

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Programowanie Zaawansowanych Aplikacji w C#

Dr Jerzy Białkowski

UMK Toruń 2013

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Spis treści

Spis treści	3
Przedmowa.....	13
Wykład	17
1. Podstawy języka C#	19
Ogólne informacje	19
Przykładowy program	19
Komentarze	21
Instrukcje	22
Instrukcje sterujące	22
Instrukcja warunkowa if / if - else	22
Instrukcja wyboru switch - case	23
Pętla while.....	24
Pętla do – while.....	24
Pętla for.....	24
Pętla foreach	25
Instrukcje skoków	25
Deklarowanie zmiennych.....	25
Operatory	26
Wyjątki.....	27
Tworzenie metod	29
Metody ze zmienną ilością argumentów	29
Metody z domyślnymi argumentami.....	30
Argumenty wejściowe, wyjściowe i referencyjne	30
Przeciążanie metod (i operatorów).....	31
Modyfikatory dostępu	33
Dziedziczenie	33
Właściwości (<i>Properties</i>)	34
Indeksery	35
Delegacje	36
2. Typy	39
Podstawowe właściwości typów języku C#.....	39
Kryteria podziału typów.....	39
Porównanie typów wartościowych i referencyjnych.....	40
Rzutowanie typów	40
Porównanie klas i interfejsów	41
Porównanie struktur i klas.....	41
Wbudowane typy proste a struktury	42
Typy mogące przyjmować wartość <i>null</i>	42
Typy wyliczeniowe	43
Statyczne pola i metody	44
Stałe i zmienne tylko do odczytu	45
Tablice	45
3. Kolekcje	49
Porównanie tablic i kolekcji.....	49
Podział kolekcji.....	49
Kolekcje niegeneryczne	49
Kolekcje generyczne	50
Porównanie popularnych kolekcji niegenerycznych i generycznych	50
Wyspecjalizowane kolekcje	51
Używanie przez kolekcje przestrzeń nazw	51

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
 realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykładowe użycia kolekcji niegenerycznych.....	51
ArrayList	51
Queue	51
Stack	52
Hashtable	52
SortedList	53
BitArray	53
Przykładowe użycia kolekcji generycznych	53
List<>	53
Queue<>	54
Stack<>	54
LinkedList<>	54
SortedList<,>	54
Struktury i klasy reprezentujące węzły	55
Struktura DictionaryEntry.....	55
Struktura KeyValuePair<TKey, TValue>	55
Klasa LinkedListNode<T>.....	55
Interfejsy implementowane w kolekcjach.....	55
ICollection.....	56
IList	56
IDictionary.....	56
IComparable	57
IComparer	57
IEqualityComparer	57
IEnumerator	57
IEnumerable	57
Przykłady definiowania kolekcji zawierających <i>enumerator</i>	57
Przykład implementacji niegenerycznej kolekcji z klasycznie zdefiniowanym enumeratorem	58
Przykład implementacji generycznej kolekcji z enumeratorem z leniwą ewaluacją	60
Składowe popularnych kolekcji niegenerycznych	62
ArrayList	62
Stack.....	63
Queue	63
Hashtable	63
SortedList	64
BitArray	65
Comparer	65
Składowe popularnych kolekcji generycznych	66
Stack<T>	66
Queue<T>	66
List<T>	66
Dictionary<TKey, TValue>	67
SortedList<TKey, TValue>	67
SortedDictionary<TKey, TValue>	68
LinkedList<T>	68
Wyspecjalizowane kolekcje	69
Klasy operujące na łańcuchach znaków	69
Klasy słownikowe	70
Tworzenie klas z kluczami opartymi na łańcuchach	70
Wyspecjalizowane struktury bitowe.....	71
4. Serializacja	73
Rodzaje serializacji i ich najważniejsze cechy	73
Serializacja binarna	73
Serializacja XML-owa	73
Serializacja SOAP-owa	73
Pojęcie serializera, najważniejsze klasy, przestrzenie, referencje	74
Serializacja typów wbudowanych.....	74
Serializacja binarna	74
Serializacja XML-owa	75
Serializacja SOAP-owa	77

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
 realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Porównanie struktury serializowanych danych	77
Serializacja własnej klasy	79
Serializacja binarna	80
Serializacja XML-owa	81
Serializacja SOAP-owa	83
Zaawansowane sposoby serializacji	84
Wybiórcza serializacja i implementacja interfejsu <i>IDeserializationCallback</i>	84
Używanie atrybutów metod	85
Implementacja interfejsu <i>ISerializable</i>	86
Opcjonalne serializowane wybranych elementów	88
XML-owa serializacja zbiorów danych	89
5. Wprowadzenie do Windows Presentation Foundation	91
Podstawowe informacje o WPF	91
XAML jako język opisu okien	92
Składowe opisu okna	92
Właściwości kontrolek na przykładzie pozycjonowania	92
Zagnieżdzanie znaczników	93
Grupowanie kontrolek i odwoływanie się do nich	95
Kolejność kontrolek – przykrywanie	95
Importowanie kontrolek z Windows Forms	95
Definiowanie i używanie stylów dynamicznych	96
Style globalne	96
Statyczne ładowanie stylów	97
Wyzwalače	97
Menu	98
Menu kontekstowe	98
Podpinanie reakcji na zdarzenia	99
Programowanie interfejsu z poziomu języka C#	99
Elementy implementacji klasy okna	100
Dostęp do kontrolek	100
Metody obsługi zdarzeń	101
Używanie kontrolki <i>MessageBox</i>	101
Otwieranie nowego okna zdefiniowanego w XAML-u	102
Tworzenie okna bez użycia XAML-a	102
Używanie kontrolki <i>Grid</i>	103
Przykłady inicjalizacji kontrolek	103
Tworzenie menu kontekstowego	104
Używanie kontrolek dziedziczących po klasie <i>FileDialog</i>	104
6. Wątki	107
Ogólne informacje o wątkach	107
Wątki a procesy	107
Wątki pierwszoplanowe i wątki drugoplanowe	108
Używanie klasy <i>Thread</i>	108
Tworzenie nowych wątków	108
Statyczne właściwości	108
Nie-statyczne właściwości	109
Wybrane nie-statyczne metody	109
Wybrane statyczne metody	109
Możliwe stany wątku	110
Możliwe priorytety wątku	111
Przykład na tworzenie wątku	111
Przykład na przekazywanie danych do metody wątku przez argument	112
Przykład na przekazywanie danych do metody wątku przez obiekt	112
Przykład na przekazanie wyniku przez obiekt	113
Przykład na przekazanie wyniku przez delegację	114
Przykład pokazujący przyjmowanie różnych stanów przez wątek	115
Tworzenie wątków przez klasę <i>ThreadPool</i>	115
Podstawowe informacje o tworzeniu wątków przez klasę <i>ThreadPool</i>	116
Wybrane statyczne metody	116

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykład tworzenia wątków przez klasę <i>ThreadPool</i>	116
Przykład na odczyt i ustawianie limitu zasobów przez klasę <i>ThreadPool</i>	117
Kiedy nie należy używać klasy <i>ThreadPool</i>	118
Mechanizmy tworzenia wątków wprowadzone od wersji 4.0 .NET Frameworka.....	118
Klasa <i>Task</i>	118
<i>Parallel.Invoke</i>	120
Synchronizacja wątków	121
Rygle.....	121
Monitory	122
Klasa <i>WaitHandle</i>	124
Muteksy	125
Semafora	128
Rygle odczytu-zapisu.....	131
Klasa <i>EventWaitHandle</i> i jej klasy pochodne (uchwyty oczekiwania na zdarzenia).....	132
Klasa <i>RegisteredWaitHandle</i>	135
Klasa <i>Interlocked</i>	136
7. Asynchroniczne uruchamianie metod.....	139
Konsepcja programowania asynchronicznego.....	139
Mechanizmy służące do wykonywania operacji asynchronicznych	139
Klasyczne rozwiązanie z jawnym użyciem wątków.....	139
Delegacje w programowaniu synchronicznym i asynchronicznym.....	139
Asynchroniczne operacje z użyciem <i>IAsyncResult</i>	139
Asynchroniczne operacje z użyciem zdarzeń	140
Obsługa asynchronicznego wywołania	140
Składowe interfejsu <i>IAsyncResult</i>	140
Przykład	140
Kontekst uruchomieniowy	141
Rola kontekstu uruchomieniowego	141
Składowe klasy <i>ExecutionContext</i>	141
Składowe klasy <i>HostExecutionContext</i>	142
Składowe klasy <i>HostExecutionContextManager</i>	142
Kontekst synchronizacji	142
Rola kontekstu synchronizacji.....	142
Składowe klasy <i>SynchronizationContext</i>	142
Klasyczny przykład (z dokumentacją) na użycie kontekstu synchronizacji.....	142
8. Domeny aplikacji.....	145
Domeny aplikacji, a procesy i wątki	145
Klasy związane z domenami aplikacji	146
Klasa <i>AppDomain</i>	146
Klasa <i>AppDomainSetup</i>	148
Przykłady	149
Przykład na dowiązywanie wartości domen aplikacji	149
Przykład na utworzenie domeny aplikacji oraz modyfikację i odczyt jej własności	149
Przykład na przypisywanie roli użytkownika w domenie aplikacji.....	150
9. Usługi windows-owe (Windows Services).....	152
Podstawowe informacje o usługach windowsowych	152
Działanie usług w systemach Windows	152
Podstawowe elementy implementacji usług.....	152
Elementarny przykład kompletnej implementacji usługi	153
Klasa usługi.....	153
Uruchamianie usługi	153
Referencje	154
Instalator	154
Rejestrowanie usługi w systemie	154
Tworzenie usług ze wzorca (w Visual Studio).....	155
Opis klas używanych w implementacji usługi	155
Klasa <i>ServiceBase</i>	155
Klasa <i>ServiceInstaller</i>	157

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
 realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Klasa <i>ServiceProcessInstaller</i>	157
Zarządzanie usługami przez klasę <i>ServiceController</i>	158
Wybrane własności klasy <i>ServiceController</i>	158
Typy usług w systemie.....	158
Statusy (stany) usług	159
Metody klasy <i>ServiceController</i>	159
Przykład użycia klasy <i>ServiceController</i>	160
10. Konfiguracja	161
Używane klasy	161
Najważniejsze klasy konfiguracji.....	161
Klasy reprezentujące strukturę pliku konfiguracyjnego	161
Typy wyliczeniowe	162
Składowe plików konfiguracyjnych	162
Wspólne ustawienia	162
Ustawienia aplikacji	164
Wczytywanie elementów konfiguracji.....	165
Wczytywanie ustawień aplikacji	165
Wczytywanie ścieżek połączeń do bazy.....	166
Wczytywanie konkretnych sekcji i kolekcji grup.....	167
Zapis konfiguracji	168
11. Instalatory	171
Mechanizmy używane przy instalacji	171
Implementacja własnego instalatora	171
Instalacja biblioteki z kodu w języku C#	174
12. Dzienniki zdarzeń	177
Podstawowe informacje o dziennikach zdarzeń.....	177
Operacje wykonywane na dziennikach zdarzeń.....	177
Tworzenie, otwieranie i podpinanie źródła	178
Zapis.....	179
Odczyt.....	179
Czyszczenie i usuwanie.....	179
Nazwy dzienników, a ustawienia regionalne	180
13. Procesy	181
Klasa <i>Process</i>	181
Metody statyczne	181
Wybrane metody nie-statyczne	181
Wybrane (niestatyczne) własności	182
Klasa <i>ProcessStartInfo</i>	183
Konstruktory	184
Wybrane własności	184
Przykłady	185
Odczyt własności bieżącego procesu	185
Pobranie procesu przez identyfikator	185
Pobranie procesów przez nazwę.....	186
Listowanie procesów.....	186
Uruchamianie procesów	187
Listowanie modułów	188
14. Liczniki (Performance Counters).....	191
Podstawowe informacje o licznikach.....	191
Informacje o sposobie używania liczników.....	191
Klasy związane z licznikami	191
Typy liczników	192
Podział liczników ze względu na rodzaj wykonywanych obliczeń	192
Składowe klasy <i>PerformanceCounter</i>	192
Metody	192
Właściwości	193
Przykłady	193

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
 realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

15. Debugowanie i śledzenie aplikacji.....	195
Dostęp do ramek stosu.....	195
Kontrola nad debugerem w debugowanym programie	196
Raportowanie w trybie debugowania.....	197
Raportowanie w trybie śledzenia	197
Przełączniki śledzenia	198
Przełączniki dwustanowe	198
Przełączniki wielostanowe	199
Obsługa wyjścia śledzenia	201
16. System plików oraz obsługa wejścia-wyjścia.....	203
Klasy operujące na strukturze plików	203
Klasa <i>Path</i>	203
Klasa <i>FileSystemInfo</i>	204
Klasa <i>File</i>	204
Klasa <i>FileInfo</i>	206
Klasa <i>Directory</i>	206
Klasa <i> DirectoryInfo</i>	207
Klasa <i> DriveInfo</i>	208
Klasa <i> FileSystemWatcher</i>	208
Klasy implementujące binarne strumienie	209
Składowe abstrakcyjnej klasy <i> Stream</i>	209
Klasy pochodne klasy <i> Stream</i>	210
Klasy implementujące formatowane dane („reader”-y i „writer”-y).....	211
Inne mechanizmy związane z wejściem-wyjściem	212
Izolowane zasobniki.....	212
Kompresja i dekompresja danych	212
Przykłady	212
Przykład na użycie klasy <i> Path</i>	212
Przykład na użycie klasy <i> File</i>	213
Przykład na użycie klas <i> DirectoryInfo</i> i <i> FileInfo</i>	215
Przykład na użycie klasy <i> DriveInfo</i>	215
Przykład na użycie klasy <i> FileWatcher</i>	216
17. Bezpieczeństwo: Code Access Security	219
Code Access Security w .NET Frameworku o wersjach niższych niż 4.0	219
Elementy CAS.....	219
Narzędzia do konfigurowania CAS.....	219
Rodzaje identyfikacji (<i>Evidence</i>)	220
Domyślne uprawnienia (<i>Permissions</i>).....	220
Domyślne zbiory uprawnień (<i>Permission Sets</i>).....	221
Przykłady domyślnych zbiorów uprawnień	221
Grupy kodu (<i>Code Groups</i>) i polityka bezpieczeństwa (<i>Security Policy</i>)	222
Wielowarstwość polityki bezpieczeństwa	222
Przestrzenie nazw związane z systemem bezpieczeństwa w .NET Frameworku	222
Przykłady na żądanie uprawnień	222
Zmiany w .NET Frameworku 4.0 w stosunku do wcześniejszych wersji	223
18. Bezpieczeństwo: Role Based Security	225
Najważniejsze przestrzenie i klasy.....	225
Klasa <i> WindowsIdentity</i>	225
Tworzenie nowych obiektów klasy <i> WindowsIdentity</i>	225
Właściwości	225
Przykłady	226
Pobranie <i> Identity</i> bieżącego użytkownika i jego <i> Principal-a</i>	226
Alternatywna metoda pobrania <i> Principal-a</i> bieżącego użytkownika	226
Klasa <i> PrincipalPermission</i>	226
Przykłady	226
Żądanie konkretnego użytkownika przez atrybut.....	226
Żądanie konkretnej roli przez atrybut.....	226
Żądanie konkretnego użytkownika przez wywołanie metody.....	226

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Principal dla pary użytkowników	226
Implementacja własnych klas	226
Implementacja własnej klasy identyfikacji.....	226
Implementacja własnej klasy identyfikującej rolę.....	227
19. Kryptografia	229
Kryptografia symetryczna	229
Używanie mechanizmów wbudowanych w system	229
Algorytmy	230
Klasy	230
Przykłady	230
Kryptografia asymetryczna	231
Algorytmy	231
Klasy	231
Przykłady	232
Funkcje jednokierunkowe	233
Algorytmy	233
Klasy	233
Przykłady	233
Podpisy.....	234
20. Integracja kodu zarządzalnego i niezarządzalnego (Interoperability)	235
Wybrane różnice pomiędzy kodem zarządzalnym i niezarządzalnym.....	235
Podstawowe podobieństwa i różnice pomiędzy bibliotekami	235
Różnice w implementacji metod	235
Obsługa wyjątków	236
Ograniczenia komponentów COM.....	236
Ograniczenia w assembly .NET używanych w komponentach COM	237
Napopularniejsze narzędzia	237
Lista narzędzi	237
Przykłady użycia	237
Ukrywanie zawartości assembly poprzez atrybut <i>ComVisible</i>	238
Przykład	239
Platform Invoke	239
Ogólna koncepcja.....	239
Zalety enkapsulacji.....	240
Zalecany schemat użycia.....	240
Przykłady	240
Konwersja typów	242
Przykłady marshalingu danych	242
Kontrola sposobu ułożenia danych w pamięci	243
Przykład przekazywania struktur	243
Używanie funkcji powrotu	244
Pobieranie kodów błędów	245
Klasa Marshal	246
Podsumowanie	247
21. Refleksje	249
Assembly	249
Podstawowa charakterystyka Assembly.....	249
Budowa Assembly jednoplikowego	249
Podstawowe informacje o Assembly wieloplikowych	249
Refleksje	250
Rola refleksji.....	250
Klasa Type	250
Wybrane metody	250
Wybrane właściwości	251
Klasa ILGenerator.....	253
Inne klasy związane z refleksami.....	253
Przykłady	253
Odczyt wybranych właściwości typu.....	256

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
 realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Pobieranie typów danych	256
Pobieranie typu bazowego	257
Pobieranie typu z assembly (biblioteki)	257
Pobieranie informacji o składowych typu	258
Pobieranie informacji o konkretnych rodzajach składowych typu	259
Właściwości jako składowe o specyficznej konstrukcji	260
Tworzenie instancji typów	260
Tworzenie instancji funkcji	261
Ładowanie assembly	261
Pobranie ciała metody	262
Pobranie atrybutów assembly	263
Dynamiczne generowanie metody	263
22. Obsługa poczty	265
Wysyłanie listu	265
Wysyłanie listu z autoryzacją	265
Edytowanie nagłówka wysyłanego listu	266
Wysyłanie listu pod kilka adresów	266
Wysyłanie oraz kopii listu	267
Dołączanie załącznika	267
Wysyłanie listu w formacie HTML	268
Wysyłanie listu z alternatywnymi formatami zawartości	268
Wysyłanie listu w szyfrowanym połączeniu	269
Wysyłanie listu w sposób asynchroniczne	269
23. Programowanie obłokowe w Windows Azure	271
Koncepcja programowania obłokowego	271
Konstrukcja Windows Azure	271
Tworzenie oprogramowania działającego pod Windows Azure	272
24. Tworzenie aplikacji dla systemu Windows 8 w Metro New UI	273
25. Windows Communication Foundation	275
Idea WCF	275
Rozróżnienie ze względu na sposób udostępniania (uruchamiania)	275
Punkty końcowe	275
26. Elementy programowania w zespole (i Team Foundation Server)	277
Modele i fazy tworzenia oprogramowania	277
Typowy cykl tworzenia oprogramowania	277
Klasyfikacja narzędzi używanych w pracy nad projektem	278
Zintegrowane środowiska programistyczne	278
Systemy kontroli wersji	278
Systemy śledzenia błędów	279
Narzędzia biurowe	279
Narzędzia wspierające zarządzanie projektem	279
Team Foundation Server jako narzędzie wspierające zarządzanie projektami programistycznymi	279
Visual Studio Ultimate	280
Team Foundation Server	280
Integracja narzędzi	280
Struktura sieci	281
Elastyczne (ang. <i>agile</i>) podejście do tworzenia oprogramowania	281
Scrum	282
Laboratorium	283
1. Laboratorium 1: zapoznanie z IDE Visual Studio	285
2. Laboratorium 2 i 3: Kolekcje, serializacja, biblioteka	286
3. Laboratorium 4: Stworzenie aplikacji w WPF	287
4. Laboratorium 5: Tworzenie i synchronizacja wątków	288

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

5.	Laboratorium 6 i 7: Tworzenie usług i instalatora, dzienniki, kontrola usługi.....	291
6.	Laboratorium 8 i 9: Procesy, pliki konfiguracyjne, przełączniki śledzenia, watcher	293
7.	Laboratorium 10: Liczniki (Performance Counters).....	294
8.	Laboratorium 11: Refleksje i obsługa poczty	295
9.	Laboratorium 12 i 13: Tworzenie i publikowanie usług w Windows Azure	296
10.	Laboratorium 14: Pisanie programów dla Windows 8 w stylu Metro New UI.....	298
11.	Laboratorium 15: WCF i kończenie pisania rozpoczętych wcześniej programów.....	299
	Literatura.....	301
	Bibliografia w języku polskim	301
	I. Książki.....	301
	II. Skrypty do szkoleń w Microsoft IT Academy (darmowy dostęp)	301
	Bibliografia w języku angielskim	302
	I. Książki.....	302
	II. Darmowe materiały do samodzielnej nauki - kursy z podziałem na laboratoria.....	302
	III. Dokumentacja w postaci elektronicznej i odsyłacze do stron WWW	302

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przedmowa

Niniejszy przedmiot przeznaczony jest dla osób, które znają już składnię języka C (lub dowolnego języka, który z niego wyewoluował, np. C++, C#, Java, D), podstawowe konstrukcje i pojęcia dotyczące programowania obiektowego oraz poznali jakiś obiektowy język programowania (np. C#, Java, D). W szczególności jest on skierowany dla studentów roku III studiów pierwszego stopnia na kierunku informatyka oraz do studentów I i II roku studiów drugiego stopnia na tym kierunku, którzy we wcześniejszych latach poznali (w ramach przedmiotu *Programowanie I*) składnię języka C oraz (w ramach przedmiotu *Programowanie II*) podstawy programowania obiektowego.

W toku nauki studenci powinni zdobyć wiedzę i umiejętności przydatne przy tworzeniu w języku C# gotowych do wdrożenia aplikacji przeznaczonych dla systemów MS Windows. W szczególności powinni nauczyć się korzystać z wbudowanych w te systemy oraz w środowisko .NET Framework mechanizmów (m.in. systemu uprawnień, logów systemowych, rejestru systemu, zarządzania konfiguracjami, transakcyjnej instalacji).

Ponieważ przewiduje się, że większość słuchaczy (np. w ramach przedmiotu *Programowanie II*) miała wcześniej styczność z językiem C# (zwłaszcza, jeśli w ramach tego przedmiotu wybrała grupę laboratoryjną z językiem C#), więc podstawy tego języka (pierwsze zagadnienia z programu wykładu) będą przedstawione w dużym skrócie.

Program wykładu podzielony jest na trzy części.

Pierwsza jest szybkim wprowadzeniem podstawowych konstrukcji języka C#, działania zintegrowanego środowiska uruchomieniowego oraz innych związanych z tym zagadnień, częściowo przypominając informacje wprowadzone w ramach przedmiotu *Programowanie II*, a częściowo je rozbudowując. W ramach tej części omawiane są następujące zagadnienia:

- Typy danych, instrukcje warunkowe i iteracyjne, kolekcje, typy generyczne.
- Tworzenie aplikacji graficznych z użyciem Windows Presentation Foundation.
- Przeciążanie operatorów, pary operatorów, serializacja danych.
- Wielowątkowość, zarządzanie wątkami, domeny aplikacji.

Druga, właściwa (z zazarem najdłuższa) część wykładu obejmuje zagadnienia potrzebne do osiągnięcia wspomnianych celów podanych dydaktycznych, w szczególności:

- Zarządzanie konfiguracją.
- Tworzenie i używanie transakcyjnych instalatorów.
- Programowanie usług systemu Windows.
- Zarządzanie logami systemowymi, licznikami, wykonywanie operacji na rejestrze.
- Śledzenie stanu uruchomionych procesów i zarządzanie nimi.
- Obsługa strumieni, wejścia-wyjścia, kompresja danych.
- Bezpieczeństwo (RBS i CAS) i kryptografia.
- Integracja kodu zarządzalnego z kodem niezarządzalnym.
- Refleksje.
- Wysyłanie poczty.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Dodatkowo, pod koniec wykładu, w ramach jego trzeciej części przedstawiane będą mechanizmy i narzędzia związane z aktualnymi trendami w programowaniu z użyciem platformy .NET i języka C#. W aktualnie opracowanej wersji materiałów poruszane są następujące zagadnienia.

- Programowanie obłokowe (Cloud Computing) z użyciem Windows Azure.
- Tworzenie aplikacji (typu Metro/Windows Store/Metro UI) dla Windows 8.
- Programowanie w zespole (Team Programming) z użyciem Windows Team Foundation Server

Zgodnie z wcześniejszymi zapowiedziami zagadnienie „programowanie urządzeń mobilnych w Windows Phone 7” zostało usunięte z niniejszych materiałów, ponieważ został utworzony nowy przedmiot, w którym zagadnienie to zostało zawarte. Należy pamiętać o tym, że materiał tej ostatniej części może się dość dynamicznie zmieniać w kolejnych cyklach kształcenia.

Wykład jest skonstruowany w ten sposób, aby osoby nie mające wcześniej do czynienia z językiem C#, ale znające język C, oraz jakiś oparty na nim obiektowy język programowania były w stanie w ciągu pierwszych zajęć opanować podstawy języka C# w stopniu wystarczającym do zrozumienia dalszej części wykładu. Na podstawie kilkuletnich obserwacji stwierdziłem, że większość studentów zna język Java, a przynajmniej wszyscy studenci, którzy nie mieli wcześniej styczności z językiem C# uczyli się Javy (łącznie ze studentami, którzy na studia II stopnia przyszli z innej uczelni). Z tego powodu w materiałach przedstawiających podstawy języka C# zakładam znajomość języka Java oraz w celu łatwiejszego przyswojenia przekazywanych treści wskazuję podobieństwa i różnice pomiędzy tymi językami.

Proponowany podział treści programowych z niniejszego opracowania pomiędzy 2-godzinne wykłady przebiega następująco:

- Wykład 1: rozdziały 1 i 2.
- Wykład 2: rozdziały 3 i 4.
- Wykład 3: rozdział 5.
- Wykład 4: rozdział 6.
- Wykład 5: rozdziały 7, 8 i 9.
- Wykład 6: rozdziały 10, 11 i 12.
- Wykład 7: rozdziały 13 i 14.
- Wykład 8: rozdziały 15 i 16.
- Wykład 9: rozdziały 17 i 18.
- Wykład 10: rozdział 19.
- Wykład 11: rozdział 20.
- Wykład 12: rozdziały 21 i 22.
- Wykład 13: rozdziały 23.
- Wykład 14: rozdziały 24.
- Wykład 15: rozdziały 25 i 26.

W przypadku niezaplanowanej konieczności zredukowania ilości wykładów (np. godziny rektorskie) uważam, że najlepszym rozwiązaniem byłoby przedstawienie zagadnień z rozdziałów 19 i 20 na jednym wykładzie.

Na koniec podam kilka informacji odnośnie literatury. Ponieważ nowa wersja środowiska programistycznego Visual Studio (o numerze 12, częściej nazwana *Visual Studio 2012*), a wraz z nią nowa wersja .NET Frameworka (o numerze 4.5) pojawiła się dopiero pod koniec 2012 roku, w chwili składania

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

niniejszych materiałów do recenzji nie ma jeszcze w języku polskim godnej polecenia pozycji opisującej te nowe standardy (zapowiadane jest wydanie [2] w maju 2013 roku). Nie jest to dużym problemem, gdyż zagadnień poruszanych w materiałach jest zgodna z wcześniejszymi standardami. Jednak dla przykładu zagadnienia rozdziału 24 przedstawiającego tworzenie aplikacji w stylu (Metro) New UI dla systemu Windows 8 dotyczą wyłącznie .NET Framework w wersji 4.5 i środowiska Visual Studio 2012. Duże zmiany zaszły też między innymi w przypadku Windows Communication Foundation (zagadnienie z rozdziału 25) - od wersji 4.5 .NET Frameworka możliwe jest używanie uproszczonej składni plików konfiguracyjnych, która jest znacznie czytelniejsza i łatwiejsza do przyswojenia przez studentów. Z tego powodu zrezygnowałem z podania przykładów w starszym standardzie, aby sprawiać niepotrzebnie wrażenia, że zawsze muszą być one bardzo skomplikowane.

Jako podręcznik dla osób, które dopiero rozpoczynają uczyć się języka C# polecałbym książki [1] i [10]. Można w tym celu użyć też książek [2] i [11], ale należy pamiętać, że część poświęcona „typowemu” WPF (rozdział 5 w moich materiałach) została w nich zastąpione częścią poświęconą tworzeniu aplikacji w (Metro) New UI (rozdział 24 w moich materiałach). Jako pozycji dla początkujących można też użyć materiałów z kursu [7]. Dobrymi pozycjami rozszerzającymi podstawową wiedzę są książki [3], [4], [12], [13], [14]. Podkreślam jednak, że obejmują one nieco inne zakresy materiału. W szczególności, osoby chcące poszerzyć wiedzę dotyczącej WPF znajdą ją wyłącznie w [4] i [12], a informacje o mechanizmach bezpieczeństwa CAS i RBS występujące w .NET Frameworku od wersji 4.0, jak również elementy kryptografii (rozdziały 17, 18, 19 z niniejszych materiałów) zawarte są w pozycji [14]. Wiele informacji zawartych w niniejszych materiałach z „właściwej części wykładu” zostało opracowane na podstawie [15]. W szczególności dotyczy to rozdziałów 10 i 20 oraz części rozdziału 17 poświęconej starszej wersji CAS. Jeśli chodzi o literaturę uzupełniającą, to polecam pozycje [16] i [6] dla osób chcących dowiedzieć się więcej o mechanizmach programowania równoległego, które można używać od wersji 4.0 .NET Frameworka oraz książki [5] i [17] dla osób chcących dogłębniej zapoznać się z zagadnieniami komunikacji poprzez WCF (i gotowe są przebrnąć przez zawiłą składnię plików konfiguracyjnych). Nazwy pozostałych pozycji z bibliografii jednoznacznie wskazują na opisywane zagadnienia. Część wykładów (rozdziały 23 i 26) bazuje na materiałach opracowanych w ramach Microsoft IT Academy (pozycje [7], [8], [9]) i nieodpłatnie udostępnianych na potrzeby przeprowadzania zajęć dydaktycznych. Niektóre wykłady i laboratoria (rozdziały 23 i 24) wykorzystują publicznie udostępnione materiały do samodzielnej nauki (pozycje [21] i [22]). Przy typowym pisaniu programów bardzo użyteczny jest bieżący dostęp do dokumentacji [23].

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Programowanie Zaawansowanych Aplikacji w C#

Wykład

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

1. Podstawy języka C#

Ogólne informacje

Język C# wyewoluował z języka C. Powstał on jako pewnego rodzaju alternatywa dla języka Java, której można używać w środowiskach Windowsowych¹. Z tego powodu w wielu aspektach te języki są do siebie podobne i będę chciał te podobieństwa wykorzystać, zakładając że większość czytelników zna podstawy języka Java.

Programy napisane w języku C# uruchamiane są we wspólnym środowisku uruchomieniowym **CLR** (ang. *Common Language Runtime*), które pełni rolę podobną jak maszyna wirtualna Javy dla języka Java. Samo środowisko CLR jest zaimplementowane w .NET Frameworku. Przed uruchomieniem programu napisanego w języku C# konieczna jest jego wcześniejsza kompilacja do *języka pośredniego* (ang. *Intermediate Language*, *Common Intermediate Language* lub *Microsoft Intermediate Language* – w skrócie **IL**, **CIL** lub **MSIL**), który jest pod względem funkcjonalnym podobny do *byte-codu* w Javie. Wartym podkreślenia jest fakt, że do języka pośredniego kompilować można też programy pisane w innych językach (np. *J#*, *F#*, Visual Basic, Visual C++), które zgodne są ze specyfikacją **CLS** (ang. *Common Language Specification*), która z kolei zawarta jest w specyfikacji **CTS** (ang. *Common Type System*). Ze względu na obecność mechanizmów kontroli oraz bezpieczeństwa programy skompilowane do języka pośredniego nazywane są często *kodem zarządzanym* (ang. *managed code*), w przeciwieństwo do programów w natywnym kodzie, zwanych *kodem niezarządzanym* (ang. *unmanaged code*)². Należy zaznaczyć, że chociaż środowisko uruchomieniowe (CLR) uruchamia programy napisane w języku pośrednim (IL) pod swoim nadzorem, to może ono, jeśli zajdzie taka potrzeba, skompilować uruchamiany kod (lub jego fragment) do języka natywnego (tzw. kompilacja *JIT* – ang. „*just-in-time*” compilation) i uruchomić go w tej postaci.

Przykładowy program

Pod względem składni język C# bardzo przypomina języki C, C++ i Javę. Rozważmy następujący kod przykładowego programu:

```
using System;

namespace PrzykladowyProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

¹ Istnieją rozwiązania pozwalające na uruchamianie programów pisanych w języku C# pod innymi systemami operacyjnymi (np. mono pod Linuxem), jednak mają one szereg ograniczeń.

² Przez skompilowane programy należy rozumieć nie tylko programy wykonywalne (zwykle z rozszerzeniem „EXE”), ale też np. biblioteki. Przykładem bibliotek DLL w kodzie niezarządzalnym są biblioteki COM (ang. *Component Object Model*).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

Zauważać w nim można metodę **Main**, do której przekazywane jest sterowanie w momencie uruchamiania programu³. Aby do metody **Main** mogło zostać przekazane sterowanie, musi ona być metodą statyczną do której jest pełen dostęp w obrębie pliku wykonywalnego (czyli musi mieć modyfikator dostępu *public* lub *internal* oraz klasa ją zawierająca też musi mieć modyfikator dostępu *public* lub *internal*⁴). Ponadto w programie⁵ powinna być zaimplementowana dokładnie jedna metoda o tej własności. Warto przy tym zaznaczyć, że C# jest obiektywym językiem programowania i podobnie jak w języku Java „zmienne” i „funkcje” mogą w nim występować wyłącznie w klasach (i nazywane są wówczas odpowiednio polami i metodami). Słowo kluczowe **namespace** definiuje *przestrzeń nazw*. Przestrzenie nazw pozwalają na większą kontrolę nad utrzymaniem złożonego kodu i tworzenie klas o tych samych nazwach. W szczególności pełna ścieżka do klasy **Program** ma postać **PrzykładowyProgram.Program**, ale w obrębie przestrzeni można się do niej odwoływać bez przedrostka przestrzeni. Innym sposobem na odwoływanie się do klas bez używania przedrostków przestrzeni jest użycie słowa kluczowego **using**⁶, podając przy nim przedrostki, które mają być domyślnie używane w programie. W tym przypadku użycie konstrukcji „**using System;**” pozwala na zastąpienie dłuższego odwołania do metody **WriteLine()** z klasy **Console** znajdującej się w przestrzeni **System**, np.

```
System.Console.WriteLine("Hello, World!");
```

krótszym

```
Console.WriteLine("Hello, World!");
```

Przestrzenie nazw można zagnieździć oraz można w ich obrębie używać konstrukcji „**using**”, np.:

```
namespace PierwszaPrzestrzen
{
    using System;

    namespace DrugaPrzestrzen.TrzeciaPrzestrzen
    {
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Hello, World!");
            }
        }
    }
}
```

³ Rozumianego w tym kontekście jako kod wykonywalny.

⁴ W tym wypadku jest to domyślny modyfikator *internal*; o modyfikatorach dostępu w języku C# będzie mowa później.

⁵ W tym miejscu rozumianym jako kod źródłowy który ma zostać skompilowany do postaci kodu wykonywanego.

⁶ Pełni ono jeszcze inną funkcję o której będzie mowa później.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W tym przypadku pełna ścieżka do klasy Program ma postać „*PierwszaPrzestrzeń.DrugaPrzestrzeń.TrzeciaPrzestrzeń.Program*”.

Programy napisane w języku C# (lub innych językach wspieranych przez .NET Framework) można kompilować do plików uruchomieniowych w formacie (posiadających rozszerzenia „.EXE”)⁷, bibliotek dynamicznych (z rozszerzeniem .DDL) oraz modułów (zwykle z rozszerzeniem „.netmodule”). Skompilowane w ten sposób pliki będziemy nazywali *asemblacjami*, lub (z języka angielskiego) *assembly*⁸. Pliki takie są samoopisywalne, co oznacza, że każdy taki plik zawiera (w części zwanej *manifestem*) informacje o znajdujących się w nim typach (klasach i strukturach)⁹. Stąd w szczególności w przypadku używania bibliotek w kodzie zarządzalnym nie ma potrzeby korzystania z plików nagłówkowych, jak to ma miejsce w przypadku wielu popularnych języków generujący kod niezarządzalny (np. języka C lub C++). Wystarczy dodawać same biblioteki¹⁰.

Komentarze

W języku C# występują takie same rodzaje komentarzy, jak w języku C++.

Komentarzem jest dowolny tekst rozpoczynający się sekwencją znaków „/*” oraz zakończony sekwencją znaków “*/”, np.:

```
/* Pierwszy komentarz */
```

Komentarzem jest też tekst rozpoczynający się od pierwszej pary znaków „//” do końca linii, np.:

```
// Drugi komentarz
```

Szczególne znaczenie mają komentarze rozpoczynające się od trzech (nie tylko dwóch) znaków „//”, gdyż można w nich podawać opisy, które będą umieszczane w automatycznie generowanej dokumentacji, np.:

```
/// <summary>
/// Metoda licząca iloczyn dwóch liczb rzeczywistych
/// </summary>
/// <param name="mnozna">pierwszy czynnik</param>
/// <param name="mnoznik">drugi czynnik</param>
/// <returns>wyliczona wartość iloczynu argumentów</returns>
double Iloczyn(double mnozna, double mnoznik)
{
    return mnozna * mnoznik;
}
```

⁷ Warto zaznaczyć, że nie wszystkie takie pliki można bezpośrednio uruchamiać w systemach Windows – w szczególności nie można bezpośrednio uruchamiać plików usług.

⁸ Przy czym dla uściślenia notacji: moduły będą „fragmentami”, z których może się składać asemblacja, a nie pełnoprawnymi asemblacjami.

⁹ Informacje te można przeglądać m.in. poprzez narzędzie ILDasm.exe (Intermediate Language Disassembler).

¹⁰ Jeśli wkompilujemy elementy bibliotek do programu, to musimy je dodać jako tzw. *referencje*. W typowym sposobie działania programy zwykle wykorzystują wyłącznie biblioteki zarządzalne zarejestrowane w GAC (ang. Global Assembly Cache). W trybie deweloperskim można używać dodatkowo niezarejestrowanych bibliotek umieszczonych w odpowiednio podanej ścieżce. Więcej informacji na temat pojawi się na laboratorium oraz dalszych wykładach.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Można dostrzec tu pewne analogie do rozwiązań używanych w języku Java związań z generowaniem dokumentacji przez *JavaDoc*. Więcej informacji dotyczących generowania dokumentacji pojawi się na laboratorium.

Instrukcje

Ogólne zasady tworzenia instrukcji są podobne jak w językach C, C++ i Java.

Instrukcje tworzymy z wyrażeń przez dodanie do nich na końcu znaku średnika, np.:

```
x = 2 + 1;
System.Console.WriteLine("x = {0}", x);
;
```

(w ostatnim przypadku mamy instrukcję pustą).

Mamy też instrukcję blokową, która grupuje instrukcje

```
{
    int x;
    x = 2 + 1;
    System.Console.WriteLine("x = {0}", x);
}
```

Instrukcje sterujące

Poza nielicznymi wyjątkami instrukcje sterujące w języku C# są takie same jak w językach C, C++ i Java.

Instrukcja warunkowa if / if - else

Składnia oraz działanie instrukcji warunkowej **if** w obu wariantach (**if** oraz **if-else**) są identyczne jak w językach C, C++ i Java:

```
if (wyrażenie)
    instrukcja
```

oraz

```
if (wyrażenie)
    instrukcja
else
    instrukcja
```

np.:

```
if (x < 0) x = -x;
if (x < y)
```



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
{
    var z = x;
    x = y;
    y = z;
}

if (x < y)
    minimum = x;
else
    minimum = y;
```

Instrukcja wyboru switch - case

Składnia oraz działanie instrukcji wyboru *switch-case* są podobne jak w językach C, C++ i Java:

```
switch (wyrażenie)
{
    case wartość:
        instrukcje
        break;
    ...
    case wartość:
        instrukcje
        break;
    default:
        instrukcje
        break;
}
```

Różnica polega na tym, że w języku C# instrukcja skoku **break** obligatoryjnie musi kończyć każdy **niepusty** blok *case* (nawet domyślny). Brak instrukcji skoku jest błędem składniowym (i uniemożliwia komplikację programu). Oto przykłady użycia instrukcji wyboru:

```
switch (x % 3)
{
    case 1:
    case 2:
        podzielosc = false;
        break;
    case 0:
        podzielosc = true;
        break;
}

switch (x % 3)
{
    case 0:
        podzielosc = true;
        break;
    default:
        podzielosc = false;
        break;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

Pętla while

Składnia oraz działanie pętli **while** są identyczne jak w językach C, C++ i Java:

```
while (wyrażenie)
  instrukcja
```

np.:

```
int i = 0, s = 1, n = 7;

while (i < n)
{
    i++;
    s *= i;
}
```

Pętla do – while

Składnia oraz działanie pętli **do-while** są identyczne jak w językach C, C++ i Java:

```
do
  instrukcja
while (wyrażenie);
```

np.:

```
int i = 0, s = 1, n = 7;

do
{
    s *= ++i;
}
while (i < n);
```

Pętla for

Składnia pętli *for* jest identyczna jak w językach C i C++:

```
for (wyrażenie1; wyrażenie2; wyrażenie3)
  instrukcja
```

Działanie tej pętli można przedstawić następującym schematem:

```
wyrażenie1;
while (wyrażenie2)
{
  instrukcja
wyrażenie3;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

np.:

```
int i, s = 1, n = 7;

for (i = 1; i < n; i++)
{
    s *= i;
}
```

Pętla foreach

W języku C# wprowadzono dodatkowo instrukcję sterującą **foreach**, której działanie jest podobne do działania w języku Java wariantu pętli for przeznaczonego dla tablic i kolekcji. Jej składnia jest następująca:

```
foreach (deklaracja zmiennej in lista/tablica/kolekcja itp.)
  instrukcja
```

Instrukcja ta pozwala na wykonanie zadanej instrukcji (tutaj *instrukcja*) dla każdego elementu struktury danych, której elementy można iterować¹¹, np.:

```
string[] tablica = {"cos", "co", "mozna", "iterowac"};

foreach (string s in tablica)
{
    System.Console.WriteLine(s);
}
```

Instrukcje skoków

W języku C# występują instrukcje skoków **break**, **continue** i **goto**, których działanie jest identyczne, jak w języku C. Zwykle używa się ich w połączeniu z instrukcją wyboru *switch—case* (zwłaszcza instrukcje *break* i *goto*) lub instrukcjami pętli (zwłaszcza instrukcja *continue*). Z wymienionych instrukcji sugeruję ograniczyć się (przynajmniej w trakcie nauki programowania) do używania instrukcji *break* w instrukcji wyboru *switch—case*. Jako instrukcje skoków można też traktować instrukcje **return** (zwrócenie wartości przez metodę) i **throw** (wyrzucenie wyjątku), których działanie jest identyczne jak w językach odpowiednio C, C++ i Java oraz C++ i Java i których używanie jest jak najbardziej zalecane. W języku C# nie ma odpowiednika instrukcji skoku **next**.

Deklarowanie zmiennych

¹¹ W tym miejscu przez słowo „iterować” należy rozumieć możliwość pobierania kolejnych elementów struktury danych (np. tablicy lub kolekcji). Używam tego słowa, gdyż może ono wywoływać odpowiednie skojarzenia u osób, które programowały w języku Java. Dla ściśleści jednak warto nadmienić, że C#-owym odpowiednikiem Jav-owych iteratorów są enumeratory, a przez iteratory w języku C# nazywamy składowe klasy pozwalające odwoływać się do jej elementów przez operator [].

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W języku C# obowiązuje standardowy sposób deklaracji zmiennych (występujący w językach C, C++, Java), w którym po nazwie typu (wartościowego lub referencyjnego) podawana jest lista zmiennych, np.:

```
int liczba;
int[] tablica;
double x, y, z;
```

Możliwe jest też inicjalizowanie wartości zmiennych w miejscu deklaracji, np.:

```
int liczba = 1;
int[] tablica = new int[5] { 1, 2, 3, 4, 5 };
double x = 3.5, y, z = 4;
```

Ponadto można używać konstrukcji deklaracji zmiennej z przypisywaniem jej wartości początkowej, w której typ zmiennej ustalany jest na podstawie przypisywanej wartości, np.

```
var liczba = 1;
var tablica3 = new[] { 1, 2, 3, 4, 5 };
var x = 3.5;
```

Deklaracje zmiennych mogą występować w dowolnym miejscu bloku, a nie tylko na jego początku.

Zmienne, którym została przypisana wartość (przy deklaracji lub później) nazywamy **zainicjalizowanymi**. Zmienne przed przypisaniem im wartości są **niezainicjalizowane**. Próba pobrania (odczytu) *niezainicjalizowanej* zmiennej spowoduje błąd komilacji.

Operatory

Operatory w języku C# można pogrupować według ich priorytetów w następującej tabeli

x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked, ->
+, -, !, ^, +x, -x, (T)x, true, false, &, sizeof
*, /, %
+, -
<<, >>
<, >, <=, >=, is, as
==, !=
&
^
&&
? :
=, +=, -=, *=, /=, %=, &=, =, ^=, <=>, ??
=>

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Większość z tych operatorów działa podobnie jak w językach C, C++, Java. Należy zaznaczyć, domyślnie język C# nie pozwala na bezpośrednie operowanie adresami pamięci (podobnie jak w języku Java pamięć jest przydzielana w momencie wejścia do bloku lub tworzenia obiektu i zwalniana przez *garbage collector*¹²). Aby zachować wsteczną zgodność języka C# z językami C i C++ wprowadzono mechanizmy pozwalające operować bezpośrednio na blokach pamięci, jednak należy je jawnie włączyć. Operatorów (np. unarne operatory & i * oraz binarny ->) i metod operujących na adresach pamięci możemy używać tylko i wyłącznie w blokach oznaczonych jako niebezpieczne (**unsafe**). Ponadto kod zawierający taki blok również musi być komplikowany z przełącznikiem pozwalającym na używanie tego rodzaju bloków. Podane przy niektórych operatorach symbole wyjaśniają sposób użycia operatorów (np. x.y oznacza odwołanie do składowej (najczęściej metody, pola lub własności) y obiektu lub klasy x, f(x) wywołuje metodę f z argumentami x, a (T)x rzutuje wartość wyrażenia x na typ T). Spośród operatorów, które nie występowały w językach C i C++ można wyróżnić następujące:

- `typeof` - zwraca typ danej klasy lub obiektu¹³,
- `checked` - włącza sprawdzanie przekroczenia zakresu dla operacji arytmetycznych (dla bloku kodu),
- `unchecked` - wyłącza sprawdzanie przekroczenia zakresu (dla bloku kodu)¹⁴,
- `is` - sprawdza, czy możliwe jest rzutowanie wyrażenia podanego w lewym argumencie na typ podany w prawym argumencie,
- `as` - zwraca wartość rzutowania wyrażenia podanego w lewym argumencie na typ podany w prawym argumencie o ile jest ono możliwe oraz `null` w pozostałych przypadkach,
- `??` - zwraca wartość wyrażenia podanego w lewym argumencie jeśli jest ono różne od wartości `null` oraz wartość wyrażenia podanego w prawym argumencie w pozostałych przypadkach,
- `=>` - tworzy lambda-wyrażenie (metodę anonimową).

Działanie pozostałych operatorów jest podobne¹⁵ jak w językach C i C++, w związku z czym ich opis zostaje pominięty.

Wyjątki

Rola i sposób działania wyjątków są podobne jak w języku Java. Najistotniejszą różnicą jest to, że w języku C# wszystkie wyjątki¹⁶ dziedziczą z klasy `Exception` i nie ma rozgraniczenia na wyjątki które mogą być przechwytywane i które nie mogą być przechwytywane.

¹² Istnieje duża ilość tłumaczeń tego terminu na język polski (m.in. odśmieczacz, zbieracz odpadków, zarządca nieużytków), jednak żadne z nich się nie przyjęło wystarczająco dobrze. Z tego powodu zdecydowałem się na używanie oryginalnej nazwy.

¹³ Więcej o typach będzie w dalszej części tego rozdziału.

¹⁴ Domyślnie sprawdzanie przekroczenia zakresu jest wyłączone.

¹⁵ W szczególności działanie operatora brania reszty zostało rozszerzone na liczby rzeczywiste - np. wyrażenie `7.5%3.1` przyjmie wartość 1.3. Niektóre operatory zostały pogrupowane parami, o czym mowa przy przeciążaniu operatorów.

