Example{}
.class public CExample extends [mscorlib]System.Object
{
    .method public static void MyAppStart() il managed
    {
        .entrypoint
        .maxstack 2
        .locals init (class [mscorlib]System.IO.FileStream V_0,
                     class [mscorlib]System.Exception V_1)
        .try
        {
            ldnull
            stloc.0      // V_0 = null
            ldloc.0
            callvirt     instance void [mscorlib]System.IO.Stream::Close()
            leave.s      IL_0018
        } // end .try
    }
}
```

```

        catch [mscorlib]System.Exception
        {
            stloc.1
            ldloc.1
            callvirt instance string [mscorlib]System.Exception::get_Message()
            call void [mscorlib]System.Console::WriteLine(string)
            leave.s IL_0018
        } // end handler
    IL_0018: ret
}
}

```

### 3.10.3 Łączenie kodu z różnych języków

#### 3.10.3.1 Zasady łączenia kodu różnych języków

Platforma .NET pozwala z niespotykaną wcześniej łatwością łączyć kod napisany w różnych językach. Dowolny kompilator może produkować biblioteki kodu, które następnie mogą być używane z poziomu innych języków. W poniższym przykładzie kod klasy napisanej w VB.NET jest wykorzystywany w programie napisanym w C#.

```

// CMainForm.vb
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Namespace MainModule

    Public Class CMainForm
        Inherits System.Windows.Forms.Form

        Public Sub New
            MyBase.New()
        End Sub

        Protected Overrides Sub OnPaint( e As PaintEventArgs )
            Dim g as Graphics = e.Graphics
            Dim f as Font      = new Font( "Times New Roman", 48 )

            Dim sf as StringFormat = new StringFormat()

            sf.Alignment        = StringAlignment.Center
            sf.LineAlignment    = StringAlignment.Center

            g.Clear( SystemColors.Control )
            g.DrawString( "VB.NET", f, Brushes.Black, _
                Me.Width / 2, Me.Height / 2, sf )

        End Sub

    End Class

End Namespace

// CExample.cs
using System;
using System.Windows.Forms;

using MainModule;

class CExample
{
    public static void Main()
    {
        Application.Run( new CMainForm() );
    }
}

```

```
C:\example>vbc.exe /target:library CMainForm.vb
Microsoft (R) Visual Basic .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.00.3705.288
Copyright (C) Microsoft Corporation 1987-2001. All rights reserved.
```

```
C:\example>csc.exe /r:CMainForm.dll CExample.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

Aby możliwa była współpraca kodu napisanego w różnych językach, oczywistym wydaje się być wymaganie, aby języki te spełniały pewne warunki. W przypadku języków projektowanych z myślą o platformie .NET sprawa jest prosta. Trudności pojawiają się w przypadku języków dostosowywanych do wymogów platformy .NET, na przykład języków funkcjonalnych. Przykład kompilatora SML.NET pokazuje, że takie trudności można z powodzeniem pokonywać.

Czy więc współistnienie wielu języków w obrębie jednej platformy oznacza, że z poziomu kodu C# można wprost wołać kod z na przykład SML.NET?

Otóż nie do końca tak jest. Aby języki mogły współpracować ze sobą, konieczne jest aby komunikacja odbywała się za pomocą dość rygorystycznych reguł określanych przez specyfikację CTS (była już o tym mowa). Oznacza to, że moduł SMLowy może dowolnie korzystać z możliwości SMLa, ale po to aby pobrać parametry i oddać wyniki do modułu C#owego, musi na przykład opakować je w klasy, o które rozszerzono składnię SMLa. Dzięki temu, że częścią CTS jest definicja typów prostych, wymiana informacji między różnymi językami nie jest trudna: typy proste przekazuje się wprost, typy złożone opakowuje się w struktury lub klasy.

Poniższy prosty przykład pokazuje jak klasę napisaną w C# można wykorzystać w kodzie SML.NET.

```
// pola.cs
namespace NExample
{
    public class CExample
    {
        // pola statyczne
        public static readonly string pole_statyczne_readonly =
            System.String.Concat("SML.", "NET");
        public static int pole_statyczne = 23;

        // pola
        public int pole;

        public CExample( int n )
        {
            pole = n;
        }
    }
}

(* pola_demo.sml *)
structure pola_demo =
struct

fun main () =
let
    val c = NExample.CExample( 156 )
in
    print ("Pole statyczne readonly " ^ valOf(NExample.CExample.pole_statyczne_readonly) ^ "\n");
    print ("Pole statyczne " ^ Int.toString(!NExample.CExample.pole_statyczne) ^ "\n");
    NExample.CExample.pole_statyczne := 17;
    print ("Pole statyczne " ^ Int.toString(!NExample.CExample.pole_statyczne) ^ "\n");
    print ("Pole " ^ Int.toString(!c.pole) ^ "\n");
end
end
```

```

    c.#pole := 77;
    print ("Pole " ^ Int.toString(!(c.#pole)) ^ "\n")

end

end

C:\Example>csc /nologo /t:library pola.cs
C:\Example>smlnet -reference:pola pola_demo
C:\Example>pola_demo.exe
Pole statyczne readonly SML.NET
Pole statyczne 23
Pole statyczne 17
Pole 156
Pole 77

```

### 3.10.3.2 Pułapki

Podczas łączenia kodu napisanego w różnych językach programista może natknąć się na różne problemy. Jednym z najsubtelniejszych z nich jest problem polegający na zbudowaniu różnych języków w oparciu o inne modele obiektowe.

Rozważmy następujący przykład kodu w C#.

```

using System;
using System.Windows.Forms;

namespace CPulapka
{
    public class A
    {
        public virtual void Q( int k )
        {
            Console.WriteLine( "A::Q(int)" );
        }
    }

    public class B : A
    {
        public virtual void Q( double d )
        {
            Console.WriteLine( "B::Q(double)" );
        }

        public static void Main()
        {
            B b = new B();
            b.Q( 1.0 ); // tu jest OK!
            b.Q( 1 );  // a tu?
        }
    }
}

C:\example>CEExample.exe
B::Q(double)
B::Q(double)

```

Taki a nie inny wynik działania programu może być w pierwszej chwili dość nieoczekiwany. W klasie **A** zdefiniowano bowiem metodę **Q(int)**, która wydaje się lepiej pasować do wywołania **b.Q( 1 )** niż przeciążona w klasie **B** metoda **Q( double )**.

Kompilator C# kieruje się jednak jednoznacznymi regułami dopasowania funkcji, określonymi w specyfikacji języka. Reguły te mówią, że jeżeli możliwe jest dopasowanie parametrów do funkcji zdefiniowanej w klasie bieżącej, to funkcja taka zostanie wywołana, mimo że w klasie bazowej może istnieć funkcja, która w danym kontekście mogłaby być bardziej właściwa (zauważmy, że konwersja **int**→**double**, która jest konieczna aby wywołać funkcję **B::Q(double)** jest



*gorsza* niż konwersja **int**→**int**, która miałaby miejsce, gdyby w **b.Q(1)** wywołana była funkcja **A::Q(int)**).

Najbardziej właściwe pytanie, które należałoby zadać w tym miejscu brzmi: a co się stanie, jeśli w innym języku programowania reguły te wyglądają inaczej i spróbujemy połączyć kod takich dwóch języków?

Cóż, przekonajmy się:

```
// CExample.cs -> CExample.dll
using System;
using System.Windows.Forms;

namespace CPulapka
{
    public class A
    {
        public virtual void Q( int k )
        {
            Console.WriteLine( "A::Q(int)" );
        }
    }

    public class B : A
    {
        public virtual void Q( double d )
        {
            Console.WriteLine( "B::Q(double)" );
        }
    }
}

// CTest.vb kompilowany z referencją do CExample.dll
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Imports CPulapka

Module MainModule

    Sub Main
        Dim b as B
        b = new B()

        b.Q( 1.0 )
        b.Q( 1 )
    End Sub

End Module

C:\example>CTest.exe
B::Q(double)
A::Q(int)
```

Wynik tekstu potwierdza, że wybierając funkcję do wywołania w danym kontekście, kompilator danego języka kieruje się swoimi własnymi regułami. W tym przypadku kompilator języka VB.NET do wywołania **b.Q(1)** dopasował funkcję **A::Q(int)** w przeciwieństwie do kompilatora C#, który (jak widzieliśmy) w tym samym przypadku wybrałby funkcję **B::Q(double)**.