¹⁶ W specyficzny sposób traktowane są wyjątki, które wystąpią poza kodem zarządzalnym – z racji tego, że występują one poza środowiskiem uruchomieniowym CLR i nie są implementowane w IL (nie występują w hierarchii klas języków zarządzalnych) tylko są utożsamiane z kodem błędów. Od wersji 2.0 .NET Framework funkcjonuje mechanizm opakowywania kodów błędów tych wyjątków w obiekty klasy `Exception`. Więcej informacji na ten temat pojawi się przy temacie poświęconym integracji kodu zarządzalnego z niezarządzalnym (interoperability).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Podobnie jak w języku Java, ogólna konstrukcja kodu z przechwytywaniem wyjątków wygląda następująco:

```
try
{
    ...
}
catch (wyjątek)
{
    ...
}
finally
{
    ...
}
```

Wyjątki wyłapywane są z bloku **try** i przekazywane do właściwego bloku **catch** (podobnie jak w językach C++ i Java może być ich więcej, argument bloku ogranicza dopuszczalne rodzaje wyjątków dla bloku, blok bez argumentu odpowiada dowolnemu rodzajowi wyjątku¹⁷). Ponadto można dodać blok **finally**, który wykonany będzie niezależnie od tego czy w bloku **try** zostanie przechwycony wyjątek, czy też nie. Przykładowo jeśli w programie o następującym kodzie źródłowym

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int x, y, z;
        try
        {
            x = Convert.ToInt32(Console.ReadLine());
            y = 0;
            z = x / y;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Błąd dzielenia przez 0");
        }
        catch (Exception e)
        {
            Console.WriteLine("Błąd \'{0}\'", e);
        }
        finally
        {
            Console.WriteLine("Sprzątanie ...");
        }
    }
}
```

¹⁷ Również z kodu niezarządzalnego, w każdej wersji Frameworka, niezależnie od przełączników.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

wczytana zostanie liczba całkowita, to zostanie przechwycony błąd przekroczenia zakresu, a jeśli wczytanie zostanie ciąg liter, to przechwycony zostanie inny błąd (wyrzucony przy próbie przeprowadzenia konwersji). W obu przypadkach na koniec zostanie wypisany komunikat „Sprzątanie ...”.

Podobnie jak w językach Java i C++ można definiować własne wyjątki (muszą one dziedziczyć z klasy *Exception*) oraz można wyrzucać wyjątki używając słowa kluczowego **throw**.

W przypadkach, w których koniecznie jest nie samo przechwycenie wyjątku, ale wykonanie pewnych operacji porządkowych (zwolnienia zasobów przydzielonych przy tworzeniu konkretnego obiektu) można użyć konstrukcji „*using (...) {...}*”, gdzie w nawiasie okrągłym przeprowadzane jest tworzenie obiektu, który alokuje zasoby, które należy na koniec zwolnić, a w nawiasie klamrowym umieszczony jest blok wykorzystujący ten obiekt. Przykładowo następujący kod może być użyty

```
using (StreamReader reader = new StreamReader(@"C:\Temp\dane.txt"))
{
    string s;
    while ((s = reader.ReadLine()) != null)
    {
        Console.WriteLine("{0}", s);
    }
}
```

zamiast standardowej konstrukcji

```
StreamReader reader = new StreamReader(@"C:\Temp\dane.txt");
try
{
    string s;
    while ((s = reader.ReadLine()) != null)
    {
        Console.WriteLine("{0}", s);
    }
}
finally
{
    reader.Close();
}
```

Tworzenie metod

Ogólne zasady dotyczące składni funkcji (metod) są takie same, jak w językach C, C++ i Java (deklaracja zwracanej wartości (**void** w przypadku braku) i argumentów, zwracanie wartości przez **return** itp.).

Metody ze zmienią ilością argumentów

Jeśli funkcja ma przyjmować zmienną ilość argumentów, to ostatni argument tej funkcji musi być zadeklarowany jako tablica, a deklaracja tego argumentu musi być poprzedzona słowem kluczowym **params** (tego słowa można użyć w tylko w stosunku do ostatniego argumentu), np.:

```
static void Max(params int[] el)
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
{
  int maksimum = Int32.MinValue;

  if (el == null) return maksimum;

  for (int i = 0; i < el.Length; i++)
  {
    if (maksimum < el[i])
    {
      maksimum = el[i];
    }
  }

  return maksimum;
}
```

Metody z domyślnymi argumentami

Od wersji 4.0 wprowadzono do języka C# domyślne argumenty (o składni podobnej jak w języku C++).

Argumenty wejściowe, wyjściowe i referencyjne

Jeśli w deklaracji metody przed argumentem nie stoi słowo kluczowe **out** ani **ref**, to argument ten jest **argumentem wejściowym**, np.:

```
static int Min(int a, int b)
{
  return a < b ? a : b;
}
```

Jeśli w deklaracji metody przed argumentem stoi słowo kluczowe **out**, to argument ten jest **argumentem wyjściowym**, czyli takim, któremu trzeba w ciele metody przypisać wartość oraz którego wartości nie można odczytać (w ciele metody) przed jego zainicjowaniem, np.:

```
static void Min(out int c, int a, int b)
{
  c = (a < b ? a : b);
}
```

Jeśli przed argumentem stoi słowo kluczowe **ref**, to jest on **argumentem referencyjnym**, czyli takim, który może być używany zarówno jako argument wejściowy jak i wyjściowy (to rozwiązanie jest bardzo podobne do przekazywania danych przez *referencję* w języku C++), np.:

```
static void Zamien(ref int a, ref int b)
{
  int c;
  c = a;
  a = b;
  b = c;
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Należy pamiętać o tym, że decydując się na użycie argumentu wyjściowego (odpowiednio referencyjnego) słowo kluczowe **out** (odpowiednio **ref**) należy umieścić zarówno w deklaracji funkcji, jak i w jej wywołaniu (w wywołaniu musi wówczas być podana I-wartość, czyli wyrażenie, któremu może zostać przypisana wartość), np.:

```
int x = 1, y = 2, c;
Zamien(ref x, ref y);
Min(out c, x, y);
```

Przeciążanie metod (i operatorów)

W języku C#, podobnie jak w większości innych języków obiektowych można *przeciążać metody*, czyli tworzyć wiele metod o tej samej nazwie różniących się typami (i/lub liczbą) argumentów¹⁸, np.:

```
static int Dodaj(int a, int b) { return a + b; }
static double Dodaj(double a, double b) { return a + b; }
```

W języku C# można przeciążać operatory, jednak możliwości takiego przeciążania są ograniczone (niektóre operatory, np. „[]” można używać tylko z wybranymi konstrukcjami, operatory ++ i -- implementuje się tylko w jednym wariantie, a część operatorów występuje w parach (np. operatory << i >>, < i >, >= i <=, == i !=). Przykładowo poniższa struktura zawiera przeciążenia kilku operatorów:

```
struct Zespolona
{
    private double re, im;
    public Zespolona(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public Zespolona(double re)
    {
        this.re = re;
        this.im = 0;
    }
    public override string ToString()
    {
        return "(" + re + "," + im + ")";
    }
    public static Zespolona operator +(Zespolona a, Zespolona b)
    {
        return new Zespolona(a.re + b.re, a.im + b.im);
    }
    /* można zrezygnować z poniższych przeciążeń
     * jeśli zdefiniuje się domyślną konwersję
    public static Zespolona operator +(Zespolona a, double b)
    {
```

¹⁸ Nie należy ich mylić z metodami o zmiennej liczbie argumentów, które omawiane były wcześniej.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    return a + new Zespolona(b);
}
public static Zespolona operator +(double a, Zespolona b)
{
    return new Zespolona(a + b.re, b.im);
} */

// definicja domyślnej konwersji
public static implicit operator Zespolona(double a)
{
    return new Zespolona(a);
}

public static Zespolona operator -(Zespolona a, Zespolona b)
{
    return new Zespolona(a.re - b.re, a.im - b.im);
}

// operatory ++ i -- wystarczy raz przeciążyć
public static Zespolona operator ++(Zespolona a)
{
    a.re++;
    return a; // w przypadku klasy należy zmienić na
              // return new Zespolona(a.re, a.im);
}
public static Zespolona operator --(Zespolona a)
{
    a.re--;
    return a; // w przypadku klasy należy zmienić
}

// operatory == i != muszą być przeciążane jednocześnie
public static bool operator ==(Zespolona a, Zespolona b)
{
    return a.re == b.re && a.im == b.im;
}
public static bool operator !=(Zespolona a, Zespolona b)
{
    return a.re != b.re || a.im != b.im;
}

override public bool Equals(object a)
{
    return this.re == ((Zespolona)a).re && this.im == ((Zespolona)a).im;
}
override public Int32 GetHashCode()
{
    return this.re.GetHashCode() + this.im.GetHashCode();
}
}
  
```

Planując użycie przeciążania operatorów w pisany oprogramowaniu należy pamiętać, że często wygodniejszym, bezpieczniejszym i dającym większe możliwości rozwiązań (od przeciążania operatorów) jest implementowanie interfejsów.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Modyfikatory dostępu

W języku C# mamy do czynienia z pięcioma rodzajami modyfikatorów dostępu:

- **public**,
- **private**,
- **protected**,
- **internal**,
- **internal protected** lub **internalprotected**.

Sposób działania pierwszych trzech modyfikatorów z wymienionych powyżej jest analogiczny jak w językach C++ i Java, tzn. **public** pozwala na dostęp do elementu klasy (lub struktury) „dla wszystkich”, **private** pozwala na dostęp do elementu klasy tylko z metod tej klasy (lub struktury), natomiast **protected** pozwala na dostęp do elementu klasy (lub struktury) tylko z jej metod oraz metod klas (lub struktur) pochodnych (patrz też – uwagi dotyczące dziedziczenia). Działanie modyfikatorów dla samych klas (oraz struktur) jest analogiczne.

Domyślnym modyfikatorem (używanym w przypadku, gdy modyfikator nie jest podany wprost) jest modyfikator **internal**. Pozwala on dostęp do klasy (lub struktury) lub jej elementu z klas (lub struktur) znajdujących się w tym samym **assembly**, czyli w tym samym (skompilowanym) pliku. W szczególności w przypadku biblioteki DLL z jej klas lub ich składników posiadających domyślny modyfikator dostępu (**internal**) można korzystać tylko w obrębie tej biblioteki. Można w tym miejscu dostrzec pewną analogię z domyślnym modyfikatorem dostępu z języka Java, który umożliwia na dostęp w obrębie *pakietu*. Należy jednak pamiętać, że w języku C# dostęp dotyczy **assembly**, a nie *przestrzeni nazw*.

Dodatkowo w języku C# został wprowadzony modyfikator składający się z dwóch słów kluczowych „**internal protected**” oraz jego alternatywna wersja powstała przez połączenie tych słów „**internalprotected**”. Elementów oznaczonych tym modyfikatorem można używać w klasach (odpowiednio strukturach), które są klasami pochodnymi **lub** znajdują się w tym samym **assembly**.

Dziedziczenie

W języku C#, podobnie jak w Javie klasa może dziedziczyć (taka klasa jest nazywana klasą pochodną) po co najwyżej jednej klasie (taka klasa jest nazywana klasą bazową). Nie ma wielodziedziczenia po klasach. Natomiast klasa (lub interfejs) mogą dziedziczyć po dowolnej liczbie interfejsów. W przeciwieństwie do Javy do wskazania klasy i/lub interfejsów bazowych nie używa się słów kluczowych **extends** ani **implements**, tylko, podobnie jak w języku C++, wypisuje się taką klasę i/lub interfejsy po znaku dwukropka.

Przypomnijmy, że w języku Java wszystkie metody były wirtualne. W języku C# do wskazania metody wirtualnej należy użyć słowa kluczowego **virtual**. Ponadto przy implementacji metody wirtualnej należy użyć słowa kluczowego **override**. Brak tego słowa traktowany jest jako usterka składniowa i powoduje utworzenie **nowej** metody o tej samej nazwie, która przykryje poprzednio utworzoną metodę wirtualną (będą do dwie różne metody o tych samych nazwach). Jeśli istotnie chcemy utworzenia nowej metody przykrywającej inną metodę, to należy w jej deklaracji użyć słowa kluczowego **new**. Metody będące implementacjami metod wirtualnych (z użyciem słowa **override**) są nadal metodami wirtualnymi

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

i w kolejnych klasach pochodnych można je ponownie implementować używając słowa **override** (o ile tylko metody te lub klasa je zawierająca nie zostały zapieczętowane).

W języku C#, podobnie jak w języku Java można używać klas i metod abstrakcyjnych. Przypomnijmy, że nie można tworzyć obiektów klas abstrakcyjnych. Klasa takie służą głównie do tworzenia klas pochodnych. Ponadto możemy używać zmiennych referencyjnych wskazujących na klasy abstrakcyjne¹⁹. Analogicznie, metody abstrakcyjne to takie metody, których nie można bezpośrednio używać z klasy w której zostały zaimplementowane. W przeciwnieństwie do języka Java, w języku C# klasa abstrakcyjna może być klasą pochodną klasy, która nie jest klasą abstrakcyjną. Do tworzenia klas (odpowiednio metod) abstrakcyjnych służy słowo kluczowe **abstract**.

Jeżeli nie chcemy dziedziczyć po pewnej klasie (lub odpowiednio implementować metody wirtualnej) to możemy taką klasę zapieczętować używając w miejscu jej deklaracji słowa kluczowego **sealed**. Mechanizm pieczętowania jest konstrukcją analogiczną do klas (odpowiednio metod) finalnych (oznaczanych słowem kluczowym **final**) w Javie. Nie można stworzyć nowej implementacji zapieczętowanej metody wirtualnej, ale można przykryć taką metodę nową metodą o tej samej nazwie.

Właściwości (*Properties*)

Właściwości (ang. *properties*) są specyficznymi dla języka C# elementami klas, do których odwoływać można się podobnie jak do pól, a które oprogramowuje się podobnie jak metody. Innymi słowy łączą one możliwości metod (w kwestii implementacji np. mechanizmów bezpieczeństwa) z prostotą użycia pól. Z tego powodu często używane są do odczytania wartości prywatnego pola lub przypisania mu wartości z jednokrotnym sprawdzeniem, czy wartość ta jest dopuszczalna. W takich sytuacjach, w języku Java trzeba zdefiniować stosowną funkcję (lub dwie).

Definicja właściwości składa się z typu (który dodatkowo może być poprzedzony modyfikatorem dostępu), nazwy oraz ciała, w którym pojawić mogą się bloki **set** oraz **get** lub tylko jeden z tych bloków. Ponadto, jeśli w ciele występują oba te bloki, to jeden z nich może być poprzedzony modyfikatorem dostępu. Blok **set** jest odpowiedzialny za pobieranie przez właściwość wartości, która jest jej przekazywana w przez słowo kluczowe **value**. Natomiast blok **get** służy wskazaniu zwracanej przez właściwość wartości, która jest podawana za słowem kluczowym **return**. Jeśli właściwość ma zdefiniowany tylko blok **get**, to nie można jej przypisać wartości. Analogicznie, jeśli posiada ona tylko blok **set**, to nie można odczytać jej wartości. Poniżej zobaczyć można przykład właściwości Zmienna przypisującej oraz odczytującej wartość zmiennej prywatnej zmienna.

```
private int zmienna;

public int Zmienna
{
    get;
    {
        return zmienna;
    }
}
```

¹⁹ Przykładowo (patrz temat dotyczący operacji w systemach plików) mamy klasę abstrakcyjną *FileSystemInfo* implementującą węzły w systemie plików oraz jej klasy pochodne *FileInfo* oraz *DirectoryInfo* implementującą węzły będące odpowiedzią (zwykłymi) plikami oraz katalogami. Zmiennej referencyjnej typu *FileSystemInfo* można przypisywać obiekty zarówno klas *FileInfo*, jak i *DirectoryInfo*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
private set
{
    if (zmienna < 0) throw new ArgumentException();
    zmienna = value;
}
```

Warto nadmienić, że można również zdefiniować właściwość, w której bloki **set** i **get** nie będą posiadać ciała, np.:

```
public int Zmienna { get; private set; }
```

Wówczas podczas komplikacji automatycznie zostanie utworzone pole, które będzie przechowywało wartość tej właściwości, a właściwość ta będzie pozwalała na dostęp do wartości tego pola zgodnie z jej modyfikatorami dostępu.

Indeksery

W języku C# używanie operatora „[]” ograniczone jest zwykle do tablic, indeksów i atrybutów²⁰. Stąd najprostszym sposobem umożliwienia odwoływania np. do wartości danych przechowywanych w implementowanej klasie kolekcji jest zdefiniowanie dla niej *indeksera*. Indeksery definiuje się podobnie, jak właściwości. Różnica polega na tym, że w deklaracji indeksera zamiast nazwy właściwości umieszcza się słowo kluczowe **this**, po którym występuje nawias kwadratowy „[]” zawierający deklarację argumentu będącego indeksem (typ i nazwę zmiennej argumentu – podobnie jak w przypadku argumentu funkcji), po którym indeksowane mają być wartości (nic nie stoi na przeszkodzie, aby indeksować wartości np. łańcuchami zamiast liczbami naturalnymi). W implementacji bloków **set** i **get** można używać zadeklarowanego w ten sposób argumentu – będzie on miał przypisaną wartość aktualnego indeksu przekazanego przez indeksera, np.:

```
struct Bool32
{
    private Int32 liczba;
    public Bool32(Int32 init)
    {
        liczba = init;
    }
    public bool this[int i]
    {
        get
        {
            if (i < 0 || i >= 32)
                throw new ArgumentOutOfRangeException("indeks");
            return ((liczba >> i) & (Int32)1) == 1;
        }
        set
        {
```

²⁰ oraz wskaźników, które jak wspomniano wcześniej można używać wyłącznie we fragmentach kodu oznaczonych jako *unsafe*.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        if (i < 0 || i >= 32)
            throw new ArgumentOutOfRangeException("indeks");
        if (value)
            liczba |= ((Int32)1) << i;
        else
            liczba &= ~((Int32)1) << i;
    }
}
```

Delegacje

W języku C# nie można używać wskaźników²¹, więc w szczególności nie używa się w nim wskaźników do funkcji. Ich rolę przejęły *delegacje* (zwane też czasami *delegatami*). Można je traktować jako pewne uogólnienie wskaźników na funkcję, ponieważ można im „przypisywać” więcej niż jedną metodę²².

Delegacjom można przypisywać tylko metody typu zgodnego z „typem” delegacji. Do zdefiniowania typu delegacji służy słowo kluczowe **delegate**, np.:

```
delegate void delegacja();
delegate int operacja(int arg1, int arg2);
```

Delegacje deklarujemy podobnie, jak inne zmienne, np.:

```
private delegacja instancjaDelegacji, drugaInstanacja;  
private operacja dodawanie;
```

Możemy utworzyć instancję delegacji podając w konstruktorze stosowną metodę, którą chcemy jej przypisać. Delegacjom możemy też wprost przypisać odpowiednie metody. Zostanie wówczas automatycznie utworzona instancja delegacji i przeprowadzone odpowiednie rzutowanie.

```
private static void nic() { }
static void cos() { Console.WriteLine("Coś"); }

...
instancjaDelegacji = new delegacja(nic);
drugaInstanacja = nic;
```

Dla delegacji zostały przeciążone operatory `+=` oraz `-=`, które pozwalają na odpowiednio podpinanie metody podanej w prawym argumencie do delegacji podanej w lewym argumencie oraz odpinanie metody podanej w prawym argumencie od delegacji podanej w lewym argumencie, np.:

```
instanciaDelegacji += cos;
```

²¹ Poza fragmentami kodu oznaczonymi jako *unsafe*.

²² Należy jednak pamiętać, że jeśli typ delegacji zwraca wartość, to przy podpięciu więcej niż jednej metody po wykonaniu delegacji otrzymany wartość zwróconą przez ostatnią z wywołanych metod (zwykle jest to metoda, która została dodana jako ostatnia).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
instancjaDelegacji == nic;
```

Delegacjom możemy przypisywać również tzw. *metody anonimowe* (zwane też czasem *lambda-wyrażeniami*), czyli definiowane w miejscu przypisania. Metody takie można tworzyć z użyciem operatora => lub słowa kluczowego **delegate**, np.:

```
instancjaDelegacji +=  
    delegate  
    {  
        Console.WriteLine("Metoda anonimowa 1");  
    };  
instancjaDelegacji +=  
    () => {  
        Console.WriteLine("Metoda anonimowa 2");  
    };
```

Aby wywołać metody podpięte do delegacji używamy w stosunku do delegacji takiej samej składni, jaką jest używana w stosunku do metod, np.:

```
this.instancjaDelegacji();
```

Poprzez delegacje można m.in. przekazywać metodę, z którą może być uruchomiony nowy wątek lub wskazywać, które metody mają być wykonywane w reakcji na zadane zdarzenie. Delegacje mogą też służyć do wykonywania wielu specyficznych operacji z użyciem podpiętych pod nie metod (np. uruchamiania asynchronicznego metod, o którym będzie mowa w późniejszej części wykładu). Po więcej informacji na temat delegacji odsyłamy czytelnika do dokumentacji systemowej języka [23].

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

2. Typy

Ten rozdział poświęcony jest przypomnieniu podstawowych informacji dotyczących typów danych w programowaniu obiektowym oraz wskazaniu specyficznych dla języka C# rozwiązań dotyczących implementacji typów.

Podstawowe właściwości typów języku C#

C# jest językiem (*silnie*) *typizowanym* (ang. *strongly typed*). Polega to na tym, że każda zmienna²³ w momencie deklaracji musi mieć zdefiniowany swój typ. Zaletami takiego podejścia są nieco większa wydajność (nie potrzeba przy każdej operacji dopasowywać typów) oraz mniejsze ryzyko popełnienia błędów (błędy polegające na przypisaniu danych niezgodnych typów mogą być wykrywane na bardzo wczesnym etapie przez kompilator lub nawet samo środowisko programistyczne w trakcie edycji kodu).

Przypomnijmy, że C# jest *obiektowym* językiem programowania, o hierarchicznej strukturze klas. Wszystkie klasy dziedziczą w nim z klasy **object**. Co więcej, każdy typ (z wyjątkiem typów interfejsów) jest typem pochodnym w stosunku do typu **object**²⁴.

W języku C# mamy mechanizm zwany *pudełkowaniem* (ang. *boxing*) pozwalający na przypisanie zmiennej referencyjnej wartości typu wartościowego. Polega on na opakowywaniu wartości typu wartościowego w „otoczkę” obiektu umieszczonego na stercie i zwracaniu referencji do tego obiektu. Dzięki tej metodzie tablicom oraz kolekcjom składającym się z elementów typu **object** można przypisywać dowolne wartości. Wadami takiego rozwiązania są między innymi mała wydajność tej metody oraz złożoność metody odwrotnej (ang. *unboxing*), która wymaga jawnego rzutowania zwracanej wartości na odpowiedni typ. Alternatywnym rozwiązaniem (do pudełkowania) jest korzystanie z *typów generycznych*.

Kryteria podziału typów

Należy pamiętać, że można stosować wiele różnych kryteriów podziału typów.

W szczególności ze względu na **sposób umieszczenia danych w pamięci** typy możemy podzielić na **typy wartościowe**, których wartości przechowywane są na stosie oraz **typy referencyjne**, w przypadku których na stosie umieszczane są jedynie referencje do wartości umieszczanych na stercie.

Ze względu na sposób **konstrukcji** typy możemy podzielić na **typy proste (pierwotne, prymitywne)**, oraz **typy złożone**.

Możemy też podzielić typy względem na **miejsce zdefiniowania** na typy **systemowe (wbudowanie)** oraz typy zdefiniowane przez użytkownika.

W dalszej części skupimy się głównie na pierwszym sposobie podziału

²³ Z wyjątkiem (statycznych) zmiennych dynamicznych wprowadzonych w wersji 4.0 .NET Frameworka.

²⁴ Dotyczy to również typów wartościowych – dziedziczą one z abstrakcyjnej klasy *ValueType*, która jest klasą pochodną klasy **object**, o czym będzie się można przekonać w trakcie wykonywania zadań z zagadnień związanych z refleksjami.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Porównanie typów wartościowych i referencyjnych

W celu przypomnienia poniższej tabelce zebrane zostały podstawowe różnice pomiędzy typami wartościowymi i typami referencyjnymi (informacje te powinny być znane ze wcześniejszych zajęć).

Typy wartościowe	Typy referencyjne
dane umieszczane na stosie	dane umieszczane na stercie
przechowywanie aktualnej wartości danych	na stosie umieszczane są wyłącznie referencje („adresy”) do danych
miejsce przydzielane jest automatycznie	wymagają przydzielenia (alokacji) miejsca na dane
zmienne są parami niezależne	wiele zmiennych może wskazywać na te same dane
przykładami w języku C# są: proste typy wbudowane (np. int , float), typy wyliczeniowe, struktury	przykłady: klasy, interfejsy, tablice, kolekcje

Rzutowanie typów

W języku C# można używać rzutowania operatorem „()”, który działa tak samo jak w językach C, C++ i Java („(typ) wartość”). Jeśli nie jest możliwe przeprowadzenie operacji rzutowania, to operacja taka wygeneruje wyjątek, który należy obsłużyć, np.:

```

object o;
int i = 12;
int a, b;
int? c;

// o = "123";
o = i;

try
{
    a = (int)o;
}
catch
{
    a = default(int);
}
  
```

Lepszym rozwiązaniem jest w tym przypadku użycie operatora **is**, dzięki któremu można sprawdzić czy rzutowanie, które ma być wykonane jest możliwe, np.:

```

if (o is int)
{
    b = (int)o;
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
else
{
    b = default(int);
}
```

Można również użyć operatora **as**, który

- ile rzutowanie jest możliwe, wykonuje je i zwraca jego wynik,
- w przeciwnym wypadku zwraca wartość **null** (używany typ musi ją dopuszczać),

np.:

```
c = o as int?;
Console.WriteLine("{0} {1} {2} {3}", o, a, b, c);
```

Porównanie klas i interfejsów

Zakładając, że pojęcia zarówno klasy jak i interfejsu są znane (zakładana jest ich znajomość przynajmniej w odniesieniu do języka Java), porównamy czym różnią się one w języku C#.

Na początku zwróćmy uwagę na to, jakie elementy składowe mogą one zawierać:

Elementy składowe klas

- pola
- metody
- własności (properties)

Elementy składowe interfejsów

- zdarzenia (bardzo szczególny rodzaj pól)
- **deklaracje** metod
- własności (properties)

Jak widać ze względu na różnice w dostępnych konstrukcjach porównanie to nieco różni się od znanego z języka Java. Jednak już pod względem funkcjonalnym porównanie klas i interfejsów wygląda tak samo:

Klasy

- elementy klasy służą do opisu stanu i zachowania obiektów
- klasy mogą dziedziczyć po klasach i interfejsach
- nie można wielodziedziczyć po klasach

Interfejsy

- elementy interfejsu opisują funkcjonalności
- metody interfejsów mogą być implementowane wyłącznie w klasach
- interfejsy mogą dziedziczyć wyłącznie po interfejsach
- możliwe jest wielodziedziczenie po interfejsach

Porównanie struktur i klas

W języku C# struktury mogą mieć wiele cech, które w języku C++ zarezerwowane były wyłącznie dla klas. W szczególności mogą posiadać nie tylko pola, ale również metody i własności, a nawet własne konstruktory. Podstawową różnicą pomiędzy strukturami i klasami jest to, że struktury są typami

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

wartościowymi, podczas gdy klasy są typami referencyjnymi. Większość pozostałych różnic jest jej konsekwencją. W poniższej tabelce zebrane zostały najważniejsze z tych różnic.

Struktury	Klasy
są typami wartościowymi	są typami referencyjnymi
instancje nazywane są <i>zmiennymi wartościowymi</i> i umieszczane na stosie	instancje nazywane są <i>obiektami</i> i umieszczane na stercie
nie można definiować własnych domyślnych konstruktorów	można definiować własne domyślne konstruktory
po zadeklarowaniu własnego konstruktora kompilator nadal tworzy domyślny konstruktor	po zadeklarowaniu własnego konstruktora kompilator nie tworzy domyślnego konstruktora
pola nie zainicjalizowanie w konstruktorze nie są automatycznie inicjalizowane	pola nie zainicjalizowanie w konstruktorze są automatycznie inicjalizowane (domyślnymi wartościami)
nie można inicjalizować pól instancji w miejscu deklaracji	można inicjalizować pola instancji w miejscu deklaracji

Wbudowane typy proste a struktury

Przypomnijmy, że w języku Java prostym typom wartościowym takim jak *int*, *double* czy *float* odpowiadały ich referencyjne odpowiedniki pisane z dużych liter - tu *Int*, *Double* oraz *Float*, które „opakowywały” wartość do postaci obiektu. W języku C# sytuacja wygląda zupełnie inaczej. Pisane małymi literami typy proste są w rzeczywistości **synonimami** (czyli są z nimi tożsame) odpowiadających im (pisanych z dużej litery) typów z przestrzeni System. Co więcej, typy proste są albo strukturami albo klasami.

Zgodnie ze specyfikacją mamy następujące przyporządkowanie typów prostych

Typ prosty	Synonim	Rodzaj
bool	System.Boolean	struktura
byte	System.Byte	struktura
decimal	System.Decimal	struktura
double	System.Double	struktura
float	System.Single	struktura
Int	System.Int32	struktura
long	System.Int64	struktura

Typ prosty	Synonim	Rodzaj
object	System.Object	klasa
sbyte	System.SByte	struktura
short	System.Int16	struktura
string	System.String	klasa
uint	System.UInt32	struktura
ulong	System.UInt64	struktura
ushort	System.UInt16	struktura

Typy mogące przyjmować wartość null

W języku C# większość wbudowanych wartościowych typów prostych posiada swoje odpowiedniki **będące również typami wartościowymi**, które mogą przyjmować wartość **null**. Typy te mają nazwy

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

utworzone z odpowiedniego typu prostego przez dodanie znaku „?”, np. **int?**, **float?**. Alternatywnym sposobem deklaracji tych typów jest skorzystanie z generycznego typu **System.Nullable<>**.

W języku C# występuje też dwu-argumentowy operator „??”, który zwraca wartość pierwszego argumentu, o ile posiada on wartość różną od „null” oraz wartość drugiego argumentu w przeciwnym wypadku.

Przykład:

```
using System;
class Program
{
    static void Main()
    {
        string s = null;
        int? x = null; // lub "System.Nullable<int> x = null;" 
        int y = x ?? default(int);
        Console.WriteLine("{0}, {1}", y, s ?? "Łańcuch pusty");
        Console.WriteLine(typeof(int?).IsValueType);
        Console.WriteLine(typeof(Nullable<double>).IsValueType);
        // ale poniższa linijka spowodowałaby błąd podczas
        // działania programu (Runtime Error):
        // Console.WriteLine(x.GetType().IsValueType);
    }
}
```

Typy wyliczeniowe

W języku C#, podobnie jak w językach C, C++ i Java możemy używać typów wyliczeniowych. Ich sposób deklaracji oraz działanie są takie same jak w językach C++ i Java (w języku C ich funkcjonalność była mocno ograniczona), np.:

```
enum Litery1 { A, B, C, D, E };
enum Litery2 { A = 5, B = 0, C = 3 };

class Program
{
    static void Main(string[] args)
    {
        Litery1 litera = Litery1.A;
        System.Console.WriteLine(litera);
    }
}
```

Przypomnijmy, że typy wyliczeniowe są typami wartościowymi, które przyporządkowują etykietom wartości całkowitoliczbowe. Sposób przypisywania wartości poszczególnym etykietom w typie wyliczeniowym przebiega zgodnie z poniższym algorytmem:

- jeśli występuje jawne przypisanie wartości, to przyjmij przypisywaną wartość
- w przeciwnym wypadku:
 - jeśli jest to pierwsza etykieta, to przypisz wartość 0,

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- a jeśli nie, to przypisz wartość o 1 większą, niż wartość przypisana poprzedniej etykiecie.
- W ramach samodzielnego zadania można zastanowić się, co wypisze w pierwszej linijce poniższy program:

```
using System;
class Program
{
    enum Litery { A, B, C, D = 1, E };
    static void Main(string[] args)
    {
        Console.WriteLine(Litera.E);
        Type typLitera = typeof(Litera);
        foreach (string s in Enum.GetNames(typLitera))
            Console.WriteLine("{0} = {1}", s,
                Convert.ToInt32(Enum.Parse(typLitera, s)));
    }
}
```

Statyczne pola i metody

Podobnie jak w języku Java, w języku C# możemy używać **statycznych** klas, pól i metod.

Podobnie jak w we wspomnianym języku statyczne składowe klas (pola, metody i własności) deklarowane są słowem ze słowem **static**. Statyczne składowe klasy (i stałe pola) wywoływane są z klasy (nie są one związane z konkretnymiinstancjami (klasy)). Pozostałe pola, metody i własności wywoływane są z obiektów (są one związane z konkretnymiinstancjami klas), np.:

```
public class Klasa
{
    public const int Max = 1024;
    public static int GlobalMax = 1024;
    public int Value = 1;

    public static int GetMax()
    {
        return Klasa.GlobalMax;
    }
    public int GetLocalValue()
    {
        return Value;
    }
}

int a = Klasa.Max;
int b = Klasa.GlobalMax;
int c = Klasa.GetMax();

Klasa kl = new Klasa();
int d = kl.Value;
int e = kl.GetLocalValue();
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Możemy też utworzyć statyczną klasę, poprzedzając jej deklarację słowem **static**. Klasa statyczna może posiadać wyłącznie statyczne składowe. Ponadto nie można tworzyć instancji klasy statycznej.

Od wersji 4.0 Frameworka w języku C# wprowadzono też statyczne pola bez kontroli typów deklarowane słowem **dynamic**.

Stałe i zmienne tylko do odczytu

Aby dane pole lub zmienną lokalną zadeklarować jako **stała** należy przy jej deklaracji użyć słowa kluczowego **const**. Takie pole lub zmienną należy zainicjować przy deklaracji, np.:

```
const int width = 1024;
public const string nazwa = "Jakiś tekst";
```

Aby dane pole lub zmienną lokalną zadeklarować jako **tylko do odczytu** należy w jej deklaracji użyć słowa kluczowego **readonly**. Osobom znającym język Java powinna wystarczyć informacja, że tego typu pola są odpowiednikami *pól finalnych* z Javy. Różnica pomiędzy polami tylko do odczytu i stałymi jest następująca:

- pole opatrzone słowem kluczowym **const** można inicjalizować wyłącznie przy deklaracji pola,
- pole opatrzone słowem kluczowym **readonly** można inicjalizować zarówno przy deklaracji pola, jak i wewnątrz konstruktora.

Tablice

W języku C# można używać typów tablicowych. Typ tablicowy tworzy się z zadanego typu przez dodanie nawiasów klamrowych „[]”²⁵. Typy tablicowe są typami referencyjnymi, zatem przed użyciem tablicy należy utworzyć jej instancję, np.:

```
int[] tablica;
tablica = new int[4];
```

Jak widać na powyższym przykładzie, instancję tablicy tworzymy używając słowa kluczowego **new**, po którym podaje się typ elementów tablicy, a następnie w nawiasie kwadratowym rozmiar tworzonej tablicy (lub rozmiary po poszczególnych współrzędnych w przypadku tablic wielowymiarowych). Używając operatora „[]” możemy odwoływać się do poszczególnych elementów utworzonej w powyższy sposób tablicy, np.:

```
for (int i = 0; i < tablica.Length; i++)
{
    tablica[i] = i*i;
}
for (int i = 0; i < tablica.Length; i++)
{
    Console.WriteLine("tablica[{0}] = {1}", i, tablica[i]);
```

²⁵ w przypadku tablic wielowymiarowych dodatkowo umieszcza się wewnątrz tych nawiasów odpowiednią liczbę przecinków.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

W podanym przykładzie używana jest własność **Length** zwracająca rozmiar utworzonej tablicy.

Tablice możemy inicjalizować początkowymi wartościami przy tworzeniu jej instancji. Można to wykonać też w miejscu deklaracji. Jeśli inicjalizujemy tablicę wartościami tego samego typu, to możemy pominąć niektóre elementy składni (są one ustalane na podstawie przypisywanych wartości), np.:

```
int[] tablica2 = new int[5] { 1, 2, 3, 4, 5 };
var tablica3 = new[] { 1, 2, 3, 4, 5 };
int[] tablica4 = { 1, 2, 3, 4, 5 };
```

W podobny sposób możemy też utworzyć tablicę zawierającą *obiekty anonimowe* (czyli obiekty klasy, która nie została zdefiniowana i której struktura jest ustalana na podstawie przypisywanych wartości), np.:

```
var studenci = new[] {
  new { Imie = "Jan",
        Nazwisko = "Kowalski" },
  new { Imie = "Anna",
        Nazwisko = "Nowak" }
};
```

W tym przypadku każdy obiekt anonimowy musi posiadać taką samą liczbę pól, pola te muszą mieć takie same nazwy i pola o tych samych nazwach muszą być tych samych typów. W szczególności poniższy kod

```
var studenci = new[] {
  new { Imie = "Jan",
        Nazwisko = "Kowalski",
        Wiek = 23 },
  new { Imie = "Anna",
        Nazwisko = "Nowak" }
};
```

jest niepoprawny, gdyż pierwszy obiekt posiada pole, które nie występuje w drugim obiekcie. Aby móc umieścić w tablicy obiekty (lub ogólniej dane) różnych typów, należy zadeklarować tablicę w ten sposób, aby typ jej elementów był typem bazowym dla wszystkich elementów, które mają być w niej umieszczone, np.:

```
var rozne = new object[] {
  new { Imie = "Jan", wiek = 23,
        Nazwisko = "Kowalski" },
  new { Imie = "Anna",
        Nazwisko = "Nowak" },
  new { Przedmiot = "Algebra",
        Godzin = "60" },
  "jakis tekst", 5, 4.01
};
```

Tablice są implementowane z użyciem abstrakcyjnej klasy **System.Array**.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W języku C# w przeciwieństwie do języków C, C++ i Java można tworzyć tablice „w pełni” wielowymiarowe (które nie są tablicami tablic), np.:

```
int[,] tablica2wym = new int[2,3] { { 1, 2, 3 }, { 4, 5, 6 } } ;
int[, ,] tablica3wym = new int[2,3,2];
```

Do tworzenia instancji tablic można też użyć publicznej metody *CreateInstance* z klasy *System.Array*. Metoda ta posiada wiele przeciążeń. W szczególności można przy jej pomocy stworzyć tablicę wielowymiarową o rozmiarach zawartych w przekazanej tablicy jednowymiarowej, np.:

```
int[] lengths = new int[2] { 2, 3, 4 } ;
Array myArray = Array.CreateInstance(typeof(int), lengths);
```

oraz tablice, których indeksy nie zaczynają się od zera, np.:

```
int[] lengths = new int[2] { 2, 3, 4 } ;
int[] lowerBounds = new int[2] { 1, 2, 3 } ;
Array myArray = Array.CreateInstance(typeof(String), lengths, lowerBounds);
```

Nie można zmieniać rozmiaru utworzonej instancji tablicy. Jeśli potrzeba użyć tablicy innego rozmiaru to należy utworzyć nową instancję tablicy i skopiować do niej stosowne wartości.

Do kopiowania zawartości tablicy (poza typową metodą polegającą na utworzeniu nowej tablicy i kopiowaniu po kolej i do niej elementów) można użyć metody *CopyTo()*, metody statycznej *Copy()* z klasy *System.Array* oraz metody *Clone()*, np.:

```
int[] tab = { 1, 2, 3, 4 } ;
int[] kopial = new int[tab.Length];
tab.CopyTo(kopial, 0);
int[] kopia2 = new int[tab.Length];
Array.Copy(tab, kopia2, tab.Length);
int[] kopia3 = (int[]) tab.Clone();
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

3. Kolekcje

W rozdziale tym przedstawione będą złożone typy służące do przechowywania danych, jakimi są kolekcje. Ich ogólna koncepcja, sposób działania oraz własności są podobne jak w przypadku Javy.

Porównanie tablic i kolekcji

Przypomnijmy, że **tablice** są zdefiniowane w klasie *System.Array*. Rozmiar tablicy jest ustalany w momencie utworzenia i nie może być (dynamicznie) zmieniany. Ponadto tablice mogą przechowywać dane tylko jednego typu, podanego przy ich deklaracji.

Kolekcje są osobnymi klasami²⁶. Większość z nich (przynajmniej jeśli chodzi o kolekcje dostarczane w standardowych dystrybucjach środowiska) zdefiniowana jest w przestrzeni *System.Collections* i jej podprzestrzeniach. Kolekcje zapewniają większą kontrolę nad sposobem przechowywania danych oraz dostępem do nich niż tablice. Ponadto zwykle można dodawać oraz usuwać elementy (dane) z kolekcji (często na różne sposoby). Często kolekcje mogą przechowywać dane różnych typów.

Podział kolekcji

Podobnie jak w języku Java kolekcje zasadniczo dzielimy na dwa rodzaje: **kolekcje niegeneryczne** oraz **kolekcje generyczne**. Ponadto wśród kolekcji generycznych możemy wyróżnić specjalną kategorię zwaną **wyspecjalizowanymi kolekcjami**.

W tej części wskazane zostaną charakterystyczne cechy wspomnianych rodzajów kolekcji (z podaniem przykładów).

Kolekcje niegeneryczne

Nie generyczne kolekcje są historycznie wcześniejsze i były dołączone do pierwszych wersji framework'a. Przechowują one dane jako obiekty klasy **System.Object**. Nie zapewnia to kontroli typów, wymaga jawnego rzutowania przy odczytywaniu wartości oraz powoduje konieczność pudełkowania w przypadku korzystania z typów wartościowych.

Przykładami popularnych kolekcji niegenerycznych są następujące klasy:

- **ArrayList** - klasa implementująca tablicę o zmiennej liczbie elementów (można dynamicznie zwiększać lub zmniejszać rozmiar, jest to odpowiednik listy);
- **Queue** - klasa implementująca kolejkę (jej elementy są usuwane zgodnie z koncepcją *FIFO* - *First In First Out*);
- **Stack** - klasa implementująca stos (jej elementy są usuwane zgodnie z koncepcją *LIFO* - *Last In First Out*);
- **Comparer** - klasa służąca do porównywania obiektów;

²⁶ W szczególnych przypadkach strukturami.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Hashtable** - klasa implementująca tablicę skojarzeniową, w kolekcji tej umieszczane są pary „klucz – wartość”, do porządkowania elementów używane są funkcje skrótu kluczy;
- **SortedList** - kolekcja składająca się z par „klucz – wartość” posortowanych po kluczach;
- **BitArray** - kolekcja przechowująca tablicę bitowych wartości reprezentowanych przez wartości **true** i **false** wspierająca operacje bitowe na danych.

Kolekcje generyczne

Tworząc nową instancję typu generycznego określa się, jakiego typu dane mają być w niej przechowywane. Kolekcje te pozwalają korzystać zarówno z typów referencyjnych, jak i typów wartościowych. Ponieważ kolekcje takie przechowują dane tylko jednego typu, podczas operacji na danych w naturalny sposób zachowane jest bezpieczeństwo typów (ang. *type safety*). Dzięki temu odczyt danych nie wymaga rzutowania, a ponadto dla typów wartościowych nie trzeba przeprowadzać pudełkowania. Zwiększa to wydajność przeprowadzanych operacji oraz zmniejsza ryzyko popełnienia błędów podczas tworzenia kodu.

Przykładami popularnych kolekcji generycznych są następujące klasy:

- **Dictionary, SortedList, SortedDictionary**, - klasy implementujące tablicę lub listę par „klucz – wartość” o zadeklarowanych przy tworzeniu typach (zarówno dla klucza jak i „wartości”), w których dane są sortowane względem kluczy;
- **List** - klasa implementująca listę składającą się z danych określonego przy inicjalizacji typu;
- **Queue** - klasa implementująca kolejkę (jej elementy są usuwane zgodnie z koncepcją *FIFO - First In First Out*) składającą się z danych tego samego typu;
- **Stack** - klasa implementująca stos (jej elementy są usuwane zgodnie z koncepcją *LIFO - Last In First Out*) składający się z danych tego samego typu;
- **LinkedList** - klasa implementująca listę dwukierunkową składającą się z danych określonego przy inicjalizacji typu.

Porównanie popularnych kolekcji niegenerycznych i generycznych

Jak widać wiele typowych struktur danych ma swoje reprezentacje zarówno w postaci kolekcji generycznych, jak i generycznych. Poniżej podajemy zestawienie dzielące wymienione powyżej kolekcje ze względu na sposób reprezentacji danych:

Rodzaj struktury danych	Kolekcje niegeneryczne	Kolekcje generyczne
Lista	ArrayList	List
Lista dwukierunkowa		LinkedList
Kolejka	Queue	Queue
Stos	Stack	Stack
Tablica skojarzeniowa	Hashtable	Dictionary
wewnętrz. drzewo poszukiwań	SortedList	SortedDictionary
wewnętrz. drzewo binarne		Sorted Dictionary
	BitArray	

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Wyspecjalizowane kolekcje

Wyspecjalizowane kolekcje są zoptymalizowane pod kątem konkretnych zastosowań. Można je traktować jako rozszerzenie kolekcji generycznych. Dokładnej zostaną one opisane w ostatnim podrozdziale tego rozdziału.

Używane przez kolekcje przestrzenie nazw

Dostarczane w standardowych bibliotekach kolekcje zostały następująco podzielone pomiędzy przestrzenie nazw:

- Definicje **nie-generycznych kolekcji** (historycznie najwcześniejszych) są umieszczone głównie w przestrzeni **System.Collections**.
- Definicje **generycznych kolekcji** (wspierających kontrolę typów) są umieszczone głównie w przestrzeni **System.Collections.Generic**.
- Definicje kolekcji zoptymalizowanych pod kątem konkretnych zastosowań są umieszczone głównie w przestrzeni **System.Collections.Specialized**.

Przykładowe użycia kolekcji niegenerycznych

W tym podrozdziale zostaną pokazane przykłady użycia wybranych kolekcji niegenerycznych.

ArrayList

```
ArrayList lista = new ArrayList();
foreach (int el in new int[] { 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 })
{
    lista.Add(el); // dodanie elementu do listy (na koniec)
}
lista.RemoveAt(7); //usunięcie elementu o indeksie 7
lista.Remove(4); //usunięcie pierwszego elementu o wartości 4
lista.Insert(lista.Count/2, 15); // wstawienie elementu o wartości 15
// w miejscu o indeksie lista.Count/2
foreach (int el in lista) // tu nie trzeba rzutować wartości !!!
{
    Console.WriteLine("{0}", el);
}
Console.WriteLine("\n");
for (int i = 0; i < lista.Count; i++)
{
    int wart = (int) lista[i]; // tu rzutowanie jest konieczne !!!
    // int wart = lista[i]; // to jest niepoprawne
    Console.WriteLine("{0}", wart);
}
Console.WriteLine("\n");
```

Queue

```
Console.WriteLine("\nKolejka (Queue) :\nDodawanie:  ");
Queue kolejka = new Queue();
foreach (int el in tablica)
{
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    kolejka.Enqueue(el); // dodanie elementu do kolejki
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\nZawartość: ");
foreach (int el in kolejka)
{
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\nZdejmowanie:");
while (kolejka.Count > 0)
{
    int el = (int)kolejka.Dequeue(); // pobranie elementu z kolejki
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\n");
  
```

Stack

```

Console.WriteLine("\nStos (Stack): \nDodawanie: ");
Stack stos = new Stack();
foreach (int el in tablica)
{
    stos.Push(el); // kładzenie elementu na stos
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\nZawartość: ");
foreach (int el in stos)
{
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\nZdejmowanie:");
while (stos.Count > 0)
{
    int el = (int)stos.Pop(); // zdejmowanie elementu ze stosu
    Console.WriteLine(" {0}", el);
}
Console.WriteLine("\n\n");
  
```

Hashtable

```

Hashtable tabSkoj = new Hashtable();
tabSkoj["indeks1"] = 1;
tabSkoj["Indeks drugi"] = 4;
tabSkoj["trzeci ind."] = 9;
tabSkoj["czwarty"] = 2;
foreach (DictionaryEntry el in tabSkoj)
{
    string nazwaIndeksu = (string)el.Key;
    int wartosc = (int)el.Value;
    Console.WriteLine("tab[{0}]= {1} ", nazwaIndeksu, wartosc);
}
Console.WriteLine("\n");
tabSkoj.Remove("trzeci ind.");
foreach (DictionaryEntry el in tabSkoj)
{
    string nazwaIndeksu = (string)el.Key;
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    int wartosc = (int)el.Value;
    Console.WriteLine("tab[{0}]={1} ", nazwaIndeksu, wartosc);
}
Console.WriteLine("\n\n");
  
```

SortedDictionary

```

SortedDictionary tabUporz = new SortedDictionary();
tabUporz["indeks1"] = 1;
tabUporz["Indeks drugi"] = 4;
tabUporz["trzeci ind."] = 9;
tabUporz["czwarty"] = 2;
foreach (DictionaryEntry el in tabUporz)
{
    string nazwaIndeksu = (string)el.Key;
    int wartosc = (int)el.Value;
    Console.WriteLine("tab[{0}]={1} ", nazwaIndeksu, wartosc);
}
Console.WriteLine("\n\n");
  
```

BitArray

```

BitArray ba = new BitArray(3); // równoważnie: BitArray(3, false);
ba.Set(0, true);
ba.Set(1, false);
// źle (za mały rozmiar tablicy): ba.Set(4, true);
ba.Length = 5;
ba.Set(4, true); // teraz dobrze
Console.WriteLine(ba.Get(4));
  
```

Przykładowe użycia kolekcji generycznych

W tym podrozdziale zostaną pokazane przykłady użycia wybranych kolekcji generycznych. Pokazane tu przykłady używają klas z przestrzeni *System.Collection.Generic*. Ponadto niektóre z tych kolekcji wykorzystują metody z biblioteki LINQ (takie jak Min(), Max(), Average(), Sum()), dlatego w używającym je programie należy podać przestrzeń System.Linq oraz dodać referencje do System.Xml.Linq.