Przykład ten jest bardzo pouczający. Świadczy on o tym, że mimo możliwości integracji w obrębie jednej platformy uruchomieniowej, języki programowania mogą zachowywać dużą dozę niezależności. W końcu gdyby każdy język trzeba było "na siłę" dopasować do jakichś reguł, zmieniając jednocześnie jego semantykę, to cała idea .NET byłaby niewiele warta - oznaczałaby bowiem powstanie tak naprawdę jednego sposobu "ewaluacji" programów, tyle że ubranego w składnię innych języków. W chwili obecnej zaś, programista chcący skorzystać z możliwości łączenia kodu różnych języków musi być po prostu świadomy możliwych problemów z tym

związanych - problemów, podkreślmy, natury dość głębokiej i wynikającej z dużych różnic koncepcyjnych pomiędzy językami i ich modelami obiektowymi. Od strony czysto "technicznej" platforma .NET daje wyjątkową możliwość bezproblemowego łączenia kodu dowolnych języków programowania.

# Bibliografia

- [1] *<http://msdn.microsoft.com>*
- [2] *<http://www.c-sharpcorner.com>*
- [3] Archer T., Whitechapel A. *Inside C#, Microsoft Press*
- [4] Eckel B. *Thinking in C#, <http://www.bruceeckel.com>*
- [5] Gunnerson E. *A Programmer's Introduction to C#*
- [6] Lidin S. *Inside Microsoft .NET IL Assembler, Microsoft Press*
- [7] Morgan M. *Poznaj język Java 1.2*
- [8] Petzold Ch. *Programming Windows, Microsoft Press*
- [9] Reilly Douglas J. *Designing Microsoft ASP.NET Applications*
- [10] Stroustrup B. *Język C++*



## Dodatek A

# Przykładowe aplikacje

Zapraszam do przeglądu interesujących przykładowych programów, które z różnych względów nie znalazły się wśród programów przedstawianych wcześniej. Każdy z tych programów demonstruje różne przydatne techniki bądź programistyczne związane z platformą .NET. Programy pozostawię bez komentarza, aby czytelnik mógł przeanalizować je samodzielnie.

### A.1 Animowany fraktalny zbiór Julii

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace vicMazeGen
{
    public class CFrmJulia : System.Windows.Forms.Form
    {
        private const int PSiz = 128;

        private double angle = 0.0, angle2 = 0.0;
        private int cx, cy;

        private Bitmap picFracSource;
        private System.Windows.Forms.PictureBox picFrac;

        public CFrmJulia()
        {
            this.SetStyle(ControlStyles.DoubleBuffer, true);
            this.SetStyle(ControlStyles.AllPaintingInWmPaint, true);

            picFracSource = new Bitmap( PSiz, PSiz, PixelFormat.Format24bppRgb );

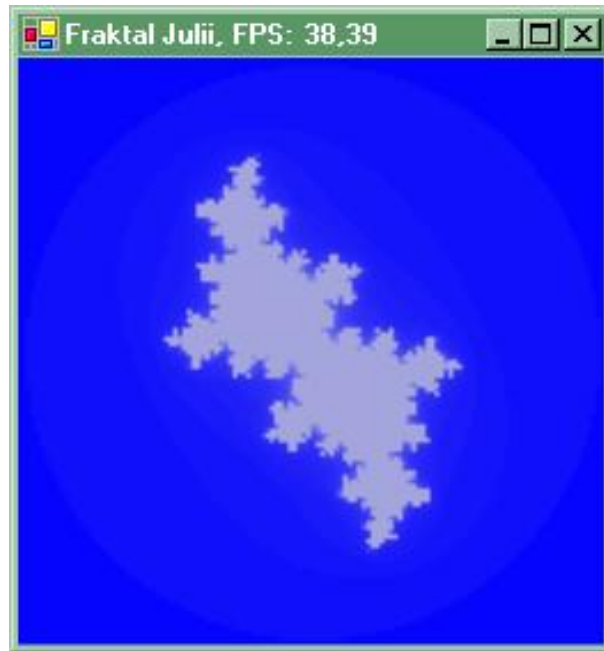
            this.picFrac = new System.Windows.Forms.PictureBox();
            this.picFrac.Dock = DockStyle.Fill;
            this.picFrac.SizeMode = System.Windows.Forms.PictureBoxSizeMode.StretchImage;