List<>

```

List<int> liczby = new List<int> {4,123,70,45,23};
liczby.Add(11);
liczby.Insert(3, 60);
liczby.Remove(70);
foreach (var el in liczby)
{
    Console.WriteLine("{0} ", el);
}
Console.WriteLine();
Console.WriteLine(
    "Min = {0}, Max = {1}, Average = {2}, Sum = {3}",
    liczby.Min(), liczby.Max(), liczby.Average(),
    liczby.Sum());
Console.WriteLine();
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Queue<>

```
Queue<int> kolejka = new Queue<int>(liczby);
Console.WriteLine("{0}", kolejka.Dequeue());
kolejka.Enqueue(8);
foreach (var el in kolejka)
{
    Console.Write("{0} ", el);
}
Console.WriteLine();
Console.WriteLine(
    "Min = {0}, Max = {1}, Average = {2}, Sum = {3}",
    kolejka.Min(), kolejka.Max(), kolejka.Average(),
    kolejka.Sum());
Console.WriteLine();
```

Stack<>

```
Stack<int> stos = new Stack<int>(kolejka);
Console.WriteLine("{0}", stos.Pop());
stos.Push(111);
foreach (var el in stos)
{
    Console.Write("{0} ", el);
}
Console.WriteLine();
Console.WriteLine(
    "Min = {0}, Max = {1}, Average = {2}, Sum = {3}",
    stos.Min(), stos.Max(), stos.Average(),
    stos.Sum());
Console.WriteLine();
```

LinkedList<>

```
LinkedList<int> lista = new LinkedList<int>(stos);
int value = 45;
LinkedListNode<int> wezel = lista.Find(value);
foreach (var el in lista)
{
    Console.Write("{0} ", el);
}
Console.WriteLine();
if (wezel != null)
{
    Console.WriteLine("{0} {1} {2}",
        wezel.Previous.Value, value,
        wezel.Next.Value);
}
Console.WriteLine();
```

SortedList<,>

```
SortedList<string, int> tabUporz = new SortedList<string, int>();
tabUporz["indeks1"] = 1;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

tabUporz["Indeks drugi"] = 4;
tabUporz["trzeci ind."] = 9;
tabUporz["czwarty"] = 2;
foreach (KeyValuePair<string, int> el in tabUporz)
{
    string nazwaIndeksu = (string)el.Key;
    int wartosc = (int)el.Value;
    Console.WriteLine("tablica[{0}] = {1}",
        nazwaIndeksu, wartosc);
}
}
  
```

Struktury i klasy reprezentujące węzły

Niektóre rodzaje kolekcji przechowują dane w specyficznej postaci, np. w postaci par „klucz-wartość”. W tym miejscu podane są składowe używanych przez te kolekcje typów danych (struktur i klas).

Struktura *DictionaryEntry*

Struktura **DictionaryEntry** (o deklaracji „public struct DictionaryEntry”) jest używana w kolekcjach **niegenerycznych**. Zawiera ona następujące publiczne składowe

- własność **Key**, która pobiera lub ustawia „klucz”,
- własność **Value**, która pobiera lub ustawia „wartość”.

Struktura *KeyValuePair<TKey, TValue>*

Struktura **KeyValuePair<TKey, TValue>** (o deklaracji „public struct KeyValuePair<TKey, TValue>”) jest używana w kolekcjach **generycznych**. Zawiera ona następujące publiczne składowe

- własność **Key**, która pobiera lub ustawia „klucz”,
- własność **Value**, która pobiera lub ustawia „wartość”.

Klasa *LinkedListNode<T>*

Struktura **LinkedListNode<T>** (o deklaracji „public sealed class LinkedListNode<T>”) jest używana w generycznej kolekcji *LinkedList<T>*. Zawiera ona następujące publiczne składowe

- własność **List**, która zwraca listę (obiekt klasy *LinkedList<T>*) do której należy dany węzeł (bieżąca instancja klasy *LinkedListNode*),
- własność **Next**, która pobiera następny węzeł z listy,
- własność **Previous**, która pobiera poprzedni węzeł z listy,
- własność **Value**, która pobiera lub ustawia „wartość”.

Interfejsy implementowane w kolekcjach

W tej części przedstawione zostaną najważniejsze interfejsy implementowane w kolekcjach. Większość z nich posiada dwa warianty: wersję niegeneryczną i wersję generyczną. Będą one przedstawiane wspólnie, ze wskazaniem podstawowych różnic.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

ICollection

Interfejs ***ICollection*** definiuje rozmiar, enumeratory oraz metody synchronizacji dla wszystkich niegenerycznych i generycznych kolekcji.

Dziedziczy on po interfejsie ***IEnumerable***, który dostarcza m.in. metodę ***GetEnumerator***. Interfejs ***ICollection*** dostarcza m.in. metodę ***CopyTo*** (skopiowanie elementów kolekcji do tablicy) i własności ***IsReadOnly*** i ***Count***. Wersja generyczna tego interfejsu dostarcza metody

- ***Add*** (dodanie elementu do kolekcji),
- ***Clear*** (usunięcie wszystkich elementów z kolekcji),
- ***Contains*** (sprawdzenie, czy kolekcja zawiera podany element),
- ***Remove*** (usunięcie pierwszego wystąpienia wybranego elementu z kolekcji).

IList

Interfejs ***IList*** jest używany do implementowania kolekcji korzystających z indeksów i pozwalających na dodawanie elementów we wskazanych miejscach oraz usuwanie wskazanych elementów.

Dziedziczy on po interfejsie ***IEnumerable*** i ***ICollection***. Interfejs ***IList*** dostarcza m.in. metody

- ***IndexOf*** (określenie indeksu określonego elementu z kolekcji),
- ***Insert*** (wstawienie do kolekcji elementu pod określonym indeksem),
- ***Contains*** (sprawdzenie, czy kolekcja zawiera podany element),
- ***RemoveAt*** (usunięcie elementu o wybranym indeksie).

Wersja niegeneryczna tego interfejsu dostarcza metody ***Add***, ***Clear***, ***Contains*** i ***Remove***, podczas gdy jego wersja generyczna dziedziczy te metody po interfejsie ***ICollection***.

IDictionary

Interfejs ***IDictionary*** jest używany do implementowania kolekcji przechowujących dane w postaci par „klucz-wartość”.

Wersja niegeneryczna tego interfejsu przechowuje dane (parę „klucz/wartość”) w instancjach struktury ***DictionaryEntry***, podczas gdy wersja generyczna przechowuje dane (parę „klucz/wartość”) w instancjach struktury ***KeyValuePair< TKey, TValue>***.

Interfejs ***IDictionary*** dziedziczy po interfejsie ***IEnumerable*** i ***ICollection***. Wersja generyczna tego interfejsu dostarcza metody:

- ***ContainsKey*** (sprawdzenie, czy kolekcja zawiera element o podanym kluczu),
- ***TryGetValue*** (pobiera wartość skojarzoną z podanym kluczem).

Wersja niegeneryczna interfejsu ***IDictionary*** dostarcza też metody

- ***Add*** (dodanie do kolekcji pary „klucz/wartość”),
- ***Clear***,
- ***Contains*** (sprawdzenie, czy kolekcja zawiera element o podanej wartości)
- ***Remove*** (usunięcie elementu o wybranym kluczu),

podczas gdy wersja generyczna dziedziczy te metody po interfejsie ***ICollection*** (metoda ***Add*** jest przeciążona).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

IComparable

Interfejs ***IComparable*** deklaruje generyczną jednoargumentową metodę *CompareTo* służącą do porównania bieżącego obiektu (lub zmiennej wartościowej) z podanym argumentem tego samego typu. Metoda ta zwraca wartość **1**, **0** lub **-1** jeśli odpowiednio wartość bieżącego obiektu jest większa, równa lub mniejsza niż wartość obiektu przekazanego w argumencie.

IComparer

Interfejs ***IComparer*** dostarcza metodę *Compare*, która

- w wersji niegenerycznej interfejsu przyjmuje dwa argumenty typu *System.Object*,
- w wersji generycznej przyjmuje dwa argumenty stosownego typu generycznego.

Metoda ta zwraca wartość **1**, **0** lub **-1** jeśli odpowiednio wartość pierwszego argument jest większa, równa lub mniejsza niż wartość drugiego argumentu.

IEqualityComparer

Interfejs ***IEqualityComparer*** definiuje metody porównywania służące do rozstrzygania równości elementów. Deklaruje on metody

- *Equals*, określającą czy podane argumenty są równe - zwracającą **true** w takim przypadku i wartość **false** w przypadku przeciwnym
- *GetHashCode*, zwracającą funkcję skrótu dla podanego obiektu.

W wersji niegenerycznej interfejsu metody te przyjmują argumenty typu *System.Object*, podczas gdy w wersji generycznej przyjmują one argumenty stosownego typu generycznego.

IEnumerator

Interfejs ***IEnumerator*** dostarcza metody

- ***MoveNext*** powodującą przesunięcie do następnego elementu kolekcji, (jeśli przesunięcie się powiedzie zwracana jest wartość **true**, w wypadku niepowodzenia zwracana jest wartość **false**),
- ***Reset*** powodującą przejście do pierwszego elementu kolekcji

oraz własność ***Current*** zwracającą bieżący element kolekcji.

Umożliwiają one iterowanie elementów kolekcji (np. w pętli *foreach*).

IEnumerable

Interfejs ***IEnumerable*** dostarcza metodę ***GetEnumerator*** zwracającą enumerator umożliwiający iterowanie elementów kolekcji (np. w pętli *foreach*).

Kolekcja musi zawierać tę metodę (metodę *GetEnumerator*), aby możliwe było takie iterowanie.

Przykłady definiowania kolekcji zawierających *enumerator*

W tej części pokazane zostaną przykłady implementacji prostych kolekcji zawierających enumeratory pozwalające na iterowanie po elementach kolekcji.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykład implementacji niegenerycznej kolekcji z klasycznie zdefiniowanym enumeratorem

```

using System;
using System.Collections;

public class Kolejka
{
    private class Wezel
    {
        public Wezel(object el)
        {
            nast = null;
            wart = el;
        }

        private Wezel nast;
        public Wezel Nast
        {
            get { return nast; }
            set { nast = value; }
        }

        private object wart;
        public object Wart
        {
            get { return wart; }
            set { wart = value; }
        }
    }

    private Wezel koniec;
    private Wezel poczatek;

    public Kolejka()
    {
        koniec = null;
        poczatek = null;
    }
    public void Dodaj(object el)
    {
        Wezel w = new Wezel(el);
        if (poczatek == null)
        {
            poczatek = w;
        }
        else
        {
            koniec.Nast = w;
        }
        koniec = w;
    }
    public object Usun()
    {
        if (poczatek == null)
        {
            throw new IndexOutOfRangeException("Kolejka jest pusta");
        }
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    }
  else
  {
    object el = poczatek.Wart;
    poczatek = poczatek.Nast;
    return el;
  }
}

public IEnumarator GetEnumerator()
{
  return new KolejkaEnum(poczatek);
}

private class KolejkaEnum : IEnumarator
{
  private Wezel poczatek;
  private Wezel pozycja;
  private bool przed;

  public KolejkaEnum(Wezel w)
  {
    poczatek = w;
    przed = true;
  }

  public void Reset()
  {
    przed = true;
    pozycja = poczatek;
  }

  public bool MoveNext()
  {
    if (przed)
    {
      pozycja = poczatek;
      przed = false;
      return true;
    }

    if (pozycja != null)
    {
      pozycja = pozycja.Nast;
      return (pozycja != null);
    }
    else
    {
      return false;
    }
  }

  public object Current
  {
    get
    {

```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

        try
        {
            return pozycja.Wart;
        }
        catch (IndexOutOfRangeException)
        {
            throw new InvalidOperationException();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Kolejka kolejka = new Kolejka();
        for (int i = 1; i <= 5; i++)
        {
            kolejka.Dodaj(i);
        }
        foreach (object i in kolejka)
        {
            Console.Write(" {0}", i);
        }
        Console.WriteLine();
        kolejka.Usun();
        foreach (var i in kolejka)
        {
            Console.Write(" {0}", i);
        }
        Console.WriteLine();
    }
}
  
```

Przykład implementacji generycznej kolekcji z enumeratorem z leniwą ewaluacją

```

using System;
using System.Collections.Generic;

namespace Stos
{
    public class Stos<T>
    {
        private class Wezel
        {
            public Wezel(T t)
            {
                nast = null;
                wart = t;
            }
            private Wezel nast;
            public Wezel Nast
            {
                get { return nast; }
            }
        }
    }
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    set { nast = value; }
}
private T wart;
public T Wart
{
    get { return wart; }
    set { wart = value; }
}
}
private Wezel szczyt;
public Stos()
{
    szczyt = null;
}
public void Poloz(T t)
{
    Wezel w = new Wezel(t);
    w.Nast = szczyt;
    szczyt = w;
}
public void Zdejmij(out T t)
{
    if (szczyt == null)
    {
        throw new IndexOutOfRangeException("Stos jest pusty");
    }
    t = szczyt.Wart;
    szczyt = szczyt.Nast;
}
public IEnumarator<T> GetEnumerator()
{
    Wezel biezacy = szczyt;
    while (biezacy != null)
    {
        yield return biezacy.Wart;
        biezacy = biezacy.Nast;
    }
}
}
class Program
{
    static void Main(string[] args)
    {
        Stos<int> stos = new Stos<int>();
        for (int i = 1; i <= 5; i++)
        {
            stos.Poloz(i);
        }
        foreach (int i in stos)
        {
            Console.WriteLine(" {0}", i);
        }
        Console.WriteLine();
        int el;
        stos.Zdejmij(out el);
    }
}

```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    foreach (int i in stos)
    {
        Console.WriteLine(" {0}", i);
    }
    Console.WriteLine();
}
}
  
```

Składowe popularnych kolekcji niegenerycznych

Ta część poświęcona jest przedstawieniu najważniejszych elementów składowych popularnych (podanych wcześniej w przykładach) kolekcji niegenerycznych.

ArrayList

Kolekcja **ArrayList** ma następującą deklarację

```
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Capacity** pozwala pobrać oraz zmienić rozmiar listy (ilość obiektów, które można na niej umieścić);
- własność **Count** zwraca liczbę obiektów umieszczonych na liście;
- własność **Item** pozwala odczytać lub przypisać element pod zadanym indeksem;
- metoda **Add** dodaje obiekt do kolekcji *ArrayList*;
- metoda **AddRange** dodaje elementy ze wskazanej kolekcji do końca listy;
- metoda **BinarySearch** wyszukuje podany obiekt na liście; posiada ona następujące przeciążenia:
 - *ArrayList.BinarySearch(Object)*;
 - *ArrayList.BinarySearch(Object, IComparer)*;
 - *ArrayList.BinarySearch(Int32, Int32, Object, IComparer)*
 (w trzecim - ostatnim wariantem podaje się początek i ilość wyszukiwań)
- metoda **Clear** usuwa wszystkie obiekty z kolekcji;
- metoda **Clone** tworzy kopię listy;
- metoda **Contains** sprawdza czy wskazany obiekt znajduje się na liście;
- metoda **GetRange** zwraca kolekcję *ArrayList* składającą się z elementów z podanego przedziału (z bieżącej listy);
- metoda **Insert** wstawia obiekt do listy pod zadanym indeksem;
- metoda **InsertRange** wstawia do listy obiekty ze wskazanej kolekcji w zadanym miejscu;
- metoda **LastIndexOf** zwraca indeks ostatniego wystąpienia na liście wskazanego obiektu;
- metoda **Remove** usuwa wskazany obiekt z listy;
- metoda **RemoveAt** usuwa z listy obiekt o wskazanym indeksie;
- metoda **RemoveRange** usuwa z listy obiekty z podanego zakresu;
- metoda **Reverse** odwraca porządek na liście (lub skazanym jej fragmencie);
- metoda **SetRange** zastępuje obiekty z podanego przedziału obiektami z podanej kolekcji na liście

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- metoda **Sort** sortuje obiekty znajdujące się na liście;
- metoda **ToArray** tworzy nową tablicę (albo elementów wskazanego typu albo elementów typu *System.Object*) i kopiuje na nią elementy listy;
- metoda **TrimToSize** zmienia (zmniejsza) rozmiar tablicy na wartość równą liczbie jego elementów.

Stack

Kolekcja niegeneryczna **Stack** ma następującą deklarację

```
public class Stack : ICollection, IEnumerable, ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Count** zwraca liczbę obiektów umieszczonych na stosie;
- metoda **Clear** usuwa wszystkie obiekty ze stosu;
- metoda **Clone** tworzy (głęboką) kopię stosu;
- metoda **Contains** sprawdza czy wskazany obiekt znajduje się na stosie;
- metoda **CopyTo** skopiowanie obiektów ze stosu do istniejącej tablicy;
- metoda **Peek** zwraca element ze szczytu stosu (bez usuwania go ze stosu);
- metoda **Pop** zdejmuje element ze szczytu stosu (i zwraca go);
- metoda **Push** kładzie element na szczyt stosu;
- metoda **ToArray** tworzy nową tablicę i kopiuje na nią elementy stosu.

Queue

Kolekcja niegeneryczna **Queue** ma następującą deklarację

```
public class Queue : ICollection, IEnumerable, ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Count** zwraca liczbę obiektów umieszczonych w kolejce;
- metoda **Clear** usuwa wszystkie obiekty z kolejki;
- metoda **Clone** tworzy (głęboką) kopię kolejki;
- metoda **Contains** sprawdza czy wskazany obiekt znajduje się w kolejce;
- metoda **CopyTo** skopiowanie obiektów z kolejki do istniejącej tablicy;
- metoda **Dequeue** usuwa obiekt z początku kolejki (i zwraca go);
- metoda **Enqueue** dodaje obiekt na koniec kolejki;
- metoda **Peek** zwraca element z początku kolejki (bez usuwania go z kolejki);
- metoda **ToArray** tworzy nową tablicę i kopiuje na nią elementy kolejki
- metoda **TrimToSize** zmniejsza rozmiar kolejki do liczby jej elementów.

Hashtable

Kolekcja niegeneryczna **Hashtable** ma następującą deklarację

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
public class Hashtable : IDictionary, ICollection, IEnumerable,
    ISerializable, IDeserializationCallback, ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Count** zwraca liczbę obiektów umieszczonych w tablicy;
- własność **Item** pozwala odczytać lub przypisać wartość skojarzoną ze wskazanym kluczem
- własność **Keys** zwraca kolekcję (dziedziczącą po interfejsie *ICollection*) zawierającą klucze (nie kopię!);
- własność **Values** zwraca kolekcję (dziedziczącą po interfejsie *ICollection*) zawierającą wartości (nie kopię!);
- metoda **Add** dodaje element o wskazanym kluczu i wartości do tablicy;
- metoda **Clear** usuwa wszystkie elementy z tablicy;
- metoda **Clone** tworzy (głęboką) kopię tablicy skojarzeniowej;
- metoda **Contains** sprawdza czy tablica zawiera element o zadanym kluczu;
- metoda **ContainsKey** sprawdza czy tablica zawiera element o zadanym kluczu;
- metoda **ContainsValue** sprawdza czy tablica zawiera element o zadanej wartości;
- metoda **CopyTo** kopiuje elementy tablicy skojarzeniowej (jako instancje struktury *DictionaryEntry*) do istniejącej 1-wymiarowej tablicy;
- metoda **KeyEquals** porównuje podany obiekt ze wskazanym kluczem z tablicy;
- metoda **Remove** usuwa z tablicy element o zadanym kluczu.

SortedList

Kolekcja niegeneryczna **SortedList** ma następującą deklarację

```
public class SortedList : IDictionary, ICollection, IEnumerable,
    ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Capacity** pozwala pobrać oraz zmienić rozmiar listy (ile obiektów można na niej umieścić);
- własność **Count** zwraca liczbę obiektów umieszczonych na liście;
- własność **Item** pozwala odczytać lub przypisać wartość skojarzoną ze wskazanym kluczem;
- własność **Keys** zwraca kolekcję (dziedziczącą po interfejsie *ICollection*) zawierającą klucze (nie kopię!);
- własność **Values** zwraca kolekcję (dziedziczącą po interfejsie *ICollection*) zawierającą wartości (nie kopię!);
- metoda **Add** dodaje element o wskazanym kluczu i wartości do listy;
- metoda **Clear** usuwa wszystkie elementy z listy;
- metoda **Clone** tworzy (głęboką) kopię posortowanej listy;
- metoda **Contains** sprawdza czy lista zawiera element o zadanym kluczu;
- metoda **ContainsKey** sprawdza czy lista zawiera element o zadanym kluczu;
- metoda **ContainsValue** sprawdza czy lista zawiera element o zadanej wartości;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- metoda **CopyTo** kopiuje elementy posortowanej listy (jako instancje struktury *DictionaryEntry*) do istniejącej 1-wymiarowej tablicy;
- metoda **GetByIndex** pobiera z listy wartość elementu o zadanym indeksie;
- metoda **GetKey** pobiera z listy klucz elementu o zadanym indeksie;
- metoda **GetKeyList** zwraca kolekcję (dziedziczącą po interfejsie *IList*) zawierającą klucze;
- metoda **GetValueList** zwraca kolekcję (dziedziczącą po interfejsie *IList*) zawierającą wartości;
- metoda **IndexOfKey** zwraca indeks elementu o zadanym kluczu;
- metoda **IndexOfValue** zwraca indeks pierwszego wystąpienia elementu o zadanej wartości;
- metoda **Remove** usuwa z listy element o zadanym kluczu;
- metoda **RemoveAt** usuwa z listy element o zadanym indeksie;
- metoda **SetByIndex** zamienia wartość elementu o zadanym indeksie;
- metoda **TrimToSize** ustawia rozmiar listy na wartość równą liczbie jego elementów.

BitArray

Kolekcja niegeneryczna **BitArray** ma następującą deklarację

```
public class BitArray : ICollection, IEnumerable, ICloneable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Count** pobiera liczbę elementów umieszczonych w tablicy bitowej;
- własność **Length** pobiera lub ustawia liczbę elementów umieszczonych w tablicy bitowej (nowo dodane elementy przyjmują wartość **false**);
- własność **Item** pozwala odczytać lub przypisać wartość skojarzoną ze wskazanym indeksem;
- metoda **And** wykonuje operację bitową „**AND**” na bitach bieżącej i wskazanej tablicy bitowej (muszą one posiadać tę samą liczbę elementów);
- metoda **Clone** tworzy (głęboką) kopię tablicy bitowej;
- metoda **CopyTo** kopiuje elementy tablicy bitowej do istniejącej tablicy;
- metoda **Get** zwraca wartość bitu stojącego na podanej pozycji;
- metoda **Or** wykonuje operację bitową „**OR**” na bitach bieżącej i wskazanej tablicy bitowej (muszą one posiadać tę samą liczbę elementów);
- metoda **Set** ustawia wartość bitu stojącego na podanej pozycji;
- metoda **SetAll** ustawia wartość wszystkich bitów na określoną wartość;
- metoda **Xor** wykonuje operację bitową „**XOR**” na bitach bieżącej i wskazanej tablicy bitowej (muszą one posiadać tę samą liczbę elementów).

Comparer

Klasa **Comparer** ma następującą deklarację

```
public class Comparer : IComparer, ISerializable
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- metoda **Compare** porównuje dwa obiekty przekazane jako argumenty i zwraca wartość **1**, **0** lub **-1** jeśli odpowiednio pierwszy argument jest większy, równy lub mniejszy niż drugi argument, przy czym pusta referencja (**null**) jest „mniejsza” od wszystkich pozostałych obiektów;
- metoda **Equals** sprawdza czy podany jako argument obiekt bieżący obiekt są równe: zwraca ona wartość **true** jeśli są równe oraz wartość **false** w przeciwnym wypadku.

Składowe popularnych kolekcji generycznych

Ta część poświęcona jest przedstawieniu najważniejszych elementów składowych popularnych (podanych wcześniej w przykładach) kolekcji generycznych.

Stack<T>

Kolekcja generyczna **Stack**<T> ma następującą deklarację

```
public class Stack<T> : I Enumerable<T>, I Collection, I Enumerable
```

Większość metod i właściwości z niegenerycznej wersji tej klasy ma swoje odpowiedniki w wersji generycznej.

Niektóre jej właściwości i metody (np. związane z dostępem synchronicznym) wymagają jawnej implementacji.

W wersji generycznej tej kolekcji pojawiły się też nowe metody. Przykładem takiej metody jest metoda **TrimExcess** zmniejszająca rozmiar kolekcji do ilości elementów, jeśli kolekcja posiada mniej elementów niż 90% jej rozmiaru. Klasa ta zawiera też dużą ilość metod rozszerzonych (np. **Min**, **Max**, **Average**, **Sum**, **Reverse**, **To Array**, **To List**, **FirstOrDefault**).

Queue<T>

Kolekcja generyczna **Queue**<T> ma następującą deklarację

```
public class Queue<T> : I Enumerable<T>, I Collection, I Enumerable
```

Większość metod i właściwości z niegenerycznej wersji tej klasy ma swoje odpowiedniki w wersji generycznej.

Niektóre jej właściwości i metody (np. związane z dostępem synchronicznym) wymagają jawnej implementacji.

W wersji generycznej tej kolekcji pojawiły się też nowe metody. Przykładem takiej metody jest metoda **TrimExcess** zmniejszająca rozmiar kolekcji do ilości elementów, jeśli kolekcja posiada mniej elementów niż 90% jej rozmiaru. Klasa ta zawiera też dużą ilość metod rozszerzonych (np. **Min**, **Max**, **Average**, **Sum**, **Reverse**, **To Array**, **To List**, **FirstOrDefault**).

List<T>

Kolekcja generyczna **List**<T> ma następującą deklarację

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable}
```

większość metod i własności z (nie-generycznej) kolekcji *ArrayList* ma swoje odpowiedniki w (generycznej) kolekcji ***List<T>***.

Niektóre jej własności i metody klasy *List<T>* wymagają jawniej implementacji.

W porównaniu z kolekcją *ArrayList*, kolekcja *List<T>* w posiada też nowe metody (np. ***Exists***, ***Find***, ***FindAll***, ***FindIndex***, ***FindLast***, ***FindLastIndex***, ***ForEach***, ***TrueForAll*** związane z predykatami).

Zamiast zawartej w klasie *ArrayList* metody *TrimToSize* klasa *List<T>* posiada metodę ***TrimExcess*** zmniejszającą rozmiar kolekcji do ilości elementów, jeśli kolekcja posiada mniej elementów niż 90% jej rozmiaru. Podobnie jak przedstawione wcześniej kolekcje generyczne kolekcja *List<T>* posiada też sporą ilość metod rozszerzonych (m.in. ***Min***, ***Max***, ***Average***, ***Sum***, ***Reverse***, ***Skip***, ***FirstOrDefault***).

Dictionary< TKey, TValue >

Kolekcja generyczna ***Dictionary< TKey, TValue >*** ma następującą deklarację

```
public class Dictionary< TKey, TValue > : IDictionary< TKey, TValue >,
    ICollection< KeyValuePair< TKey, TValue >>,
    IEnumerable< KeyValuePair< TKey, TValue >>,
    IDictionary, ICollection, IEnumerable,
    ISerializable, IDeserializationCallback;
```

Kolekcja ta jest odpowiednikiem niegenerycznej kolekcji *Hashtable*.

Kolekcja *Dictionary< TKey, TValue >* do porównywania kluczy używa metod dziedziczonych z generycznego interfejsu *IEqualityComparer*. Jeśli stosowny interfejs nie zostanie wyspecyfikowany w implementacji, to jest używany domyślny interfejs *EqualityComparer.Default*.

Większość metod i własności kolekcji *Hashtable* ma swoje odpowiedniki w kolekcji *Dictionary*.

Niektóre jej własności i metody klasy *Dictionary* wymagają jawniej implementacji.

W porównaniu z kolekcją *Hashtable*, kolekcja *Dictionary* w posiada też nowe metody (np. ***TryGetValue***).

Kolekcja *Dictionary* posiada też sporą ilość metod rozszerzonych (m.in. ***Min***, ***Max***, ***Average***, ***Sum***, ***Reverse***, ***Skip***, ***FirstOrDefault***), ale niektóre z nich wymagają jawniej implementacji.

SortedList< TKey, TValue >

Kolekcja generyczna ***SortedList< TKey, TValue >*** ma następującą deklarację

```
public class Dictionary< TKey, TValue > : IDictionary< TKey, TValue >,
    ICollection< KeyValuePair< TKey, TValue >>,
    IEnumerable< KeyValuePair< TKey, TValue >>,
    IDictionary, ICollection, IEnumerable;
```

Kolekcja ta jest odpowiednikiem niegenerycznej kolekcji *SortedList*.

Kolekcja *SortedList< TKey, TValue >* do porównywania kluczy używa metod dziedziczonych z generycznego interfejsu *IComparer*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Do przechowywania danych używa ona wewnętrznej tablicy, której rozmiar początkowy domyślnie wynosi 16, a podczas dodawania nowych elementów jej rozmiar jest automatycznie zwiększany.

Większość metod i własności negenerycznej wersji kolekcji *SortedDictionary* ma swoje odpowiedniki w wersji generycznej tej kolekcji.

Niektóre jej własności i metody generycznej kolekcji *SortedDictionary* wymagają jawniej implementacji.

Wersja generyczna klasy *SortedDictionary* w posiada też nowe metody (np. **TryGetValue**) w stosunku do swojej wersji niegenerycznej.

Wersja generyczna kolekcji *SortedDictionary* zamiast metody *TrimToSize* z wersji niegenerycznej posiada metodę **TrimExcess** zmniejszającą rozmiar kolekcji do ilości elementów, jeśli kolekcja posiada mniej elementów niż 90% jej rozmiaru.

Wersja generyczna kolekcji *SortedDictionary* posiada też dużą ilość metod rozszerzonych (m.in. **Min**, **Max**, **Average**, **Sum**, **Reverse**, **Skip**, **FirstOrDefault**), ale niektóre z nich wymagają jawniej implementacji.

SortedDictionary< TKey, TValue >

Kolekcja generyczna ***SortedDictionary< TKey, TValue >*** ma następującą deklarację

```
public class SortedDictionary< TKey, TValue > : IDictionary< TKey, TValue >,  

    ICollection< KeyValuePair< TKey, TValue > >,  

    IEnumerable< KeyValuePair< TKey, TValue > >,  

    IDictionary, ICollection, IEnumerable;
```

Kolekcja ta jest bardzo podobna do generycznej kolekcji *SortedDictionary*.

Używa ona szybszych algorytmów sortowania niż kolekcja *SortedDictionary*, ale wymagających więcej zasobów pamięciowych.

W przeciwnieństwie do kolekcji *SortedDictionary*, elementy kolekcji *SortedDictionary* nie są indeksowane, co przyśpiesza m.in. dodawanie nowych elementów.

W aktualnej wersji .NET Frameworka nie ma wsparcia dla kolekcji *SortedDictionary* w wersach dla XNA i .NET Compact Framework.

LinkedList< T >

Kolekcja generyczna ***LinkedList< T >*** ma następującą deklarację

```
public class LinkedList< T > : ICollection< T >, IEnumerable< T >,  

    ICollection, IEnumerable, ISerializable,  

    IDeserializationCallback
```

oraz następujące składowe o podanych poniżej funkcjonalnościach:

- własność **Count** zwraca liczbę obiektów umieszczonych na liście;
- własność **First** zwraca pierwszy węzeł z listy;
- własność **Last** zwraca ostatni węzeł z listy;
- metoda **AddAfter** dodaje nowy węzeł lub wartość po wskazanym węźle;
- metoda **AddBefore** dodaje nowy węzeł lub wartość przed wskazanym węzłem;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- metoda **AddFirst** dodaje nowy węzeł lub wartość na początek listy;
- metoda **AddLast** dodaje nowy węzeł lub wartość na koniec listy;
- metoda **AddRange** dodaje elementy ze wskazanej kolekcji do na koniec listy
- metoda **Clear** usuwa wszystkie obiekty z listy;
- metoda **Contains** sprawdza czy wskazany obiekt znajduje się na liście;
- metoda **CopyTo** kopiuje całą listę do jednowymiarowej tablicy typu zgodnego z generycznym poczynając od wskazanego indeksu;
- metoda **Find** znajduje pierwszy węzeł zawierający wskazaną wartość;
- metoda **FindLast** znajduje ostatni węzeł zawierający wskazaną wartość;
- metoda **Remove** usuwa pierwsze wystąpienie węzła lub wartości z listy;
- metoda **RemoveFirst** usuwa ostatni węzeł z listy;
- metoda **RemoveLast** usuwa pierwszy węzeł z listy;

Kolekcja ta posiada też sporą ilość metod rozszerzonych (m.in. **Min**, **Max**, **Average**, **Sum**, **Reverse**, **ToArrayList**, **Skip**, **FirstOrDefault**).

Wyspecjalizowane kolekcje

Wyspecjalizowane kolekcje są zoptymalizowane pod kątem konkretnych zastosowań. Można je traktować jako rozszerzenie kolekcji generycznych. Pod względem rodzaju danych, na których one operują kolekcje te możemy podzielić na cztery rodzaje:

- wyspecjalizowane klasy łańcuchowe (operujące na łańcuchach znaków);
- wyspecjalizowane klasy słownikowe;
- wyspecjalizowane nazwane kolekcje;
- wyspecjalizowane struktury bitowe.

W tym podrozdziale podane zostaną najczęściej używane kolekcje z tych rodzajów.

Klasy operujące na łańcuchach znaków

Wśród kolekcji operujących na łańcuchach wyróżnić możemy następujące klasy:

StringCollection

Klasa **StringCollection** ma taką samą funkcjonalność, jak generyczna kolekcja *List*, dla której elementy posiadają typ *string* (jej wewnętrzna konstrukcja wykorzystuje wbudowaną kolekcję *ArrayList*).

StringDictionary

Klasa **StringCollection** ma taką samą funkcjonalność, jak generyczna kolekcja *Dictionary*, dla której klucze i wartości są typu *string*.

StringEnumerator

Klasa **StringEnumerator** pozwala na iterowanie kolekcji działających na łańcuchach znaków.

CollectionsUtil

Klasa **CollectionsUtil** pozwala na tworzenie kolekcji *Hashtable* i *SortedList*, w których porównywanie nie rozróżnia dużych i małych liter.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Klasy słownikowe

Wśród kolekcji słownikowych warto zwrócić uwagę na następujące klasy:

ListDictionary

Klasa ***ListDictionary*** jest prostą implementacją interfejsu *Idictionary* używającą listy jednokierunkowej. Dla rozmiaru do 10 elementów jej obiekty zajmują mniej miejsca, a operacje są wykonywane szybciej niż w przypadku obiektów klasy *Hashtable*. Elementy w klasie *ListDictionary* są przechowywane jako pary „klucz-wartość” w strukturach *DictionaryEntry* (tak samo jak dla *Hashtable*).

HybridDictionary

Klasa ***HybridDictionary*** dostosowuje wewnętrzną strukturę do swojego rozmiaru. Kolekcje *HybridDictionary* składające się z nie więcej niż 10 elementów używają wewnętrz klasy *ListDictionary*. Przy większej liczbie elementów *HybridDictionary* zmienia automatycznie wewnętrzną konstrukcję na *Hashtable*.

OrderedDictionary

Klasa ***OrderedDictionary*** służy do przechowywania par „klucz-wartość” (w obiektach *DictionaryEntry*). Operacje na elementach kolekcji (dodawanie, pobieranie, sortowanie) można przeprowadzać zarówno używając kluczy, jak i indeksów.

Tworzenie klas z kluczami opartymi na łańcuchach

Wyspecjalizowane nazwane kolekcje służą głównie tworzeniu kolekcji przechowujących dane w postaci par „klucz-wartość”, gdzie „klucze” są łańcuchami znaków. Wśród tych kolekcji można wskazać następujące klasy:

NameObjectCollectionBase

NameObjectCollectionBase jest abstrakcyjną klasą służącą do definiowania kolekcji działających jak tablice asocjacyjne o kluczach będących łańcuchami znaków (domyślnie bez rozróżniania małych i dużych liter) (z możliwością indeksowania również liczbami naturalnymi).

NameObjectCollectionBase.KeysCollection

Klasa ***NameObjectCollectionBase.KeysCollection*** przedstawia kolekcję łańcuchów znaków będących kluczami wymienionej powyżej kolekcji. Można ją otrzymać poprzez własność *Keys* klasy *NameObjectCollectionBase*.

NameValueCollection

Klasa ***NameValueCollection*** jest klasą pochodną klasy *NameObjectCollectionBase*. W przeciwieństwie do np. klasy *StringDictionary* może ona przechowywać wiele wartości pod jednym kluczem. Ponadto dopuszcza również wartość **null** (zarówno dla „klucza”, jak i „wartości”).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Wyspecjalizowane struktury bitowe

Wśród kolekcji operujących na polach bitowych wyróżnić można struktury **BitVector32** oraz **BitVector32.Section**. Mogą być one szczególnie przydatne przy obróbce danych zawierających nagłówki, w których znajdują się zarówno pola bitowe, jak i wartości całkowito-liczbowe o różnych rozmiarach (które nie muszą być potęgami liczby 2).

BitVector32

Struktura **BitVector32** służy do przechowywania maksymalnie 32 wartości „prawda/fałsz” jako ciąg binarny zer i jedynek. Ze względu na ustalony sztywno rozmiar operacje na tej strukturze są szybsze niż operacje na kolekcji *BitArray*. W strukturze *BitVector32* mamy następujące składowe

- własność **Data** pobiera wartość struktury jako liczby całkowitej;
- własność **Item** pobiera/zmienia wartość wskazanej sekcji lub bitów (z maski);
- metoda **CreateMask** tworzy maskę służącą do operowania na wybranych bitach;
- metoda **CreateSection** tworzy nową (następną) sekcję zawierającą „małą” liczbę całkowitą.

BitVector32.Section

Struktura **BitVector32.Section** odpowiada wydzielonemu fragmentowi struktury **BitVector32** i może przechowywać małą liczbę całkowitą. W strukturze *BitVector32.Section* występują następujące składowe

- własność **Mask** pobiera maskę wydzielającą sekcję ze struktury *BitVector32*;
- własność **Offset** pobiera przesunięcie początku sekcji względem struktury *BitVector32*.

Przykład użycia struktur bitowych

Poniższy przykład prezentuje przykładowe użycie struktur bitowych.

```
using System;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        BitVector32 wektor = new BitVector32(5);

        BitVector32.Section sekcja1 = BitVector32.CreateSection(8);
        BitVector32.Section sekcja2 = BitVector32.CreateSection(15, sekcja1);
        BitVector32.Section sekcja3 = BitVector32.CreateSection(20, sekcja2);

        Console.WriteLine("Wartości początkowe:");
        Console.WriteLine(" sekcja1: {0}, ", wektor[sekcja1]);
        Console.WriteLine(" sekcja2: {0}, ", wektor[sekcja2]);
        Console.WriteLine(" sekcja3: {0}", wektor[sekcja3]);

        Console.WriteLine("Zmiany wartości w każdej sekcji"
            + " (na odpowiednio 3, 7, 15):");
        Console.WriteLine(" {0}", wektor.ToString());
        wektor[sekcja1] = 3;
        Console.WriteLine(" {0}", wektor.ToString());
        wektor[sekcja2] = 7;
        Console.WriteLine(" {0}", wektor.ToString());
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

wektor[sekcja3] = 15;
Console.WriteLine("  {0}", wektor.ToString());

Console.WriteLine("Nowe wartości:");
Console.WriteLine("  sekcja1: {0},", wektor[sekcja1]);
Console.WriteLine("  sekcja2: {0},", wektor[sekcja2]);
Console.WriteLine("  sekcja3: {0}", wektor[sekcja3]);
}
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

4. Serializacja

Serializacja jest mechanizmem pozwalającym na zapisanie (np. do pliku) stanu obiektu (w niektórych przypadkach struktury) w celu jego późniejszego odtworzenia. Proces odtworzenia stanu obiektu, będący odwrotnością serializacji nazywany jest **deserializacją**. Zakłada się, że czytelnik spotkał już wcześniej z tymi pojęciami (np. w trakcie nauki języka Java), dlatego pominięte zostanie szczegółowe ich wyjaśnienie.

Wiele rozważanych tu mechanizmów i konstrukcji zostanie omówione w bardzo skrótwowej formie lub podane w postaci przykładów.

Rodzaje serializacji i ich najważniejsze cechy

W języku C# mamy możliwość użycia trzech rodzajów serializacji:

- serializacja binarna,
- serializacja XML-owa,
- serializacja SOAP-owa.

W tej krótszej części zostaną przedstawione najważniejsze ich cechy.

Serializacja binarna

Serializacja binarna przechowuje dane serializowanych obiektów (odpowiednio zmienne strukturalne) w postaci dokładnego odwzorowania bitów danych z zajmowanych przez te obiekty obszarów pamięci. Ten rodzaj serializacji można zastosować do najszerzej rodziny obiektów i zmiennych, ponieważ wszystkie one w jakiś sposób są umieszczone w pamięci. Ten rodzaj serializacji jest też zwykle najszybszy, gdyż jest podczas niego przeprowadzana minimalna ilość konwersji. Najistotniejszą wadą serializacji binarnej jest jej najmniejsza przenośność. Dane serializowane w ten sposób muszą być odtwarzane w zbliżonym środowisku i są bardzo wrażliwe na kwestie reprezentacji binarnej typów.

Serializacja XML-owa

Serializacja XML-owa, jak jej nazwa wskazuje, przechowuje dane w postaci tekstu w znacznikach XML-owych. Dane są zwykle w miarę możliwości przetwarzane do postaci czytelnej (i zrozumiałej) przez użytkownika. Taki sposób przechowywania danych zapewnia niezależność od sprzętu i największą przenośność. Jednak ten rodzaj serializacji ma bardzo ograniczone możliwości zastosowania ze względu na bardzo wąski zakres tego co oraz w jaki sposób można serializować²⁷.

Serializacja SOAP-owa

Można powiedzieć, że *serializacja SOAP-owa* jest czymś pośrednim pomiędzy serializacją binarną i serializacją XML-ową starając się przejąć zalety każdej z nich. Dane są w niej przechowywane w znacznikach SOAP-owych XML-owych w postaci tekstu, ale w postaci możliwe „mało przetworzonej”

²⁷ W szczególności serializacji XML-owej może być podany tylko jeden obiekt lub zmienna (na strumień serializacji), zachowane mogą być tylko wartości publicznych (!) pól, a serializacja i deserializacja złożonych typów danych (np. zaimplementowanych przez użytkownika klas) jest bardzo skomplikowana.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

(często w postaci wartości liczbowej odpowiadającej konkretnej reprezentacji bitowej). Pozwala to uniezależnić się w znacznej mierze od sprzętu bez nakładania nadmiernych ograniczeń na zakres serializowanych danych. Z punktu widzenia programisty możliwości oraz sposób użycia serializacji SOAP-owej jest zbliżony do serializacji binarnej. Wadą serializacji SOAP-owej w stosunku do pozostałych rodzajów serializacji jest to, że zwykle przechowywane za jej pomocą dane zajmują najwięcej miejsca.

Pojęcie serializera, najważniejsze klasy, przestrzenie, referencje

W odniesieniu do wszystkich trzech rodzajów serializacji schemat wykonywania operacji serializacji oraz deserializacji jest podobny. Najpierw tworzony jest obiekt **serializera** odpowiedniego typu oraz otwierany jest strumień serializacji. Następnie w przypadku serializacji serializer zapisuje dane do strumienia, a w przypadku deserializacji pobiera dane ze strumienia. Na koniec strumień jest zamknięty. W zależności od rodzaju serializacji serializer jest obiektem z następujących klas:

- dla *serializacji binarnej* jest to klasa **BinaryFormatter** z przestrzeni *System.Runtime.Serialization.Formatters.Binary*;
- dla *serializacji XML-owej* jest to klasa **XmlSerializer** z przestrzeni *System.Xml.Serialization*;
- dla *serializacji binarnej* jest to klasa **SapFormatter** z przestrzeni *System.Runtime.Serialization.Formatters.Sap*.

Używanie serializacji binarnej oraz XML-owej nie wymaga dodatkowych referencji. Aby można było używać serializacji SOAP-owej niezbędne jest dodanie referencji *System.Runtime.Serialization.Formatters.Sap*.

Serializacja typów wbudowanych

Najprostszym rodzajem danych do serializacji są zwykle typy wbudowane. W tym podrozdziale pokazane zostaną przykłady serializacji i deserializacji takich danych z poniżej wymienionych czterech typów:

- liczby całkowitej typu **int**,
- daty w postaci struktury **DateTime**,
- łańcucha znaków typu **string**,
- tablicy czterech elementów typu **double**.

Zostanie też pokazana struktura serializowanych danych.

Serializacja binarna

W przypadku serializacji binarnej w tym samym strumieniu może być umieszczonych więcej danych. W tym przypadku (i następnych) serializowane dane będą zapisywane do pliku (lub plików).

Oto przykład serializacji binarnej typów wbudowanych

```

int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };

FileStream fs = new FileStream("dane.dat", FileMode.Create);
BinaryFormatter bf = new BinaryFormatter();
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
bf.Serialize(fs, liczba);
bf.Serialize(fs, data);
bf.Serialize(fs, lancuch);
bf.Serialize(fs, tablica);
fs.Close();
```

a to jest przykład ich deserializacji

```
int liczba;
DateTime data;
string lancuch;
double[] tablica;

FileStream fs = new FileStream("dane.dat", FileMode.Open);
BinaryFormatter bf = new BinaryFormatter();

liczba = (int)bf.Deserialize(fs);
data = (DateTime)bf.Deserialize(fs);
lancuch = (string)bf.Deserialize(fs);
tablica = (double[])bf.Deserialize(fs);
fs.Close();

Console.WriteLine("liczba: {0}", liczba);
Console.WriteLine("data: {0}", data);
Console.WriteLine("łańcuch: {0}", lancuch);
Console.WriteLine("tablica: ");
for (int i = 0; i < tablica.Length; i++)
{
    Console.Write("\t " + tablica[i]);
}
Console.WriteLine();
```

Serializacja XML-owa

Serializacja XML-owa pozwala na przechowywanie w strumieniu serializacyjnym tylko jednego obiektu, z tego powodu w poniższym przykładzie serializowane dane zapisywane są do osobnych plików:

```
int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };

//XmlSerializer xs = new XmlSerializer(); // błąd
XmlSerializer xsInt = new XmlSerializer(typeof(int));
XmlSerializer xsDateTime = new XmlSerializer(typeof(DateTime));
XmlSerializer xsString = new XmlSerializer(typeof(string));
XmlSerializer xsTable = new XmlSerializer(typeof(double[]));

FileStream fs1 = new FileStream("dane1.xml", FileMode.Create);
xsInt.Serialize(fs1, liczba);
//xsInt.Serialize(fs1, 1234567);
fs1.Close();
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
FileStream fs2 = new FileStream("dane2.xml", FileMode.Create);
xsDateTime.Serialize(fs2, data);
fs2.Close();

FileStream fs3 = new FileStream("dane3.xml", FileMode.Create);
xsString.Serialize(fs3, lancuch);
fs3.Close();

FileStream fs4 = new FileStream("dane4.xml", FileMode.Create);
xsTable.Serialize(fs4, tablica);
fs4.Close();
```

W analogiczny sposób przebiega ich deserializacja:

```
int liczba, liczba2;
DateTime data;
string lancuch;
double[] tablica;

//XmlSerializer xs = new XmlSerializer(); // błąd
XmlSerializer xsInt = new XmlSerializer(typeof(int));
XmlSerializer xsDateTime = new XmlSerializer(typeof(DateTime));
XmlSerializer xsString = new XmlSerializer(typeof(string));
XmlSerializer xsTable = new XmlSerializer(typeof(double[]));

FileStream fs1 = new FileStream("dane1.xml", FileMode.Open);
liczba = (int)xsInt.Deserialize(fs1);
//liczba2 = (int)xsInt.Deserialize(fs1);

Console.WriteLine("liczba: {0}", liczba);
fs1.Close();

FileStream fs2 = new FileStream("dane2.xml", FileMode.Open);
data = (DateTime)xsDateTime.Deserialize(fs2);
Console.WriteLine("data: {0}", data);
fs2.Close();

FileStream fs3 = new FileStream("dane3.xml", FileMode.Open);
lancuch = (string)xsString.Deserialize(fs3);
Console.WriteLine("łańcuch: {0}", lancuch);
fs3.Close();

FileStream fs4 = new FileStream("dane4.xml", FileMode.Open);
tablica = (double[])xsTable.Deserialize(fs4);
Console.Write("tablica: ");
for (int i = 0; i < tablica.Length; i++)
{
    Console.Write("\t " + tablica[i]);
}
Console.WriteLine();
fs4.Close();
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Serializacja SOAP-owa

Serializacja SOAP-owa, podobnie jak serializacja binarnej pozwala na umieszczanie w tym samym strumieniu danych większej liczby obiektów lub zmiennych. Sam sposób używania serializatora też jest bardzo podobny.

Oto przykład serializacji SOAP-owej typów wbudowanych

```
int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };

FileStream fs = new FileStream("dane.dat", FileMode.Create);
SoapFormatter sf = new SoapFormatter();

sf.Serialize(fs, liczba);
sf.Serialize(fs, data);
sf.Serialize(fs, lancuch);
sf.Serialize(fs, tablica);
fs.Close();
```

a to jest przykład ich deserializacji

```
int liczba;
DateTime data;
string lancuch;
double[] tablica;

FileStream fs = new FileStream("dane.dat", FileMode.Open);
SoapFormatter sf = new SoapFormatter();

liczba = (int)sf.Deserialize(fs);
data = (DateTime)sf.Deserialize(fs);
lancuch = (string)sf.Deserialize(fs);
tablica = (double[])sf.Deserialize(fs);
fs.Close();

Console.WriteLine("liczba: {0}", liczba);
Console.WriteLine("data: {0}", data);
Console.WriteLine("łańcuch: {0}", lancuch);
Console.Write("tablica: ");
for (int i = 0; i < tablica.Length; i++)
{
    Console.Write("\t " + tablica[i]);
}
Console.WriteLine();
```

Porównanie struktury serializowanych danych

Na koniec przyjrzymy się temu, w jaki sposób zapisywane są serializowane dane. Warto przy tym zwrócić szczególną uwagę na sposób reprezentacji daty przechowywanej w zmiennej strukturalnej typu **DateTime**.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Podczas serializacji binarnej dane zapisane są w postaci binarnej i można je zobaczyć w odpowiednich edytora:

000000000	00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00 00
000000010	00 04 01 00 00 00 0C 53 79 73 74 65 6D 2E 49 6E System.In
000000020	74 33 32 01 00 00 00 07 6D 5F 76 61 6C 75 65 00	t32....m_value.
000000030	08 39 30 00 00 0B 00 01 00 00 00 FF FF FF FF 01	.90.....
000000040	00 00 00 00 00 00 00 04 01 00 00 00 OF 53 79 73 Sys
000000050	74 65 6D 2E 44 61 74 65 54 69 6D 65 02 00 00 00	tem.DateTime....
000000060	05 74 69 63 6B 73 08 64 61 74 65 44 61 74 61 00	.ticks.dateData.
000000070	00 09 10 FC A8 3B 9F B8 CB CF 08 FC A8 3B 9F B8;.....;
000000080	CB CF 88 0B 00 01 00 00 00 FF FF FF FF 01 00 00
000000090	00 00 00 00 00 06 01 00 00 00 10 C5 81 61 C5 84a..
0000000a0	63 75 63 68 20 64 61 6E 79 63 68 0B 00 01 00 00	cuch danych.....
0000000b0	00 FF FF FF FF 01 00 00 00 00 00 00 00 00 OF 01 00
0000000c0	00 00 04 00 00 00 06 00 00 00 00 00 00 00 F0 3F 00?.
0000000d0	00 00 00 00 00 04 40 9A 99 99 99 99 99 09 40 66@.....@f
0000000e0	66 66 66 66 66 10 40 0B	fffff.f.

W wyniku serializacji XML-owej dane zapisane zostały do osobnych plików, o następujących zawartościach:

```

<?xml version="1.0"?>
<int>12345</int>

<?xml version="1.0"?>
<dateTime>2013-01-28T21:54:01.140625+01:00</dateTime>

<?xml version="1.0"?>
<string>Łańcuch danych</string>

<?xml version="1.0"?>
<ArrayOfDouble xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <double>1</double>
  <double>2.5</double>
  <double>3.2</double>
  <double>4.1</double>
</ArrayOfDouble>
  
```

W wyniku serializacji SOAP-owej dane zapisane zostaną zapisane do pojedynczego pliku, którego zawartość będzie miała następującą postać:

```

<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <xsd:int id="ref-1">
      <m_value>12345</m_value>
    </xsd:int>
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<xsd:dateTime id="ref-1">
<ticks>634950070421250000</ticks>
<dateData>9858322107276025808</dateData>
</xsd:dateTime>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<SOAP-ENC:string id="ref-1">Łańcuch danych</SOAP-ENC:string>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:double[4]">
<item>1</item>
<item>2.5</item>
<item>3.2</item>
<item>4.1</item>
</SOAP-ENC:Array>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Serializacja własnej klasy

Rozważmy teraz sytuację, w której programista musi serializować obiekty zaimplementowanej przez siebie klasy. W zależności od rodzaju serializacji sposób serializacji i deserializacji oraz wymagania nakładane na klasę, której obiekty mają być serializowane są różne.

W przykładach zakładamy, że klasa ma zawierać cztery pola o takich samych typach (i przechowywanych wartościach), jak poprzednich przykładach.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Serializacja binarna

Klasa, której obiekty mają być poddane serializacji binarnej musi mieć ustawiony atrybut **Serializable**. Sama klasa ani jej pola nie muszą być publiczne. Klasa nie musi też posiadać zdefiniowanego przez programistę bezargumentowego konstruktora.

```
[Serializable] // Musi być ustawiony atrybut Serializable
class Dane // Nie musi być publiczna
{
    private int liczba; // Nie muszą być publiczne
    private DateTime data;
    private string lancuch;
    protected double[] tablica;

    public int Liczba { get { return liczba; } }
    public DateTime Data { get { return data; } }
    public string Lancuch { get { return lancuch; } }
    public double[] Tablica { get { return tablica; } }

    public Dane(int liczba, DateTime data, string lancuch, double[] tablica)
    {
        this.liczba = liczba;
        this.data = data;
        this.lancuch = lancuch;
        this.tablica = tablica;
    }
    // Nie musi być zdefiniowany
    //public Dane()
    //{
    //    this.liczba = 0;
    //    this.data = DateTime.MinValue;
    //    this.lancuch = String.Empty;
    //    this.tablica = null;
    //}
}
}
```

Sama serializacja binarna

```
int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };
Dane dane = new Dane(liczba, data, lancuch, tablica);

FileStream fs = new FileStream("dane.dat", FileMode.Create);
BinaryFormatter bf = new BinaryFormatter();

bf.Serialize(fs, dane);

fs.Close();
```

oraz deserializacja

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
FileStream fs = new FileStream("dane.dat", FileMode.Open);
BinaryFormatter bf = new BinaryFormatter();

Dane dane = (Dane)bf.Deserialize(fs);
fs.Close();

Console.WriteLine("liczba: {0}", dane.Liczba);
Console.WriteLine("data: {0}", dane.Data);
Console.WriteLine("łańcuch: {0}", dane.Lancuch);
Console.WriteLine("tablica: ");
for (int i = 0; i < dane.Tablica.Length; i++)
{
    Console.Write("\t " + dane.Tablica[i]);
}
Console.WriteLine();
```

takiej klasy nie różnią się zbytnio od serializacji typów prostych.

Serializacja XML-owa

Klasa, której obiekty mają przejść proces serializacji XML-owej nie musi mieć ustawionych żadnych atrybutów. Musi być jednak klasą publiczną. Ponadto jeśli jest klasą wewnętrzną, to wszystkie klasy w których jest ona zawarta również muszą być klasami publicznymi. Klasa taka musi posiadać bezargumentowy konstruktor (taki konstruktor musi być jawnie zdefiniowany przez programistę o ile klasa posiada jakikolwiek inny konstruktor). Ostatnim i najbardziej uciążliwym wymogiem jest to, że wszystkie pola takiej klasy muszą być publiczne, aby można było je serializować. W istotny sposób kłuci się to z zasadą hermetyzacji klas oraz podważa zasadność używania własności do kontroli dostępu do wybranych pól.

Oto przykład klasy, której obiekty będą mogły być serializowane w taki sposób

```
public class Dane // Musi być publiczna
{
    //int liczba;
    //DateTime data;
    //string lancuch;
    //double[] tablica;

    public int Liczba { get { return liczba; } }
    public DateTime Data { get { return data; } }
    public string Lancuch { get { return lancuch; } }
    public double[] Tablica { get { return tablica; } }

    public int liczba; // Muszą być publiczne
    public DateTime data;
    public string lancuch;
    public double[] tablica;

    public Dane(int liczba, DateTime data, string lancuch, double[] tablica)
    {
        this.liczba = liczba;
        this.data = data;
        this.lancuch = lancuch;
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        this.tablica = tablica;
    }

    public Dane() // Musi być zdefiniowany
    {
        this.liczba = 0;
        this.data = DateTime.MinValue;
        this.lancuch = String.Empty;
        this.tablica = null;
    }
}
```

Aby przeprowadzić XML-ową serializację obiektu takiej klasy należy serializatorowi przekazać informację o wszystkich używanych przez tą klasę typach:

```
int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };

Dane dane = new Dane(liczba, data, lancuch, tablica);
FileStream fs = new FileStream("dane.xml", FileMode.Create);
Type[] dodatkoweTypy = new Type[4]
{
    typeof(int),
    typeof(DateTime),
    typeof(string),
    typeof(double[])
};
XmlSerializer xs = new XmlSerializer(typeof(Dane), dodatkoweTypy);
xs.Serialize(fs, dane);
fs.Close();
```

Dane te trzeba też przekazać przy deserializacji

```
FileStream fs = new FileStream("dane.xml", FileMode.Open);

Type[] dodatkoweTypy = new Type[4]
{
    typeof(int),
    typeof(DateTime),
    typeof(string),
    typeof(double[])
};

XmlSerializer xs = new XmlSerializer(typeof(Dane), dodatkoweTypy);
Dane dane = (Dane)xs.Deserialize(fs);
fs.Close();

Console.WriteLine("liczba: {0}", dane.Liczba);
Console.WriteLine("data: {0}", dane.Data);
Console.WriteLine("łańcuch: {0}", dane.Lancuch);
Console.Write("tablica: ");
for (int i = 0; i < dane.Tablica.Length; i++)
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
  Console.WriteLine("\t " + dane.Tablica[i]);
}
Console.WriteLine();

```

Serializacja SOAP-owa

W przypadku serializacji SOAP-owej klas obowiązują te same zasady, co przy serializacji binarnej. Klasę

```

[Serializable] // Musi być ustawiony atrybut Serializable
class Dane // Nie musi być publiczna
{
  int liczba; // Nie muszą być publiczne
  DateTime data;
  string lancuch;
  double[] tablica;

  public int Liczba { get { return liczba; } }
  public DateTime Data { get { return data; } }
  public string Lancuch { get { return lancuch; } }
  public double[] Tablica { get { return tablica; } }

  public Dane(int liczba, DateTime data, string lancuch, double[] tablica)
  {
    this.liczba = liczba;
    this.data = data;
    this.lancuch = lancuch;
    this.tablica = tablica;
  }
  // Nie musi być zdefiniowany
  //public Dane()
  //{
  //  this.liczba = 0;
  //  this.data = DateTime.MinValue;
  //  this.lancuch = String.Empty;
  //  this.tablica = null;
  //}
}

```

serializujemy w następujący sposób

```

int liczba = 12345;
DateTime data = DateTime.Now;
string lancuch = "Łańcuch danych";
double[] tablica = { 1.0, 2.5, 3.2, 4.1 };
Dane dane = new Dane(liczba, data, lancuch, tablica);

FileStream fs = new FileStream("dane.dat", FileMode.Create);
SoapFormatter sf = new SoapFormatter();

sf.Serialize(fs, dane);

fs.Close();

```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

i deserializujemy następująco

```
FileStream fs = new FileStream("dane.dat", FileMode.Open);
SoapFormatter sf = new SoapFormatter();

Dane dane = (Dane) sf.Deserialize(fs);
fs.Close();

Console.WriteLine("liczba: {0}", dane.Liczba);
Console.WriteLine("data: {0}", dane.Data);
Console.WriteLine("łańcuch: {0}", dane.Lancuch);
Console.Write("tablica: ");
for (int i = 0; i < dane.Tablica.Length; i++)
{
    Console.Write("\t " + dane.Tablica[i]);
}
Console.WriteLine();
```

Zaawansowane sposoby serializacji

Ostatnia część tego rozdziału poświęcona została przedstawieniu wybranych zaawansowanych mechanizmów serializacji.

Wybiórcza serializacja i implementacja interfejsu *IDeserializationCallback*

Przy serializacji binarnej oraz SOAP-owej możemy atrybutami **NonSerialized** wskazywać pola, które nie mają być serializowane (np. w przypadku, gdy ich wartości mogą być w prosty sposób wyliczone na podstawie wartości innych pól). Istnieje wiele sposobów, na które można albo w trakcie deserializacji albo tuż po niej inicjalizować wartości takich pól. Jednym z rozwiązań jest zaimplementowanie dla serializowanej klasy metody **OnDeserialization** z interfejsu **IDeserializationCallback** (z przestrzeni *System.Runtime.Serialization*). Wówczas wspomniana metoda będzie wywoływana po deserializacji.

Oto przykład użycia atrybutu *NonSerialized* oraz implementacji interfejsu *IDeserializationCallback*:

```
[Serializable] // Musi być ustwiony atrybut Serializable
class Dane : IDeserializationCallback // Nie musi być publiczna
{
    private int liczba1, liczba2; // Nie muszą być publiczne
    [NonSerialized]
    private int suma, roznica, iloczyn;

    public int Liczba1 { get { return liczba1; } }
    public int Liczba2 { get { return liczba2; } }
    public int Suma { get { return suma; } }
    public int Roznica { get { return roznica; } }
    public int Iloczyn { get { return iloczyn; } }

    public Dane(int liczba1, int liczba2)
    {
        this.liczba1 = liczba1;
        this.liczba2 = liczba2;
        suma = liczba1 + liczba2;
    }

    void IDeserializationCallback.OnDeserialization(object sender, DeserializationEventArgs e)
    {
        suma = e.GetValue("suma");
        roznica = e.GetValue("roznica");
        iloczyn = e.GetValue("iloczyn");
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    roznica = liczba1 - liczba2;
    iloczyn = liczba1 * liczba2;
}
public void OnDeserialization(object sender)
{
    Console.WriteLine("Wyliczanie operacji");
    suma = liczba1 + liczba2;
    roznica = liczba1 - liczba2;
    iloczyn = liczba1 * liczba2;
}
}
  
```

Używanie atrybutów metod

Innym sposobem na wykonywanie dodatkowych operacji w trakcie deserializacji lub serializacji jest stworzenie w klasie odpowiednich metod i opatrzenie ich stosownymi atrybutami. W szczególności:

- metoda opatrzona atrybutem **OnSerializing** zostanie wywołana w trakcie (na początku) serializacji;
- metoda opatrzona atrybutem **OnSerialized** zostanie wywołana zaraz po zakończeniu serializacji;
- metoda opatrzona atrybutem **OnDeserializing** zostanie wywołana w trakcie (na początku) deserializacji;
- metoda opatrzona atrybutem **OnDeserialized** zostanie wywołana zaraz po zakończeniu deserializacji.

Aby lepiej zrozumieć działanie tych atrybutów posłużymy się przykładem. Podczas serializacji następującej klasy

```

[Serializable] // Musi być ustawiony atrybut Serializable
class Dane : IDeserializationCallback // Nie musi być publiczna
{
    private int liczba1, liczba2; // Nie muszą być publiczne
    [NonSerialized]
    private int suma, roznica, iloczyn;

    public int Liczba1 { get { return liczba1; } }
    public int Liczba2 { get { return liczba2; } }
    public int Suma { get { return suma; } }
    public int Roznica { get { return roznica; } }
    public int Iloczyn { get { return iloczyn; } }

    public Dane(int liczba1, int liczba2)
    {
        this.liczba1 = liczba1;
        this.liczba2 = liczba2;
        suma = liczba1 + liczba2;
        roznica = liczba1 - liczba2;
        iloczyn = liczba1 * liczba2;
    }

    public void OnDeserialization(object sender)
    {
        Console.WriteLine("Wyliczanie operacji");
        suma = liczba1 + liczba2;
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    roznica = liczba1 - liczba2;
    iloczyn = liczba1 * liczba2;
}

[OnSerializing]
internal void MyOnSerializing(StreamingContext context)
{
    Console.WriteLine("MyOnSerializing");
}

[OnSerialized]
internal void MyOnSerialized(StreamingContext context)
{
    Console.WriteLine("MyOnSerialized");
}

[OnDeserializing]
internal void MyOnDeserializing(StreamingContext context)
{
    Console.WriteLine("MyOnDeserializing");
}

[OnDeserialized]
internal void MyOnDeserialized(StreamingContext context)
{
    Console.WriteLine("MyOnDeserialized");
}
}
  
```

zostanie wypisany na standardowe wyjście następujący komunikat

```

MyOnSerializing
MyOnSerialized
  
```

Natomiast przy deserializacji tej klasy otrzymamy następującą informację

```

MyOnDeserializing
MyOnDeserialized
Wyliczanie operacji
  
```

Implementacja interfejsu **ISerializable**

Jeszcze innym sposobem na zaimplementowanie własnych mechanizmów serializacji jest przekazywanie zdefiniowanych przez użytkownika dodatkowych danych do strumienia serializacji oraz późniejsze ich wydobywanie w odpowiednim konstruktorze przy deserializacji. Dołączanie dodatkowych informacji do strumienia serializacji może zostać zrealizowane przez odpowiednie zaimplementowanie metody **GetObjectData** z interfejsu **ISerializable**, a pobieranie tych informacji może zostać przeprowadzone w konstruktorze, którego argumenty są odpowiednio typów **SerializationInfo** oraz **StreamingContext**. Zostało to zilustrowane na poniższym przykładzie.

```
[Serializable]
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

class Dane : ISerializable
{
    [NonSerialized]
    private int liczba1, liczba2;
    [NonSerialized]
    private int suma, roznica, iloczyn;

    public int Liczba1 { get { return liczba1; } }
    public int Liczba2 { get { return liczba2; } }
    public int Suma { get { return suma; } }
    public int Roznica { get { return roznica; } }
    public int Iloczyn { get { return iloczyn; } }

    public Dane(int liczba1, int liczba2)
    {
        this.liczba1 = liczba1;
        this.liczba2 = liczba2;
        suma = liczba1 + liczba2;
        roznica = liczba1 - liczba2;
        iloczyn = liczba1 * liczba2;
    }

    public Dane(SerializationInfo sinfo, StreamingContext ctx)
    {
        Console.WriteLine("    pobieranie informacji");
        liczba1 = sinfo.GetInt32("L1");
        liczba2 = sinfo.GetInt32("L2");
        suma = liczba1 + liczba2;
        roznica = liczba1 - liczba2;
        iloczyn = liczba1 * liczba2;
    }

    // [SecurityPermissionAttribute(SecurityAction.Demand,
    // SerializationFormatter = true)]
    void ISerializable.GetObjectData(SerializationInfo sinfo, StreamingContext
ctx)
    {
        Console.WriteLine("    dodawanie informacji");
        sinfo.AddValue("L1", liczba1);
        sinfo.AddValue("L2", liczba2);
    }

    // Nie dziedziczymy po interfejsie IDeserializationCallback,
    // na skutek czego poniższa metoda nie zostanie wywołana
    public void OnDeserialization(object sender)
    {
        Console.WriteLine("Wyliczanie operacji");
        suma = liczba1 + liczba2;
        roznica = liczba1 - liczba2;
        iloczyn = liczba1 * liczba2;
    }

    [OnSerializing]
    internal void MyOnSerializing(StreamingContext context)
    {
        Console.WriteLine("    MyOnSerializing");
    }
}

```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

}

[OnSerialized]
internal void MyOnSerialized(StreamingContext context)
{
    Console.WriteLine("  MyOnSerialized");
}

[OnDeserializing]
internal void MyOnDeserializing(StreamingContext context)
{
    Console.WriteLine("  MyOnDeserializing");
}

[OnDeserialized]
internal void MyOnDeserialized(StreamingContext context)
{
    Console.WriteLine("  MyOnDeserialized");
}

}
  
```

W wyniku serializacji powyższej klasy zostanie wypisany komunikat

```

MyOnSerializing
    dodawanie informacji
MyOnSerialized
  
```

Natomiast podczas jej deserializacji otrzymamy następującą informację

```

MyOnDeserializing
    pobieranie informacji
MyOnDeserialized
  
```

Opcjonalne serializowane wybranych elementów

W podanych powyżej przykładach pola opatrzone atrybutem *NonSerialized* nie były serializowane. Wstawienie wówczas jakichkolwiek informacji o tych polach do strumienia serializacji spowodowałoby błąd przy deserializacji, gdyż liczba danych w strumieniu byłaby większa od danych, które należałyby deserializować.

Czasami w przypadku pól, których wartości mogą być wyliczone w inny sposób wygodniej jest tak zaimplementować klasę, aby zarówno umieszczenie w strumieniu serializacji wartości tych pól, jak ich brak nie powodowały błędów. W tym celu wystarczy takie pola opatrzyć atrybutami *OptionalField* z przestrzeni *System.Runtime.Serialization*. Przy serializacji informacje o wartości takich pól domyślnie będą dołączane do strumienia serializacji.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

XML-owa serializacja zbiorów danych

Dla serializacji XML-owej zostały zaimplementowane specjalne konstrukcje ułatwiające serializację i deserializację danych o strukturze tabel z relacyjnej bazy danych. Tego typu dane przechowywane są w postaci zbiorów (typu **DataSet**) podzielonych na tabele (typu **DataTable**), w których z kolei dane ułożone są w wierszach (typu **DataRow**) i kolumnach (typu **DataColumn**).

Poniżej przedstawiony został przykład, w którym przeprowadzana jest serializacja zorganizowanych w ten sposób danych

```
XmlSerializer xs = new XmlSerializer(typeof(DataSet));
DataSet ds = new DataSet("Zbior");
DataTable dt = new DataTable("Tabela");
DataColumn dc1 = new DataColumn("PierwszaKolumna");
DataColumn dc2 = new DataColumn("DrugaKolumna");
dt.Columns.Add(dc1);
dt.Columns.Add(dc2);
ds.Tables.Add(dt);
DataRow dr;
for (int i = 0; i < 5; i++)
{
    dr = dt.NewRow();
    dr[0] = "Element A" + i.ToString();
    if (i < 4) dr[1] = "Element B" + i.ToString();
    dt.Rows.Add(dr);
}
TextWriter tw = new StreamWriter("dane.xml");
xs.Serialize(tw, ds);
tw.Close();
```

oraz przykład z przeprowadzeniem deserializacji tych danych.

```
XmlSerializer xs = new XmlSerializer(typeof(DataSet));
TextReader tr = new StreamReader("dane.xml");
DataSet ds = (DataSet)xs.Deserialize(tr);
tr.Close();

Console.WriteLine(ds.DataSetName);
foreach (DataTable dt in ds.Tables)
{
    Console.WriteLine(" " + dt.TableName);
    foreach (DataRow dr in dt.Rows)
    {
        //foreach (var element in dr.ItemArray)
        //{
        //    Console.WriteLine(" " + element);
        //}
        Console.WriteLine(" Nowy wiersz:");
        foreach (DataColumn dc in dt.Columns)
        {
            string field = dr.Field<string>(dc.ColumnName);
            if (field != null)
            {
                Console.WriteLine(" " + dc.ColumnName + ": " + field);
            }
        }
    }
}
```



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        }  
    }  
    //Console.WriteLine("      " + dr.ColumnName + ": " + dr);  
}  
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

5. Wprowadzenie do Windows Presentation Foundation

Materiał zawarty w niniejszym temacie został podzielony na dwie części: język opisu interfejsu XAML oraz programowanie w („czystym”) języku C#. Pomijam generowanie kodu metodą „przeciągnij-upuść” w „Designerze”. Jak sama nazwa tematu sugeruje, zagadnienia będą omawiane wyjątkowo pobiżnie. Nacisk będzie położony na wskazanie elementów, które mogą być pomocne do tworzenia interfejsu aplikacji (np. kontrolki, style, menu), przy założeniu, że zainteresowane osoby będą mogły we własnym zakresie wyszukać szczegółowe informacje np. o sposobie używania konkretnych kontrolek. Istotną część wykładu powinny stanowić prezentacje uruchamiania kodu w środowisku Visual Studio (nie zawarte w niniejszym skrypcie).

Podstawowe informacje o WPF

WPF (*Windows Presentation Foundation*) służy do budowy interfejsu graficznego aplikacji. W zamierzeniach miał/ma być następcą *Windows Forms*.

Do opisu tworzonego w WPF dla aplikacji interfejsu użyto języka **XAML** (*eXtensible Application Markup Language*). Jak sama nazwa wskazuje XAML jest językiem znaczników (bazuje na XML-owych znacznikach). Programista w WPF-ie ma pełen dostęp do języka opisu interfejsu co daje mu większą kontrolę nad generowanym kodem niż miało to miejsce w przypadku *Windows Forms*.

Jest wiele czynników, które powodują, że nadal często wybiera się *Windows Forms* zamiast WPF. Są to między innymi:

- przyzwyczajenia programistów,
- duża ilość stworzonego wcześniej kodu w *Windows Forms* (zwłaszcza w przypadku kontynuacji starszych projektów),
- duża ilość kontrolek stworzonych dla *Windows Forms*,
- możliwości implementacji (np. w mono nie można tworzyć aplikacji WPF, ale można Windows Forms).

Programista może tworzyć interfejs WPF na wiele różnych sposobów. Używając samego środowiska Visual Studio może on:

- projektować interfejs metodą „przeciągnij i upuść” w graficznym oknie Design, przeciągając potrzebne kontrolki z rozwijanej (lub dokowalnej) zakładki ToolBox (podobnie jak w *Windows Forms*);
- dodawać potrzebne kontrolki i inne elementy interfejsu (np. style) modyfikować elementy już istniejące bezpośrednio w kodzie opisującym interfejs w oknie XAML (czego w Windows Forms nie było);
- modyfikować właściwości kontrolek w oknie Properties (podobnie jak w *Windows Forms*);
- dodawać kontrolki i modyfikować ich właściwości z menu kontekstowego;
- używać równocześnie dwóch lub więcej z wymienionych mechanizmów, co często jest najefektywniejszym rozwiązaniem.

Interfejs użytkownika Visual Studio został zaprojektowany tak, aby jednoczesne korzystanie ze wspomnianych rozwiązań było jak wygodniejsze. Dla projektów w WPF domyślnie główna część

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

interfejsu IDE jest współdzielona pomiędzy okna *Design* i *XAML* podczas gdy okno *Properties* znajduje się z prawej strony, a z lewej strony można rozwijać (lub zadokować) okno *ToolBox* z potrzebnymi kontrolkami. Dostępne są też przyciski pozwalające na szybkie przełączanie widoków, np. aby ukryć okno *Design* lub *XAML* lub zmienić sposób współdzielenia przez nie głównej przestrzeni. Oczywiście użytkownik ma dużą kontrolę nad sposobem rozmieszczenia poszczególnych okien, jednak indywidualne ich przenoszenie, dokowanie oraz zmiana rozmiaru zajmuje niewątpliwie znacznie więcej czasu.

XAML jako język opisu okien

W niniejszych materiałach zakłada się, że przeznaczenie i ogólne zasady działania typowych rodzajów kontrolek takich jak *Label* (etykieta), *TextBox* (pole tekstowe), *Button* (przycisk), *RadioButton*, *CheckBox*, *ComboBox* są czytelnikowi znane.

Składowe opisu okna

W momencie utworzenia nowego projektu aplikacji WPF zostanie wygenerowany następujący kod dla głównego okna (na razie pustego) tej aplikacji:

```
<Window x:Class="EmptyWpfApplication.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
    <Grid>
    </Grid>
</Window>
```

W parametrach znacznika *Window* wyszukać można właściwości okna (tutaj: wyświetlany nagłówek, wysokość i szerokość). Zauważać też można przestrzenie nazw, do których można się odwoływać wewnątrz okna.

Wewnątrz znacznika *Window* umieszczany jest zwykle znacznik kontrolki, w której osadzać można właściwe elementy okna. W tym przypadku jest to kontrolka *Grid*, użyteczna zwłaszcza przy osadzaniu typowych kontrolek interfejsu użytkownika. W przypadku tworzenia okien z grafiką wygodniej byłoby w tym celu użyć kontrolki *Canvas*.

W samym znaczniku okna można też zagnieździć (tutaj nieobecny) znacznik zasobów

```
<Window.Resources></Window.Resources>
```

w którym umieszczać można zasoby używane dla tego okna, takie jak np. omawiane później statyczne style kontrolek lub menu kontekstowe.

Właściwości kontrolek na przykładzie pozycjonowania

Ten fragment ma służyć wskazaniu problemów, jakie można napotkać podczas używania niektórych właściwości kontrolek. Dodatkowym jego celem jest zasygnalizowanie sposobu pozycjonowania kontrolek w wyświetlanym oknie.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Zauważmy, że w większości przypadków użytkownik ma możliwość zmiany wysokości i szerokości okna aplikacji, co może powodować potrzebę zmiany położenia znajdujących się w nim kontrolek. Położenie oraz rozmiary kontrolki możemy ustalać albo jako ściśle ustalone albo jako zależne od rozmiaru kontrolki, w której się ona znajduje. Przykładowo, podobnie jak przy ustawianiu rozmieszczenia tekstu możemy chcieć, aby kontrolka była albo wyrównywana do lewego marginesu albo do prawego marginesu albo justowana albo centrowana, podając jej argument `HorizontalAlignment="ppoz"`, gdzie „`ppoz`” przyjmuje jedną z wartości odpowiednio „`Left`”, „`Right`”, „`Stretch`” lub „`Center`”. W analogiczny sposób możemy ustawać ułożenie w pionie, podając argument `VerticalAlignment="ppion"`, gdzie „`ppion`” przyjmuje jedną z wartości odpowiednio „`Top`”, „`Bottom`”, „`Stretch`” lub „`Center`”.

Niektóre argumenty mogą mieć przypisywane wartości na wiele różnych sposobów. Dla przykładu wartości marginesu kontrolki możemy podawać w argumencie **margin** na trzy różne sposoby.:

```
Margin="x1,y1,x2,y2"
Margin="x,y"
Margin="w"
```

gdzie

- „`x1,y1,x2,y2`” są wartościami odpowiednio lewego, górnego, prawego i dolnego marginesu,
- „`x`” jest wartością lewego oraz prawego marginesu,
- „`y`” jest wartością górnego oraz dolnego marginesu,
- „`w`” jest wartością marginesu z każdej z czterech stron.

Należy też pamiętać, że kontrolki mogą mieć różne własności, których ustawienia czasem mogą się wzajemnie wykluczać. W takich wypadkach zwykle ustawienia jednych własności mają większy priorytet nad pozostałymi. Dla przykładu szerokość kontrolki może być kontrolowana przez własności **Width**, **MaxWidth**, **MinWidth** (zakładam, że rola tych własności jest dla wszystkich oczywista – przypominam, że rozmiar kontrolki może ulegać zmianie wraz ze zmianą rozmiarów kontrolki, w której jest ona zawarta), przy czym najwyższy priorytet mają wartości podane we właściwości **MinWidth**, później we właściwości **MaxWidth**, a na końcu we właściwości **Width**. Czyli np. kontrolka, której przypiszemy we właściwości **MinWidth** wartość 100, a we właściwościach **MaxWidth** i **Width** odpowiednio 5 i 10 będzie miała szerokość 100.

W analogiczny sposób wysokość kontrolki może być kontrolowana przez właściwości **Height**, **MaxHeight**, **MinHeight**, które zostały tu podane zgodnie z rosnącym priorytetem.

Zagnieżdżanie znaczników

Należy pamiętać o tym, że zwykle ten sam efekt można osiągnąć na więcej niż jeden sposób. W szczególności od programisty zależy, czy będzie on używał konwencji z możliwie „płaską” strukturą dokumentu XML-owego w którym występuje mniejsza liczba znaczników, które posiadają dużo parametrów, czy też zdecyduje się na wariant, w którym będzie dużo zagnieżdzonych znaczników XML-owych, które będą miały niewiele parametrów (lub wcale ich nie będą miały). Przykładowo następujący „płaski” opis kontrolki przycisku

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
<Button FontSize="22" Background="Blue" Name="przycisk1" Content="Przycisk"/>
```

jest równoważny następującemu „zagnieżdzonemu” opisowi

```
<Button>
  <Button.FontSize>
    22
  </Button.FontSize>
  <Button.Background>
    <SolidColorBrush Color="Blue"/>
  </Button.Background>
  <Button.Name>
    przycisk1
  </Button.Name>
  <Button.Content>
    Przycisk
  </Button.Content>
</Button>
```

W powyższym przykładzie hierarchiczna struktura ostatniego wariantu jest mniej czytelna i zaciemnia kod. Jednak w wielu przypadkach, zwłaszcza kontrolek, które posiadają wiele opcji wyboru, zagnieżdzony sposób definiowania jest bardzo wygodny, np.:

```
<ComboBox Height="23" HorizontalAlignment="Right" Margin="0,106,12,0"
  Name="comboBox1" VerticalAlignment="Top" Width="112">
  <ComboBox.Items>
    <ComboBoxItem>
      Wybór 1
    </ComboBoxItem>
    <ComboBoxItem>
      Wybór 2
    </ComboBoxItem>
    <ComboBoxItem>
      Wybór 3
    </ComboBoxItem>
  </ComboBox.Items>
</ComboBox>
```

Ponadto w wielu przypadkach sposób zagnieżdżenia kontrolek w istotny sposób wpływa na sposób interakcji między nimi. Dla przykładu jest tak dla zgrupowanych razem kontrolek *RadioButton*, np.:

```
<GroupBox Header="GroupBox" Margin="50,0,0,27" Name="gBox1" Height="86"
  HorizontalAlignment="Left" VerticalAlignment="Bottom" Width="75">
  <StackPanel Height="66" Name="stackPanel1" Width="64">
    <RadioButton Height="15" Name="rB1" Width="50">1</RadioButton>
    <RadioButton Height="15" Name="rB2" Width="50">2</RadioButton>
    <RadioButton Height="15" Name="rB3" Width="50">3</RadioButton>
    <RadioButton Height="15" Name="rB4" Width="50">4</RadioButton>
  </StackPanel>
</GroupBox>
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Grupowanie kontrolek i odwoływanie się do nich

Do kontrolek odwołujemy się zwykle (zwłaszcza w kodzie w języku C#) po ich nazwach zdefiniowanych parametrem *Name*. W przypadku niektórych kontrolek zachodzi potrzeba ich jawnego lub niejawnego pogrupowania. Przykładowo jest tak w przypadku kontroliki *RadioButton*, która charakteryzuje się tym, że w każdym momencie powinna być zaznaczona dokładnie jedna kontrolka tego typu z grupy. Zwykle, jeśli nie wskażemy inaczej, to grupowane są razem wszystkie kontroliki z danego poziomu zagłędnienia. W przypadku tej kontroliki poprzez parametr *GroupName* możemy jawnie wskazać, do której grupy chcemy ją przypisać. Należy pamiętać, że przy wskazywaniu grupy możemy ją podać w sposób względny lub bezwzględny. Jako zadanie do zastanowienia podaję poniższy przykład z pytaniem na ile grup (i jakie) zostały podzielone kontroliki *RadioButton* z poniższego kodu:

```
<StackPanel Height="66" Name="stackPanel1" Width="64">
  <RadioButton Height="15" Name="rB1" Width="50">1</RadioButton>
  <RadioButton Height="15" Name="rB2" Width="50">2</RadioButton>
  <RadioButton Height="15" Name="rB3" Width="50"
    GroupName="A">3</RadioButton>
  <RadioButton Height="15" Name="rB4" Width="50"
    GroupName="A">4</RadioButton>
</StackPanel>
<StackPanel Height="66" Name="stackPanel2" Width="64">
  <RadioButton Height="15" Name="rB5" Width="50">5</RadioButton>
  <RadioButton Height="15" Name="rB6" Width="50">6</RadioButton>
  <RadioButton Height="15" Name="rB7" Width="50"
    GroupName="A">7</RadioButton>
  <RadioButton Height="15" Name="rB8" Width="50"
    GroupName="stackPanel1.A">8</RadioButton>
</StackPanel>
```

Kolejność kontrolek – przykrywanie

Domyślnie kolejne kontroliki przykrywają poprzednio umieszczone kontroliki (na tych obszarach, na których na nie nadchodzą). Używając własności **Panel.ZIndex** można bezpośrednio, przypisać „głębokość” umieszczenia kontroliki, wpływając na kolejność nachodzenia na siebie kontrolek

Panel.ZIndex = "głębokość"

Kontrolki mogą mieć też ustawiony poziom przezroczystości. Poziom przezroczystości można ustawać między innymi jako jedną ze składowych koloru.

Importowanie kontrolek z Windows Forms

W Windows Presentation Foundation można używać kontrolek z Windows Forms. W tym celu potrzeba wykonać kilka operacji. Przede wszystkim należy do projektu WPF dodać referencje *WindowsFormsIntegration* oraz *System.Windows.Forms*. Dla uproszczenia zapisu warto jest w deklaracji przestrzeni nazw dodać nową przestrzeń do Windows Forms np.:

xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

(tutaj została ona oznaczona jako **wf**). Następnie importowaną kontrolkę ze zdefiniowanej wcześniej przestrzeni wstawiamy w kontrolce *WindowsFormsHost*, np.:

```
<WindowsFormsHost Margin="150,70,100,0"
    Name="windowsFormsHost1"
    Height="30" VerticalAlignment="Top">
    <wf:DateTimePicker Name="wstawianaData"/>
</WindowsFormsHost>
```

Definiowanie i używanie stylów dynamicznych

Jeśli chcemy, aby użyta przez nas kontrolka wyglądała lub zachowywała się w nieco inny niż domyślny sposób, to możemy zdefiniować dla niej odpowiedni styl. Najprościej jest umieścić taki styl bezpośrednio w zasobach tej kontrolki i ładować go jako zasób dynamiczny, np.:

```
<Button Style="{DynamicResource myButtonStyle}"
    Height="23" Width="75" ...>
    <Button.Resources>
        <Style x:Key="myButtonStyle">
            <Setter Property="Button.Background"
                Value="#00808080" />
            <Setter Property="Button.Foreground"
                Value="White" />
            <Setter Property="Button.FontFamily"
                Value="Comic Sans MS" />
        </Style>
    </Button.Resources>
    Klawisz ...
</Button>
```

Warto zauważyć, że zamiana w powyższym przykładzie rodzaju ładowanego zasobu z „*DynamicResource*” na „*StaticResource*” spowodowałaby błąd uruchomienia, ponieważ styl jest definiowany w zasobach kontrolki i nie jest znany przed rozpoczęciem jej ładowania.

Style globalne

Zaletą dynamicznego ładowania stylu jest to, że ładowany jest on dopiero przy ładowaniu kontrolki, przez co proces uruchamiania aplikacji jest szybszy. Ma to niestety też swoje wady. Samo ładowanie kontrolki jest wolniejsze. Ponadto takiego stylu nie można użyć w innych kontrolkach, co wymusza definiowanie stylów osobno dla każdej kontrolki. W efekcie przy wielu różnych stylach dynamicznych każdy styl musi być ładowany osobno, co nie jest zbyt wydajnym rozwiązaniem.

Można temu zaradzić przez zdefiniowanie stylu w bardziej globalnych zasobach – dla przykładu zasobach okna albo nawet zasobach aplikacji. Poniższy przykład pokazuje jak przenieść styl do zasobów okna.

```
<Window.Resources>
    <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="Background"
            Value="#00808080" />
        <Setter Property="Foreground"
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    Value="White" />
<Setter Property="FontFamily"
       Value="Comic Sans MS" />
</Style>
</Window.Resources>
```

W przykładzie typ kontrolki „*Button*” został przeniesiony z przestrzeni nazw poszczególnych właściwości do ustawień stylu (argumentu *TargetType*). Warto w tym miejscu zaznaczyć, że aby używać tego stylu do innych rodzajów kontrolek można zmienić docelowy typ kontrolki (podawany w *TargetType*) na bardziej ogólny typ, np. *Control*.

Statyczne ładowanie stylów

Poprzednio zwróciliśmy uwagę, że dynamiczne ładowanie stylów w przypadku stylów, które są często używane jest mało wydajne. Jeśli dany styl został wcześniej zdefiniowany, w szczególności, jeśli został on zdefiniowany globalnie, to można go załadować statycznie. W tym celu należy wpisać

```
Style="{StaticResource nazwaStylu}"
```

zamiast

```
Style="{DynamicResource nazwaStylu}"
```

Należy pamiętać, że próba przypisania statycznie stylu który nie został wcześniej zdefiniowany może skończyć się błędem uruchomieniowym (*runtime error*).

Wyzwalacze

W stylach można definiować wyzwalacze powodujące zmiany w stylu w przypadku wystąpienia określonych stanów, np.:

```

<Window.Resources>
  <Style x:Key="buttonStyle" TargetType="Button">
    <Setter Property="Background"
           Value="#00808080" />
    <Setter Property="Foreground"
           Value="White" />
    <Setter Property="FontFamily"
           Value="Comic Sans MS" />
    <Style.Triggers>
      <Trigger Property="IsMouseOver"
                Value="True">
        <Setter Property="Background"
               Value="Green" />
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Menu

W WPF-ie używając kontrolki *Menu* w prosty sposób definiować można typowe rozwijane menu. Zwykle osadza się je w kontrolce *DockPanel* pełniącej rolę belki. Poszczególne wpisy realizowane są przez kontrolki *MenuItem*, które można zagnieździć. Własność *Header* tych kontrolek służy do podania wyświetlanej w menu nazwy, przy czym dodatkowo symbolem podkreślenia „_” można wskazać literę skrótu klawiaturowego odpowiadającego wybraniu stosowej opcji. Własność *Click* służy do wskazania metody wywoływanej w przypadku wcisnięcia danej pozycji z menu (więcej informacje dotyczące implementacji tych metod pojawią się w dalszej części tego tematu). W kontrolkach *MenuItem* można osadzać też np. ikonki pojawiające się przy pozycji z menu oraz belki separatora oddzielające poszczególne pozycje wpisów. Oto przykład definicji menu.:

```
<DockPanel Height="20" Margin="0" Name="dockPanel1"
  VerticalAlignment="Top">
  <Menu Height="Auto" Name="menu1" Width="Auto">
    <MenuItem Height="Auto" Header="_File" Width="Auto">
      <MenuItem Header="_Otwórz" />
      <MenuItem Header="Zapisz" Click="ZapiszDane">
        <MenuItem.Icon>
          <Image Source="ikonka.bmp"/>
        </MenuItem.Icon>
      </MenuItem>
      <Separator/>
      <MenuItem Header="_Zakończ" Name="Zakoncz"
        Click="MenuItem_Click"/>
    </MenuItem>

    <MenuItem Header="_Help" >
      <MenuItem Header="Informacje _o Aplikacji"
        Click="Informacje_O_Aplikacji"/>
    </MenuItem>
  </Menu>
</DockPanel>
```

Menu kontekstowe

W podobny sposób można definiować też menu kontekstowe, czyli menu pojawiające się w momencie wcisnięcia prawnego klawisza myszki. Można je definiować dla całego okna, dla poszczególnych kontrolek lub dla pewnej grupy kontrolek. W poniższym przykładzie zdefiniowano globalnie w zasobach okna kontrolkę menu kontekstowego, która jest podpinana do kontrolki pola tekstowego umieszczonego wewnętrz tego okna (oczywiście nic nie stoi na przeszkodzie, aby kontrolkę tę podpiąć również pod inne kontrolki z tego okna).

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="Control">
    ...
  </Style>
  <ContextMenu x:Key="textBoxMenu"
    Style="{StaticResource myStyle}">
    <MenuItem Header="Czyszczenie Pola" Name="clearField"
      Click="clearField_Click"/>
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

<Separator/>
<MenuItem Header="Coś jeszcze ..." Name="cosJeszcze"/>
</ContextMenu>
</Window.Resources>

...

<TextBox Style="{StaticResource myStyle}" Panel.ZIndex="1"
Height="23" Margin="152,33,106,0" Name="textBox1"
VerticalAlignment="Top"
ContextMenu="{StaticResource textBoxMenu}"/>

...
  
```

Podpinanie reakcji na zdarzenia

Odpowiednie własności kontrolek służą do wskazania metod, które mają być wywoływane w przypadku wystąpienia określonego zdarzenia. W szczególności własność *Click* kontrolki *Button* służy do wskazania metody, która ma być wywołana w momencie wcisnięcia przycisku, np.:

```

<Button ... Name="button1" Click="uruchomCos">
  Włącz
</Button>
  
```

natomiast własność *Closing* kontrolki *Window* służy do wskazania metody, która ma być wywołana w przypadku podjęcia próby zamknięcia okna, np.:

```

<Window x:Class="WpfApplication1.Window1"
  ...
  Title="Moje Okno 1" Height="300" Width="430"
  Closing="zamknienieOkna">
  ...
</Window>
  
```

Sposób implementacji metod wywoływanych w momencie nadejścia tych zdarzeń będzie omawiany w kolejnym zagadnieniu.

Programowanie interfejsu z poziomu języka C#

Język XAML jest bardzo wygodny do tworzenia samego interfejsu graficznego aplikacji (a dokładniej jego wyglądu). Jednak zwykle potrzeba jeszcze zaimplementować „logikę” stojącą za tym interfejsem (np. metody, które będą uruchamiane w momencie wciskania odpowiednich przycisków, czy wybrania opcji). Ta jest już definiowana w plikach napisanych w języku C#. Oczywiście można w kodzie języka C# stworzyć też wszystkie pozostałe elementy interfejsu (o czym będzie się można przekonać poniżej), ale jest to rozwiązanie mało wygodne.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Elementy implementacji klasy okna

Jeśli stworzyliśmy w Visual Studio w XAML-u okienko aplikacji w WPF-ie, to możemy przejść do implementacji jego „logiki”. W tym celu należy w oknie „*Solution Explorer*” rozwinąć listę znajdująca się pod nazwą pliku utworzonego okna (założymy, że plik ten ma domyślną nazwę „*Window1.xaml*”). Powinien się nam pokazać plik o tej samej nazwie z dodanym rozszerzeniem „.cs” (czyli w tym przypadku „*Window1.xaml.cs*”). Po przejściu do edycji tego pliku zobaczymy definicję **częściowej** klasy naszego okna, np.:

```
namespace EmptyWpfApplication
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

Słowo kluczowe „***partial***” w powyższym przykładzie oznacza, że mamy do czynienia z częściową definicją, czyli część elementów klasy zdefiniowana jest w innym miejscu (np. w innym pliku). W tym konkretnym przypadku reszta klasy jest generowana automatycznie na podstawie liku XAML-owego.

Sama klasa okna dziedziczy (jak widać w tym przykładzie) po klasie *Window*.

W konstruktorze klasy jest wywoływana metoda *InitializeComponent()* służąca do inicjalizacji okna.

Dodatkowo w definicji klasy mogą znaleźć się np. metody do obsługi zdarzeń, które zostaną automatycznie wygenerowane w trakcie edycji umieszczanych w oknie kontrolek (np. klikając w Designerze na dodaną kontrolkę przycisku możemy wygenerować metodę obsługi zdarzenia wciśnięcia przycisku i przejść do jej edycji).

Warto wiedzieć, że w celu zamknięcia okna z poziomu kodu należy wywołać metodę *Close()* z obiektu okna.

Dostęp do kontrolek

Jak zostało wcześniej wspomniane, do kontrolek zdefiniowanych w XAML-u można się odwoływać przez obiekty o nazwach takich samych, jak nazwy podane pod XAML-em we właściwościach „***Name***” tych kontrolek. Przykładowo, w następujący sposób można zmieniać właściwości kontrolki etykiety o nazwie ***label1***:

```
label1.Content = "Uwaga";
label1.FontSize = 20;
label1.FontWeight = FontWeights.Bold;
label1.Foreground = Brushes.Red;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W analogiczny sposób można pobrać wartości własności kontrollek. Należy pamiętać, że często wartości wpisane w kontrolkach wymagają odpowiedniej konwersji przed ich użyciem. W poniższym przykładzie pobierana jest wartość z pola tekstowego o nazwie **textBox1** i przekonwertowywana na wartość całkowitoliczbową.

```
int wartosc = Convert.ToInt32(textBox1.Text);
```

Metody obsługi zdarzeń

W części poświęconej językowi XAML zostało wspomniane, że kontrolkom można przypisywać metody wykonywane w momencie wystąpienia określonych zdarzeń. Tego typu metody muszą przyjmować dwa argumenty. W pierwszym z nich przekazywana jest referencja do wywołującego zdarzenie obiektu (np. do przycisku, który został wciśnięty), a w drugim argumencie przekazywany jest obiekt zdarzenia, które wystąpiło (np. wciśnięcia przycisku). Przykładowo poniżej metoda może być podpięta pod zdarzenie wciśnięcia przycisku lub wybrania opcji z menu. W wyniku wywołania tej metody podejmowana jest próba zamknięcia okna aplikacji (w którym wywołana została ta metoda).

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

W powyższym przykładzie argumenty metody nie były wykorzystane. Warto zaznaczyć, że czasami zachodzi użycia tych argumentów. Dla przykładu metoda może być wywoływana w wyniku zdarzenia polegającego na zmianie zawartości kontrolki i musi sprawdzić, z jakiej kontrolki została wywołana i jaka jest aktualna wartość tej kontrolki. Często trzeba wówczas rzutować argumenty na odpowiednie typy lub odpowiednio zmienić typ argumentu metody. Poniższy przykład pokazuje jak powinna wyglądać konstrukcja metody, która może dopuszczać anulowanie zdarzenia, które ją wywołało:

```
private void zamkniecieOkna(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    ...
}
```

Używanie kontrolki **MessageBox**

Do wyświetlania okienek z informacjami oraz zapytaniami można używać klasy **MessageBox**. Do wyświetlania okienka służy statyczna metoda **Show** tej klasy, która posiada kilkanaście przeciążonych wariantów. W poniższym przykładzie kontrolka ta została użyta do potwierdzenia zamiaru zamknięcia okna:

```
private void zamkniecieOkna(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    MessageBoxResult key = MessageBox.Show(
        "Czy na pewno chcesz zamknąć okno ?",
        "Potwierdzenie",
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    MessageBoxButton.YesNo,
    MessageBoxButtonImage.Question,
    MessageBoxButtonResult.No);
e.Cancel = (key == MessageBoxButtonResult.No);
}
  
```

Otwieranie nowego okna zdefiniowanego w XAML-u

Do projektu aplikacji WPF można dodać nowe okno wybierając kolejno z menu:

Project ⇒ Add window ⇒ Window (WPF)

lub z okna „Solution explorer”:

Add ⇒ Window ⇒ Window (WPF)

Wówczas można tworzyć nowe obiekty z klasy tego okna oraz można je pokazywać przez wywołanie metody **ShowDialog()**. Przykładowo nowe okienko można utworzyć w następujący sposób z kodu w języku C# (zakładamy, że klasa okna została nazwana *MojeOkno*):

```

MojeOkno noweOkno = new MojeOkno();
noweOkno.ShowDialog();
  
```

Tworzenie okna bez użycia XAML-a

W podobny sposób można utworzyć nowe okno, wypełnić je kontrolkami oraz wyświetlić jego zawartość. Obiekt okna tworzymy wówczas z klasy **Window**. Należy pamiętać, że kontrolki interfejsu powinny zwykle być umieszczane w kontrolce umożliwiającej ich dokowanie, np. *Grid* lub *Canvas*. W celu wyświetlenia okna należy wywołać metodę **Show()**. Poniżej podany został przykład utworzenia okna zgodnie z powyższymi wskazówkami.