            this.ClientSize = new System.Drawing.Size(256, 256);
            this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;

            this.Controls.Add( this.picFrac );
        }

        void JuliaPaint()
        {
            angle += 0.023; angle2 += 0.027;
            cx = (int)( 800.0 * Math.Sin( angle+1.0 ) );
            cy = (int)( 800.0 * Math.Cos( angle2 ) );

            Rectangle bounds =
```



Rysunek A.1: Animowany fraktalny zbiór Julii

```

    new Rectangle( new Point(0, 0),
    new Size( picFracSource.Width, picFracSource.Height ) );
    BitmapData bData =
        picFracSource.LockBits( bounds, ImageLockMode.ReadWrite,
                                PixelFormat.Format24bppRgb );

    byte[] picData = new byte[PSiz * PSiz * 3];
    int iDex = 0;

    int iterNo, i, j;
    int x, y, xn, yn, x2, y2;
    for (i=0; i<PSiz; i++)
        for (j=0; j<PSiz; j++)
        {
            iterNo = 0;

            x = (i<<5)-2048;
            y = (j<<5)-2048;
            x2 = x*x;
            y2 = y*y;
            while (
                ( iterNo++ < 32 ) &&
                ( Math.Abs(x2+y2) < 4000000 )
            )
            {
                x2 = x*x; y2 = y*y;
                xn = ((x2-y2)>>10) + cx;
                yn = ((x*y)>>9) + cy;
                x = xn; y = yn;
            }

            picData[iDex++] = Convert.ToByte(255-iterNo);
            picData[iDex++] = Convert.ToByte(5*iterNo);
            picData[iDex++] = Convert.ToByte(5*iterNo);
        }

    Marshal.Copy ( picData, 0, bData.Scan0, PSiz * PSiz * 3 );
    picFracSource.UnlockBits( bData );

```

```

        this.picFrac.Image = picFracSource;
        this.picFrac.Invalidate();
    }

    public static void Main()
    {
        CFrmJulia frm = new CFrmJulia();
        frm.Show();

        DateTime start = DateTime.Now;
        int frame = 0;
        while ( frm.Created )
        {
            frm.JuliaPaint();
            Application.DoEvents();

            frm.Text = String.Format( "Fraktal Julii, FPS: {0:N}",
                                     frame/((TimeSpan)(DateTime.Now-start)).TotalSeconds );
            frame++;
        }
    }
}

```

## A.2 Bezpośredni dostęp do nośnika danych w Windows NT

```

#include <windows.h>
#include <stdio.h>

void ShowErrorMessage()
{
    char* lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
    );

    // Pokaż komunikat
    MessageBox( NULL, lpMsgBuf, "GetLastError", MB_OK|MB_ICONINFORMATION );

    // Zwolnij pamięć
    LocalFree( lpMsgBuf );

    exit( 1 );
}

int main( int argc, char **argv )
{
    BOOL bResult;

    const int nBytesToRead = 512;
    unsigned long nBytesRead;
    unsigned long nBytesWrote;
    char inBuffer[nBytesToRead];

    HANDLE hFloppy = CreateFile( "\\.\a:",
        GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL, OPEN_EXISTING, 0, NULL );
    if ( hFloppy == INVALID_HANDLE_VALUE ) ShowErrorMessage();

    HANDLE hData = CreateFile( "read.bin",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,

```

```
        NULL, CREATE_ALWAYS, 0, NULL );
if ( hData == INVALID_HANDLE_VALUE ) ShowErrorMessage();

// kopiuj obraz dyskietki do pliku
do
{
    bResult = ReadFile( hFloppy, &inBuffer, nBytesToRead, &nBytesRead, NULL ) ;
    if ( bResult )
        WriteFile( hData, inBuffer, nBytesRead, &nBytesWrote, NULL );
} while ( bResult != 0 && nBytesRead > 0 );

CloseHandle( hFloppy );
CloseHandle( hData );

return 0;
}
```