```

Window myWindow = new Window();
myWindow.Title = "Nowe Okno";
myWindow.Width = 250;
myWindow.Height = 100;

Grid myGrid = new Grid();
myGrid.Width = 250;
myGrid.Height = 100;

TextBlock windowText = new TextBlock();
windowText.Text = "Przykładowe okienko";
windowText.FontSize = 20;
windowText.FontWeight = FontWeights.Bold;
myGrid.Children.Add(windowText);

myWindow.Content = myGrid;
myWindow.Show();
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Używanie kontrolki *Grid*

Użyta w powyższym przykładzie kontrolka *Grid*, jak sama jej nazwa wskazuje, umożliwia układanie zawartych w niej kontrolek w rzędach i kolumnach. Ponadto można „sklejać” sąsiadujące komórki, co widać na poniższym przykładzie.

```
ColumnDefinition colDef1 = new ColumnDefinition();
ColumnDefinition colDef2 = new ColumnDefinition();
myGrid.ColumnDefinitions.Add(colDef1);
myGrid.ColumnDefinitions.Add(colDef2);

RowDefinition rowDef1 = new RowDefinition();
RowDefinition rowDef2 = new RowDefinition();
myGrid.RowDefinitions.Add(rowDef1);
myGrid.RowDefinitions.Add(rowDef2);

TextBlock txt1 = new TextBlock();
txt1.Text = "nagłówek - połączone pola";
Grid.SetColumnSpan(txt1, 3);
Grid.SetRow(txt1, 0);
myGrid.Children.Add(txt1);

TextBlock txt2 = new TextBlock();
txt2.Text = "jedno pole";
Grid.SetColumn(txt2, 0);
Grid.SetRow(txt2, 1);
myGrid.Children.Add(txt2);

RadioButton rB1 = new RadioButton();
rB1.Content = "A";
Grid.SetRow(rB1, 1);
Grid.SetColumn(rB1, 1);
myGrid.Children.Add(rB1);
```

Przykłady inicjalizacji kontrolek

W podobny sposób mogą być tworzone lub inicjalizowane inne kontrolki. Poniżej podane zostały przykłady inicjalizacji wybranych typów kontrolek.

```
private string[] wybory = { "tekst 1", "tekst 2", "tekst 3", "tekst 4" };
private string[] elListy = { "element 1", "element 2", "element 3",
                           "element 4", "element 5", "element 6",
                           "element 7", "element 8", "element 9",
                           "element 10", "element 11" };

...

comboBox2.Items.Clear();
foreach (string nazwaWyboru in wybory)
{
    comboBox2.Items.Add(nazwaWyboru);
}
comboBox2.SelectedIndex = 0;
listBox1.Items.Clear();
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

CheckBox element;
foreach (string nazwaElementu in elListy)
{
    element = new CheckBox();
    element.Margin = new Thickness(0, 0, 0, 10);
    element.Content = nazwaElementu;
    listBox1.Items.Add(element);
}
System.Windows.Forms.DateTimePicker mojaData =
windowsFormsHost1.Child as System.Windows.Forms.DateTimePicker;
mojaData.Value = DateTime.Today;
  
```

Tworzenie menu kontekstowego

W analogiczny sposób można tworzyć w kodzie źródłowym menu i menu kontekstowe. W poniższym przykładzie przedstawiono możliwy sposób implementacji menu kontekstowego z poziomu wyłącznie kodu w języku C#.

```

private ContextMenu windowContextMenu = null;

...
MenuItem saveMenuItem = new MenuItem();
saveMenuItem.Header = "Zapisz Dane";
saveMenuItem.Click += new RoutedEventHandler(zapiszDane);

MenuItem clearMenuItem = new MenuItem();
clearMenuItem.Header = "Czyść formularz";
clearMenuItem.Click += new RoutedEventHandler(czyscFormularz);

windowContextMenu = new ContextMenu();
windowContextMenu.Items.Add(saveMenuItem);
windowContextMenu.Items.Add(clearMenuItem);

this.ContextMenu = windowContextMenu;
  
```

Używanie kontrollek dziedziczących po klasie *FileDialog*

W języku C# mamy do dyspozycji klasę abstrakcyjną *FileDialog*, z której dziedziczą kontrolki okien dialogowych otwarcia oraz zapisania pliku – odpowiednio *OpenFileDialog* i *SaveFileDialog*. Kontrolki te wspierają między innymi wyszukiwanie plików według zadanej maski, weryfikację poprawności nazwy pliku, otwarcie strumienia do zapisu lub odczytu danych (w przypadku pierwszego z wymienionych okien dialogowych jest możliwe też otwarcie strumieni ze wszystkimi wybranymi plikami). Przykładowe użycie klasy *SaveFileDialog* zostało przedstawione w poniższym kodzie:

```

SaveFileDialog saveDialog = new SaveFileDialog();
saveDialog.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
saveDialog.DefaultExt = "txt";
saveDialog.AddExtension = true;
saveDialog.FileName = "nazwa_pliku";
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

saveDialog.InitialDirectory =
    @"C:\Documents and Settings\guest\Moje dokumenty\";
saveDialog.OverwritePrompt = true;
saveDialog.ValidateNames = true;
saveDialog.Title = "Nazwa okienka zapisu";

if (saveDialog.ShowDialog().Value)
{
    //using (Stream myStream = saveDialog.OpenFile())
    //{
    //    ...
    //}

    using (StreamWriter writer = new StreamWriter(saveDialog.FileName))
    {
        writer.WriteLine("Zapis do pliku {0}", saveDialog.FileName);
        System.Windows.Forms.DateTimePicker zapisywanaData =
            windowsFormsHost1.Child as System.Windows.Forms.DateTimePicker;
        writer.WriteLine(zapisywanaData.Value.ToString());

        ...
        MessageBox.Show("Zapisano dane do pliku", "Zapis");
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

6. Wątki

Często w programie zachodzi potrzeba podzielenia wykonywanego kodu pomiędzy osobne potoki sterowania, które będą przetwarzane równolegle lub symultanicznie. W .NET Frameworku (i w szczególności języku C#) jest to realizowane przez wątki, podobnie jak w Javie i w większości innych podobnych języków programowania.

Ogólne informacje o wątkach

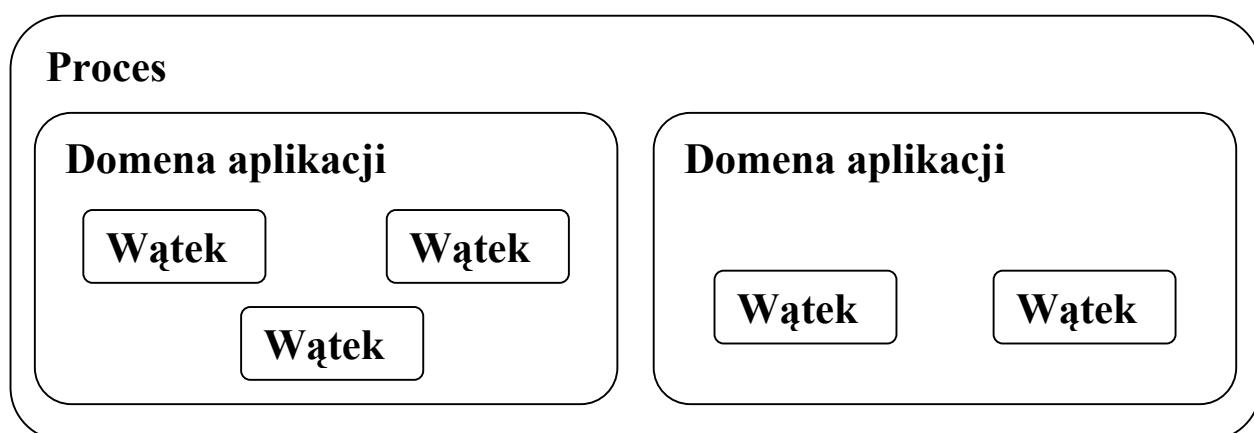
Sama idea wątków jest podobna jak w Javie. Występują jednak pewne różnice w sposobie działania wątkach używanych w języku C# i innych językach z .NET Frameworku (i ogólniej systemu Windowsowego), a wątkami z Javy (i ogólniej – wątkami działającymi w systemach UNIX-owych i UNIX-o podobnych).

Wątki a procesy

W Javie (i w systemach w systemach UNIX-owych i UNIX-o podobnych) wątki są uruchamiane w obrębie konkretnego procesu. Sam proces jest realizacją w pamięci operacyjnej pewnego programu uruchomieniowego. Wątki współdzielą zasoby (np. obszar sterty) procesu w obrębie którego działają.

W .NET Frameworku (i zarządzalnym środowisku windowsowym) do tego modelu dołącza jeszcze jeden pośredniczący element. Wątki również są uruchamiane w obrębie procesu, ale są umieszczane w konkretnej domenie aplikacji, nazywanej też czasem dziedziny aplikacji. To właśnie domena aplikacji jest realizacją w pamięci operacyjnej pewnego programu uruchomieniowego. W procesie może występować więcej niż jedna domena aplikacji, a nawet można je zagnieżdżać. Więcej informacji na temat domen aplikacji pojawi się w jednym z następnych tematów.

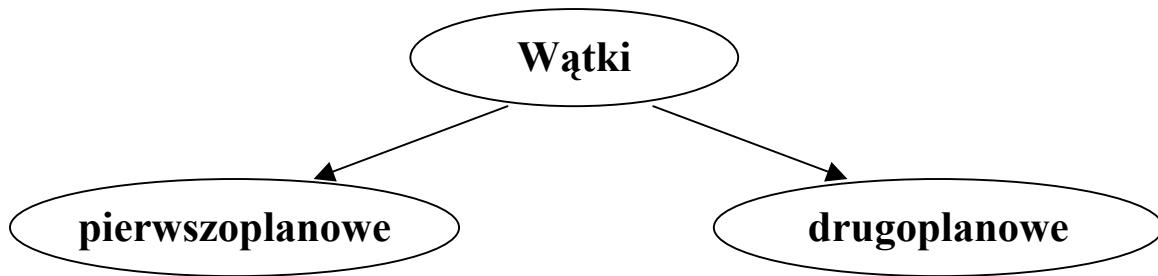
Wzajemne zależności pomiędzy procesem, dziedzinami aplikacji, a wątkami (w jednym z najprostszych przypadków) został zobrazowany na poniższym schemacie



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Wątki pierwszoplanowe i wątki drugoplanowe

Wątki działające w środowisku uruchomieniowym .NET Frameworka podzielić możemy na dwie grupy: *wątki pierwszoplanowe* i *wątki drugoplanowe*.



Pomimo tego, że nazwy te mogą rodzić skojarzenia z podziałem procesów w środowiskach UNIX-owych i UNIX-o podobnych kryteria tego podziału są zupełnie inne. Najistotniejszą różnicą pomiędzy tymi rodzajami wątków jest to, że proces działa dopóki działają w nim wątki pierwszoplanowe. Kiedy wszystkie wątki pierwszoplanowe w procesie zarządzalnym zostaną zatrzymane, to proces ten zatrzymuje wszystkie wątki drugoplanowe i kończy swoje działanie.

Używanie klasy *Thread*

Najistotniejszą klasą służącą do tworzenia nowych wątków i zarządzania nimi jest klasa ***Thread***. Jej obiekty powiązane są z konkretnymi wątkami bieżącego procesu.

Tworzenie nowych wątków

Nowo tworzone wątki uruchamiane są ze wskazaną metodą. W przypadku tworzenia wątku przez klasę *Thread* podania tej metody służą delegacje

- dla metod bezargumentowych delegacja

```
public delegate void ThreadStart();
```

- dla metod posiadających argument delegacja

```
public delegate void ParameterizedThreadStart(Object obj)
```

Statyczne właściwości

Klasa *Thread* posiada następujące statyczne właściwości o podanych poniżej funkcjonalnościach:

- CurrentContext*** - pobiera (w postaci obiektu klasy *Context*) informacje dotyczące środowiska i polityki bieżącego wątku;
- CurrentPrincipal*** - pobiera lub ustawia wartość identyfikującą rolę w aktualnych ustawieniach polityki;
- CurrentThread*** - zwraca obiekt reprezentujący bieżący wątek.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Nie-statyczne własności

Klasa *Thread* posiada następujące nie-statyczne własności o podanych poniżej funkcjonalnościach:

- **CurrentCulture** - pobiera lub zmienia ustawienia regionalne;
- **CurrentUICulture** - pobiera lub zmienia ustawienia regionalne używane przez menadżera zasobów;
- **ExecutionContext** - pobiera (w obiekcie klasy *ExecutionContext*) różne informacje dotyczące środowiska wątku;
- **IsAlive** - pobiera informację o stanie wątku;
- **IsBackground** - pobiera lub ustawia wartość wskazującą czy wątek jest uruchomiony w tle;
- **IsThreadPoolThread** - pobiera wartość wskazującą czy wątek należy do zarządzalnej puli wątków (*ThreadPool*);
- **Name** - pobiera lub ustawia nazwę wątku;
- **Priority** - pobiera lub ustawia wartość opisującą priorytet wątku;
- **ThreadState** - zwraca wartość opisującą stan wątku.

Wybrane nie-statyczne metody

Wśród niestatycznych metod klasy *Thread* najistotniejszą rolę odgrywają następujące metody:

- **Abort** - wywołuje wyjątek *ThreadAbortException* w wątku dla którego metoda została wywołana;
- **Interrupt** - przerywa stan oczekiwania dla wątku znajdującego się w stanie *WaitSleepJoin*;
- **Join** - blokuje wywołujący wątek (zwykle macierzysty) do czasu zakończenia wskazanego wątku (zwykle potomnego);
- **Start** - powoduje przesłanie wątku do kolejki zadań oczekujących na wykonanie (uruchamia wątek).

Ponadto następujące niestatyczne metody tej klasy kontrolują umieszczenie wątku w jednowątkowym lub wielowątkowym kontenerze:

- **GetApartmentState** - pobiera wartość opisującą rodzaj kontenera wątku
- **SetApartmentState** - ustawia rodzaj kontenera wątku przed jego uruchomieniem; może być wywołana tylko raz;
- **TrySetApartmentState** - ustawia rodzaj kontenera wątku (przed jego uruchomieniem).

Rodzaj kontenera jest wyznaczany przez wartości typu wyliczeniowego **ApartmentState**, który może przyjmować wartości:

- **STA** (jednowątkowy kontener, ang. *Single-Thread Apartment*),
- **MTA** (wielowątkowy kontener, ang. *Multi-Thread Apartment*),
- **Unknown** (nieokreślony, przedinicjalizacją).

Wybrane statyczne metody

Spośród statycznych metod klasy *Thread* wyróżnić można następujące metody:

- **GetDomain** - zwraca bieżącą domenę aplikacji (obiekt klasy **AppDomain**) dla wywołującego wątku;
- **GetDomainID** - zwraca identyfikator (liczbę całkowitą) bieżcej domeny aplikacji dla wywołującego wątku;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **MemoryBarrier** - synchronizuje dostęp do pamięci w ten sposób, że dostęp do pamięci przed wywołaniem tej metody nie może zostać zrealizowany po dostępie umieszczonym po jej wywołaniu;
- **ResetAbort** - przerywa żądanie przerwania wywołane metodą Abort (bez wywołania ResetAbort wyjątek ThreadAbortException jest ponownie rzucany w miejscu, w którym został pierwotnie rzucony);
- **Sleep** - blokuje działanie bieżącego wątku na zadaną ilość milisekund;
- **SpinWait** - blokuje działanie bieżącego wątku przez zadaną ilość iteracji;
- **VolatileRead** - odczytuje ostatnio zapisaną pod zmienną wartość (niezależnie od liczby procesorów i stanu cache);
- **VolatileWrite** - natychmiast zapisuje wartość do pamięci (tak, aby od razu była widziana przez wszystkie procesory).

Ponadto następujące statyczne metody tej klasy służą do wskazania oznaczenia szczególnych regionów w kodzie aktualnie wykonywanego wątku:

- **BeginCriticalSection** - sygnalizuje rozpoczęcie fragmentu kodu mającego wpływ na stabilność całej domeny aplikacji;
- **BeginThreadAffinity** - sygnalizuje rozpoczęcie fragmentu kodu, który może być zależny od systemu operacyjnego;
- **EndCriticalSection** - sygnalizuje zakończenie fragmentu kodu mającego wpływ na stabilność całej domeny aplikacji;
- **EndThreadAffinity** - sygnalizuje zakończenie fragmentu kodu, który może być zależny od systemu operacyjnego;

Możliwe stany wątku

Stany wątku są reprezentowane przez wartości typu wyliczeniowego

```
public enum ThreadState
```

Instancje tego typu mogą przyjmować następujące wartości

- **Running** - wątek został uruchomiony, nie jest blokowany i nie obsługuje wyjątku ThreadAbortException;
- **StopRequested** - nakazano zatrzymanie wątku (ten stan jest tylko „do użytku wewnętrznego”)
- **SuspendRequested** - nakazano zawieszenie wątku;
- **Background** - wątek jest uruchamiany w tle (w drugim planie) (stan ten jest zależny od właściwości Thread.IsBackground);
- **Unstarted** - nie uruchomiono jeszcze wątku (nie wywołano metody Thread.Start);
- **Stopped** - wątek został zatrzymany (zakończył się);
- **WaitSleepJoin** - wątek jest blokowany;
- **Suspended** - wątek jest zawieszony;
- **AbortRequested** - wywołano metodę Thread.Abort, ale wątek nie obsłużył jeszcze wyjątku ThreadAbortException;
- **Aborted** - wątek przeszedł przez stan AbortRequested i się zakończył, ale jego stan nie zmienił się jeszcze na Stopped;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

przy czym wytłuszczone zostały te stany, które można zaobserwować podczas normalnej pracy uruchamianych w standardowy sposób wątków, natomiast oznaczone jedynie kursywą zostały te stany, których zaobserwowanie jest bardzo trudne w typowych warunkach lub które są przyjmowane przez wątki uruchamiane w specyficzny sposób (lub przełączone w inny tryb).

Możliwe priorytety wątku

Priorytety wątku są reprezentowane przez wartości typu wyliczeniowego

```
public enum ThreadPriority
```

Instancje tego typu mogą przyjmować następujące wartości

- **Lowest** - wątek umieszczany jest w kolejce wykonywania po wątkach z innymi priorytetami;
- **BelowNormal** - wątek umieszczany jest w kolejce wykonywania po wątkach o priorytecie *Normal*, ale przed wątkami o priorytecie *Lowest*;
- **Normal** - wątek umieszczany jest w kolejce wykonywania po wątkach o priorytecie *AboveNormal*, ale przed wątkami o priorytecie *BelowNormal*; jest to domyślny priorytet;
- **AboveNormal** - wątek umieszczany jest w kolejce wykonywania po wątkach o priorytecie *Highest*, ale przed wątkami o priorytecie *Normal*;
- **Highest** - wątek umieszczany jest w kolejce wykonywania przed wątkami o innym priorytecie.

Należy pamiętać, że „rzeczywisty priorytet” działania wątku wyliczany jest w oparciu zarówno o priorytet wątku, jak i priorytet procesu (o którym będzie mowa w jednym z następnych tematów).

Przykład na tworzenie wątku

```
using System;
using System.Threading;

namespace WatkiPrzyklad1
{
    class Program
    {
        static void MetodaWatklu()
        {
            for (int i = 1; i < 8; i++)
            {
                Console.WriteLine("Wątek Potomny ({0})", i);
                Thread.Sleep(100);
            }
        }
        static void Main(string[] args)
        {
            Thread t = new Thread(new ThreadStart(MetodaWatklu));
            t.Start();

            for (int i = 1; i < 5; i++)
            {
                Console.WriteLine("Wątek Przodka ({0})", i);
                Thread.Sleep(100);
            }
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        Console.WriteLine("Wątek Przodka: oczekiwanie na zakończenie"
                           + " wątku potomka.");
        t.Join();
        Console.WriteLine("Wątek Przodka: wątek potomka się zakończył.");
    }
}
```

Przykład na przekazywanie danych do metody wątku przez argument

```
using System;
using System.Threading;

class Program
{
    static void MetodaWatkus()
    {
        Console.WriteLine("Wątek uruchomiony bez argumentu");
    }
    static void MetodaWatkus1(object o)
    {
        //string tekst = o as string;
        Console.WriteLine("Wątek uruchomiony z argumentem \'{0}\'", o);
    }
    void MetodaWatkus2(object o)
    {
        Console.WriteLine("Wątek uruchomiony z argumentem \'{0}\'", o);
    }
    static void Main(string[] args)
    {
        Thread t = new Thread(new ThreadStart(MetodaWatkus));
        //Thread t = new Thread(MetodaWatkus);
        t.Start();
        Thread t1 =
            new Thread(new ParameterizedThreadStart(Program.MetodaWatkus1));
        //Thread t1 = new Thread(new ParameterizedThreadStart(MetodaWatkus1));
        //Thread t1 = new Thread(Program.MetodaWatkus1);
        //Thread t1 = new Thread(MetodaWatkus1);
        t1.Start("jakiś argument ...");
        Program p = new Program();
        Thread t2 = new Thread(new ParameterizedThreadStart(p.MetodaWatkus2));
        //Thread t2 = new Thread(p.MetodaWatkus2);
        t2.Start(12.34);
        //t.Join(); t1.Join(); t2.Join();
    }
}
```

Przykład na przekazywanie danych do metody wątku przez obiekt

```
using System;
using System.Text;
using System.Threading;

namespace WatkiPrzyklad3
{
    class KlasaWatku
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
    private String argument = null;
    public KlasaWatku(String arg)
    {
        argument = arg;
    }
    public void Obliczenia()
    {
        Console.WriteLine("Wykonywane obliczenia dla: \'{0}\'", this.argument);
    }
}

class Program
{
    static void Main(string[] args)
    {
        KlasaWatku w = new KlasaWatku("dane");
        Thread t = new Thread(w.Obliczenia);
        t.Start();
    }
}
}
  
```

Przykład na przekazanie wyniku przez obiekt

```

using System;
using System.Threading;

namespace WatkiPrzyklad4
{
    public class Dane
    {
        private int a, b;
        private int wynik;
        public Dane (int a, int b)
        {
            this.a = a;
            this.b = b;
        }
        public int Wartosc1
        {
            get { return this.a; }
        }
        public int Wartosc2
        {
            get { return this.b; }
        }
        public int Wynik
        {
            set { wynik = value; }
            get { return this.wynik; }
        }
    }

    class Program
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykład na przekazanie wyniku przez delegację

```
using System;
using System.Threading;

public delegate void delegacja(int wartosc);

public class KlasaWatku
{
    private string opis;
    private int wartosc;

    private delegacja wywolanieZwrotne;

    public KlasaWatku(string opis, int wart, delegacja del)
    {
        this.opis = opis;
        this.wartosc = wart;
        wywolanieZwrotne = del;
    }

    public void WykonanieWatku()
    {
        Console.WriteLine("Wykonanie wątku {0}", opis);
        if (wywolanieZwrotne != null)
            wywolanieZwrotne(wartosc);
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
class Program
{
    public static void ZapisanieWyniku(int wartosc)
    {
        Console.WriteLine("Wynik wątku: {0}", wartosc);
    }
    static void Main(string[] args)
    {
        KlasaWatku w = new KlasaWatku(
            "Watek ze zwracaniem wartosci",
            //12345, ZapisanieWyniku); // tak też można
            12345, new delegacja (ZapisanieWyniku));
        //Thread t = new Thread(w.WykonanieWatku);
        Thread t = new Thread(new ThreadStart(w.WykonanieWatku));
        t.Start();
        t.Join();
    }
}
```

Przykład pokazujący przyjmowanie różnych stanów przez wątek

```
using System;
using System.Threading;

class Program
{
    static void MetodaWatku()
    {
        int j = 1;
        Thread.Sleep(50);
        for (int i = 1; i < 100000000; i++)
        {
            j *= i;
        }
    }

    static void Main(string[] args)
    {
        Thread t = new Thread(new ThreadStart(MetodaWatku));
        Console.Write("Stan przed uruchomieniem:\n  ");
        Console.WriteLine(t.ThreadState);
        t.Start();
        Console.Write("Stan w uśpieniu:\n  ");
        Console.WriteLine(t.ThreadState);
        Thread.Sleep(60);
        Console.Write("Stan odczas pracy:\n  ");
        Console.WriteLine(t.ThreadState);
        t.Join();
        Console.Write("Stan po zatrzymaniu:\n  ");
        Console.WriteLine(t.ThreadState);
    }
}
```

Tworzenie wątków przez klasę *ThreadPool*

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Nowe wątki można też tworzyć przez klasę **ThreadPool**.

Podstawowe informacje o tworzeniu wątków przez klasę **ThreadPool**

Klasa **ThreadPool** służy do tworzenia wyłącznie wątków drugoplanowych. Wątki tworzone tą metodą umieszczane są w tzw. „pułi wątków”. Wspomniana pula jest wspólna dla procesu.

Wątki z puli uruchamiane są kolejno, w sytuacji, gdy dostępne są zasoby. Uruchamianie nowych wątków zwykle odbywa się po pewnym czasie oczekiwania. Można jednak wpływać na ilość wątków z puli, które są przygotowane do uruchomienia.

Do podania metody z którą ma być uruchamiany utworzony tak wątek służy delegacja

```
public delegate void WaitCallback(Object obj)
```

Wybrane statyczne metody

Klasa **ThreadPool** posiada następujące statyczne metody o podanych poniżej funkcjonalnościami:

- **GetAvailableThreads** - pobiera różnicę pomiędzy maksymalną ilością wątków a ilością wykonywanych wątków;
- **GetMaxThreads** - pobiera maksymalną ilość wątków z puli, które można wykonywać współbieżnie;
- **GetMinThreads** - pobiera minimalną ilość gotowych do uruchomienia wątków;
- **QueueUserWorkItem** - tworzy i dodaje do puli nowy wątek;
- **RegisterWaitForSingleObject** - rejestruje delegację, która wywoła wątek, gdy nastąpi stosowne zdarzenie (lub przeterminowanie) (patrz później: użycie klasy *RegisteredWaitHandle* pod koniec tego rozdziału);
- **SetMaxThreads** - ustawia maksymalną ilość wątków z puli, które można wykonywać współbieżnie;
- **SetMinThreads** - ustawia minimalną ilość gotowych do uruchomienia wątków.

W powyższych metodach **wraz z pobieraniem i ustawianiem liczby wątków pobiera i ustawia się liczbę asynchronicznych operacji wejścia/wyjścia**.

Przykład tworzenia wątków przez klasę **ThreadPool**

```
using System;
using System.Threading;

namespace WatkiPrzyklad6
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.QueueUserWorkItem(new WaitCallback(MetodaWatku),
                "argument");
            ThreadPool.QueueUserWorkItem(new WaitCallback(MetodaWatku));
            // Thread.Sleep(500);
            Thread.Sleep(100);
            Console.WriteLine("Zakończenie wątku głównego");
        }

        static void MetodaWatku(object argument)
        {
            Console.WriteLine("Wątek o numerze {0} został uruchomiony.", argument);
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        }
    static void MetodaWatku(Object o)
    {
        Console.WriteLine("Wywołanie wątku (z argumentem \"{0}\")",
            o ?? "<null>");
        Thread.Sleep(200);
        Console.WriteLine("Wywołanie wątku (2)");
    }
}
```

Przykład na odczyt i ustawianie limitu zasobów przez klasę *ThreadPool*

```
using System;
using System.Threading;

namespace WatkiPrzyklad8
{
    class Program
    {
        static void MetodaWatku(Object o)
        {
            int watki, komunikacja;
            ThreadPool.GetAvailableThreads(out watki, out komunikacja);
            Console.WriteLine("Wywołanie wątku z argumentem \"{0}\"" +
                " Available: {1}, {2}", o, watki, komunikacja);
            Thread.Sleep((int)o);
            Console.WriteLine("Zakończenie wątku z argumentem \"{0}\", {0}", o);
        }

        static void Main(string[] args)
        {
            int watki, komunikacja;

            ThreadPool.GetMaxThreads(out watki, out komunikacja);
            Console.WriteLine("Max: {0}, {1}", watki, komunikacja);

            ThreadPool.GetMinThreads(out watki, out komunikacja);
            Console.WriteLine("Min: {0}, {1}", watki, komunikacja);

            ThreadPool.GetAvailableThreads(out watki, out komunikacja);
            Console.WriteLine("Available: {0}, {1}", watki, komunikacja);

            for (int i = 601; i < 605; i++)
            {
                ThreadPool.QueueUserWorkItem(new WaitCallback(MetodaWatku), i);
            }

            ThreadPool.GetAvailableThreads(out watki, out komunikacja);
            Console.WriteLine("Available: {0}, {1}", watki, komunikacja);

            Thread.Sleep(600);

            ThreadPool.GetAvailableThreads(out watki, out komunikacja);
            Console.WriteLine("Available: {0}, {1}", watki, komunikacja);
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        ThreadPool.GetMinThreads(out watki, out komunikacja);
        if (ThreadPool.SetMinThreads(3, komunikacja))
        {
            Console.WriteLine("Ustawiono minimalną liczbę wątków");
        }
        else
        {
            Console.WriteLine("Nie ustawiono minimalnej liczby wątków");
        }
        ThreadPool.GetMinThreads(out watki, out komunikacja);
        Console.WriteLine("Min: {0}, {1}", watki, komunikacja);

        Thread.Sleep(2100);
        Console.WriteLine("Zakończenie wątku głównego");
    }
}
```

Kiedy nie należy używać klasy *ThreadPool*

Należy pamiętać o tym, że możliwości użycia klasy *ThreadPool* są ograniczone specyfczną konstrukcją puli wątków. W szczególności nie należy tej klasy używać w następujących przypadkach:

- jeśli potrzebujemy stworzyć wątek pierwszoplanowy;
 - jeśli potrzebujemy stworzyć wątek o określonym priorytecie;
 - jeśli mamy zadania mogące blokować wątek przez długi czas (w takim przypadku może dojść do osiągnięcia maksymalnej liczby uruchomionych wątków, co spowoduje że nowe wątki nie będą uruchamiane);
 - jeśli musimy umieścić wątek w jedno-wątkowym kontenerze;
 - jeśli chcemy identyfikować utworzony wątek.

Mechanizmy tworzenia wątków wprowadzone od wersji 4.0 .NET Framework

W wersji 4.0 .NET Frameworka wprowadzono nowe klasy *Task* i *Parallel*, przeznaczone głównie do tworzenia i uruchamiania równoległych zadań. Bieżący podrozdział został poświęcony przedstawieniu wybranych ich możliwości.

Klasa Task

Klasa **Task** została wyposażona w szereg mechanizmów do tworzenia wątków i uruchamiania w nich zadań. Używając tej klasy można w szczególności:

- tworzyć i uruchamiać wątek asynchronicznie używając metody `Task.Factory.StartNew`;
 - uruchamiać utworzony wcześniej wątek asynchronicznie wywołując metodę `Start`;
 - uruchamiać synchronicznie (czekając na zakończenie) przez wywołanie metody `RunSynchronously`.

Poniżej zamieszczony został przykład użycia klasy *Task*.

```
using System;
using System.Threading;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Action<object> action = (object obj) =>
        {
            Console.WriteLine("Zadanie={0}, Argument={1}, Wątek={2}",
                Task.CurrentId, obj.ToString(),
                Thread.CurrentThread.ManagedThreadId);
        };

        Task t = Task.Factory.StartNew(Metoda);
        //Task t = Task.Factory.StartNew(() => { Console.WriteLine("Zadanie..."); });
        t.Wait();

        Task t1 = new Task(action, "pierwszy");
        Task t2 = Task.Factory.StartNew(action, "drugi");
        //...
        t2.Wait();
        t1.Start();
        // ...
        t1.Wait();
        Task t3 = new Task(action, "trzeci");
        t3.RunSynchronously();
        // Jeśli chcemy pobrać informacje o zakończeniu:
        t3.Wait();
    }

    static void Metoda()
    {
        Console.WriteLine("Id wątku = {0}",
            Thread.CurrentThread.ManagedThreadId);
    }
}
  
```

Warto zwrócić w tym miejscu uwagę na użycie wprowadzonej w wersji 4.0 .NET Frameworka klasy *Action*, której obiektom można przypisywać lambda-wyrażenia (metody anonimowe).

Klasa *Task* może też zostać użyta do równoległego uruchamiania zadań, w sposób pokazany na poniższym przykładzie

```

Task[] zadania = new Task[3]
{
    Task.Factory.StartNew(() => Metoda1()),
    Task.Factory.StartNew(() => Metoda2()),
    Task.Factory.StartNew(() => Metoda3())
};
Task.WaitAll(zadania);
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Parallel.Invoke

W tej samej wersji języka wprowadzono też inne sposoby do równoległego uruchamiania zadań. W tym celu specjalnie została utworzona metoda *Parallel.Invoke*. Z jej użyciem można tworzyć i uruchamiać równolegle nowe zadania w postaci wątków tworzonych z metodami podanymi w jej argumentach. Poniżej został pokazany przykład takiego użycia tej metody.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        try
        {
            Parallel.Invoke(
                Zadanie,
                delegate()
                {
                    Console.WriteLine("Metoda 2, Wątek {0}",
                        Thread.CurrentThread.ManagedThreadId);
                },
                () =>
                {
                    Console.WriteLine("Metoda 3, Wątek {0}",
                        Thread.CurrentThread.ManagedThreadId);
                }
            );
        }
        catch (AggregateException e)
        {
            Console.WriteLine("Wyjątek w potomku:\n{0}",
                e.InnerException.ToString());
        }
    }

    static void Zadanie()
    {
        Console.WriteLine("Metoda 1, Wątek {0}",
            Thread.CurrentThread.ManagedThreadId);
    }
}
```

Klasa *Parallel* dostarcza również metody **For** i **ForEach**, będące implementacjami zrównoleglonych pętli *for* i *foreach*. Przykładowo zamiast klasycznego wywołania pętli *foreach*:

```
foreach (var element in kolekcja)
{
    Metoda(element);
}
```

Można użyć równoległego wywołania metody *Parallel.ForEach*:

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
Parallel.ForEach(kolekcja, element => Metoda(element));
```

Synchronizacja wątków

Bardzo istotną rolę przy pisaniu wielowątkowych programów odgrywa kwestia synchronizacji wątków, np. dla zachowania wyłącznego dostępu do niepodzielnych zasobów. W języku C# jest wiele mechanizmów pozwalających na synchronizację wątków. Są to między innymi

- rygle,
- monitory (implementowane przez klasę *Monitor*),
- muteksy (implementowane przez klasę *Mutex*),
- semafory (implementowane przez klasę *Semaphore*),
- rygle odczytu-zapisu (implementowane przez klasę *ReaderWriterLock*),
- zdarzenia.

Ich opisowi poświęcony został niniejszy podrozdział.

Rygle

Jednym z najprostszych mechanizmów synchronizacji wątków są rygle.

Mechanizm rygli zapewnia wyłączny dostęp do bloku rygla dla jednego wątku. Blok rygla musi być opatrzony słowem kluczowym **lock** ze wskazanym obiektem (może to być obiekt dowolnej klasy), np.:

```
Object obiektRygla = new Object();

lock (obiektRygla)
{
    // sekcja krytyczna
}
```

Należy pamiętać, że w związku ze sposobem użycia **rygle służą wyłącznie do synchronizacji wątków w obrębie jednej dziedziny aplikacji**. Poniżej pokazany został przykład użycia rygli w programie.

```
using System;
using System.Threading;

class Program
{
    private static int przedzial;
    private static Object o = new object();

    static void MetodaWatku()
    {
        for (int i = 1; i < 3; i++)
        {
            lock (o)
            {
                Console.WriteLine("Wejście do sekcji krytycznej ({0}) w wątku {1}",
                    Thread.CurrentThread.Name, i);
            }
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    przedzial = przedzial + 100;
    Thread.Sleep(200 + przedzial);
    Console.WriteLine("Wyjście z sekcji krytycznej ({1}) w wątku {0}",
                      Thread.CurrentThread.Name, i);
}
}
}
static void Main(string[] args)
{
    for (int i = 1; i < 5; i++)
    {
        Console.WriteLine("Tworzenie wątku {0}", i);
        Thread t = new Thread(new ThreadStart(MetodaWatku));
        t.Name = "W" + i;
        t.Start();
    }
}
}
  
```

Monitory

Klasa **Monitor** jest klasą statyczną a zatem posiada jedynie statyczne metody. Używanie metod *Enter* i *Exit* tej klasy zapewnia funkcjonalność identyczną z ryglami (*lock*):

```

Object obiektMonitora = new Object();

Monitor.Enter(obiektMonitora)
    // sekcja krytyczna
Monitor.Exit(obiektMonitora)
  
```

Co więcej, w samym języku rygle są zaimplementowane z użyciem monitorów. Poniższy przykład przedstawia alternatywną wersję programu z ostatniego przykładu, tym razem z użyciem klasy *Monitor* zamiast rygla:

```

using System;
using System.Threading;

class Program
{
    private static int przedzial;
    private static Object o = new object();

    static void MetodaWatku()
    {
        for (int i = 1; i < 3; i++)
        {
            Monitor.Enter(o);
            // Patrz też Monitor.Wait(), Monitor.Pulse(), Monitor.PulseAll()
            Console.WriteLine("Wejście do sekcji krytycznej ({1}) w wątku {0}",
                              Thread.CurrentThread.Name, i);
            przedzial = przedzial + 100;
            Thread.Sleep(200 + przedzial);
            Console.WriteLine("Wyjście z sekcji krytycznej ({1}) w wątku {0}",
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    Thread.CurrentThread.Name, i);
Monitor.Exit(o);
}
}
static void Main(string[] args)
{
  for (int i = 1; i < 5; i++)
  {
    Console.WriteLine("Tworzenie wątku {0}", i);
    Thread t = new Thread(new ThreadStart(MetodaWatku));
    t.Name = "W" + i;
    t.Start();
  }
}
}
  
```

Klasa *Monitor* posiada również inne metody służące do synchronizacji wątków. W szczególności posiada następujące metody:

- **Enter** - jeśli obiekt jest nie jest zablokowany, to go blokuje (ustawia rygiel) i zwraca wartość **true** i wchodzi do sekcji krytycznej (jako jedyny wątek); jeśli obiekt jest zablokowany to wątek czeka, aż zostanie zwolniony i wówczas go blokuje;
- **Exit** - odblokowuje rygiel (zwalnia obiekt) i kończy blok krytyczny (warto podawać tę metodę w bloku *finally* konstrukcji *try--catch--finally*);
- **Pulse** - wysyła sygnał informujący o zmianie stanu ryglia do **przynajmniej jednego** wątku oczekującego na metodzie *Wait*;
- **PulseAll** - wysyła sygnał informujący o zmianie stanu ryglia do **wszystkich** wątków oczekujących na metodzie *Wait*;
- **TryEnter** - jeśli obiekt jest nie jest zablokowany, to go blokuje i zwraca wartość **true**; jeśli obiekt jest zablokowany i został podany czas oczekiwania (w postaci liczby milisekund lub w strukturze *TimeSpan*), to wątek czeka podany okres czasu i jeśli w jego trakcie obiekt zostanie zwolniony, to jest on blokowany i jest zwracana wartość **true**; w przeciwnym wypadku jest zwracana wartość **false**;
- **Wait (Object)** - odblokowuje rygiel i zatrzymuje wątek do czasu odebrania notyfikacji (od metody *Pulse* lub *PulseAll*)
- **Wait (Object, Int32)**, **Wait (Object, TimeSpan)** - odblokowuje rygiel i zatrzymuje wątek do czasu odebrania notyfikacji lub upłynięcia podanego czasu (wg. struktury lub liczby milisekund);
- **Wait (Object, Int32, Boolean)**, **Wait (Object, TimeSpan, Boolean)** - odblokowuje rygiel i zatrzymuje wątek do czasu odebrania notyfikacji lub upłynięcia podanego czasu; w czasie oczekiwania można zażądać opuszczenia kontekstu synchronizacji.

Osoby, które wcześniej pisały programy wielowątkowe dla systemów UNIX-owych lub UNIX-o podobnych (zwłaszcza linux-a) mogą zauważać podobieństwa w działaniu niektórych z wymienionych metod do POSIX-owych mutexów i zmiennych warunku.

Klasa *Monitor*, podobnie jak rygle służy wyłącznie do synchronizacji wątków w obrębie jednej dziedziny aplikacji.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Klasa **WaitHandle**

Klasa **WaitHandle** jest klasą abstrakcyjną. Jest ona klasą bazową dla klas takich jak *Mutex*, *Semaphore* i *EventWaitHandle*. Elementy dziedziczone z tej klasy są pomocne do przekazywania powiadomień do wątków podczas synchronizacji. Mogą one być też stosowane do blokowania wątków do czasu zwolnienia zasobów. W szczególności klasa *WaitHandle* posiada metody *SignalAndWait*, *WaitAll*, *WaitAny* i *WaitOne*. Poniżej przedstawione zostały wspomniane elementy.

Klasa *WaitHandle* posiada własność ***SafeWaitHandle***, poprzez którą można pobierać lub ustawiać (współdzielony w systemie operacyjnym) uchwyt do zaimplementowanego zasobu. Po przypisaniu nowej wartości, poprzedni uchwyt może być w bezpieczny sposób usunięty przez „garbage collector”. Klasa ta posiada też następujące niestatyczne metody:

- ***Close*** - zwalnia wszystkie zasoby używane przez bieżący obiekt klasy *WaitHandle* (zatem wszelkie późniejsze odwołania do tego uchwytu - również w innych wątkach mogą powodować powstanie błędów);
- ***WaitOne()*** - blokuje wywołujący wątek do czasu odebrania sygnału na bieżącym obiekcie klasy *WaitHandle*;
- ***WaitOne(Int32, Boolean)***, ***WaitOne(TimeSpan, Boolean)*** - blokuje wywołujący wątek do czasu odebrania sygnału na bieżącym obiekcie klasy *WaitHandle* lub upłynięcia podanego czasu; można zarządzać opuszczenia kontekstu synchronizacji na czas oczekiwania

Klasa *WaitHandle* posiada również następujące statyczne metody:

- ***SignalAndWait*** - blokuje bieżący wątek do czasu nadania sygnału na drugim z podanych uchwytów i wówczas wysyła sygnał na pierwszy z podanych uchwytów (operacja odebrania sygnału i oczekiwania jest atomiczna);
- ***SignalAndWait(WaitHandle, WaitHandle, Int32, Boolean)***,
SignalAndWait(WaitHandle[], WaitHandle, TimeSpan, Boolean) - atomicznie odbiera sygnał na drugim uchwycie i wysyła sygnał na pierwszy sygnał na trzecim argumencie podawany jest maksymalny czas oczekiwania na ostatnim argumencie; można zażądać opuszczenia kontekstu synchronizacji na czas oczekiwania;
- ***WaitAll(WaitHandle[])*** - blokuje wywołujący wątek do czasu odebrania sygnału na wszystkich uchwyttach z podanej w argumencie tablicy *WaitHandle*;
- ***WaitAll(WaitHandle[], Int32, Boolean)***, ***WaitAll(WaitHandle[], TimeSpan, Boolean)*** - blokuje wywołujący wątek do czasu odebrania sygnału na wszystkich uchwyttach z podanej w pierwszym argumencie tablicy lub upłynięcia podanego czasu, w czasie oczekiwania można zażądać opuszczenia kontekstu synchronizacji;
- ***WaitAny(WaitHandle[])*** - blokuje wywołujący wątek do czasu odebrania sygnału na dowolnym uchwycie z podanej w argumencie tablicy *WaitHandle*;
- ***WaitAny(WaitHandle[], Int32, Boolean)***, ***WaitAny(WaitHandle[], TimeSpan, Boolean)*** - blokuje wywołujący wątek do czasu odebrania sygnału na dowolnym uchwycie z podanej w pierwszym argumencie tablicy lub upłynięcia podanego czasu; w czasie oczekiwania można zażądać opuszczenia kontekstu synchronizacji.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Muteksy

Klasa **Mutex** służy do synchronizacji wątków i posiada podobną funkcjonalność do klasy *Monitor*. Jednak w przeciwieństwie do niej, wątki synchronizowane przez klasę *Mutex* nie muszą należeć do tej samej dziedziny aplikacji. Można tworzyć dwa rodzaje muteksów:

- *lokalne* (bez nazwy)
- systemowe (*globalne*, z nazwą).

Muteksy lokalne mogą być używane wyłącznie w procesie, w którym zostały utworzone. **Muteksy systemowe** mogą być używane przez różne procesy z systemu operacyjnego.

Poniższy przykład przedstawia użycie nienazwanych muteksów (z klasy *Mutex*).

```
using System;
using System.Threading;

namespace MuteksyPrzyklad
{
    class Program
    {
        private static Mutex mutex = new Mutex();
        static void MetodaWatku()
        {
            for (int i = 1; i < 8; i++)
            {
                mutex.WaitOne();
                Console.WriteLine("Wejście do sekcji krytycznej w wątku {0} ({1})",
                    Thread.CurrentThread.Name, i);
                Thread.Sleep(100);
                Console.WriteLine("Wyjście z sekcji krytycznej w wątku {0} ({1})",
                    Thread.CurrentThread.Name, i);
                mutex.ReleaseMutex();
            }
        }

        static void Main(string[] args)
        {
            for (int i = 1; i < 5; i++)
            {
                Console.WriteLine("Tworzenie wątku {0}", i);
                Thread t = new Thread(new ThreadStart(MetodaWatku));
                t.Name = "W" + i;
                t.Start();
            }
        }
    }
}
```

Następny przykład przedstawia użycie nazwanych muteksów.

```
using System;
using System.Threading;

class Program
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
    static void MetodaWatku()
    {
        bool czyWlasciciel;
        Mutex mutex = new Mutex(false, "MojMutex", out czyWlasciciel);
        if (czyWlasciciel) Console.WriteLine("Watek {0}: utworzono muteks",
                                             Thread.CurrentThread.Name);
        for (int i = 1; i < 8; i++)
        {
            mutex.WaitOne();
            Console.WriteLine("Wejście do sekcji krytycznej w wątku {0} ({1})",
                               Thread.CurrentThread.Name, i);
            Thread.Sleep(100);
            Console.WriteLine("Wyjście z sekcji krytycznej w wątku {0} ({1})",
                               Thread.CurrentThread.Name, i);
            mutex.ReleaseMutex();
        }
        mutex.Close();
    }

    static void Main(string[] args)
    {
        for (int i = 1; i < 5; i++)
        {
            Console.WriteLine("Tworzenie wątku {0}", i);
            Thread t = new Thread(new ThreadStart(MetodaWatku));
            t.Name = "W" + i;
            t.Start();
        }
    }
}
  
```

Klasa **Mutex** posiada następujące pięć konstruktorów:

- **Mutex()** - inicjalizuje instancję muteksu z domyślnymi ustawieniami;
- **Mutex(Boolean)** - inicjalizuje instancję muteksu z wartością sygnalizującą, czy wywołujący wątek ma stać się właścicielem muteksu;
- **Mutex(Boolean, String)** - inicjalizuje instancję nazwanego muteksu z wartością sygnalizującą, czy wywołujący wątek ma przejąć prawa właściciela muteksu muteksu
- **Mutex(Boolean, String, out Boolean)** - inicjalizuje instancję nazwanego muteksu z wartością sygnalizującą, czy wywołujący wątek ma przejąć prawa właściciela muteksu i pobiera informację, czy wątek stał się właścicielem muteksu;
- **Mutex(Boolean, String, out Boolean, MutexSecurity)** - inicjalizuje instancję nazwanego muteksu z wartością sygnalizującą, czy wywołujący wątek ma przejąć prawa właściciela muteksu oraz zadanymi ustawieniami kontroli i pobiera informację, czy wątek stał się właścicielem muteksu.

Posiada ona również następujące metody:

- **GetAccessControl** - zwraca obiekt klasy **MutexSecurity** odpowiadający ustawieniom kontroli dostępu muteksu;
- **OpenExisting(String)** - otwiera istniejący muteks z nazwą;
- **OpenExisting(String, MutexRights)** - otwiera istniejący muteks z nazwą z zadanymi prawami dostępu (typ wyliczeniowy);

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **ReleaseMutex** - zwalnia zablokowany muteks;
- **SetAccessControl** - ustawia kontrolę dostępu zgodnie z podanym obiektem klasy *MutexSecurity* (zawierającym m.in. kolekcję ACL-i w obiektach klasy *MutexAccessRule*).

W powyższych metodach używany był typ wyliczeniowy **MutexRights** posiadający następujące etykiety:

- **Modify** - prawa do zwolnienia nazwanego muteksu;
- **Delete** - prawa do usunięcia nazwanego muteksu;
- **ReadPermissions** - prawa do otwarcia i odczytu uprawnień kontroli dostępu i uprawnień nazwanego muteksu;
- **ChangePermissions** - prawa do zmiany uprawnień kontroli dostępu i uprawnień skojarzonych z nazwanym muteksem;
- **TakeOwnership** - prawa do zmiany właściciela nazwanego muteksu;
- **Synchronize** - prawa do oczekiwania na nazwany muteks;
- **FullControl** - pełne prawa kontroli do nazwanego muteksu.

Poniższy przykład pokazuje użycie tego typu wyliczeniowego oraz metod do kontroli dostępu do muteksu:

```

using System;
using System.Threading;
using System.Security.AccessControl;
using System.Security.Principal;

public class Program
{
    public static void Main()
    {
        string user = Environment.UserDomainName + @"\\" + Environment.UserName;

        MutexSecurity security = new MutexSecurity();

        MutexAccessRule rule = new MutexAccessRule(user,
            MutexRights.Synchronize | MutexRights.Modify,
            AccessControlType.Allow);
        security.AddAccessRule(rule);

        rule = new MutexAccessRule(user, MutexRights.ChangePermissions,
            AccessControlType.Deny);
        security.AddAccessRule(rule);

        rule = new MutexAccessRule(user, MutexRights.ReadPermissions,
            AccessControlType.Allow);
        security.AddAccessRule(rule);

        foreach (MutexAccessRule ar in
            security.GetAccessRules(true, true, typeof(NTAccount)))
        {
            Console.WriteLine("Użytkownik: {0}", ar.IdentityReference);
            Console.WriteLine("Typ: {0}", ar.AccessControlType);
            Console.WriteLine("Uprawnienia: {0}", ar.MutexRights);
            Console.WriteLine();
        }
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

Semafony

Klasy **Semaphore** można używać do kontroli dostępu do zasobu więcej niż jednego wątku jednocześnie.

Obiekty tej klasy realizują dostęp zgodnie z modelem **semafora zliczającego** (patrz: wykład z przedmiotu *Programowanie Równoległe i Rozproszone*).

Wątki synchronizowane przez te klasę **nie muszą** należeć do tej samej dziedziny aplikacji. Można tworzyć dwa rodzaje semaforów: *lokalne* (bez nazwy) i *systemowe* (z nazwą). Różnice pomiędzy nimi są analogiczne jak pomiędzy lokalnymi i systemowymi muteksami.

Poniżej został przedstawiony przykładowy program, w którym użyte zostały semafory bez nazwy.

```
using System;
using System.Threading;

class Program
{
    private static Semaphore semafor;
    private static int przedzial;

    static void Main(string[] args)
    {
        semafor = new Semaphore(0, 3);
        for (int i = 1; i < 7; i++)
        {
            Thread t = new Thread(MetodaWatku);
            t.Start(i);
        }
        Console.WriteLine("Wątki utworzone ...");
        Thread.Sleep(1000);
        Console.WriteLine("Zwolnienie semafora - zwiększenie wartości o 3");
        semafor.Release(3);
    }

    static void MetodaWatku(object o)
    {
        Console.WriteLine("Wątek {0} - początek", o);
        semafor.WaitOne();
        int p = Interlocked.Add(ref przedzial, 100);
        Console.WriteLine("Wątek {0} - użycie semafora", o);
        Thread.Sleep(1000 + p);
        Console.WriteLine("Wątek {0} - zwolnienie semafora", o);
        Console.WriteLine("Wątek {0} - poprzednia wartość semafora: {1}",
            o, semafor.Release());
    }
}
```

Użyta w tym przykładzie klasa *Interlocked* zostanie omówiona dokładniej pod koniec tego rozdziału.

Poniżej przedstawiona została wersja tego samego przykładu, ale bez użycia pól statycznego przechowujących referencję do obiektu semafora oraz wartość zmiennej. Stosowne dane zostały „owrapowane” i przekazane do wątków przez argument metody wątku.



Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

using System;
using System.Threading;

class Program
{
    private class Arg
    {
        private Semaphore semafor;
        public int przedzial;
        private int numer;
        public Arg(Semaphore semafor)
        {
            this.semafor = semafor;
            this.przedzial = 0;
            this.numer = 0;
        }
        public Semaphore Semafor
        {
            get { return semafor; }
        }
        public int Numer
        {
            get { return numer; }
        }
        public void Inc()
        {
            numer++;
        }
    }

    static void Main(string[] args)
    {
        Arg argument = new Arg(
            new Semaphore(0, 3));

        for (int i = 1; i < 7; i++)
        {
            Thread t = new Thread(MetodaWatku);
            argument.Inc();
            t.Start(argument);
        }
        Console.WriteLine("Wątki utworzone ...");

        Thread.Sleep(1000);

        Console.WriteLine("Zwolnienie semafora -"
            + " zwiększenie wartości o 3");
        argument.Semafor.Release(3);
    }

    static void MetodaWatku(object o)
    {
        Arg a = o as Arg;
        Console.WriteLine("Wątek {0} - początek",
            a.Numer);
        a.Semafor.WaitOne();
    }
}

```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

int p = Interlocked.Add(ref a.przedzial,
                        100);
Console.WriteLine(
    "Wątek {0} - użycie semafora", a.Numer);
Thread.Sleep(1000 + p);
Console.WriteLine("Wątek {0} - zwolnienie"
                  + " semafora", a.Numer);
Console.WriteLine("Wątek {0} - poprzednia"
                  + " wartość semafora: {1}",
                  a.Numer, a.Semafor.Release());
}
}
  
```

Klasa *Semaphore* posiada następujące konstruktory:

- **Semaphore(Int32, Int32)** - inicjalizuje instancję semafora z zadanymi wartościami początkową i maksymalną;
- **Semaphore(Int32, Int32, String)** - inicjalizuje instancję nazwanego semafora z zadanymi wartościami początkową i maksymalną;
- **Semaphore(Int32, Int32, String, out Boolean)** - inicjalizuje instancję nazwanego semafora z zadanymi wartościami początkową oraz maksymalną i zwraca wartość informującą czy semafor został utworzony (**true**), czy też istniał już wcześniej (**false**);
- **Semaphore(Int32, Int32, String, out Boolean, SemaphoreSecurity)** - inicjalizuje instancję nazwanego semafora z zadanymi wartościami początkową i maksymalną oraz zadanymi ustawieniami kontroli (w stosownych ACL-ach) i zwraca wartość informującą czy semafor został utworzony (**true**), czy też istniał już wcześniej (**false**).

Posiada ona również następujące metody:

- **GetAccessControl** - zwraca obiekt klasy *SemaphoreSecurity* odpowiadający ustawieniom kontroli dostępu semafora;
- **OpenExisting(String)** - otwiera istniejący semafor z nazwą;
- **OpenExisting(String, SemaphoreRights)** - otwiera istniejący semafor z nazwą zadanymi prawami dostępu (typ wyliczeniowy);
- **ReleaseMutex** - zwalnia zablokowany semafor;
- **Release()** - zwalnia semafor (zwiększa licznik o 1) i zwraca poprzednią wartość licznika;
- **Release(Int32)** - zwalnia semafor zwiększając licznik o zadaną wartość i zwraca poprzednią wartość licznika;
- **SetAccessControl** - ustawia kontrolę dostępu zgodnie z podanym obiektem klasy *SemaphoreSecurity* (zawierającym m.in. kolekcję ACL-i w obiektach klasy *SemaphoreAccessRule*).

W powyższych metodach używany był typ wyliczeniowy **SemaphoreRights** posiadający następujące etykiety (dualne do etykiet przedstawionego wcześniej typu wyliczeniowego *MutexRights*):

- **Modify** - prawa do zwolnienia nazwanego semafora;
- **Delete** - prawa do usunięcia nazwanego semafora;
- **ReadPermissions** - prawa do otwarcia i odczytu uprawnień kontroli dostępu i uprawnień nazwanego semafora;
- **ChangePermissions** - prawa do zmiany uprawnień kontroli dostępu i uprawnień skojarzonych z nazwanym semafora;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **TakeOwnership** - prawa do zmiany właściciela nazwanego semafora;
- **Synchronize** - prawa do oczekiwania na nazwany semafora;
- **FullControl** - pełne prawa kontroli do nazwanego semafora.

Sposób używania tego typu wyliczeniowego oraz metod do kontroli dostępu do semafora jest podobny jak sposób używania przedstawionych uprzednio dualnych mechanizmów dla muteksów.

Rygle odczytu-zapisu

Klasa **ReaderWriterLock** implementuje rozwiązywanie problemu pisarzy i czytelników.

- Klasy **ReaderWriterLock** można używać do synchronizacji odczytu i zapisu, przy czym w tym samym momencie jeden wątek może mieć albo prawo do zapisu albo do odczytu (nie oba naraz).
- Dostęp do odczytu może być przydzielony jednocześnie dla więcej niż jednego wątku (podobnie jak to ma miejsce w przypadku klasy **Semaphore**).
- Dostęp do zapisu w określonym momencie może być przydzielony dla co najwyżej jednego wątku.
- Jeśli jakiś wątek ma przydzielony dostęp do zapisu, to żaden wątek nie może mieć w tym samym momencie przydzielonego dostępu do odczytu.

Klasa **ReaderWriterLock** posiada następujące własności:

- **IsReaderLockHeld** - zwraca wartość informującą o tym, czy bieżący wątek ma przydzielony dostęp do odczytu (założony rygiel odczytu);
- **IsWriterLockHeld** - zwraca wartość informującą o tym, czy bieżący wątek ma przydzielony dostęp do zapisu (założony rygiel zapisu);
- **WriterSeqNum** - zwraca wartość informującą o tym ile razy bieżący wątek miał przydzielony dostęp do zapisu (założony rygiel zapisu);

Posiada ona również następujące metody:

- **AcquireReaderLock** - próbuje w ciągu zadanego okresu czasu (albo w strukturze **TimeSpan** albo w postaci liczby milisekund) ustawić blokadę do odczytu (jeśli jest ustawiona, to zwiększa jej licznik);
- **AcquireWriterLock** - próbuje w ciągu zadanego okresu czasu (albo w strukturze **TimeSpan** albo w postaci liczby milisekund) ustawić blokadę do zapisu (jeśli jest ustawiona, to zwiększa jej licznik);
- **AnyWritersSince** - zwraca wartość informującą o tym czy rygiel odczytu-zapisu miał przydzielony dostęp do zapisu (miał założony rygiel zapisu) w czasie od podanej liczby sekwencyjnej;
- **DowngradeFromWriterLock** - zwalnia rygiel zapisu i przywraca rygiel odczytu, argumentem tej metody jest obiekt klasy **LockCookie** zwrócony przez metodę **UpgradeToWriterLock**;
- **ReleaseLock** - zwalnia blokadę z rygla niezależnie od liczby żądań założenia blokady wywołanych w wątku;
- **ReleaseReaderLock** - zmniejsza licznik blokady odczytu o 1, jeśli licznik osiągnie wartość zero blokada jest zdejmowana;
- **ReleaseWriterLock** - zmniejsza licznik blokady zapisu o 1, (jeśli licznik osiągnie wartość zero blokada jest zdejmowana);
- **RestoreLock** - przywraca rygiel do stanu poprzedzającego wywołanie metody **ReleaseLock**, (argumentem tej metody jest obiekt klasy **LockCookie** zwrócony przez metodę **ReleaseLock**);
- **UpgradeToWriterLock** - próbuje w ciągu zadanego okresu czasu (albo w strukturze **TimeSpan** albo w postaci liczby milisekund) atomicznie zwolnić blokadę odczytu (niezależnie od stanu licznika) i ustawić blokadę do zapisu.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Klasa *EventWaitHandle* i jej klasy pochodne (uchwyty oczekiwania na zdarzenia)

Klasa ***EventWaitHandle*** jest klasą pochodną klasy *WaitHandle*. Służy ona do przesyłania powiadomień pomiędzy wątkami i procesami (również w tym przypadku mamy wersję lokalną (bez nazwy) i globalną (z nazwą)). Posiada ona dwie klasy pochodne: *AutoResetEvent* i *ManualResetEvent*.

W przypadku klasy ***AutoResetEvent*** uchwyt zdarzenia blokuje się automatycznie po odebraniu sygnału przez dowolny wątek.

Przy używaniu klasy ***ManualResetEvent*** po odebraniu sygnału przez wątek uchwyt zdarzenia należy „ręcznie” zablokować (metodą *Reset*).

W poniższym przykładzie używana jest klasa *EventWaitHandle*. W momencie tworzenia obiektu tej klasy (w konstruktorze) wskazywane jest, czy uchwyt zdarzenia ma być automatycznie blokowany po odebraniu sygnału.

```

using System;
using System.Threading;

class Program
{
    const int IlWatkow = 5;
    private class Arg
    {
        private EventWaitHandle wh1;
        private EventWaitHandle wh2;
        public long IlBlokad;

        public Arg(EventWaitHandle wh1, EventWaitHandle wh2)
        {
            this.wh1 = wh1;
            this.wh2 = wh2;
            this.IlBlokad = 0;
        }
        public EventWaitHandle Uchwyty1 { get { return wh1; } }
        public EventWaitHandle Uchwyty2 { get { return wh2; } }
    }

    public static void Main()
    {
        Arg argument = new Arg(
            //new EventWaitHandle(false, EventResetMode.AutoReset),
            //new EventWaitHandle(false, EventResetMode.AutoReset)
            new EventWaitHandle(false, EventResetMode.ManualReset),
            new EventWaitHandle(false, EventResetMode.ManualReset)
        );

        for (int i = 1; i <= IlWatkow; i++)
        {
            Thread watek = new Thread(
                new ParameterizedThreadStart(MetodaWatku)
            );
            watek.Name = "Watek " + i;
            watek.Start(argument);
        }
    }
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    while (Interlocked.Read(ref argument.IlBlokad) < IlWatkow)
    {
        Thread.Sleep(500);
    }
    while (Interlocked.Read(ref argument.IlBlokad) > 0)
    {
        Console.WriteLine("Wciśnij ENTER aby zwolnić zablokowane wątki.");
        Console.ReadLine();
        WaitHandle.SignalAndWait(argument.Uchwytl, argument.Uchwyt2);
    }
}
public static void MetodaWatku(object o)
{
    Arg a = o as Arg;
    Console.WriteLine("Blokowanie {0}", Thread.CurrentThread.Name);
    Interlocked.Increment(ref a.IlBlokad);
    a.Uchwytl.WaitOne();
    Console.WriteLine("Kończenie {0}", Thread.CurrentThread.Name);
    Interlocked.Decrement(ref a.IlBlokad);
    a.Uchwyt2.Set();
}
}
  
```

W kolejnym przykładzie używana jest klasa *AutoResetEvent*.

```

using System;
using System.Threading;

class Program
{
    const int IlWatkow = 5;
    private class Arg
    {
        private AutoResetEvent wh1;
        private AutoResetEvent wh2;
        public long IlBlokad;

        public Arg(AutoResetEvent wh1, AutoResetEvent wh2)
        {
            this.wh1 = wh1;
            this.wh2 = wh2;
            this.IlBlokad = 0;
        }
        public AutoResetEvent Uchwytl { get { return wh1; } }
        public AutoResetEvent Uchwyt2 { get { return wh2; } }
    }

    public static void Main()
    {
        Arg argument = new Arg(
            new AutoResetEvent(false),
            new AutoResetEvent(false)
        );
        for (int i = 1; i <= IlWatkow; i++)
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    {
        Thread watek = new Thread(
            new ParameterizedThreadStart(MetodaWatku)
        );
        watek.Name = "Watek " + i;
        watek.Start(argument);
    }
    while (Interlocked.Read(ref argument.IlBlokad) < IlWatkow)
    {
        Thread.Sleep(500);
    }
    while (Interlocked.Read(ref argument.IlBlokad) > 0)
    {
        Console.WriteLine("Wciśnij ENTER aby zwolnić zablokowane wątki.");
        Console.ReadLine();
        WaitHandle.SignalAndWait(argument.Uchwytl, argument.Uchwytn);
    }
}
public static void MetodaWatku(object o)
{
    Arg a = o as Arg;
    Console.WriteLine("Blokowanie {0}", Thread.CurrentThread.Name);
    Interlocked.Increment(ref a.IlBlokad);
    a.Uchwytl.WaitOne();
    Console.WriteLine("Kończenie {0}", Thread.CurrentThread.Name);
    Interlocked.Decrement(ref a.IlBlokad);
    a.Uchwytn.Set();
}
}
  
```

W ostatnim przykładzie używana jest klasa *ManualResetEvent*.

```

using System;
using System.Threading;

class Program
{
    const int IlWatkow = 5;
    private class Arg
    {
        private ManualResetEvent wh1;
        private ManualResetEvent wh2;
        public long IlBlokad;

        public Arg(ManualResetEvent wh1, ManualResetEvent wh2)
        {
            this.wh1 = wh1;
            this.wh2 = wh2;
            this.IlBlokad = 0;
        }
        public ManualResetEvent Uchwytl { get { return wh1; } }
        public ManualResetEvent Uchwytn { get { return wh2; } }
    }
    public static void Main()
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
    Arg argument = new Arg(
        new ManualResetEvent(false),
        new ManualResetEvent(false)
    );

    for (int i = 1; i <= IlWatkow; i++)
    {
        Thread watek = new Thread(
            new ParameterizedThreadStart(MetodaWatku)
        );
        watek.Name = "Watek " + i;
        watek.Start(argument);
    }
    while (Interlocked.Read(ref argument.IlBlokad) < IlWatkow)
    {
        Thread.Sleep(500);
    }
    while (Interlocked.Read(ref argument.IlBlokad) > 0)
    {
        Console.WriteLine("Wciśnij ENTER aby zwolnić zablokowane wątki.");
        Console.ReadLine();
        WaitHandle.SignalAndWait(argument.Uchwyty1, argument.Uchwyty2);
    }
}
public static void MetodaWatku(object o)
{
    Arg a = o as Arg;
    Console.WriteLine("Blokowanie {0}", Thread.CurrentThread.Name);
    Interlocked.Increment(ref a.IlBlokad);
    a.Uchwyty1.WaitOne();
    Console.WriteLine("Kończenie {0}", Thread.CurrentThread.Name);
    Interlocked.Decrement(ref a.IlBlokad);
    a.Uchwyty2.Set();
}
}
  
```

Klasa **RegisteredWaitHandle**

Klasa **RegisteredWaitHandle** implementuje uchwyty zwracane metodą *RegisterWaitForSingleObject* klasy *ThreadPool*. Jest ona klasą pochodną klasy *WaitHandle*.

Metoda **RegisterWaitForSingleObject** służy do zarezerwowania miejsca w puli (kolejce) wątków klasy *ThreadPool*. Uchwyty zdarzenia przekazywany jako argument tej metody służy do przesyłania powiadomienia o zwolnieniu zasobu wymaganego przez wątek (wówczas wątek zajmuje stosowne miejsce w kolejce).

Delegacja rejestrowanej metody ma następującą deklarację

```
public delegate void WaitOrTimerCallback(Object state, bool timedOut)
```

Miejsce w puli (kolejce) wątków można też zwolnić (bez uruchamiania wątku) wywołując metodę **Unregister**.

Poniżej podany został przykład użycia klasy *RegisteredWaitHandle* do przesłania notyfikacji dla wątku z puli.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

using System;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i <= 1; i++)
        {
            AutoResetEvent zdarzenie = new AutoResetEvent(false);

            RegisteredWaitHandle handle =
                ThreadPool.RegisterWaitForSingleObject(
                    zdarzenie, MetodaWatku, null, 1000,
                    (i == 0)); //false == ponawiać

            Thread.Sleep(4000);
            Console.WriteLine("Wysłanie notyfikacji.");
            zdarzenie.Set();

            Thread.Sleep(2000);
            Console.WriteLine("Koniec oczekiwania w wątku głównym.");
        }
    }

    static void MetodaWatku(object o, bool timedOut)
    {
        if (!timedOut)
        {
            Console.WriteLine("Wątek roboczy: WŁAŚCIWA PRACA.");
        }
        else
        {
            Console.WriteLine("Wątek roboczy: przeterminowanie.");
        }
    }
}
  
```

Klasa **Interlocked**

Klasa **Interlocked** służy do przeprowadzania **atomicznych** (czyli inaczej **niepodzielnych**) operacji na współdzielonych zmiennych. Klasa ta posiada następujące statyczne metody.

- **Add** - dodaje wskazaną wartość całkowitoliczbową do zmiennej;
- **CompareExchange** - porównuje wartość zmiennej z zadana wartością; jeśli są one równe, to zmiennej przypisywana jest zadana wartość (podana jako dodatkowy argument);
- **Decrement** - zmniejsza zmiennej całkowitoliczbową o 1;
- **Exchange** - przypisuje zmiennej wskazaną wartość i zwraca poprzednio przechowywaną pod tą zmiennej wartość;
- **Increment** - zwiększa zmiennej całkowitoliczbową o 1;
- **Read** - zwraca wartość zmiennej 64-bitowej (w systemach 32 bitowych zwykłe odczytanie wartości zmiennej 64 bitowej nie musi być atomiczne).



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykłady użycia tej klasy pojawiły się już w kodach zamieszczanych jako przykłady użycia klas *Semaphore*, *EventWaitHandle*, *AutoResetEvent* i *ManualResetEvent*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

7. Asynchroniczne uruchamianie metod

Mając wprowadzone wątki możemy przejść do koncepcji wykonywania operacji asynchronicznych w języku C#.

Koncepcja programowania asynchronicznego

Przypomnijmy najpierw pojęcia używane w tym rozdziale.

W modelu **synchronicznym** podzielonym na zadania po uruchomieniu zadania oczekujemy na jego wynik przed przejściem do kolejnych zadań.

W modelu **asynchronicznym** po uruchomieniu zadania możemy przejść do kolejnych zadań nie czekając na wynik uruchomionego zadania. Musimy tylko zapewnić odebranie wyniku przed uruchomieniem kolejnych zadań, które będą go potrzebować.

Mechanizmy służące do wykonywania operacji asynchronicznych

Klasyczne rozwiązanie z jawnym użyciem wątków

Naturalnym sposobem implementacji asynchronicznych zadań wydaje się umieszczenie ich w osobnych metodach, uruchamianie ich w nowych wątkach (np. z użyciem klasy *Thread* lub *Task*) i zbieraniem ich wyników, kiedy zostaną otrzymane (np. przez zdarzenia) lub oczekiwaniem na ich otrzymanie (np. wywołaniem metody *Join*) kiedy będą one potrzebne. To rozwiązanie jest proste w założeniach, ale jego realizacja powoduje znaczny rozrost kodu. Znacznie łatwiej jest też popełnić błędy w tak pisanych programach. Z tego powodu tego typu rozwiązania są zwykle stosowane wyłącznie dla wybranych lub bardziej złożonych operacji.

Delegacje w programowaniu synchronicznym i asynchronicznym

Bardzo użytecznych mechanizmów do uruchamiania zadań dostarczają delegacje.

Przypomnijmy, że pod delegację możemy podpisać jedną lub więcej metod. Wywołując delegację wywołujemy podpięte pod nią metody. Do bezpośredniego wywoływanego podpiętej pod delegację metody lub metod mogą też służyć metody *Invoke* oraz *BeginInvoke* delegacji.

W modelu **synchronicznym** wywołujemy metodę ***Invoke*** delegacji, która uruchamia podpiętą pod delegację metodę (lub metody) w **biejącym wątku**. W modelu **asynchronicznym** używamy metody ***BeginInvoke***, która powoduje utworzenie nowego wątku i uruchomienie w nim podpiętej pod delegację metody oraz metody ***EndInvoke***, która służy do odebrania wyniku.

Asynchroniczne operacje z użyciem *IAsyncResult*

Do prowadzenia asynchronicznych operacji można używać elementów interfejsu *IAsyncResult* (np. *FileStream.BeginRead* i *FileStream.EndRead*).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Asynchroniczne operacje z użyciem zdarzeń

Do przeprowadzania asynchronicznych operacji można używać zdarzeń.

Obsługa asynchronicznego wywołania

W przypadku wykonywania operacji asynchronicznych konieczne jest dostarczenie mechanizmów pozwalających na sprawdzenie czy nastąpiło zakończenie asynchronicznego wywołania lub wstrzymanie bieżącego wątku do tego czasu (na synchronizację). W języku C# można to wykonać na cztery sposoby:

- wywołując metodę **EndInvoke** w głównym wątku, który wywołał metodę **BeginInvoke** (zwłaszcza jeśli w wywołaniu **BeginInvoke** nie podano delegacji zwrotnej);
- używając uchwytu oczekiwania (obiektu z klasy dziedziczącej po klasie **WaitHandle**) do przesyłania notyfikacji;
- sprawdzając własność **IsCompleted** obiektu (klasy dziedziczącej po interfejsie **IAsyncResult**) zwróconego przez metodę **BeginInvoke**;
- używając delegacji zwrotnej (obiektu klasy **AsyncCallback**) przy wywoływaniu metody **BeginInvoke**.

Składowe interfejsu **IAsyncResult**

Interfejs **IAsyncResult** posiada następujące własności:

- **AsyncResult** - zwraca obiekt podany jako ostatni argument metody inicjalizującej wywołanie asynchroniczne;
- **AsynWaitHandle** - zwraca uchwyt przeznaczony do oczekiwania na zakończenie operacji asynchronicznej
- **CompletedSynchronously** - zwraca wartość informującą o tym, czy operacja asynchroniczna się zakończyła się w sposób synchroniczny (większość implementacji tego interfejsu nie używa tej własności i powinna ona wówczas zwracać wartość **false**, jednak niektóre implementacje mogą realizować operacje w sposób synchroniczny np. przy „prostych” operacjach na wejściu/wyjściu);
- **IsCompleted** - zwraca wartość informującą o tym, czy operacja asynchroniczna się zakończyła.

Przykład

```
using System;
using System.Threading;

class Program
{
    delegate int Delegacja(int liczba);
    static Delegacja delegacja;
    static int Fibonacci(int liczba)
    {
        if (liczba < 1)
            return 0;
        else if (liczba == 1 || liczba == 2)
            return liczba;
        else
            return Fibonacci(liczba - 2) + Fibonacci(liczba - 1);
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

static void WyswietlWynik(IAsyncResult asRes)
{
    int wynik = delegacja.EndInvoke(asRes);
    Console.WriteLine(
        "Wartość {0}-go elementu w ciągu Fibonaciego wynosi {1}",
        asRes.AsyncState.ToString(), wynik);
}
static void Main(string[] args)
{
    delegacja = new Delegacja(Fibonacci);
    AsyncCallback wywolanieZwrotne = new AsyncCallback(WyswietlWynik);
    Console.Write("Podaj liczbę naturalną: ");
    int liczba = int.Parse(Console.ReadLine());
    int priorytet = liczba;
    delegacja.BeginInvoke(priorytet, wywolanieZwrotne, liczba);
    Console.WriteLine("Oczekiwanie na wykonanie operacji");
    Thread.Sleep(5000);
}
}
  
```

Kontekst uruchomieniowy

Rola kontekstu uruchomieniowego

Każdy wątek uruchamiany jest w pewnym konkretnym kontekście uruchomieniowym. Taki kontekst zawiera między innymi informacje o kontroli bezpieczeństwa, transakcjach, synchronizacji i ustawieniach lokalnych wątku.

Przy asynchronicznym wywoływaniu metod, metody mogą być uruchamiane w innym kontekście uruchomieniowym, niż kontekst wywołującego wątku.

Klasa **ExecutionContext** pozwala kontrolować przepływ (propagację) kontekstu uruchomieniowego, klasa **HostExecutionContext** przechowuje stan kontekstu, natomiast klasa **HostExecutionContextManager** pozwala na zarządzanie kontekstem.

Składowe klasy **ExecutionContext**

Klasa *ExecutionContext* posiada następujące metody:

- **Capture** - pobiera kontekst uruchomieniowy z bieżącego wątku;
- **CreateCopy** - tworzy kopię kontekstu uruchomieniowego;
- **IsFlowSuppressed** - informuje o tym, czy przepływ kontekstu uruchomieniowego jest ukrywany (pozwala na stwierdzenie, czy kontekst uruchomieniowy będzie przenoszony w asynchronicznych wywołaniach);
- **RestoreFlow** - pozwala na przywrócenie przepływu kontekstu uruchomieniowego przy wywoływaniu wątków asynchronicznych (zamiast tej metody należy wywoływać metodę *Undo* obiektu *AsyncFlowControl* zwróconego przez *SuppressFlow*);
- **Run** - uruchamia metodę (zadaną przez delegację *ContextCallback*) w zadanym kontekście uruchomieniowym;
- **SuppressFlow** - blokuje przepływ kontekstu uruchomieniowego do wywoływanych asynchronicznych wątków.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Składowe klasy *HostExecutionContext*

Klasa *HostExecutionContext* posiada między innymi następujące składowe:

- metodę **CreateCopy**, która tworzy kopię kontekstu uruchomieniowego;
- własność **State**, która pobiera lub ustawia stan kontekstu.

Składowe klasy *HostExecutionContextManager*

Klasa *HostExecutionContextManager* posiada następujące metody:

- **Capture** - pobiera kontekst uruchomieniowy z bieżącego wątku;
- **Revert** - pozwala przywrócić kontekst uruchomieniowy do poprzedniego stanu;
- **SetHostExecutionContext** - ustawia bieżący kontekst uruchomieniowy na kontekst podany w argumencie.

Kontekst synchronizacji

Rola kontekstu synchronizacji

Klasa **SynchronizationContext** służy do synchronizacji wątków korzystających z tych samych zasobów. Pozwala ona na kontrolę dostępu zarówno dla operacji synchronicznych, jak i asynchronicznych.

Do uruchamiania operacji synchronicznych służy do tego metoda *Send*, a do inicjowania operacji asynchronicznych służy do tego metoda *Post*. Obydwie ze wspomnianych metod jako argument przyjmują delegacje będące obiektami klasy *SendOrPostCallback*.

Składowe klasy *SynchronizationContext*

Klasa *SynchronizationContext* posiada statyczną własność **Current**, która pobiera obiekt opisujący kontekst synchronizacji bieżącego wątku. Ponadto klasa ta posiada następujące metody:

- **CreateCopy** - tworzy kopię kontekstu synchronizacji;
- **IsWaitNotificationRequired** - zwraca wartość **true** jeśli wymagane jest wysyłanie powiadomienia o oczekiwaniu;
- **OperationCompleted** - odpowiada na powiadomienie o zakończeniu operacji;
- **OperationStarted** - odpowiada na powiadomienie o rozpoczęciu operacji;
- **Post** - uruchamia nowy wątek w tym samym kontekście, co kontekst wątku wywołującego;
- **Send** - uruchamia synchronicznie metodę w kontekście synchronizacji bieżącego wątku;
- **SetSynchronizationContext** - ustawia kontekst synchronizacji bieżącego wątku;
- **Wait** - wysyła powiadomienie do wątków i zatrzymuje pozostałe wątki do czasu zakończenia ich operacji.

Klasyczny przykład (z dokumentacji) na użycie kontekstu synchronizacji

```
using System;
using System.Threading;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

class KontoBankowe
{
    public decimal saldo;

    public void Wplata(decimal kwota) { saldo += kwota; }
    public void Wyplata(decimal kwota) { saldo -= kwota; }
}

class Program
{
    static KontoBankowe konto;

    static void Main(string[] args)
    {
        konto = new KontoBankowe();
        SendOrPostCallback wpłata = new SendOrPostCallback(Wplata);
        SendOrPostCallback wyplata = new SendOrPostCallback(Wyplata);
        SynchronizationContext kontekst = new SynchronizationContext();
        kontekst.Post(wpłata, 100);
        kontekst.Post(wyplata, 100);
        Console.ReadLine();
    }

    static void Wyplata(object kwotaOperacji)
    {
        Console.WriteLine("Wyplata: bieżące saldo = {0:C}", konto.saldo);
        konto.Wyplata(decimal.Parse(kwotaOperacji.ToString()));
        Console.WriteLine("Wyplata: nowe saldo = {0:C}", konto.saldo);
    }

    static void Wplata(object kwotaOperacji)
    {
        Console.WriteLine("Wpłata: bieżące saldo = {0:C}", konto.saldo);
        konto.Wplata(decimal.Parse(kwotaOperacji.ToString()));
        Console.WriteLine("Wpłata: nowe saldo = {0:C}", konto.saldo);
    }
}
  
```

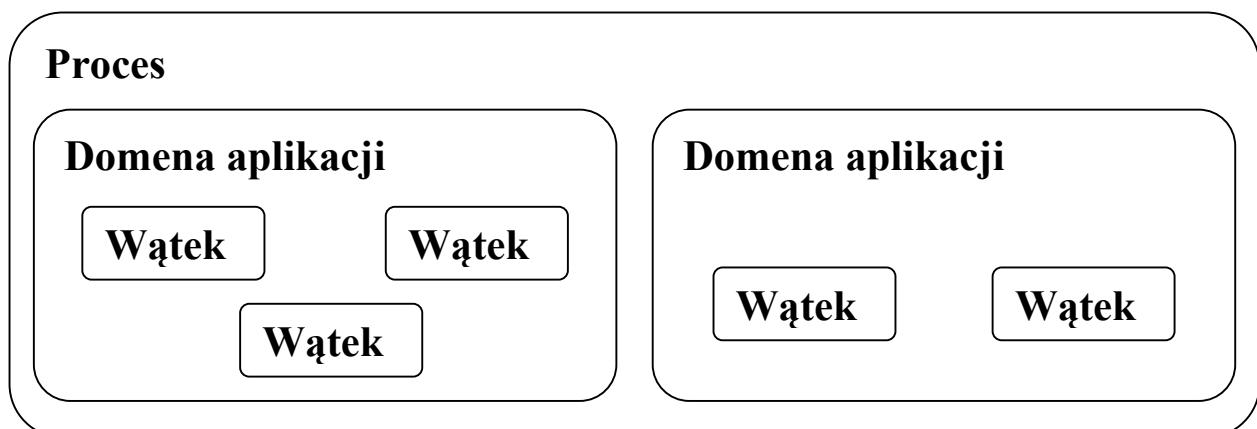


Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

8. Domeny aplikacji

Przypomnijmy najpierw, że zgodnie z informacjami wprowadzonymi w temacie szóstym „Wątki” domeny aplikacji grupują wątki w obrębie procesu, jak to zostało przedstawione na poniższym schemacie

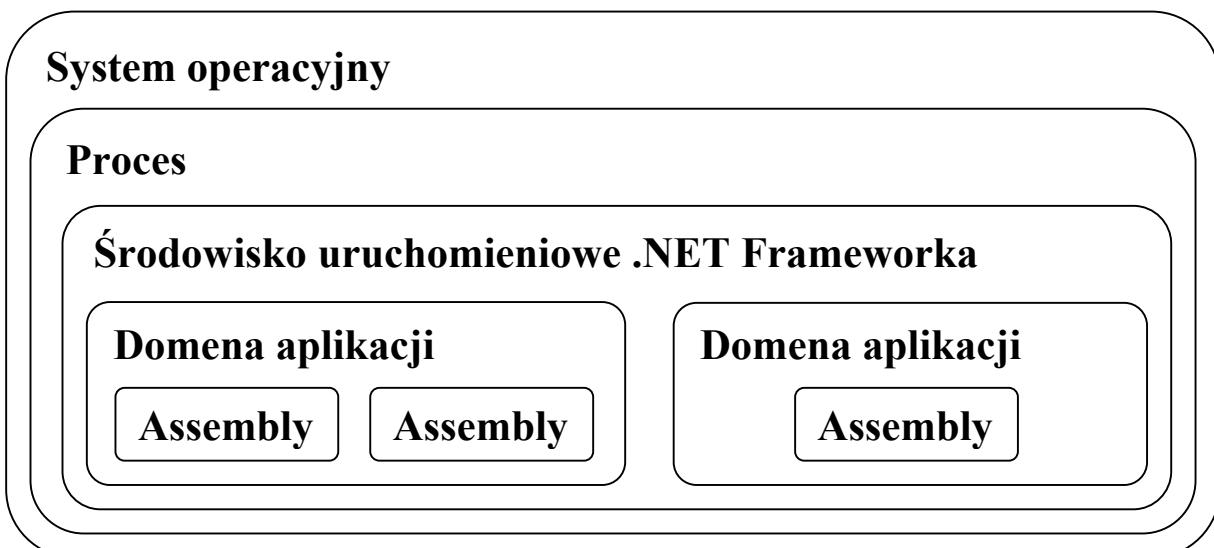


W dalszej części rozdziału dokładniej zagłębimy się w rolę domen aplikacji oraz poznamy sposoby zarządzania nimi.

Domeny aplikacji, a procesy i wątki

Na początku należy uzmysolić sobie, że w .NET Framework „jednostką uruchomieniową” (izolowanym środowiskiem, w którym uruchamiana jest aplikacja) nie są procesy, ale dziedziny aplikacji.

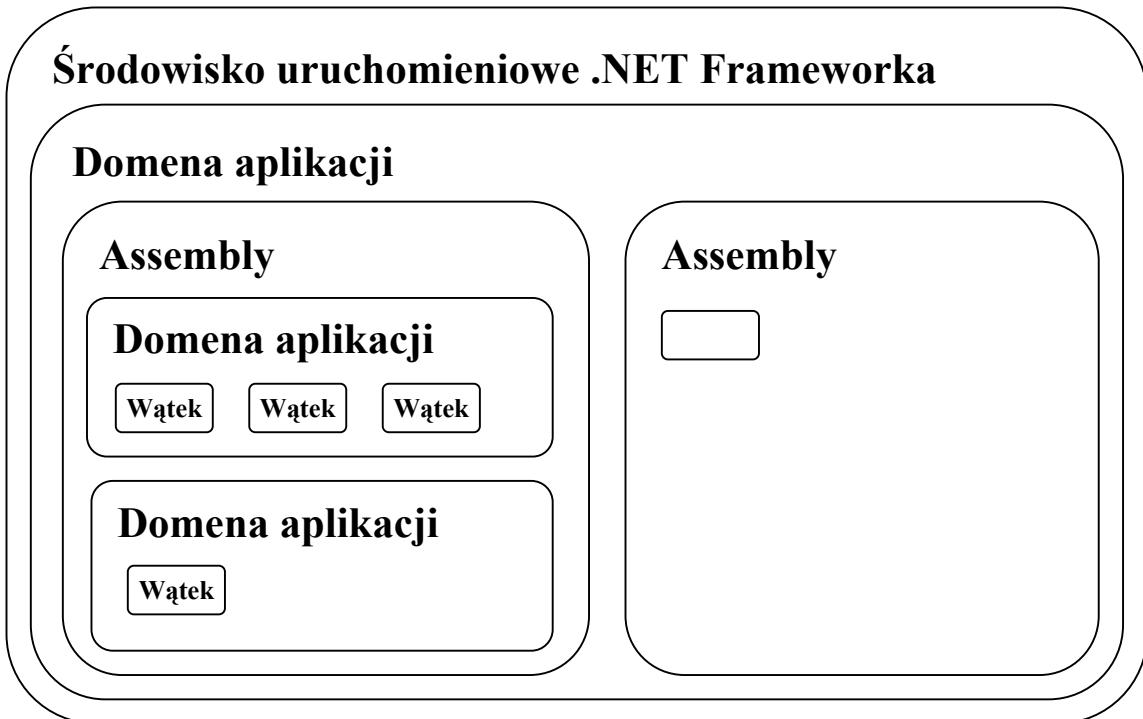
Poniższy schemat dokładniej przedstawia „umiejscowienie” kodu programu wykonywalnego w pamięci operacyjnej komputera.



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Nadrzędną rolę odgrywa system operacyjny komputera. W systemie operacyjnym działają procesy, w obrębie których alokowane są zasoby (np. obszar pamięci). O ile w proces używa kodu zarządzanego, to w procesie działa środowisko uruchomieniowe .NET Frameworka (zamiast albo oprócz niego mogą występować inne elementy, które na razie nie są przez nas rozważane). Uruchamiany program (kod wykonywalny lub biblioteka) w kodzie zarządzalnym jest ładowane w pewnej domenie aplikacji, która pełni rolę „środowiska” w którym umieszczany jest kod (nazywamy go tu *assembly*). Jak zostało pokazane na powyższym schemacie, w procesie może być umieszczona więcej niż jedna domena aplikacji, a ponadto w domenie aplikacji można załadować więcej niż jedno *assembly*.

Co więcej, domeny aplikacji można zagnieździć, co zostało pokazane na poniższym schemacie.



Klasy związane z domenami aplikacji

Do zarządzania domenami aplikacji służą głównie następujące dwie klasy

- **AppDomain** - klasa reprezentująca domenę aplikacji - izolowane środowisko w którym uruchamiana jest aplikacja;
- **AppDomainSetup** - klasa reprezentująca dowiązywanie do instancji klasy *AppDomain* informacje o uruchamianym assembly.

Klasa *AppDomain*

Klasa *AppDomain* posiada następujące własności:

- **ActivationContext** - pobiera kontekst aktywacji z dziedziny aplikacji;
- **ApplicationIdentity** - pobiera informacje identyfikujące aplikację z domeny aplikacji;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **ApplicationTrust** - pobiera informacje opisujące poziom uprawnień nadanych aplikacji (w szczególności, czy jest na tyle zaufana, że poziom uprawnień wystarcza do uruchomienia);
- **BaseDirectory** - pobiera ścieżkę do głównego katalogu przeszukiwanego w celu znalezienia assembly;
- **CurrentDomain** - pobiera obiekt reprezentujący domenę aplikacji bieżącego wątku;
- **DomainManager** - pobiera obiekt klasy *AppDomainManager* opisujący udostępnionego w momencieinicjalizacji domeny aplikacji menadżera domeny;
- **DynamicDirectory** - pobiera ścieżkę do katalogu przeszukiwanego w celu znalezienia dynamicznie generowanych assembly;
- **Evidence** - pobiera informacje związane z polityką bezpieczeństwa;
- **FriendlyName** - pobiera zaprzyjaźnioną nazwę domeny aplikacji;
- **Id** - pobiera wartość całkowitoliczbową jednoznacznie identyfikującą dziedzinę aplikacji wewnętrz procesu;
- **RelativeSearchPath** - pobiera ścieżkę w obrębie katalogu głównego assembly do katalogu przeszukiwanego w celu znalezienia dynamicznie prywatnych assembly;
- **SetupInformation** - pobiera informacje o konfiguracji (obiekt klasy *AppDomainSetup*) domeny aplikacji;
- **ShadowCopyDirectories** - pobiera oraz ustawia ścieżkę do katalogu zawierającego assembly dla tworzenia ukrytych kopii;
- **ShadowCopyFiles** - pobiera oraz ustawia informację o tym, czy domena aplikacji jest skonfigurowana aby tworzyć ukryte kopie plików (assembly tego typu mogą być m.in. uaktualniane bez ich usuwania (unloading) z domeny aplikacji).

Klasa *AppDomain* posiada też następujące metody:

- **ApplyPolicy** - zwraca nazwę assembly po zastosowaniu polityki (tłumaczy nazwę assembly z postaci zgodnej z właściwością *Assembly.FullName*);
- **CreateComInstanceFrom** - tworzy nową instancję obiektu COM;
- **CreateDomain** - tworzy nową domenę aplikacji;
- **CreateInstance** - tworzy nową instancję wskazanego typu zdefiniowanego w podanym assembly (po nazwie zgodnej z *Assembly.FullName*);
- **CreateInstanceAndUnwrap** - tworzy nową instancję wskazanego typu zdefiniowanego w podanym assembly (po nazwie zgodnej z *Assembly.FullName*);
- **CreateInstanceFrom** - tworzy nową instancję wskazanego typu zdefiniowanego w podanym assembly (przez plik i ścieżkę);
- **CreateInstanceFromAndWrap** - tworzy nową instancję wskazanego typu zdefiniowanego w podanym assembly (przez plik i ścieżkę);
- **DefineDynamicAssembly** - definiuje dynamiczne assembly o podanej nazwie i trybie dostępu;
- **DoCallBack** - wykonuj kod innej domeny aplikacji wskazanej przez podaną delegację;
- **ExecuteAssembly** - wykonuje assembly umieszczone we wskazanym pliku;
- **ExecuteAssemblyByName** - wykonuje assembly o nazwie zgodnej z *Assembly.FullName*;
- **GetAssemblies** - pobiera tablicę obiektów klasy *Assembly* reprezentujących assembly załadowane do kontekstu wykonywania dziedziny aplikacji;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **GetLifetimeService** - pobiera obiekt klasy *LifetimeServices* opisujący czas życia usługi (ustalający czas życia dla tej instancji);
- **GetData** - zwraca wartość przypisaną w dziedzinie aplikacji pod zadaną nazwą;
- **InitializeLifetimeService** - ustawia czas życia na nieskończoność
- poprzez zapobieganie ustanawianiu czasu dzierżawy;
- **IsDefaultAppDomain** - zwraca wartość informującą o tym, czy dziedzina aplikacji jest domyślną dziedziną aplikacji procesu;
- **IsFinalizingForUnload** - zwraca wartość informującą o tym, czy dziedzina aplikacji jest w trakcie usuwania (unloading) i obiekty w niej zawarte są finalizowane przez CLR;
- **Load** - wczytuje assembly do dziedziny aplikacji;
- **ReflectionOnlyGetAssemblies** - pobiera tablicę obiektów klasy *Assembly* reprezentujących assembly załadowane do kontekstu „reflection-only” dziedziny aplikacji (będzie o tym mowa w temacie poświęconym refleksjom);
- **SetAppDomainPolicy** - ustawia poziom polityki bezpieczeństwa dla dziedziny aplikacji przypisuje pod zadaną nazwą podaną wartość w dziedzinie aplikacji;
- **SetData** - przypisuje pod zadaną nazwą podaną wartość w dziedzinie aplikacji;
- **SetDynamicBase** - ustawia podaną ścieżkę jako miejsce do przechowywania i używania dynamicznie tworzonych plików;
- **SetPrincipalPolicy** - określa jak obiekty związane z polityką (role i obiekty identyfikujące) mają być dołączane, kiedy wątek będzie próbował je dowiązać w trakcie uruchamiania dziedziny aplikacji;
- **SetThreadPrincipal** - ustawia domyślną rolę przypisywaną wątkowi w momencie, gdy ten próbuje dowiązać rolę w momencie wykonywania operacji w dziedzinie aplikacji
- **Unload** - usuwa wskazaną domenę aplikacji.

Klasa AppDomainSetup

Klasa *AppDomainSetup* posiada następujące własności:

- **ActivationArguments** - pobiera kontekst aktywacji z dziedziny aplikacji;
- **AppDomainInitializer** - pobiera lub ustawia delegację, która odpowiada metodzie zwrotnej (*callback*) wywoływanej w momencie inicjalizacji domeny aplikacji;
- **AppDomainInitializerArguments** - pobiera lub ustawia argumenty delegacji, która odpowiada metodzie zwrotnej (*callback*) wywoywanej w momencie inicjalizacji domeny aplikacji;
- **ApplicationName** - pobiera lub ustawia nazwę aplikacji;
- **ApplicationTrust** - pobiera lub ustawia informacje opisujące poziom uprawnień nadanych aplikacji (w szczególności, czy jest na tyle zaufana, że poziom uprawnień wystarcza do uruchomienia);
- **ConfigurationFile** - pobiera lub ustawia nazwę pliku konfiguracyjnego dla domeny aplikacji;
- **DisallowApplicationBaseProbing** - określa czy ścieżka główna i prywatna mają być sprawdzane przy przeszukiwaniu assemblies do wczytania;
- **DisallowBindingRedirects** - pobiera lub ustawia wartość decydującą o tym, czy domena aplikacji pozwala assembly na dowiązywanie przekierowań;
- **DisallowCodeDownload** - pobiera lub ustawia wartość decydującą o tym, czy domena aplikacji pozwala na wczytywanie assembly przez protokół HTTP;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **DisallowPublisherPolicy** - pobiera lub ustawia wartość decydującą o tym, czy sekcja polityki dostawcy pliku konfiguracyjnego ma zastosowanie do domeny aplikacji domena aplikacji pozwala assembly na dowiązywanie przekierowań;
- **DynamicBase** - pobiera lub ustawia katalog dla dynamicznie generowanych plików;
- **LicenseFile** - pobiera lub ustawia lokalizację pliku z licencją skojarzonego z domeną aplikacji;
- **LoaderOptimization** - określa politykę optymalizacji używaną do wczytywania pobiera lub ustawia katalog dla dynamicznie generowanych plików;
- **PrivateBinPath** - pobiera lub ustawia listę katalogów wewnątrz głównego (bazowego) katalogu aplikacji przeszukiwanych w poszukiwaniu prywatnych assembly.

Posiada ona też inne własności (m.in. *PrivateBinPathProbe*, *ShadowCopyDirectories*, *ShadowCopyFiles*), o działaniu podobnym jak odpowiadające im własności klasy *AppDomain*.

Przykłady

Przykład na dowiązywanie wartości domen aplikacji

```
using System;

class Program
{
    public static void Main()
    {
        AppDomain currentDomain = AppDomain.CurrentDomain;

        // Przypisanie nowej wartości dla domeny aplikacji
        currentDomain.SetData("PRZYPISANA_WARTOSC",
            "Przykładowa wartość");

        // Odczytanie wartości przypisanej domenie aplikacji
        Console.WriteLine("PRZYPISANA_WARTOSC = \"{0}\",",
            currentDomain.GetData("PRZYPISANA_WARTOSC"));

        // Próba odczytania wartości nie przypisanej domenie
        Console.WriteLine("PRZYPISANA_WARTOSC_2 = \"{0}\",",
            currentDomain.GetData("PRZYPISANA_WARTOSC_2"));

        // Pobranie wartości dowiązanej do domeny aplikacji
        Console.WriteLine("LOADER_OPTIMIZATION: {0}",
            currentDomain.GetData("LOADER_OPTIMIZATION"));
    }
}
```

Przykład na utworzenie domeny aplikacji oraz modyfikację i odczyt jej własności

```
using System;
using System.Security.Policy; // evidence
using System.Security; // securityzone
using System.Collections; // IEnumarator

class Program
{
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

public static void Main()
{
    AppDomainSetup domaininfo = new AppDomainSetup();
    domaininfo.ApplicationBase = System.Environment.CurrentDirectory;
    domaininfo.ConfigurationFile =
        System.Environment.CurrentDirectory +
        "\\DomenaAplikacji.exe.config";
    domaininfo.ApplicationName = "MyApplication";
    domaininfo.LicenseFile = System.Environment.CurrentDirectory +
        "\\license.txt";

    Evidence adevidence = AppDomain.CurrentDomain.Evidence;
    adevidence.AddHost(new Uri("http://www.example.com"));
    adevidence.AddHost(new Zone(SecurityZone.Internet));

    AppDomain newDomain = AppDomain.CreateDomain("MyDomain", adevidence,
                                                domaininfo);

    Console.WriteLine("Domena hosta: " +
                      AppDomain.CurrentDomain.FriendlyName);
    Console.WriteLine("Domena potomka: " + newDomain.FriendlyName);
    Console.WriteLine();
    Console.WriteLine("ApplicationBase: " +
                      newDomain.SetupInformation.ApplicationBase);
    Console.WriteLine("ConfigurationFile: " +
                      newDomain.SetupInformation.ConfigurationFile);
    Console.WriteLine("ApplicationName: " +
                      newDomain.SetupInformation.ApplicationName);
    Console.WriteLine("LicenseFile: " +
                      newDomain.SetupInformation.LicenseFile);
    IEnumrator newevidenceenum = newDomain.Evidence.GetEnumrator();
    while (newevidenceenum.MoveNext())
        Console.WriteLine(newevidenceenum.Current);

    AppDomain.Unload(newDomain);
}
}
  
```

Przykład na przypisywanie roli użytkownika w domenie aplikacji

```

using System;
using System.Security.Principal;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(new ThreadStart(PrintPrincipalInformation));
        t.Start();
        t.Join();
        AppDomain currentDomain = AppDomain.CurrentDomain;
        currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
        t = new Thread(new ThreadStart(PrintPrincipalInformation));
        t.Start();
        t.Join();
    }
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
    IIdentity identity = new GenericIdentity("InnyUzytkownik");
    IPrincipal principal = new GenericPrincipal(identity, null);
    currentDomain.SetThreadPrincipal(principal);
    t = new Thread(new ThreadStart(PrintPrincipalInformation));
    t.Start();
    t.Join();
}
static void PrintPrincipalInformation()
{
    IPrincipal curPrincipal = Thread.CurrentPrincipal;
    if (curPrincipal != null)
    {
        Console.WriteLine("Typ: {0}", curPrincipal.GetType().Name);
        Console.WriteLine("Nazwa: {0}", curPrincipal.Identity.Name);
        Console.WriteLine("Czy przeszedł autentykację: {0}\n",
                          curPrincipal.Identity.IsAuthenticated);
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

9. Usługi windows-owe (Windows Services)

Ten rozdział poświęcony jest implementacji i zarządzaniu usługami windowsowymi (*Windows Services*).

Podstawowe informacje o usługach windowsowych

Przypomnijmy, jak działają usługi (demony usług) w systemach UNIX-owych. Procesy usług pod systemami UNIX-owymi i UNIX-o podobnymi poza drobnymi różnicami (np. odłączenie od grupy procesów, odłączenie od konsoli, przekierowanie standardowego wejścia i wyjścia itp.) nie różnią się typowych procesów (aplikacji) uruchamianych przez użytkownika lub administratora.

Działanie usług w systemach Windows

W przypadku usług windowsowych (*Windows Services*) wygląda to zupełnie inaczej. Assemblies (pliki binarne) usług pomimo tego samego rozszerzenia nie są typowymi programami uruchomieniowymi. Nie można ich uruchomić jak innych aplikacji (próba takiego uruchomienia zakończy się niepowodzeniem i wywołaniem komunikatu o błędzie). Usługi można uruchamiać wyłącznie przez wbudowane w system operacyjny mechanizmy zarządzania usługami. Wcześniej należy w systemie zarejestrować plik usługi i usługę.

W środowisku Windows XP użytkownik może przeglądać listę usług z poziomu konsoli zarządzania **MMC** (*Microsoft Management Console*). W tym celu należy kliknąć prawym klawiszem myszki na „*Mój komputer*”, wybrać „*Zarządzaj*”, a następnie rozwinąć grupę „*Usługi i aplikacje*” i wybrać podgrupę „*Usługi*”. Pojawi się wówczas lista usług. Można przeglądać ich nazwy, opisy, ustawienia i stany. Można je też uruchamiać i zatrzymywać. (Do wykonywania tych operacji mogą być dodatkowo potrzebne stosowne uprawnienia, ponadto niektóre usługi na poziomie implementacji mają wyłączone niektóre opcje, np. wyłączanie).

Podstawowe elementy implementacji usług

Tworząc implementację własnej usługi należy zadbać o kilka elementów.

Pierwszym i zarazem najistotniejszym elementem takiej implementacji jest klasa usługi. Musi ona dziedziczyć po klasie **ServiceBase**. Instancje klasy usługi muszą ustawać (własnością **ServiceName**) unikalny identyfikator usługi. Warto przy tym zaznaczyć, że unikalne w tej nazwie musi być pierwsze sześć znaków. Nazwy posiadające taki sam sześcioczątkowy początek są ze sobą utożsamiane(!). W usłudze tej należy też zaimplementować wariant metody **OnStart()**, która będzie uruchamiana w momencie startu usługi. Ponadto, o ile usługa ma udostępnić opcję zatrzymania, to należy w niej zaimplementować metodę **OnStop()**, wywoływaną w momencie zatrzymywania usługi. Można też zaimplementować inne metody, które mogą wywoływać się w reakcji na pewne zdarzenia (np. zmiana zasilania, rozpoczęcie zamykania systemu, zmiana stanu sesji). Metody te zostaną wymienione w późniejszej części rozdziału. Metody te muszą działać krótko i w przypadku potrzeby wykonania dłużej działających operacji wywoływać je w sposób asynchroniczny (np. uruchamiając właściwą część usługi w osobnym wątku). Jest to o tyle istotne, że zwykle system operacyjny ogranicza ilość metod obsługi zdarzeń w usługach (z wyłączeniem i wyłączeniem wyłącznie), które mogą być jednocześnie uruchamiane

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

do jednej(!). jeśli wywołania metod *OnStart* lub *OnStop* trwają „długo”, to są w stanie zablokować menadżera usług, a przez to operacje na innych usługach (do czasu przeterminowania, o ile usługa nie będzie zgłaszała zapotrzebowania na dodatkowy czas).

Kolejnym elementem, który powinien się znaleźć w implementacji usługi jest uruchomienie żądanej usługi lub usług. Należy utworzyć obiekt lub obiekty tych usług, a następnie z klasy bazowej (*ServiceBase*) wywołać metodę *Run* (*ServiceBase.Run*) podając jej jako argument referencję do obiektu usługi lub tablicę referencji do obiektów usług.

Implementacja usługi powinna wreszcie zawierać klasę instalatora. Klasa ta musi dziedziczyć po klasie *Installer* i musi być opatrzona atrybutem *RunInstall*. Instancja tej klasy przy inicjalizacji (w konstruktorze) powinna instalować każdą z implementowanych usług (z użyciem obiektu klasy *ServiceInstaller*) oraz sam plik zawierający implementacje usług (z użyciem obiektu klasy *ServiceProcessInstaller*).

Implementacja usługi powinna też zawierać referencje (używane przez instalatory).

Elementarny przykład kompletnej implementacji usługi

W niniejszym podrozdziale przedstawiony został kompletny przykład prostej implementacji usługi. Poniższe pliki wystarczy wprowadzić do projektu generycznego (pustego) typu.

Klasa usługi

```
using System.ServiceProcess;

namespace MojaUsluga
{
    public class MojaUsluga : ServiceBase
    {
        public const string NazwaUslugi = "Mój Pierwszy Serwis";
        public MojaUsluga()
        {
            this.ServiceName = NazwaUslugi;
        }
        protected override void OnStart(string[] args)
        {
            // ...
        }
        protected override void OnStop()
        {
            // ...
        }
    }
}
```

Uruchamianie usługi

```
using System.ServiceProcess;

namespace MojaUsluga
{
    static class Program
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
{
    static void Main()
    {
        //ServiceBase[] ServicesToRun;
        //ServicesToRun = new ServiceBase[]
        //{
        //    new MojaUsluga()
        //};
        //ServiceBase.Run(ServicesToRun);
        ServiceBase.Run(new MojaUsluga());
    }
}
```

Referencje

- **System.ServiceProcess;**
- **System.Configuration.Install.**

Instalator

```
using System.ComponentModel;
using System.Configuration.Install;
using System.ServiceProcess;

namespace MojaUsluga
{
    [RunInstaller(true)]
    public class ProjectInstaller : Installer
    {
        public ProjectInstaller()
        {
            ServiceProcessInstaller serviceProcessInstaller =
                new ServiceProcessInstaller();
            ServiceInstaller serviceInstaller = new ServiceInstaller();
            //serviceProcessInstaller.Account = ServiceAccount.LocalService;
            //serviceProcessInstaller.Account = ServiceAccount.User;
            serviceProcessInstaller.Account = ServiceAccount.LocalSystem;
            serviceInstaller.DisplayName = MojaUsluga.NazwaUslugi;
            serviceInstaller.StartType = ServiceStartMode.Manual;
            serviceInstaller.ServiceName = MojaUsluga.NazwaUslugi;
            this.Installers.Add(serviceProcessInstaller);
            this.Installers.Add(serviceInstaller);
        }
    }
}
```

Rejestrowanie usługi w systemie

Instalację i deinstalację usługi z linii poleceń (z „Visual Studio Tools” → „Visual Studio 20XX Command Prompt”) można przeprowadzić poleceniem **installutil.exe**.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Tworzenie usług ze wzorca (w Visual Studio)

Implementacja usługi może też być przeprowadzona z domyślnego wzorca w środowisku Visual Studio. Wykonać wówczas należy następujące kroki:

- Tworzymy nowy projekt typu *Windows Service*.
- Ustawiamy własności *DisplayedName* i *ServiceName* dla serwisu *Service1* (nazwa usługi).
- Ustawiamy inne potrzebne własności, takie jak *AutoLog* i *CanPauseAndContinue* dla serwisu.
- Zmieniamy własność *Startup* na własność utworzoną w poprzednim kroku.
- Implementujemy metodę *OnStart* zawierającą kod wykonywany w momencie uruchamiania usługi.
- Implementujemy metodę *OnStop* zawierającą kod wykonywany w momencie kończenia pracy usługi.
- Dodajemy instalator i instalowanie w nim usług(i) przez *ServiceInstaller* oraz pliku z usługą/ami przez *ServiceProcessInstaller*.

Opis klas używanych w implementacji usługi

Klasa *ServiceBase*

W tym miejscu podane zostaną wybrane składowe klasy *ServiceBase*.

Klasa ta posiada statyczne stałe pole ***MaxNameLength*** określające maksymalną długość nazwy dla usługi. Posiada ona również następujące własności:

- ***AutoLog*** - określa czy raportować wywołania komend *Start*, *Stop*, *Pause* i *Continue* do dziennika zdarzeń;
- ***CanHandlePowerEvent*** - pozwala sprawdzić lub ustawić czy usługa ma obsługiwać notyfikacje dotyczące zmian w stanie zasilania;
- ***CanHandleSessionChangeEvent*** - pozwala sprawdzić lub ustawić wartość określającą czy usługa ma obsługiwać notyfikacje dotyczące zmian w stanie sesji (wysyłane przez *Terminal Server*);
- ***CanPauseAndContinue*** - pozwala sprawdzić lub ustawić; wartość określającą czy usługa może zostać zatrzymana i wznowiona;
- ***CanRaiseEvents*** - pobiera wartość określającą czy usługa może wywoływać zdarzenia (dziedziczone z klasy *Component*);
- ***CanShutdown*** - pozwala sprawdzić lub ustawić wartość określającą czy usługa powinna otrzymywać notyfikacje o zatrzymywaniu systemu;
- ***CanStop*** - pozwala sprawdzić lub ustawić wartość określającą czy usługa może zostać zakończona (jeśli zostanie uruchomiona);
- ***Container*** - zwraca kontener zawierający komponent (dziedziczone z klasy *Component*);
- ***DesignMode*** - zwraca wartość określającą czy komponent znajduje się aktualnie w trybie projektowym (ang. *design mode*) (dziedziczone z klasy *Component*);
- ***EventLog*** - zwraca obiekt opisujący log zdarzeń do którego zapisywane są informacje o wykonywanych poleceniach (takich jak *Start* i *Stop*);
- ***Events*** - pobiera tablicę zdarzeń dowiązanych do komponentu;
- ***ExitCode*** - pobiera lub ustawia kod zakończenia usługi;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **ServiceHandle** - zwraca uchwyt kontrolny usługi (*Service Control Manager* używa go do komunikacji z usługą, np. do aktualniania statusu usługi poprzez wywołania niezarządzalnej funkcji *SetServiceStatus*);
- **ServiceName** - pobiera lub ustawia skróconą nazwę usługi służącą do identyfikacji usługi w systemie;
- **Site** - pobiera lub ustawia lokalizację (*ISite*) komponentu.

Klasa *ServiceBase* posiada też następujące metody:

- **GetService** - zwraca obiekt reprezentujący usługę dostarczony przez komponent lub kontener;
- **OnContinue** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie wysłania przez **SCM** (*Service Control Manager*) żądania wznowienia usługi;
- **OnCustomCommand** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie wysłania przez **SCM** (*Service Control Manager*) żądania wykonania zdefiniowanego przez użytkownika polecenia (*custom command*) (identyfikator polecenia przekazywany jest tej metodzie jako argument);
- **OnPause** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie wysłania przez **SCM** (*Service Control Manager*) żądania zatrzymania (bez jej kończenia) usługi;
- **OnPowerEvent** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie wykrycia zmian w stanie zasilania (np. przełączania się na zasilanie baterijne w laptopie);
- **OnSessionChange** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie zmiany stanu sesji (np. *zalogowanie*, *wylogowanie*, *zablokowanie sesji*) wysłanym przez *Terminal Server*;
- **OnShutdown** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie zamykania systemu; powinna implementować akcje które należy wykonać bezpośrednio przed zamknięciem systemu (np. zapisanie plików);
- **OnStart** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie ręcznego uruchomienia usługi przez **SCM** (*Service Control Manager*) lub startu systemu w przypadku automatycznego uruchamiania;
- **OnStop** - metoda wirtualna, która po zaimplementowaniu będzie wywoływana w momencie wysłania przez **SCM** (*Service Control Manager*) żądania końca działania usługi;
- **RequestAdditionalTime** - metoda żądająca przydzielenia dodatkowego czasu działania (podawanego w milisekundach); zwykle wywoływana jest w implementacjach metod *OnContinue*, *OnPause*, *OnStart* oraz *OnStop* zanim **SCM** (*Service Control Manager*) oznaczy usługę jako „nie dającą odpowiedzi”;
- **Run** - uruchamia podaną usługę lub zbiór usług (w szczególności rejestruje) pod **SCM** (*Service Control Manager*); jest to metoda statyczna;
- **ServiceMainCallback** - rejestruje uchwyt wywołania i uruchamia usługę (do użytku raczej przez Framework, nie przez programistę);
- **Stop** - zatrzymuje działanie usługi.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Klasa *ServiceInstaller*

Klasa **ServiceInstaller** jest klasą pochodną klasy *ComponentInstaller*. Służy ona do instalacji klasy implementującej usługę (klasy rozszerzającą klasę *ServiceBase*). W szczególności zapisuje ona dane do rejestrów w obrębie ścieżki HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services. Klasa ta jest używana przez narzędzie do instalacji programu usługi.

Klasa *ServiceInstaller* posiada następujące własności:

- **Context** - pobiera lub ustawia informacje o instalacji (m.in. położenie pliku logów, pliku z informacjami potrzebnymi dla metody *Uninstall*, informacje o sposobie wywołania instalatora) w obiekcie klasy *InstallContext* przekazywanym metodom *Install*, *Commit*, *Rollback* i *Uninstall*;
- **Description** - pobiera lub ustawia opis usługi;
- **DisplayName** - pobiera lub ustawia nazwę identyfikującą usługę (przez użytkownika);
- **HelpText** - pobiera informacje o sposobie używania instalatora (np. znaczenie argumentów i opcji instalatora);
- **Installers** - zwraca kolekcję instalatorów zawartego w instalatorze;
- **Parent** - pobiera lub ustawia instancję klasy *Installer* zawierającą kolekcję do której należy instalator (lub **null** jeśli nie należy do kolekcji lub jest rodzicem);
- **ServiceName** - pobiera lub ustawia nazwę identyfikującą usługę (przez system);
- **ServiceDependedOn** - pobiera lub ustawia tablicę usług, które muszą działać, aby można było uruchomić usługę;
- **StartType** - pobiera lub ustawia sposób uruchamiania usługi (*Manual*, *Automatic*, *Disabled*).

Klasa *ServiceInstaller* posiada też następujące metody:

- **Commit** - kończy transakcję instalacji, jest wywoływana wyłącznie gdy metody *Install* wszystkich instalatorów kolekcji *InstallerCollection* zwróconych przez własność *Installers*;
- **Install** - instaluje usługę zapisując odpowiednie informacje do rejestrów (jest przeznaczona do automatycznego przetwarzania);
- **Rollback** - usuwa informacje zapisane do rejestrów (przywraca do poprzedniego stanu) (jest przeznaczona do automatycznego przetwarzania);
- **Uninstall** - odinstalowuje usługę usuwając informacje o usłudze zapisane do rejestrów; wcześniej stara się zatrzymać usługę, jeśli jest uruchomiona.

Powysze metody warto zapamiętać, gdyż będą się pojawiały przy innych klasach instalatorów.

Klasa *ServiceProcessInstaller*

Klasa **ServiceProcessInstaller** jest klasą pochodną klasy *ComponentInstaller*. Służy ona do instalacji pliku zawierającego klasę implementującą usługę (klasy rozszerzającą klasę *ServiceBase*). Klasa ta jest używana przez narzędzie do instalacji programu usługi.

Wiele elementów klasy *ServiceProcessInstaller* jest odpowiednikami elementów klasy *ServiceInstaller*, które działają w podony sposób. W szczególności dotyczy to metod *Install*, *Commit*, *Rollback* i *Uninstall*.

Klasa *ServiceProcessInstaller* dostarcza mechanizmów pozwalających kontrolować sposób uruchamiania usług. Posiada ona własność **Account** opisującą kontekst bezpieczeństwa, w którym

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

uruchamiana jest usługa. Za pomocą tej własności można pobrać lub przypisać wartość typu wyliczeniowego **ServiceAccount**, który może przyjmować następujące wartości:

- *LocalService* - na lokalnej maszynie uprawnienia zwykłego użytkownika, przy zdalnych połączeniach przedstawia się jako użytkownik anonimowy;
- *LocalSystem* - na lokalnej maszynie uprawnienia zaawansowanego użytkownika, przy zdalnych połączeniach przedstawia się jako komputer;
- *NetworkService* - rozszerzone uprawnienia lokalne, przy zdalnych połączeniach uwierzytelnia się jako komputer;
- *User* – uwierzytelnianie jako zadany użytkownik.

W przypadku, gdy własność *ServiceAccount* przyjmuje wartość *User*, do autentykacji użytkownika, z którego uprawnieniami ma być uruchamiana usługa, można użyć własności *Username* i *Password*.

Zarządzanie usługami przez klasę **ServiceController**

Do zarządzania usługami w systemie służy klasa *ServiceController*. Za jej pomocą można w szczególności pobierać listę usług z systemu, sprawdzać stan konkretnych usług lub wpływać na jego zmianę (np. uruchamiając lub zatrzymując usługę). Ten podrozdział poświęcony jest zarządzaniu zainstalowanymi usługami.

Wybrane własności klasy **ServiceController**

Klasa *ServiceController* posiada szereg własności, wśród których możemy wyróżnić:

- **CanPauseAndContinue** - pozwala sprawdzić, czy usługa może zostać zatrzymana i wznowiona (przywrócona);
- **CanShutdown** - informuje o tym, czy usługa powinna dostawać notyfikację o zamknięciu systemu;
- **CanStop** - pozwala sprawdzić, czy usługa może być zatrzymana (po uruchomieniu);
- **DependantServices** - pobiera lub ustawia zbiór usług zależnych od usługi powiązanej z bieżącą instancją klasy *ServiceController*;
- **DisplayName** - pobiera lub ustawia „zaprzyjaźnioną” nazwę usługi (identyfikowaną przez użytkownika, która jest wyświetlana na liście usług);
- **MachineName** - pobiera lub ustawia nazwę komputera, na którym rezyduje usługa;
- **ServiceName** - pobiera lub ustawia nazwę usługi wskazywanej przez tę instancję.

Typy usług w systemie

Własność **ServiceType** klasy *ServiceController* informuje o roli usługi w systemie.

Może ona przyjmować wartości typu wyliczeniowego **ServiceType** związane z następującymi polami bitowymi:

- **Adapter** - usługa dla urządzenia sprzętowego wymagającego własnego sterownika;
- **FileSystemDriver** - sterownik systemu plików, który pełni też rolę sterownika jądra systemu;
- **InteractiveProcess** - usługa mogąca się komunikować z pulpitem;
- **KernelDriver** - niskopoziomowy sterownik sprzętowy jądra;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **RecognizedDriver** - sterownik systemu plików używany podczas startu w celu rozpoznania systemów plików w systemie
- **Win32OwnProcess** - usługa Win32 (COM) zgodna z „*service control protocol*”, uruchamiana (wyłącznie) jako osobny proces;
- **Win32ShareProcess** - usługa Win32 (COM), która może współdzielić proces z inną usługą *Win32*.

Statusy (stany) usług

Właściwość **Status** klasy *ServiceController* pozwala na sprawdzenie stanu usługi.

Może ona przyjmować następujące wartości typu wyliczeniowego **ServiceControllerStatus**:

- **ContinuePending** - stan „pośredni” przy wznowieniu działania (metodą *Continue*) przy przełączaniu ze stanu *Paused* do *Running*; jest to odpowiednik stałej **SERVICE_CONTINUE_PENDING** z Win32 zdefiniowanej jako **0x00000005**;
- **Paused** - informuje o wstrzymaniu usługi; jest to odpowiednik stałej **SERVICE_PAUSED** z Win32 zdefiniowanej jako **0x00000007**;
- **PausePending** - stan „pośredni” przed przejęciem do stanu *Pause*; jest to odpowiednik stałej **SERVICE_PAUSE_PENDING** z Win32 zdefiniowanej jako **0x00000006**;
- **Running** - informuje o działaniu usługi; jest to odpowiednik stałej **SERVICE_RUNNING** z Win32 zdefiniowanej jako **0x00000004**;
- **StartPending** - informuje o uruchamianiu usługi; jest to odpowiednik stałej **SERVICE_START_PENDING** z Win32 zdefiniowanej jako **0x00000002**;
- **Stopped** - informuje o tym, że usługa nie działa; jest to odpowiednik stałej **SERVICE_STOPPED** z Win32 zdefiniowanej jako **0x00000001**;
- **StopPending** - informuje o tym, że usługa jest zatrzymywana; jest to odpowiednik stałej **SERVICE_STOP_PENDING** z Win32 zdefiniowanej jako **0x00000003**;

Metody klasy *ServiceController*

Klasa *ServiceController* posiada także wiele metod, wśród których znajdują się:

- **Close** - odłącza instancję klasy *ServiceController* od usługi i zwalnia zasoby zaalokowane dla tej instancji;
- **Continue** - wznowia wstrzymaną usługę (jeśli właściwość *CanPauseAndContinue* przyjmuje wartość **false**, działanie usługi nie może zostać wznowione);
- **ExecuteCommand** - wykonuje definiowaną komendę usługi;
- **GetDevices** - pobiera tablicę usług pełniących rolę sterowników sprzętowych;
- **GetServices** - pobiera tablicę usług będących sterownikami, ale nie sterownikami sprzętowymi oraz usług nie będących sterownikami;
- **Pause** - metoda służąca do wstrzymania działania usługi;
- **Refresh** - odświeża wartości wszystkich właściwości, przypisując im aktualne wartości;
- **Start** - metoda służąca do uruchomienia usługi (jeśli właściwość *CanPauseAndContinue* przyjmuje wartość **false** usługa nie może zostać uruchomiona);
- **Stop** - metoda służąca do zatrzymania usługi;

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **WaitForStatus** - wstrzymuje działanie aplikacji do przyjęcia przez usługę wskazanego statusu (typu *ServiceControllerStatus*).

Przykład użycia klasy *ServiceController*

Poniższy przykład, będący drobną modyfikacją zamieszczonego w dokumentacji przykładu, pokazuje, w jaki sposób można pobrać informacje o sterownikach sprzętowych w systemie.:

```

using System;
using System.ServiceProcess; // + referencja do "System.ServiceProcess"

class Program
{
    static void Main(string[] args)
    {
        ServiceController[] scDevices;
        scDevices = ServiceController.GetDevices();
        int numFileSystem = 0, numKernel = 0
        /*, numAdapter = 0, numRecognizer = 0 */;

        Console.WriteLine("Device driver services on the local computer:");

        foreach (ServiceController scTemp in scDevices)
        {
            Console.WriteLine(" [{0}] {1}", scTemp.Status, scTemp.DisplayName);
            Console.WriteLine("           Type = {0}", scTemp.ServiceType);

            if ((scTemp.ServiceType & ServiceType.FileSystemDriver) != 0)
            {
                numFileSystem++;
            }
            if ((scTemp.ServiceType & ServiceType.KernelDriver) != 0)
            {
                numKernel++;
            }
        }

        Console.WriteLine("\nTotal of {0} device driver services",
                        scDevices.Length);
        Console.WriteLine("  {0} are file system drivers", numFileSystem);
        Console.WriteLine("  {0} are kernel drivers", numKernel);
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

10. Konfiguracja

Środowisko .NET posiada mechanizmy wspierające używanie XML-owych plików konfiguracyjnych przechowujących konfigurację. Dla aplikacji domyślna nazwa pliku konfiguracyjnego jest tworzona z nazwy pliku wykonywalnego aplikacji przez dodanie przyrostka „.config”, przy czym zachowanie poprzedniego przyrostka jest opcjonalne. Przykładowo plikiem konfiguracyjnym dla programu „**Aplikacja.exe**” może być plik „**Aplikacja.exe.config**” lub „**Aplikacja.config**”.

W tym rozdziale przedstawione zostały wybrane elementy plików konfiguracyjnych oraz przykłady ich używania. Niektóre inne elementy plików konfiguracyjnych pojawią się w dalszych rozdziałach tych materiałów (np. przełączniki śledzenia pojawią się w rozdziale poświęconym śledzeniu działania programów).

Używane klasy

Na początku przedstawione zostaną klasy używane przy zarządzaniu konfiguracją aplikacji. Zasadniczo klasy te zlokalizowane są w przestrzeni *System.Configuration*²⁸.

Najważniejsze klasy konfiguracji

Najważniejszymi klasami służącymi do zarządzania konfiguracją są klasy *Configuration* i *ConfigurationManager*. Obiekty klasy **Configuration** opisują konkretne zestawy ustawień oraz pozwalają na wykonywanie operacji na nich (np. odczyt pewnych elementów lub zapis ustawień do pliku). Natomiast klasa **ConfigurationManager** jest klasą statyczną pozwalającą na wykonywanie operacji na istniejącym pliku konfiguracyjnym takich jak jego odczytanie i utworzenie obiektu klasy *Configuration* lub odczyt konkretnego elementu pliku konfiguracyjnego (ale nie udostępnia możliwości zapisania pliku konfiguracyjnego ani zmiany jego zawartości).

Klasy reprezentujące strukturę pliku konfiguracyjnego

Do reprezentacji najbardziej ogólnego elementu pliku konfiguracyjnego służy klasa abstrakcyjna *ConfigurationElement*. W szczególności z tej klasy dziedziczy klasa abstrakcyjna *ConfigurationSection* reprezentująca sekcję w pliku konfiguracyjnym oraz klasa abstrakcyjna *ConfigurationElementCollection* reprezentująca kolekcję (zagnieżdżonych potomnych) elementów pliku konfiguracyjnego.

Elementy plików konfiguracyjnych są opisywane przez klasy pochodne powyżej wymienionych klas. Przykładami takich klas są klasy **AppSettingsSection** reprezentująca sekcję ustawień aplikacji oraz klasa **ConnectionStringSection** reprezentująca sekcję zawierającą ścieżki połączeń do baz danych (wraz z parametrami tych połączeń).

Ponadto do reprezentowania powiązanych sekcji pliku konfiguracyjnego służy klasa *ConfigurationSectionGroup*, a zapieczętowane klasy *ConfigurationSectionCollection*

²⁸ Należy pamiętać, że istnieją też inne przestrzenie zawierające klasy pozwalające na zarządzanie konfiguracją lub jej elementami. Przykładowo w przestrzeni *System.Web.Configuration* zlokalizowana jest klasa służąca do odczytania serwisu sieciowego, a w przestrzeni *System.Diagnostic*, klasy *BooleanSwitch* i *TraceSwitch* służące do zarządzania przełącznikami śledzenia.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

ConfigurationSectionGroupCollection służą do reprezentowania kolekcji odpowiednio sekcji i grup sekcji pliku konfiguracyjnego i są używane do iterowania po tych kolekcjach.

Typy wyliczeniowe

Mamy dwa typy wyliczeniowe związane z plikiem konfiguracyjnym

- typ **public enum ConfigurationUserLevel** określa zasięg użytkowników, których dotyczy konfiguracja i posiada etykiety:
 - **None** (dotyczy wszystkich),
 - **PerUserRoaming**,
 - **PerUserRoamingAndLocal**;
- typ **public enum ConfigurationSaveMode** określa zakres zapisywanych zmian do pliku konfiguracyjnego i posiada etykiety:
 - **Modified**,
 - **Minimal**,
 - **Full**.

Składowe plików konfiguracyjnych

Konfiguracja jest przechowywana w plikach XML-owych. Właściwe ustawienia są w nich umieszczane wewnątrz znaczników `<configuration></configuration>`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
</configuration>
```

Wspólne ustawienia

Niektóre z ustawień zawartych w pliku konfiguracyjnym dotyczą nie tyle samego działania aplikacji, ale działania środowiska uruchomieniowego CLR. W tej części przytoczone zostały przykłady takich ustawień pobrane z opracowania [15].

Przykładowo w sekcji **startup** umieszczone są ustawienia, które powinny być zastosowane jeszcze przed uruchomieniem aplikacji. Przykładem takich ustawień jest numer wspieranej wersja .NET Frameworka. Przykład podania tej wersji został przedstawiony poniżej:

```
<?xml version = "1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v2.0.50727" />
  </startup>
</configuration>
```

Działanie tej opcji jest następujące:

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Założymy, że aplikacja została skompilowana z wersją .NET Framework A, a w pliku konfiguracyjnym został podany numer wspieranej wersji .NET Framework B. Wówczas wersja z którą jest uruchamiana aplikacja jest ustalana według następujących reguł:

- jeżeli na maszynie jest zainstalowana wersja framework A, to aplikacja jest z nią uruchamiana;
- w przeciwnym wypadku aplikacja jest uruchamiana z wersją B, o ile jest ona zainstalowana na maszynie;
- o ile na maszynie nie ma zainstalowanej ani wersji A ani wersji B, to jest ona uruchamiana z najwyższym numerem wersji .NET Framework zainstalowanej na maszynie.

Innym przykładem sekcji nie związanej bezpośrednio z ustawieniami aplikacji jest sekcja **runtime**, w której umieszczane są ustawienia dotyczące sposobu działania aplikacji. Jednym z takich ustawień jest przełącznik **developmentMode** pozwalający na przełączenie aplikacji w tryb „developerski”, w którym używać można nie tylko bibliotek zarejestrowanych w **GAC** (ang. *Global Assembly Cache*), ale również (niezarejestrowanych) bibliotek umieszczonych w katalogu wskazywanym przez zmienną systemową **DEVPATH**. Przykład użycia wspomnianego przełącznika został podany poniżej:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <developmentMode developerInstallation="true" />
  </runtime>
</configuration>
```

W sekcji *runtime* można również podawać dowiązania do assemblacji (zwykle bibliotek DLL, ale mogą to też być np. pliki EXE) niezbędnych do poprawnego działania aplikacji, np.:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="schemaname">
      <dependentAssembly>
        <assemblyIdentity name="myprogram"
          publicKeyToken="xxxxxxxxxx" culture="en-us" />
        <codeBase version="x.0.0.0"
          href="http://www.moja.strona.pl/myprogram.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Pewne ustawienia pliku konfiguracyjnego pozwalają na manipulację sposobem działania środowiska CLR. Przykładowo poniższy plik konfiguracyjny wyłącza tryb współbieżny *Garbage Collectora*

```
<configuration>
  <runtime>
    <gcConcurrent enabled="false" />
  </runtime>
</configuration>
```

(co swoją drogą zwykle nie jest zbyt dobrym pomysłem).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W sekcji **system.runtime.remoting** ustawiane są parametry dotyczące połączeń (np. przez WCF) w modelu klient-serwer. Zwykle te części konfiguracji dla serwera oraz klienta zawierają nieco inną składnię, gdyż w przypadku serwera należy podać sposób uruchomienia usługi, np.:

```
<system.runtime.remoting>
  <application name = "MyApplication">
    <service>
      <wellknown type="FullyQualifiedNamespace,AssemblyName"
        mode="Singleton" objectUri="MyClass.rem"/>
    </service>
  </application>
</system.runtime.remoting>
```

a w przypadku klienta należy wskazać parametry do połączenia z usługą, np.:

```
<system.runtime.remoting>
  <application name = "MyClientApplication">
    <service>
      <wellknown type="FullyQualifiedNamespace, AssemblyName"
        url="http://localhost:5000/MyClass.rem"/>
    </service>
  </application>
</system.runtime.remoting>
```

Ustawienia aplikacji

W tym rozdziale przedstawione zostaną dwa rodzaje ustawień związanych z działaniem aplikacji: będą to *ustawienia aplikacji* podawane w sekcji **appSettings** oraz ścieżki połączeń do baz danych ustawiane w sekcji **connectionStrings**, których przykłady użycia zostały podane poniżej

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Zmienna" value="Jakaś wartość"/>
  </appSettings>
  <connectionStrings>
    <clear/>
    <add name="SciezkaDoBazy"
      providerName="System.Data.SqlClient"
      connectionString="Data Source=localhost;
        Initial Catalog=MojaBaza;
        Integrated Security=true"/>
  </connectionStrings>
</configuration>
```

oraz definiowalne przez użytkownika sekcje i grupy sekcji.

W rozdziale 15 zostaną później jeszcze podane ustawienia związane ze śledzeniem i debugowaniem aplikacji, które można umieszczać w sekcji **system.diagnostic**.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Wczytywanie elementów konfiguracji

W niniejszym podrozdziale przedstawione zostaną sposoby odczytu wybranych ustawień z plików konfiguracyjnych (oraz zapisu plików konfiguracyjnych). Przykładowe kody źródłowe używają klas zdefiniowanych w między innymi w przestrzeni *System.Configuration*, *System.Collection.Specialized* (w przypadku kolekcji związanych z sekcjami), *System.Collection* oraz *System.Data* i wymagają dodania referencji do biblioteki *System.Configuration*.

Wczytywanie ustawień aplikacji

Załóżmy, że aplikacja ma plik konfiguracyjny taki, jak w poprzednim przykładzie:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Zmienna" value="Jakaś wartość"/>
  </appSettings>
  <connectionStrings>
    <clear/>
    <add name="SciezkaDoBazy"
      providerName="System.Data.SqlClient"
      connectionString="Data Source=localhost;
        Initial Catalog=MojaBaza;
        Integrated Security=true"/>
  </connectionStrings>
</configuration>
```

W pierwszych wersjach .NET Frameworka można było odczytać wartości poprzez klasę **ConfigurationSettings**, np.:

```
String OdczytanaWartosc =
  ConfigurationSettings.AppSettings["Zmienna"];
```

Obecnie to rozwiązanie jest uznawane za przestarzałe oraz zdecydowanie odradzane. Zamiast tego można pobrać sekcję ustawień aplikacji poprzez właściwość **AppSettings** klasy **ConfigurationManager**, np.:

```
NameValueCollection AllAppSettings =
  ConfigurationManager.AppSettings;
```

Można też użyć jednej z metod *OpenExeConfiguration()* *OpenMachineConfiguration()*, *OpenMappedExeConfiguration()* *OpenMappedMachineConfiguration()* klasy **ConfigurationManager** do pobrania obiektu klasy **Configuration** opisującego (całą) konfigurację i z niego wydobyć właściwością **AppSettings** sekcję ustawień aplikacji. Następnie można odwołać się do ustawienia konkretnego parametru albo po jego nazwie albo po jego numerze, np.:

```
Console.WriteLine(AllAppSettings["Zmienna"]);
Console.WriteLine(AllAppSettings[0]);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Poniższy kod pokazuje w jaki sposób można odczytać wszystkie ustawienia aplikacji

```
NameValueCollection AllAppSettings = ConfigurationManager.AppSettings;
Int32 Counter = 0;
IEnumerator SettingsEnumerator = AllAppSettings.Keys.GetEnumerator();
while (SettingsEnumerator.MoveNext())
{
    Console.WriteLine("Item: {0}; Value: {1}",
        AllAppSettings.Keys[Counter],
        AllAppSettings[Counter]);
    Counter++;
}
foreach (string key in AllAppSettings.Keys)
{
    Console.WriteLine("Item: {0}; Value: {1}",
        key, AllAppSettings[key]);
}
```

Wczytywanie ścieżek połączeń do bazy

W podobny do powyższego sposób można wczytywać ścieżki połączeń do bazy danych.

Rozważmy następujący przykład zawierający bardziej rozbudowaną sekcję ze ścieżkami połączeń do bazy, w której znajdują się połączenia do różnych rodzajów baz.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <connectionStrings>
        <clear/>
        <add name="SciezkaDoBazy"
            providerName="System.Data.SqlClient"
            connectionString="Data Source=localhost;
Initial Catalog=MojaBaza;
Integrated Security=true"/>
        <add name="SqlServer2005ConnectionString"
            providerName="System.Data.SqlClient"
            connectionString=
                "Server=MyServer1;Database=
                pubs;Trusted_Connection=True;MultipleActiveResultSets=true" />
        <add name="OdbcConnectionString"
            providerName="System.Data.Odbc"
            connectionString=
                "Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\adatabase.mdb;
                Uid=Admin;Pwd=Pi3rwszeChasl0DoBa2yDanyh;" />
        <add name="AccessConnectionString"
            providerName="System.Data.OleDb"
            connectionString="Provider=Microsoft.Jet.OLEDB.4.0;
                Data Source=\PathOrShare\mydb.mdb;
                User Id=admin;Password=Drug13Hasl0DoB4zyDanych;" />
        <add name="OracleConnectionString"
            providerName="System.Data.OracleClient"
            connectionString="Data Source=MyOracleDB;Integrated Security=yes;" />
    </connectionStrings>
</configuration>
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Można z niego odczytać ścieżki w poniższy sposób

```
ConnectionStringSettingsCollection MySettings =
    ConfigurationManager.ConnectionStrings;

if (MySettings != null)
{
    StringBuilder sb = new StringBuilder();
    foreach (ConnectionStringSettings
        individualSettings in MySettings)
    {
        sb.Append("Full Connection String: " +
            individualSettings.ConnectionString + "\r\n");
        sb.Append("Provider Name: " +
            individualSettings.ProviderName + "\r\n");
        sb.Append("Section Name: " +
            individualSettings.Name + "\r\n");
    }
    Console.WriteLine(sb.ToString());
}
```

Następnie w zależności od wartości ustawionej w polu dostawcy można nawiązać połączenia różnego rodzaju, np.:

```
IDbConnection MyConnection = null;
switch (individualSettings.ProviderName)
{
    case "System.Data.SqlClient":
        MyConnection = new SqlConnection(
            individualSettings.ConnectionString);
        break;
    case "System.Data.OracleClient":
        MyConnection = new OracleConnection(
            individualSettings.ConnectionString);
        break;
    case "System.Data.OleDb":
        MyConnection = new OleDbConnection(
            individualSettings.ConnectionString);
        break;
    case "System.Data.Odbc":
        MyConnection = new OdbcConnection(
            individualSettings.ConnectionString);
        break;
}
```

Wczytywanie konkretnych sekcji i kolekcji grup

Rozważmy następujący plik konfiguracyjny:

```
<configuration>
    <configSections>
        <sectionGroup name="MyFirstSectionGroup">
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

<section name="MyFirstSection"
  type="ConnectionStringDemo.MyFirstSectionHandler, ConnectionStringDemo"/>
</sectionGroup>
</configSections>
<MyFirstSectionGroup>
  <MyFirstSection>
    <Value>
      <Identifier>111</Identifier>
      <SettingValue>System.Data.SqlClient</SettingValue>
    </Value>
    <Value>
      <Identifier>112</Identifier>
      <SettingValue>System.Data.OleDb</SettingValue>
    </Value>
    <Value>
      <Identifier>113</Identifier>
      <SettingValue>System.Data.Odbc</SettingValue>
    </Value>
  </MyFirstSection>
</MyFirstSectionGroup>
</configuration>
  
```

Można z niego wczytać konkretną sekcję w następujący sposób

```
ConfigurationManager.GetSection("MyFirstSectionGroup/MyFirstSection");
```

oraz grupę sekcji w poniższy sposób:

```

Configuration config =
  ConfigurationManager.OpenExeConfiguration(
    ConfigurationUserLevel.None);

ConfigurationSectionGroupCollection
  DemoGroups = config.SectionGroups;

foreach (String groupName in DemoGroups.Keys)
{
  Console.WriteLine(groupName);
}
  
```

Zapis konfiguracji

Poniższy przykład pokazuje w jaki sposób można do istniejącej instancji pliku konfiguracyjnego dodać nową sekcję i dokonać zapisu tego pliku:

```

ConfigurationSection customSection;
// Pobranie aktualnej zawartości pliku konfig.
System.Configuration.Configuration config =
  ConfigurationManager.OpenExeConfiguration(
    ConfigurationUserLevel.None);

if (config.Sections["CustomSection"] == null)
{
  
```



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
customSection = new CustomSection();
config.Sections.Add("CustomSection",
    customSection);
customSection.SectionInformation.ForceSave =
    true;
config.Save(ConfigurationSaveMode.Full);
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

11. Instalatory

Zagadnienie instalatorów i instalacji pojawiło się już przy omawianiu usług (i ich instalacji).

W tym rozdziale zostanie ono nieco rozbudowane.

Mechanizmy używane przy instalacji

W systemach Windows mamy szereg mechanizmów pozwalających na instalację programów.

Podstawowym mechanizmem do przeprowadzania operacji związanych z instalacją programów jest usługa **Microsoft Windows Installer**, będąca częścią systemów Windows od wersji 2000, Me i XP wzwyż.

Użytkownik może ręcznie zainstalować plik (moduł, bibliotekę czy usługę) narzędziem **InstallUtil.exe** uruchamianym z linii poleceń, które było wspomniane w temacie poświęconym usługom. Korzystanie z tego narzędzia powinno zostać wcześniej przećwiczone w ramach laboratorium.

W Visual Studio w grupie „*Other Project Type*” mamy podgrupę „*Setup and Deployment*” zawierającą szereg typów projektów instalatorów. Tworzenie projektów z tej podgrupy (przynajmniej „*Setup Project*” oraz ewentualnie „*Setup Wizard*”) pozostawiamy jako zadanie na laboratorium.

Programista może zaimplementować w kodzie źródłowym własny instalator.

Wreszcie języku C# zawiera klasy pozwalające na instalację plików (np. bibliotek) lub modułów bezpośrednio w kodzie.

W tym rozdziale skupimy się na dwóch ostatnich rozwiązańach (wcześniej dwa powinny zostać omówione i przećwiczone na laboratorium).

Implementacja własnego instalatora

Programista może stworzyć implementację własnego instalatora. W tym celu należy utworzyć klasę dziedziczącą po klasie *Installer*, odpowiednio nadpisać metody dziedziczone z klasą *Installer* i nadać nowo utworzonej klasie atrybut *System.ComponentModel.RunInstaller*.

W jednym ze wcześniejszych rozdziałów zostały przedstawione klasy *ServiceInstaller* oraz *ServiceProcessInstaller* dziedziczące (pośrednio – przez klasę *ComponentInstaller*) po klasie *Installer* i ich składowe. Przedstawione przy tym zostały metody

- **Commit** - kończy transakcję instalacji, jest wywoływana wyłącznie gdy metody *Install* wszystkich instalatorów kolekcji *InstallerCollection* zwróconych przez własność *Installers* zakończą się prawidłowo;
- **Install** - instaluje usługę zapisując odpowiednie informacje do rejestrów (jest przeznaczona do automatycznego przetwarzania);
- **Rollback** - usuwa informacje zapisane do rejestrów (przywraca do poprzedniego stanu) (jest przeznaczona do automatycznego przetwarzania);
- **Uninstall** - odinstalowuje usługę usuwając informacje o usłudze zapisane do rejestrów; wcześniej stara się zatrzymać usługę, jeśli jest uruchomiona dziedziczone z klasą *Installer* i nadpisywane w klasach pochodnych. Klasa *Installer* posiada też metody wirtualne
- **OnCommitting** – metoda wywoływana przed zatwierdzeniem instalacji;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **OnCommitted** – metoda wywoływana po zatwierdzeniu instalacji;
- **OnAfterInstall** – metoda wywoywana po instalacji;
- **OnAfterRollback** – metoda wywoywana przed anulowaniem wprowadzonych zmian;
- **OnAfterUninstall** – metoda wywoywana po deinstalacji;
- **OnBeforeInstall** – metoda wywoywana przed instalacją;
- **OnBeforeRollback** – metoda wywoywana po anulowaniu wprowadzonych zmian;
- **OnBeforeUninstall** – metoda wywoywana przed deinstalacją,

które można nadpisać. Posiada ona również zdarzenia

- **Committing** – zdarzenie przychodzące przed zatwierdzeniem instalacji;
- **Committed** – zdarzenie przyjące po zatwierdzeniu instalacji;
- **AfterInstall** – zdarzenie przyjące po instalacji;
- **AfterRollback** – zdarzenie przyjące przed anulowaniem wprowadzonych zmian;
- **AfterUninstall** – zdarzenie przyjące po deinstalacji;
- **BeforeInstall** – zdarzenie przyjące przed instalacją;
- **BeforeRollback** – zdarzenie przyjące po anulowaniu wprowadzonych zmian;
- **BeforeUninstall** – zdarzenie przyjące przed deinstalacją,

pod które można podpiąć odpowiednie metody, które mają być wywoływane, kiedy nadejdą stosowne zdarzenia.

Poniższy kod przedstawia przykład implementacji instalatora. Projekt ten jest komplikowany do postaci biblioteki (DLL). Przy której instalacji tej biblioteki dodawane są odpowiednie klucze do rejestru systemu, a przy deinstalacji klucze te są usuwane.

```

using System;
using System.Collections;
using System.Configuration.Install;
using Microsoft.Win32;

namespace CustomInstaller
{
    [System.ComponentModel.RunInstaller(true)]
    public class CustomInstaller : Installer
    {
        public CustomInstaller()
            : base()
        {
            // Dowiązywanie zdarzenia "Committed"
            this.Committed +=
                new InstallEventHandler(CustomInstaller_Committed);
            // Dowiązywanie zdarzenia "Committing"
            this.Committing +=
                new InstallEventHandler(CustomInstaller_Committing);
        }
        // Uchwyt dla zdarzenia "Committing"
        private void CustomInstaller_Committing(object sender,
                                                EventArgs e)
        {
            // Wystąpienie zdarzenia "Committing"
            Console.WriteLine("Committing...");
        }
    }
}

```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

// Uchwyt dla zdarzenia "Committed"
private void CustomInstaller_Committed(object sender,
                                         EventArgs e)
{
    // Wystąpienie zdarzenia "Committed"
    Console.WriteLine("Committed..."); 
}
// Nadpisywanie metody "OnCommitting"
protected override void OnCommitting(IDictionary savedState)
{
    Console.WriteLine("OnCommitting (1)");
    base.OnCommitting(savedState);
    // Add steps to be done before committing an application.
    Console.WriteLine("OnCommitting (2)");
}
// Nadpisywanie metody "OnCommitted"
protected override void OnCommitted(IDictionary savedState)
{
    Console.WriteLine("OnCommitted (1)");
    base.OnCommitted(savedState);
    // Add steps to be done before committing an application.
    Console.WriteLine("OnCommitted (2)");
}
// Implementacja metody Install
public override void Install(IDictionary savedState)
{
    base.Install(savedState);
    Console.WriteLine("Tworzę klucz ...");
    RegistryKey key = Registry.CurrentUser.CreateSubKey("AAA");
    using (RegistryKey subkey = key.CreateSubKey("KluczTestowy"),
          subkey2 = key.CreateSubKey("InnyKlucz"))
    {
        subkey.SetValue("klucz1", "Wartość pierwszego klucza");
        subkey.SetValue("klucz2", "Wartość drugiego klucza");
        subkey.SetValue("klucz3", 12345);
        key.SetValue("klucz", "Jakaś wartość");
    }
}
// Implementacja metody Commit
public override void Commit(IDictionary savedState)
{
    Console.WriteLine("Commit (1)");
    base.Commit(savedState);
    Console.WriteLine("Commit (2)");
}
// Implementacja metody Rollback
public override void Rollback(IDictionary savedState)
{
    base.Rollback(savedState);
}
// Implementacja metody Uninstall
public override void Uninstall(IDictionary savedState)
{
    base.Uninstall(savedState);
    Registry.CurrentUser.DeleteSubKeyTree("AAA");
    //RegistryKey key = Registry.CurrentUser.OpenSubKey("AAA");
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
// ...
//Registry.CurrentUser.DeleteSubKey( "AAA" );
}
}
```

Do skompilowania tego przykładu potrzebne jest dodanie referencji **System.Configuration.Install**.

Powyższy przykład pozwala prześledzić kolejność wywoływanego przedstawionych poprzednio metod oraz nadchodzenia zdarzeń. W wyniku instalacji lub deinstalacji otrzymanej biblioteki powinniśmy dostać następujący komunikat:

```
Commit (1)
OnCommitting (1)
Committing...
OnCommitting (2)
OnCommitted (1)
Committed...
OnCommitted (2)
Commit (2)
```

Instalacja biblioteki z kodu w języku C#

W języku C# do instalacji pliku (assembly, np. biblioteki lub usługi) można użyć klasy **AssemblyInstaller** dziedziczącej (bezpośrednio) po klasie *Installer*. Informacje o instalowanym pliku możemy przekazywać albo w konstruktorze albo przez odpowiednie własności. Klasa ta zawiera między innymi następujące elementy:

- public AssemblyInstaller();
- public AssemblyInstaller(Assembly assembly, string[] commandLine);
- public AssemblyInstaller(string fileName, string[] commandLine);
- public Assembly Assembly { get; set; };
- public string[] CommandLine { get; set; };
- public string Path { get; set; };
- public bool UseNewContext { get; set; }
- public static void CheckIfInstallable(string assemblyName) .

W poniższym przykładzie klasa **AssemblyInstaller** jest używana do instalacji klasy zaimplementowanej w poprzednim przykładzie

```
using System;
using System.Collections;
using System.Configuration.Install;

namespace CustomInstaller
{
    class Program
    {
        static void Main(string[] args)
        {
            IDictionary actions = new Hashtable();
```



Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
try
{
    Console.WriteLine(
        "Enter - instalacja; inny klawisz - deinstalacja");
    Console.ReadKey(cki = Console.ReadKey(true));
    if (cki.Key == ConsoleKey.Enter)
    {
        // Stworzenie nowej instancji instalatora
        // dla assembly "CustomInstaller"
        AssemblyInstaller customAssemblyInstaller = new
            AssemblyInstaller("CustomInstaller.dll", args);
        // Użycie nowego kontekstu do instalacji
        customAssemblyInstaller.UseNewContext = true;
        // Instalacja assembly "CustomInstaller"
        customAssemblyInstaller.Install(actions);
        // Zatwierdzenie instalacji assembly "CustomInstaller"
        customAssemblyInstaller.Commit(actions);
    }
    else
    {
        // Stworzenie nowej instancji instalatora
        // dla assembly "CustomInstaller"
        AssemblyInstaller customAssemblyInstaller = new
            AssemblyInstaller("CustomInstaller.dll", args);
        // Użycie nowego kontekstu instalacji do deinstalacji
        customAssemblyInstaller.UseNewContext = true;
        // Deinstalacja assembly "CustomInstaller"
        customAssemblyInstaller.Uninstall(actions);
        // Zatwierdzenie deinstalacji assembly "CustomInstaller"
        customAssemblyInstaller.Commit(actions);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

12. Dzienniki zdarzeń

Zwykle najważniejsze informacje o działaniu systemu (w szczególności o błędach, które wystąpiły) są zapisywane do tzw. **dzienników zdarzeń** lub **logów**, którym poświęcony jest niniejszy rozdział.

Podstawowe informacje o dziennikach zdarzeń

Użytkownik może przeglądać logi z poziomu konsoli zarządzania **MMC** (*Microsoft Management Console*). W tym celu należy kliknąć prawym klawiszem myszki na „*Mój komputer*”, wybrać „*Zarządzaj*”, a następnie pod grupą „*Narzędzia systemowe*” rozwinąć podgrupę „*Podgląd zdarzeń*”. Pojawi się wówczas lista dzienników, które można przeglądać (do przeglądania dzienników mogą być dodatkowo potrzebne stosowne uprawnienia). Użytkownik systemu Windows XP zobaczy między innymi dziennik **System**, w którym znajdują się wpisy dotyczące zdarzeń, które wystąpiły w obrębie działania systemu operacyjnego oraz dziennik **Aplikacja**, przechowujący informacje o zdarzeniach związanych z działaniem uruchamianych w tym systemie aplikacji.

Informacje przechowywane są w dziennikach w postaci poszczególnych wpisów. Każdy wpis zawiera następujące elementy:

- typ zdarzenia (np. błąd, ostrzeżenie, informacja),
- data wystąpienia zdarzenia,
- godzina wystąpienia zdarzenia,
- źródło wystąpienia zdarzenia,
- kategoria zdarzenia,
- identyfikator zdarzenia (w postaci liczbowej),
- użytkownik,
- komputer,
- opis.

Przy pewnych rodzajach zdarzeń niektóre z tych elementów mogą nie być związane ze zdarzeniem (*użytkownik* lub *kategoria* dla przynajmniej części zdarzeń systemowych) dlatego elementy te mogą przyjmować wartości puste. *Identyfikator* zdarzenia domyślnie przyjmuje wartość 0, jeśli nie jest podawany. Jednak większość elementów wpisu jest obligatoryjna (w szczególności typ, źródło, data i godzina).

Operacje wykonywane na dziennikach zdarzeń

Z poziomu konsoli zarządzania można przeglądać i usuwać poszczególne wpisy.

Z poziomu programu można wykonywać znacznie więcej operacji. Podstawową klasą używaną do operowania na dziennikach zdarzeń jest klasa **EventLog**, dziedzicząca po klasie *Component* i interfejsie *ISupportInitializeInitialize*. Ponadto istotną rolę odgrywa przy tym klasa **EventLogEntry**, której obiekty mogą być powiązywane z konkretnymi wpisami w dziennikach. Do operowania na wpisach (a w szczególności do ich tworzenia) używany jest też typ wyliczeniowy **EventLogEntryType**, którego wartości powiązane są z dopuszczalnymi typami zdarzeń.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

W tej części materiałów pokazane zostaną przykłady operowania na dziennikach zdarzeń z użyciem wspomnianych klas i typów. Pominięty przy tym zostanie tutaj szczegółowy opis elementów tych klas (metod, pól i własności).

Tworzenie, otwieranie i podpinanie źródła

We wszystkich przykładach zakładam, że mam dane następujące stałe

```
const string LogName = "C#DemoLog";
const string LogSource = "Przyklad";
const string LogSource2 = "Przyklad2";
```

przechowujące odpowiednio nazwę dziennika oraz dwie nazwy źródeł. Istotne jest to, że podobnie jak w przypadku nazw usług, pierwsze sześć znaków z tych nazw musi być unikalne, gdyż dzienniki oraz źródła są identyfikowane po nich (jako obiekty w systemie operacyjnym). Co więcej, identyfikatory różnych rodzajów obiektów współdzielą tę samą przestrzeń wpisów, przez co np. utworzenie i zarejestrowanie źródła o tym samym początku nazwy, jaki posiada istniejący dziennik uniemożliwi wykonywanie jakichkolwiek operacji na tym dzienniku. Nazwy źródeł podane w powyższym kodzie nie będą rozróżniane przez system, chociaż wpisy z nich będą miały wyświetlane różne nazwy źródeł.

Obiekt dziennika powiązujemy z logiem o określonej nazwie albo w momencie jego tworzenia przez podanie nazwy dziennika w konstruktorze, np.:

```
EventLog DemoLog = new EventLog(LogName);
```

albo przez właściwość **Log**, np.:

```
EventLog DemoLog = new EventLog();
DemoLog.Log = LogName;
```

Jeśli log o zadanej nazwie nie istnieje, to zostanie on utworzony w momencie dokonywania pierwszego wpisu.

Każdy wpis jest powiązany z konkretnym źródłem, więc przed dokonaniem wpisu należy wskazać źródło zdarzenia. Jeśli chcemy użyć własnego źródła, to należy to źródło wcześniej utworzyć i **powiązać z dziennikiem**, np.:

```
if (!EventLog.SourceExists(LogSource, "."))
{
    EventLog.CreateEventSource(LogSource, LogName);
    Console.WriteLine("Utworzono nowe źródło \"{0}\" dla logu \"{1}\",
                      LogSource, LogName);
    return;
}
```

(przy tym drobne uwagi: powyższa konstrukcja może być użyta **przed** utworzeniem logu, jeśli chcemy użyć konkretnego źródła i logu do dokonania pierwszego wpisu; pierwszy wpis musi być dokonany z istniejącego źródła (lub powiązanego z logiem) źródła). Źródło może zostać podane w konstruktorze lub przez właściwość **Source**.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Zapis

Do dokonywania wpisów w dzienniku służy metoda *WriteEntry()*. Posiada ona szereg przeciążeń, z których wszystkie zawierając argument podający treść opisu (tym niemniej może on być pusty). Wśród dodatkowych argumentów wymienić można typ wpisu, identyfikator, kategorię oraz tablicę surowych danych, która może być dołączona do opisu. Poniżej podany jest przykład dokonywania zapisów do dziennika:

```
EventLog DemoLog = new EventLog(LogName);
DemoLog.Source = LogSource;
DemoLog.WriteEntry("Zapis zdarzenia",
    EventLogEntryType.Information);
DemoLog.WriteLine("Zapis zdarzenia (z numerem ID)",
    EventLogEntryType.Information, 100);
DemoLog.Source = LogSource2;
DemoLog.WriteLine("Zapis ostrzeżenia z innego źródła",
    EventLogEntryType.Warning, 200);
Console.WriteLine("Umieszczone wpisy w logu \'{0}\'", 
    DemoLog.Log);
```

Odczyt

Wpisy dziennika można pobrać w postaci kolekcji (klasy *EventLogEntryCollection*) obiektów typu *EventLogEntry* z własności **Entries**, np.:

```
EventLog DemoLog = new EventLog();
DemoLog.Log = LogName;
foreach (EventLogEntry DemoEntry in DemoLog.Entries)
{
    Console.WriteLine(DemoEntry.Source + ":" +
        DemoEntry.Message + ":" + DemoEntry.TimeWritten);
}
```

Czyszczenie i usuwanie

Do usunięcia wszystkich wpisów z dziennika służy metoda *Clear()*

```
EventLog DemoLog = new EventLog(LogName);
DemoLog.Clear();
```

Do usunięcia całego dziennika służy metoda statyczna *Delete()*

```
EventLog.Delete(LogName);
```

Do usunięcia źródła dziennika służy metoda statyczna *DeleteEventSource()*

```
EventLog.DeleteEventSource(LogSource);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przeciążenia wspomnianych metody statycznych pozwalają też na wskazanie maszyny, której dziennik lub źródło ma zostać usunięte.

Nazwy dzienników, a ustawienia regionalne

Jako ciekawostkę można potraktować fakt, że nazwy dzienników mogą mieć swoje nazwy regionalne. W szczególności w poniższym przykładzie uruchomionym pod systemem z polską lokalizacją możemy się przekonać, że dziennik „Application” będzie wyświetlany pod nazwą „Aplikacja”.

```
EventLog DemoLog = new EventLog("Application");
Console.WriteLine("Zarejestrowana nazwa logu \"{0}\" to \"{1}\",
  DemoLog.Log, DemoLog.LogDisplayName);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

13. Procesy

Ten rozdział poświęcony jest z zapoznaniem z mechanizmami służącymi do

- uruchamiania i zatrzymywania procesów,
- pobierania informacji o procesach,
- modyfikowania własności procesów (np. ustawienie klasy priorytetu lub rdzeni, na których może działać proces).

Klasa *Process*

Podstawową klasą służącą do zarządzania procesami jest klasa ***Process***. Jej obiekty są powiązywane z konkretnymi procesami działającymi w systemie (lub służą do uruchamiania procesów). Posiada ona liczne składowe, które zostały przedstawione poniżej.

Metody statyczne

Klasa *Process* posiada następujące statyczne metody:

- ***EnterDebugMode*** - przełącza komponent klasy w specjalny stan (tryb debugowania) pozwalający na interakcję z procesami systemu operacyjnego działającymi w specjalnym trybie (włącza natywne uprawnienia *SeDebugPrivilege* dla bieżącego wątku)
- ***GetCurrentProcess*** - tworzy nową instancję klasy powiązaną z bieżącym procesem;
- ***GetProcessById*** - tworzy nową instancję klasy powiązaną istniejącym procesem o zadanym identyfikatorze;
- ***GetProcesses*** - tworzy tablicę obiektów powiązanych z procesami działającymi w systemie operacyjnym;
- ***GetProcessesByName*** - tworzy tablicę obiektów powiązanych z procesami o zadanej nazwie działającymi w systemie operacyjnym;
- ***LeaveDebugMode*** - wychodzi ze specjalnego trybu debugowania;
- ***Start*** - tworzy nowy proces uruchamiając wskazany program lub otwierając podany dokument i zwraca nowy obiekt klasy *Process* powiązany z tym procesem.

Wybrane metody nie-statyczne

Klasa *Process* posiada również następujące niestatyczne metody:

- ***Close*** - zwalnia zasoby związane z instancją klasy (obiektem);
- ***CloseMainWindow*** - zamkna proces posiadający interfejs użytkownika poprzez wysłanie polecenia zamknięcia głównego okna aplikacji tego procesu;
- ***GetService*** - zwraca obiekt reprezentujący usługę (jeśli proces realizuje usługę, w przeciwnym wypadku zwraca **null**);
- ***Kill*** - natychmiast zatrzymuje proces związany z obiektem, z którego jest wywoływana metoda;
- ***OnExited*** – jest wywoływaną przez zdarzenie *Exited*;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Refresh** - powoduje ponowne odczytanie buforowanych wartości własności obiektu klasy, z którego metoda była wywołana;
- **WaitForExit** - blokuje bieżący wątek do czasu zakończenia procesu powiązanego z obiektem, na którym wywołano metodę lub do upłynięcia czasu przeterminowania podanego w argumencie;
- **WaitForInputIdle** - powoduje oczekiwanie na przejście procesu w stan bezczynności;
- **Start** - uruchamia proces z programem lub dokumentem wskazywanym przez własność *StartInfo*.

Wybrane (niestatyczne) własności

Klasa *Process* posiada też następujące własności:

- **BasePriority** - pobiera bazowy priorytet dla procesu (wartość liczbową wyliczoną na podstawie wartości własności *PriorityClass*):
 - 4 - Idle,
 - 8 - Normal,
 - 13 - High,
 - 24 - RealTime);

(niedostępne dla procesu uruchomionego z własnością *ProcessStartInfo.UseShellExecute* ustawioną na **true**);
- **CanRaiseEvents** - pozwala sprawdzić, czy proces może wywoływać zdarzenia;
- **EnableRaisingEvents** - pobiera lub ustawia wartość, od której zależy, czy proces w momencie zakończenia procesu powinno być wywoływane zdarzenie *Exited*;
- **ExitCode** - pobiera kod zakończenia procesu (po jego zakończeniu);
- **ExitTime** - pobiera czas, w którym proces się zakończył;
- **Handle** - zwraca natywny uchwyt procesu (przypisany procesowi w momencie startu i używany do utrzymywania jego atrybutów; może być użyty do inicjalizowania uchwytów *WaitHandle* lub wywoływania natywnych metod);
- **HandleCount** - pobiera ilość uchwytów (uchwytów zasobów takich jak uchwyty plików, kolejek komunikatów itp.) otwartych przez proces;
- **HasExited** - zwraca wartość pozwalającą na sprawdzenie, czy proces się zakończył;
- **Id** - zwraca (unikalny) identyfikator procesu;
- **MachineName** - zwraca nazwę maszyny, na której działa proces;
- **MainModule** - pobiera główny moduł procesu (moduł użyty do uruchomienia procesu) (niedostępne dla procesu uruchomionego z własnością *ProcessStartInfo.UseShellExecute* ustawioną na **true**)
- **MainWindowHandle** - pobiera uchwyt głównego okna aplikacji procesu (niedostępne dla procesu uruchomionego z własnością *ProcessStartInfo.UseShellExecute* ustawioną na **true**);
- **MainWindowTitle** - pobiera nagłówek głównego okna aplikacji procesu (niedostępne dla j.w.);
- **MaxWorkingSet** - pobiera lub ustawia maksymalną wielkość „working set” (zbioru stron pamięci widocznego dla procesu w fizycznej pamięci RAM);
- **MinWorkingSet** - pobiera lub ustawia minimalną wielkość „working set”;
- **Modules** - pobiera tablicę/kolekcję modułów wczytanych przez proces;
- **NonpagedSystemMemorySize64** - pobiera rozmiar niestronicowanej pamięci zaallokowanej przez proces;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **PagedMemorySize64** - pobiera rozmiar stronicowanej pamięci zaallokowanej przez proces;
- **PagedSystemMemorySize64** - pobiera rozmiar stronicowanej pamięci systemowej zaallokowanej dla procesu;
- **PeakPagedMemorySize64** - pobiera maksymalny rozmiar pamięci zaallokowanej w pamięci wirtualnej pliku wymiany (od czasu uruchomienia);
- **PeakVirtualMemorySize64** - pobiera maksymalny rozmiar pamięci wirtualnej używanej przez proces;
- **PeakWorkingSet64** - pobiera maksymalny rozmiar pamięci fizycznej używanej przez proces;
- **PriorityBoostEnabled** - pobiera lub ustawia wartość indykującą, czy priorytet skojarzonego procesu powinien być tymczasowo zwiększały (przez system operacyjny) w czasie gdy główne okno procesu jest aktywne (jest wybrane);
- **PriorityClass** - pozwala pobierać lub ustawiać klasę priorytetu dla procesu (typ wyliczeniowy *ProcessPriorityClass: Normal / Idle / High / RealTime / BelowNormal / AboveNormal*)
- **PrivateMemorySize64** - pobiera ilość pamięci zaallokowanej przez proces która nie może być współdzielona z innymi procesami;
- **PrivilegedProcessorTime** - pobiera ilość czasu jaką proces działał w obrębie rdzenia systemu operacyjnego (czas systemowy);
- **ProcessName** - pobiera nazwę procesu;
- **ProcessorAffinity** - pobiera lub ustawia maskę reprezentującą procesory na których mogą działać wątki procesu;
- **Responding** - zwraca wartość informującą o tym, czy interfejs użytkownika procesu reaguje (na bieżąco (!)) na wejście użytkownika;
- **SessionId** - pobiera numer sesji (terminala/usługi terminalowej) procesu;
- **StandardError** - pobiera strumienia do odczytu z wyjścia błędów aplikacji;
- **StandardInput** - pobiera strumienia do zapisu na standardowe wejście aplikacji;
- **StandardOutput** - pobiera strumienia do odczytu ze standardowego wyjścia aplikacji;
- **StartInfo** - pobiera lub ustawia dane, z którymi uruchamiany jest proces (np. argumenty z linii wywołania, nazwa pliku (dokumentu) do otwarcia, nazwa użytkownika, hasło, profil użytkownika);
- **StartTime** - pobiera czas uruchomienia procesu;
- **Threads** - pobiera lub przypisuje tablicę wątków działających w procesie;
- **TotalProcessorTime** - pobiera łączny czas procesora działania procesu;
- **UserProcessorTime** - pobiera czas „użytkownika” działania procesu;
- **VirtualMemorySize64** - pobiera ilość pamięci wirtualnej zaallokowanej przez proces;
- **WorkingSet64** - pobiera ilość fizycznej pamięci zaallokowanej przez proces.

Klasa **ProcessStartInfo**

Klasa **ProcessStartInfo** służy do zarządzania parametrami, z którymi tworzone są procesy. Procesy są uruchamiane metodą *Process.Start* w sposób zadany przez obiekt klasy *ProcessStartInfo* który albo został przekazany jako argument metody *Start* (przy wywołaniu statycznego wariantu metody) albo przez

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

własność *StartInfo* obiektu z którego była wywołana ta metoda (przy wywołaniu niestatycznego wariantu metody *Start*).

Konstruktory

Klasa *ProcessStartInfo* posiada trzy warianty konstruktora:

- ***ProcessStartInfo()*** – tworzona instancja będzie musiała być zainicjalizowana przez odpowiednie własności;
- ***ProcessStartInfo(string)*** - argumentem jest nazwa programu lub dokumentu do uruchomienia;
- ***ProcessStartInfo(string, string)*** - argumentem jest nazwa programu lub dokumentu do uruchomienia oraz argumenty.

Wybrane własności

Klasa *ProcessStartInfo* wiele właściwości pozwalających na ustawianie wielu różnych parametrów uruchamiania procesu. Wśród nich znajdują się:

- ***Arguments*** - podaje lub pobiera argumenty z linii wywołania;
- ***CreateNoWindow*** - informuje, czy proces ma być uruchamiany w nowym oknie;
- ***Domain*** - domena (Active Directory), która ma być użyta przez uruchamiany proces;
- ***EnvironmentVariables*** - pobiera zmienne środowiskowe;
- ***ErrorDialog*** - pobiera lub ustawia wartość odpowiedzialna za to, czy ma być otwierane okienko, jeśli proces nie może zostać uruchomiony;
- ***ErrorDialogParentHandle*** - pobiera lub ustawia uchwyt okna otwieranego kiedy proces nie może zostać uruchomiony;
- ***FileName*** - pobiera lub ustawia nazwę programu lub dokumentu do „uruchomienia”;
- ***LoadUserProfile*** - pobiera lub ustawia wartość odpowiedzialną za to, czy ma dane profilu użytkownika mają być wczytywane z rejestrów;
- ***Password*** - pobiera lub ustawia bezpieczny łańcuch zawierający hasło używane podczas uruchamiania procesu;
- ***RedirectStandardError*** - pobiera lub ustawia wartość odpowiedzialną za to, czy standardowe wyjście błędów ma zostać przekierowane do strumienia *Process.StandardError*;
- ***RedirectStandardInput*** - pobiera lub ustawia wartość odpowiedzialną za to, czy standardowe wyjście uruchamianego procesu ma zostać przekierowane do strumienia *Process.StandardInput*;
- ***RedirectStandardOutput*** - pobiera lub ustawia wartość odpowiedzialną za to, czy standardowe wejście uruchamianego procesu ma zostać przekierowane do strumienia *Process.StandardOutput*;
- ***StandardErrorEncoding*** - pobiera lub ustawia domyślny format kodowania standardowego wyjścia błędów;
- ***StandardOutputEncoding*** - pobiera lub ustawia domyślny format kodowania standardowego wyjścia;
- ***UserName*** - pobiera lub ustawia nazwę użytkownika używaną podczas uruchamiania procesu;
- ***UseShellExecute*** - pobiera lub ustawia wartość odpowiedzialną za to, czy przy uruchamianiu procesu ma być używana powłoka systemu operacyjnego;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Verb** - pobiera lub ustawia akcję jaka ma być podejmowana przy uruchamianiu procesu z dokumentem (zadanym przez własność *FileName*);
- **Verbs** - pobiera tablicę akcji jakie mogą być podjęte przy uruchamianiu procesu z dokumentem (zadanym przez własność *FileName* - rozpoznawane po rozszerzeniu);
- **WindowState** - pobiera lub ustawia stan okna z jakim ma być uruchamiany proces;
- **WorkingDirectory** - pobiera lub ustawia ścieżkę do katalogu roboczego dla uruchamianego procesu.

Przykłady

Odczyt własności bieżącego procesu

```
using System;
using System.Diagnostics;

namespace BiezacyProces
{
    class Program
    {
        static void Main(string[] args)
        {
            Process ThisProcess = Process.GetCurrentProcess();
            Console.WriteLine("Nazwa procesu: " + ThisProcess.ToString());
            Console.WriteLine("Identyfikator procesu: " + ThisProcess.Id);
        }
    }
}
```

Pobranie procesu przez identyfikator

```
using System;
using System.Diagnostics;

namespace WybraniePrzezIdentyfikator
{
    class Program
    {
        static void Main(string[] args)
        {
            Process SpecificProcess = null;
            try
            {
                SpecificProcess = Process.GetProcessById(2000);
                Console.WriteLine(SpecificProcess.ProcessName);
            }
            catch (ArgumentException Problem)
            {
                Console.WriteLine(Problem.Message);
            }

            try
            {
                //SpecificProcess = Process.GetProcessById(2, "machinename");
            }
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        SpecificProcess = Process.GetProcessById(2000, "acer");
        Console.WriteLine(SpecificProcess.ProcessName);
    }
    catch (ArgumentException Problem)
    {
        Console.WriteLine(Problem.Message);
    }
}
```

Pobranie procesów przez nazwę

```
using System;
using System.Diagnostics;

namespace WybraniePrzezNazwe
{
    class Program
    {
        static void Main(string[] args)
        {
            Process[] SpecificProcesses = null;
            //SpecificProcesses = Process.GetProcessesByName("explorer", "acer");
            //SpecificProcesses = Process.GetProcessesByName("svchost", "acer");
            SpecificProcesses = Process.GetProcessesByName("acrord32", "acer");
            foreach (Process ThisProcess in SpecificProcesses)
            {
                //Console.WriteLine(ThisProcess.ProcessName + " " +
                //                  ThisProcess.TotalProcessorTime + " " + ThisProcess.Id);
                Console.WriteLine(ThisProcess.ProcessName + ":" + 
                                  ThisProcess.MainWindowTitle + "; Id: " +
                                  ThisProcess.Id);
            }
        }
    }
}
```

Listowanie procesów

```
using System;
using System.Diagnostics;

namespace Procesy
{
    class Program
    {
        static void Main(string[] args)
        {
            GetAllProcessesWithoutMachineName();
            //GetAllProcessesWithMachineName(".");
            //GetAllProcessesWithMachineName("acer");
        }

        private static void GetAllProcessesWithoutMachineName()
        {
            Process[] AllProcesses = null;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

try
{
    AllProcesses = Process.GetProcesses();
    foreach (Process Current in AllProcesses)
    {
        //Console.WriteLine(Current.ProcessName);
        Console.WriteLine("{0,-15} {1,5}", Current.ProcessName,
                          Current.Id);
    }
}
catch (ArgumentException Problem)
{
    Console.WriteLine(Problem.Message);
}
private static void GetAllProcessesWithMachineName(string machineName)
{
    Process[] AllProcesses = null;
    try
    {
        AllProcesses = Process.GetProcesses(machineName);
        //AllProcesses = Process.GetProcesses("acer");
        foreach (Process Current in AllProcesses)
        {
            Console.WriteLine(Current.ProcessName);
        }
    }
    catch (ArgumentException Problem)
    {
        Console.WriteLine(Problem.Message);
    }
}
}
}
  
```

Uruchamianie procesów

```

using System;
using System.Diagnostics;
using System.ComponentModel;

namespace UruchamianieProcesow
{
    class MyProcess
    {
        static void Main()
        {
            MyProcess myProcess = new MyProcess();

            myProcess.OpenApplication();
            myProcess.OpenWithArguments();
            myProcess.OpenWithStartInfo();
        }

        /// <summary>
        /// Uruchomienie aplikacji / dokumentu (bez argumentów)
        /// </summary>
    }
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
/// </summary>
void OpenApplication()
{
    // Pytanie na inteligencję czy, a jeśli tak, to w których
    // z poniższych wywołań wystąpią błędy?
    Process.Start("IExplore.exe");
    string mySamplePath =
        Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
    Process.Start(mySamplePath);
    Process.Start("moodle.mat.umk.pl");
    Process.Start("http://moodle.mat.umk.pl");
    Process.Start(
        "C:\\Documents and Settings\\jb\\Moje dokumenty\\Test.html");
}

/// <summary>
/// Uruchomienie przeglądarki z argumentami
/// </summary>
void OpenWithArguments()
{
    // URL-e nie są traktowane jako dokumenty
    // należy je otwierać przekazując je jako argumenty
    Process.Start("IExplore.exe", "www.mat.umk.pl");
    Process.Start("IExplore.exe", "moodle.mat.umk.pl");

    // Przeglądarkę można też uruchomić ze wskazanym plikiem
    Process.Start("IExplore.exe",
        "C:\\Documents and Settings\\jb\\Moje dokumenty\\Test.html");
}

/// <summary>
/// Użycie klasy ProcessStartInfo do uruchomiania
/// nowych procesów w postaci zminimalizowanej
/// </summary>
void OpenWithStartInfo()
{
    ProcessStartInfo startInfo = new ProcessStartInfo("IExplore.exe");
    startInfo.WindowStyle = ProcessWindowStyle.Minimized;
    Process.Start(startInfo);
    startInfo.Arguments = "moodle.mat.umk.pl";
    Process.Start(startInfo);
}
```

Listowanie modułów

```
using System;
using System.Diagnostics;

namespace ListowanieModulow
{
    class Program
    {
        static void Main(string[] args)
```



Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
{  
    Process[] ProcesyPrzegladarki = null;  
    bool czyKrotko = true;  
  
    ProcesyPrzegladarki = Process.GetProcessesByName("firefox");  
  
    foreach (Process ProcesPrzegladarki in ProcesyPrzegladarki)  
    {  
        Console.WriteLine("Nazwa procesu: " +  
                          ProcesPrzegladarki.ProcessName);  
        Console.WriteLine("Nagłówek: " +  
                          ProcesPrzegladarki.MainWindowTitle);  
        Console.WriteLine("Identyfikator procesu: " +  
                          ProcesPrzegladarki.Id);  
        Console.WriteLine();  
  
        ProcessModuleCollection moduly = ProcesPrzegladarki.Modules;  
  
        Console.WriteLine("MODUŁY:");  
        if (czyKrotko)  
        {  
            foreach (ProcessModule modul in moduly)  
            {  
                Console.WriteLine("{0,-12}{1,9} bajtów pamięci",  
                                 modul.ModuleName, modul.ModuleMemorySize);  
            }  
        }  
        else  
        {  
            foreach (ProcessModule modul in moduly)  
            {  
                Console.WriteLine(  
                    "Moduł: {0}:\nŚcieżka: {1}\nInformacje:\n{2}\n",  
                    modul.ModuleName, modul.FileName,  
                    modul.FileVersionInfo);  
            }  
        }  
    }  
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

14. Liczniki (Performance Counters)

Liczniki (ang. *Performance Counters*) są mechanizmem pozwalającym na monitorowanie stanu zasobów. Występują w dość dużej liczbie w systemach Windows-owych. Dlatego został im poświęcony ten rozdział.

Podstawowe informacje o licznikach

Informacje o sposobie używania liczników

Liczniki są używane w dość szczególny sposób. Nie występują one w systemie osobno, lecz są umieszczane w kategoriach. Aby użyć licznika z konkretnej kategorii najpierw należy utworzyćinstancję tej kategorii lub podłączyć się pod istniejącą już instancję. Niektóre kategorie mogą posiadać wyłącznie jedną instancję, a inne wiele instancji. Można wskazać pewną analogię z programowaniem obiektowym, gdzie kategoriom odpowiadają klasy, instancjom kategorii odpowiadają obiekty, kategorie jednoinstancyjne to singletony, a liczniki odgrywają rolę pól lub własności.

Istnieje kilkadesiąt dostępnych typów liczników. Liczniki z tej samej kategorii mogą mieć powiązaną rolę. Aby zobrazować to przykładem można sobie wyobrazić kategorię mającą na celu wyliczanie wartości średniej czasu potrzebnego na obsłużenie zapytania przez serwer oraz wartości średniej ilości przesyłanych danych przy operacji. Kategoria taka może zawierać trzy liczniki, z których jeden będzie zliczał czas operacji, drugi ilość przesyłanych danych, a trzeci liczbę wykonywanych operacji. W takim wypadku poszukiwanymi danymi będą ilorazy wartości licznika pierwszego i trzeciego oraz drugiego i trzeciego. Operacja wyliczenia wartości średniej może być jawnie wykonana w programie, ale może być też wykonana przez środowisko uruchomieniowe. W tym drugim przypadku liczniki powinny być odpowiednich typów, zgodnie z przeznaczeniem, np. odpowiednio *AverageTimer32*, *AverageCount64* i *AverageBase*.

Nie można dodawać liczników do istniejącej już kolekcji, ani ich z niej usuwać. Jedynym rozwiązaniem w przypadku, gdy chcemy dodać lub usunąć licznik z kategorii jest usunięcie tej kategorii i utworzenie w jej miejsce nowej o żądanej zawartości. Instancje, które nie są podpięte w żadnym procesie mają wyzerowane wartości.

Klasy związane z licznikami

W języku C# mamy następujące klasy i struktury związane z licznikami:

- **PerformanceCounter** - klasa implementująca licznik systemowy;
- **PerformanceCounterCategory** - klasa implementująca kategorię liczników;
- **CounterCreationData** - klasa opisująca dane potrzebne do utworzenia licznika;
- **CounterCreationDataCollection** - klasa implementująca kolekcję danych potrzebną do utworzenia kategorii liczników;
- **CounterSample** - struktura reprezentująca próbkę danych licznika.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Typy liczników

Typ licznika jest określany na podstawie wartości typu wyliczeniowego **PerformanceCounterType** podowanej przy tworzeniu licznika. Typ ten może przyjmować następujące wartości:

- *NumberOfItems32*
- *NumberOfItems64*
- *NumberOfItemsHEX32*
- *NumberOfItemsHEX64*
- *RateOfCountsPerSecond32*
- *RateOfCountsPerSecond64*
- *CountPerTimeInterval32*
- *CountPerTimeInterval64*
- *RawFraction*
- *RawBase*
- *AverageTimer32*
- *AverageBase*
- *AverageCount64*
- *SampleFraction*
- *SampleCounter*
- *SampleBase*
- *CounterTimer*
- *CounterTimerInverse*
- *Timer100Ns*
- *Timer100NsInverse*
- *ElapsedTime*
- *CounterMultiTimer*
- *CounterMultiTimerInverse*
- *CounterMultiTimer100Ns*
- *CounterMultiTimer100NsInverse*
- *CounterMultiBase*
- *CounterDelta32*
- *CounterDelta64*

Podział liczników ze względu na rodzaj wykonywanych obliczeń

Liczniki możemy podzielić ze względu na następujące rodzaje

- wyliczające **średnią** wartość pomiarów (ang. *average*); z każdym takim licznikiem jest skojarzony licznika bazowy (ang. *base*) zliczający ilość pomiarów;
- wyliczające **różnicę** dwóch ostatnich pomiarów (ang. *difference*); jeśli jest ona dodatnia to jest ona zwracana jako wynik, w przeciwnym wypadku zwracana jest wartość zerowa;
- zwracające najbardziej **aktualny** pomiar (ang. *instantaneous*);
- zwracające wartość w postaci **procentowej** (ang. *percentage*);
- wyliczające **tempo** zliczeń w jednostce czasu (ang. *rate*).

Składowe klasy **PerformanceCounter**

Metody

Klasa **PerformanceCounter** posiada następujące metody:

- **BeginInit** – rozpoczyna inicjalizację instancji licznika;
- **Close** – zamyka licznik i zwalnia przydzielone przez tę instancję licznika zasoby;
- **CloseSharedResources** – zwalnia zasoby z biblioteki współdzielonej zaalokowane przez licznik;
- **Decrement** – w atomicznej (niepodzielnej) operacji zmniejsza wartość licznika o 1;
- **EndInit** – kończy inicjalizację instancji licznika;
- **Increment** – w atomicznej (niepodzielnej) operacji zwiększa wartość licznika o 1;
- **IncrementBy** – w atomicznej (niepodzielnej) operacji zwiększa wartość licznika o zadaną wartość;
- **NextSample** – pobiera próbkę danych i zwraca jej surową lub nieobliczoną wartość;
- **NextValue** – pobiera próbkę danych i zwraca jej wyliczoną wartość;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **RemoveInstance** – usuwa instancję **kategorii** do której przypisany jest licznik (o nazwie wyznaczonej przez własność *InstanceName*).

Właściwości

Klasa *PerformanceCounter* posiada następujące właściwości:

- **CategoryName** – pobiera lub ustawia nazwę kategorii licznika dla licznika;
- **CounterName** – pobiera lub ustawia nazwę licznika skojarzoną z instancją licznika;
- **CounterType** – pobiera typ skojarzonego licznika;
- **InstanceLifetime** – pobiera lub ustawia czas życia procesu;
- **InstanceName** – pobiera lub ustawia nazwę instancji dla licznika;
- **MachineName** – pobiera lub ustawia nazwę maszyny dla licznika;
- **RawValue** – pobiera lub przypisuje surową lub niewyliczoną wartość licznika;
- **ReadOnly** – pobiera lub ustawia wartość odpowiedzialną za to czy licznik jest w trybie wyłącznego odczytu.

Przykłady

Poniższy program wypisze nazwy wszystkich kategorii liczników (wraz z ich typami).

```
using System;
using System.Diagnostics;

namespace Bateria
{
    class OdczytNazwLicznikow
    {
        static void Main(string[] args)
        {
            PerformanceCounterCategory[] pcc = PerformanceCounterCategory.GetCategories();
            foreach (var pc in pcc)
            {
                Console.WriteLine(pc.CategoryName + " " + pc.CounterType);
            }
        }
    }
}
```

Następujący program wypisze dla każdej baterii informacje o jej stanie odczytane z liczników powiązanych z tą baterią (m.in. aktualną pojemność, aktualny prąd ładowania, aktualny prąd rozładowania itp.):

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace Liczniki
{
    class OdczytBaterii
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
  static void Main(string[] args)
  {
    PerformanceCounterCategory bcc = new PerformanceCounterCategory("BatteryStatus");
    string[] instancje = bcc.GetInstanceNames();
    foreach (var instancja in instancje)
    {
      Console.WriteLine("{1}{0}", instancja, bcc.CategoryName);
      PerformanceCounter[] bcic = bcc.GetCounters(instancja);
      foreach (var pc in bcic)
      {
        Console.WriteLine("{0,30} {1,6}", pc.CounterName, pc.NextValue());
      }
    }
  }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

15.Debugowanie i śledzenie aplikacji

Na początku zwróćmy uwagę na zasadniczą różnicę pomiędzy procesami debugowania oraz śledzenia aplikacji. W obu przypadkach mamy do czynienia z wykonywaniem szeregu operacji na działającym w pamięci operacyjnej programie mających na celu zebranie informacji o sposobie jego działania (zwykle w celu znalezienia błędów w kodzie). Jednak proces **debugowania** ingeruje w sposób działania programu (najczęściej mamy do czynienia z pracą krokową, ale mogą występować też np. bezpośrednie przypisania danych do pamięci lub rejestrów procesora). Natomiast proces **śledzenia** nie ingeruje w sposób działania programu (poza wykonywaniem operacji niezbędnych do przekazania informacji sposobie działania programu).

Przypomnijmy też, że budowa (kompilacja) aplikacji może być przeprowadzana w dwóch „konfiguracjach”: „**Debug**” oraz „**Release**”. W trakcie komplikacji w konfiguracji *Debug* są dodawane informacje używane przez debugger (a czasem też pewne dodatkowe mechanizmy) oraz nie jest przeprowadzana optymalizacja. Natomiast w trakcie komplikacji w konfiguracji *Release* nie są dodawane informacje dla debugowania oraz jest przeprowadzana optymalizacja.

Dostęp do ramek stosu

W języku C# można pobierać informacje o zawartości stosu. Służą do tego klasy **StackTrace** i **StackFrame**. Poniższy kod pokazuje ich przykładowe użycie:

```
using System;
using System.Diagnostics;

class StackExample
{
    [STAThread]
    static void Main(string[] args)
    {
        method1(2);
    }

    static void method1(int arg)
    {
        if (arg > 0)
        {
            StackExample se = new StackExample();
            se.method2(arg - 1);
        }
        else
        {
            StackTrace st = new StackTrace(true);
            for (int i = 0; i < st.FrameCount; i++)
            {
                StackFrame sf = st.GetFrame(i);
                Console.WriteLine(" Metoda: {0}",
                    sf.GetMethod());
                Console.WriteLine(" Plik: {0}",
                    sf.GetFileName());
            }
        }
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

        sf.GetFileName();
        Console.WriteLine(" Linia: {0}",
                           sf.GetFileLineNumber());
        Console.WriteLine();
    }
}

void method2(int arg)
{
    method1(arg - 1);
}
}

```

Kontrola nad debugerem w debugowanym programie

Klasa **Debugger** służy do zarządzania debugerem. W szczególności z jej użyciem można z działającego kodu uruchomić debuger i przejść do debugowania tego kodu oraz ustawić w nim pułapkę, co zostało pokazane na poniższym przykładzie:

```

using System;
using System.Diagnostics;

namespace Debugowanie
{
    class Program
    {
        static void LaunchDebuger()
        {
            if (Debugger.Launch())
            {
                Console.WriteLine("Debugger was started");
            }
            if (Debugger.IsLogging())
            {
                Debugger.Log(1, "aaa", "aaa");
            }
            if (Debugger.IsAttached)
            {
                Debugger.Break();
            }
        }

        static void Main(string[] args)
        {
            LaunchDebuger();
        }
    }
}

```

(Kompilacja powyższego programu w trybie *Release* i późniejsze uruchomienie są możliwe, ale w tym wypadku wywołany debuger nie będzie mógł w pełni kontrolować kodu.)

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Raportowanie w trybie debugowania

Klasa **Debug** służy do logowania informacji przez program, jeśli został on skompilowany w konfiguracji „*Debug*”. Poniższy sposób pokazuje jak można używać tej klasy.

```
using System;
using System.Data;
using System.Diagnostics;

namespace Debugowanie
{
    class Program
    {
        static void Main(string[] args)
        {
            Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
            Debug.AutoFlush = true;
            Debug.Indent();
            Debug.WriteLine("Entering Main");
            Console.WriteLine("Hello World.");
            Debug.WriteLine("Exiting Main");
            Debug.Unindent();
        }
    }
}
```

W powyższym przykładzie warto zwrócić uwagę na użycie właściwości **Listeners**, która przechowuje „cele”, do których mają być przekazywane informacje. Więcej informacji na ich temat pojawi się pod koniec tego rozdziału.

W programach komplikowanych w konfiguracji „*Release*” żadne informacje nie będą logowane poprzez klasę *Debug*.

Raportowanie w trybie śledzenia

Klasa **Trace** ma funkcjonalność podobną, jak przedstawiona powyżej klasa *Debug*. Jednak w przeciwieństwie do niej, klasa Trace umożliwia logowanie informacji w programach komplikowanych w konfiguracji „*Release*”, czyli w szczególności w programach, których kod w procesie komplikacji poddany został optymalizacji.

```
using System;
using System.Diagnostics;

namespace TraceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Trace.Listeners.Add(new TextWriterTraceListener(Console.Out));
            Trace.AutoFlush = true;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    Trace.Indent();
    Trace.WriteLine("Entering Main");
    Console.WriteLine("Hello World.");
    Trace.WriteLine("Exiting Main");
    Trace.Unindent();
}
}
}
  
```

Należy jednak podkreślić, że aby możliwe było logowanie informacji przez klasę *Trace*, to podczas komplikacji programu musi być włączony przełącznik **TRACE** (przy komplikacji z linii poleceń ustawiany parametrem „/d:TRACE”, a przy komplikacji w środowisku Visual Studio ustawiany w oknie wybieranym z *Project* (belka) ➔ *Properties* (ostatnia opcja) ➔ *Build* (druga zakładka w pionie) ➔ *TRACE* (checkbox)).

Sam sposób logowania informacji można kontrolować również poprzez plik konfiguracyjny. Służy do tego znacznik **trace** w sekcji **system.diagnostic**. Oto przykład użycia tego znacznika:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="false" indentsize="3" />
  </system.diagnostics>
</configuration>
  
```

Przełączniki śledzenia

Czasami zachodzi potrzeba zmiany zakresu logowania informacji albo całkowitego wyłączenia logowania (np. w celu zwiększenia wydajności programu). Każdorazowa modyfikacja oraz komplikacja kodu źródłowego w celu przeprowadzenia takiej zmiany jest bardzo mało wygodnym rozwiązaniem. Aby temu zapobiec wprowadzony został mechanizm zwany przełącznikami śledzenia. Można podawać ich aktualne wartości w pliku konfiguracyjnym w podsekcji **switches** w sekcji **system.diagnostic**. Wyróżniamy dwa rodzaje przełączników: dwustanowe (włączony-wyłączony) i wielostanowe.

Przełączniki dwustanowe

Przełączniki dwustanowe realizowane są przez obiekty klasy **BooleanSwitch**. Przełączniki te mogą przyjmować tylko dwa stany: włączony – reprezentowany przez wartość **true** lub **1** oraz wyłączony – któremu odpowiada wartość **false** lub **0**. Poniższy przykład pokazuje przypisywanie wartości przełącznikom w pliku konfiguracyjnym:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="Przelacznik" value="1"/>
      <add name="P2" value="True"/>
      <add name="P3" value="0"/>
    </switches>
  </system.diagnostics>
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

</configuration>

Do wartości przełącznika można się odwoływać przez własność **Enabled** obiektów klasy *BooleanSwitch*. Przy tworzeniu tych obiektów można przypisywać w konstruktorze (w trzecim argumencie) domyślną wartość licznika (przypisywaną, gdy jego wartość nie jest podana w pliku konfiguracyjnym). Poniżej pokazany jest przykład użycia tych liczników:

```
using System;
using System.Diagnostics;

namespace BooleanSwitchExample
{
    class Program
    {
        static BooleanSwitch przełącznik = new BooleanSwitch(
            "Przelacznik", "Przelacznik dwustanowy");
        static BooleanSwitch przełącznik2 = new BooleanSwitch(
            "P2", "Drugi przełącznik", "0");

        public static void Main(string[] args)
        {
            if (przełącznik.Enabled) Console.WriteLine("Przełącznik włączony");
            Console.WriteLine("Stan drugiego przełącznika: "
                + przełącznik2.Enabled.ToString());
        }
    }
}
```

Przełączniki wielostanowe

Znacznie większą funkcjonalność zapewniają przełączniki wielostanowe. Są one realizowane są przez obiekty klasy **TraceSwitch**. Przełączniki te mogą przyjmować pięć stanów reprezentowanych przez następujące wartości typu wyliczeniowego **TraceLevel**:

- **Off** (odpowiada mu wartość liczbową 0),
- **Error** (odpowiada mu wartość liczbową 1),
- **Warning** (odpowiada mu wartość liczbową 2),
- **Info** (odpowiada mu wartość liczbową 3),
- **Verbose** (odpowiada mu wartość liczbową 4).

Sposób przypisywania wartości tym licznikom oraz ich inicjowania jest podobny jak w przypadku liczników dwustanowy. Przy tym jako wartość licznika można podawać zarówno nazwę stanu, jak i odpowiadającą mu wartość liczbową. Poniższy przykład pokazuje przypisywanie wartości przełącznikom w pliku konfiguracyjnym:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="Pierwszy" value="1"/>
      <add name="Drugi" value="Warning"/>
    </switches>
  </system.diagnostics>
</configuration>
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
</system.diagnostics>
</configuration>
```

W tym przypadku przełącznikowi **Pierwszy** została przypisana wartość **Error** (odpowiadająca wartości liczbowej **1**), a przełącznikowi **Drugi** wartość **Warning** (odpowiadająca wartości liczbowej **2**). Tworząc obiekt klasy *TraceSwitch* podajemy w konstruktorze nazwę przełącznika. Ponadto opcjonalnie możemy przy tym podać opis przełącznika oraz domyślną wartość przełącznika, jaka ma zostać przyjęta przez przełącznik w przypadku, jeśli żadna wartość nie zostanie przypisana w pliku konfiguracyjnym. Klasa *TraceSwitch* posiada własności

- **Level** zwracającą wartość przełącznika;
- **DisplayName** zwracającą nazwę przełącznika;
- **Description** zwracającą opis przełącznika.

Ponadto mamy w niej też własności

- **TraceError**,
- **TraceWarning**,
- **TraceInfo**,
- **TraceVerbose**,

które zwracają wartość **true**, jeśli przełącznik znajduje się w zadanym stanie lub w stanie o wartości wyższej od zadanego stanu (np. w stanie *Info* lub *Verbose* dla własności *TraceInfo*) oraz wartość **false** w przeciwnym wypadku. Przykładowo, dla następujący program

```
using System;
using System.Diagnostics;

namespace TraceSwitchExample
{
    class Program
    {
        public static void Main(string[] args)
        {
            TraceSwitch[] przełączniki = {
                new TraceSwitch("Pierwszy", "Pierwszy przełącznik"),
                new TraceSwitch("Drugi", "Drugi przełącznik", "Error"),
                new TraceSwitch("Trzeci", "Trzeci przełącznik", "3")
            };

            foreach (TraceSwitch przełącznik in przełączniki)
            {
                // TraceLevel: 0 - Off, 1 - Error,
                // 2 - Warning, 3 - Info, 4 - Verbose
                Console.WriteLine("Poziom przełącznika {0} ({1}): {2}",
                    przełącznik.DisplayName, przełącznik.Description,
                    przełącznik.Level);
                if (przełącznik.TraceWarning)
                    Console.WriteLine(przełącznik.Description
                        + ": przynajmniej ostrzeżenie");
                if (przełącznik.TraceInfo)
                    Console.WriteLine(przełącznik.Description
                        + ": przynajmniej informacja");
            }
        }
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
    }
}
```

uruchomiony z podanym wcześniej plikiem konfiguracyjnym powinien wypisać następujące linijki:

```
Poziom przełącznika Pierwszy (Pierwszy przełącznik): Error
Poziom przełącznika Drugi (Drugi przełącznik): Warning
Drugi przełącznik: przynajmniej ostrzeżenie
Poziom przełącznika Trzeci (Trzeci przełącznik): Info
Trzeci przełącznik: przynajmniej ostrzeżenie
Trzeci przełącznik: przynajmniej informacja
```

Obsługa wyjścia śledzenia

Do nasłuchiwanego na komunikaty wysyłane w ramach debugowania oraz śledzenia programów służą klasy pochodne klasy abstrakcyjnej **TraceListener**. Można do nich zaliczyć następujące klasy:

- **DefaultTraceListener** - generuje poprzez metody *Write* i *Writeline* komunikaty kierowane do metod *OutputDebugString* oraz *Debugger.Log*, komunikaty te są wyświetlane w Visual Studio w oknie *Output*, analogicznie przetwarzane są komunikaty *Fail* i *Assert*. Jako jedyny „nasłuchiwacz” jest automatyczniełączany do każdej kolekcji *Listeners* (pozostałe wymagają dodania do kolekcji *Listeners* w celu użycia).
- **EventLogTraceListener** - przekierowuje wyjście do (systemowego) logu zdarzeń.
- **ConsoleTraceListener** - przekierowuje „śledzone” lub „debugowane” wyjście albo na standardowe wyjście albo na standardowe wyjście błędów (jest to klasa pochodna klasy *TextWriterTraceListener*).
- **TextWriterTraceListener** - przekierowuje wyjście do instancji klasy *TextWriter* lub *Stream* (w szczególności pozwala na zapis zarówno na konsolę, jak i do pliku).
- **DelimitedListTraceListener** - przekierowuje wyjście do obiektu klasy *TextWriter* lub *Stream* (np. *FileStream*), którego szczegóły mogą zostać podane przez właściwości *Delimiter*.
- **XmlWriterTraceListener** - zapisuje wyjście w postaci danych *XML* do obiektów klasy *TextWriter* lub *Stream*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

16. System plików oraz obsługa wejścia-wyjścia

Niniejszy rozdział poświęcony jest podaniu mechanizmów pozwalających na operowanie na strukturze plików w systemach windows-owych oraz przedstawieniu podstawowych mechanizmów służących do obsługi operacji wejścia-wyjścia.

Klasy operujące na strukturze plików

Język C# posiada w standardowych bibliotekach następujące klasy służące do operowania na windowsowym systemie plików:

- **Path** - klasa statyczna dostarczająca operacja na ścieżkach;
- **FileSystemInfo** - abstrakcyjna klasa dostarczająca metod do operowania na plikach i katalogach, jest ona klasą bazową dla klas *FileInfo* i *DirectoryInfo*;
- **File** - klasa statyczna dostarczająca operacje na plikach;
- **FileInfo** - klasa pochodna klasy *FileSystemInfo* umożliwiająca wykonywanie operacji na plikach;
- **Directory** - klasa statyczna umożliwiająca wykonywanie operacji na katalogach;
- **DirectoryInfo** - klasa pochodna klasy *FileSystemInfo* umożliwiająca wykonywanie operacji na katalogach;
- **DriveInfo** - klasa dostarczającą informacji o napędach;
- **FileSystemWatcher** - klasa pozwalająca na odbieranie notyfikacji o zmianach w systemie plików.

Ten podrozdział służy ich przedstawieniu.

Klasa Path

Klasa *Path* posiada następujące pola do wyłącznego odczytu:

- **AltDirectorySeparatorChar** - przechowuje alternatywny znak służący do rozdzielenia katalogów w ścieżce (wartość pola może być taka sama jak pola *DirectorySeparatorChar* lub różna - np. '\' dla Unix-a, '/' dla Windows);
- **DirectorySeparatorChar** - przechowuje znak służący do rozdzielenia katalogów w ścieżce (domyślnie '\' dla Windows);
- **PathSeparator** - przechowuje znak służący do oddzielenia ścieżek do plików lub katalogów (domyślnie dla systemów Windows ';');
- **VolumeSeparatorChar** - przechowuje znak służący do oddzielenia znaku reprezentującego dysk (wolumin) od pozostałej części ścieżki (domyślnie dla systemów Windows ':').

Posiada ona też następujące metody:

- **ChangeExtension** - zmienia rozszerzenie w podanej ścieżce (lub usuwa je, jeśli za nowe rozszerzenie podano wartość **null**);
- **Combine** - łączy dwa łańcuchy znaków opisujące ścieżki;
- **GetDirectoryName** - zwraca katalog z podanej ścieżki (lub **null** w przypadku katalogu głównego);
- **GetExtension** - pobiera rozszerzenie pliku z podanej ścieżki;
- **GetFileName** - pobiera nazwę pliku wraz z rozszerzeniem z podanej ścieżki;
- **GetFileNameWithoutExtension** - pobiera nazwę pliku bez rozszerzenia z podanej ścieżki;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **GetFullPath** - pobiera bezwzględną ścieżkę dla podanej ścieżki;
- **GetInvalidFileNameChars** - zwraca tablicę znaków niedozwolonych w nazwie pliku;
- **GetInvalidPathChars** - zwraca tablicę znaków niedozwolonych w nazwie ścieżki;
- **GetPathRoot** - pobiera główny katalog dla podanej ścieżki;
- **GetRandomFileName** - zwraca losowy ciąg znaków, który może być nazwą pliku lub katalogu;
- **GetTempFileName** - tworzy nowy, pusty plik o losowo ustalonej nazwie i zwraca pełną ścieżkę do niego;
- **GetTempPath** - zwraca ścieżkę do bieżącego tymczasowego katalogu w systemie;
- **HasExtension** - sprawdza czy w podanej ścieżce znajduje się rozszerzenie nazwy pliku;
- **IsPathRooted** - zwraca wartość informującą o tym, czy podany łańcuch znaków jest ścieżką bezwzględną, czy względową.

Klasa *FileSystemInfo*

Klasa *FileSystemInfo* jest klasą abstrakcyjną. Posiada ona następujące chronione pola (wykorzystywane w klasach pochodnych):

- **FullPath** - pełna ścieżka do pliku lub katalogu;
 - **OriginalPath** - ścieżka podana pierwotnie przez użytkownika (względna lub bezwzględna)
- oraz (między innymi) następujące metody:
- **Delete** - usuwa plik lub katalog;
 - **Refresh** – „odświeża” stan obiektu (uaktualnia informacje o fizycznym zasobie z którym skojarzony jest obiekt);

Posiada ona też następujące własności:

- **Attributes** - pobiera lub ustawia atrybuty reprezentowane przez obiekt klasy *FileAttributes*;
- **CreationTime** - pobiera lub ustawia czas utworzenia;
- **CreationTimeUtc** - pobiera lub ustawia czas utworzenia w formacie UTC;
- **Exists** - pobiera wartość informującą o tym, czy plik lub katalog istnieje;
- **Extension** - pobiera łańcuch odpowiadający rozszerzeniu pliku;
- **FullName** - pobiera pełną ścieżkę do pliku lub katalogu;
- **LastAccessTime** - pobiera lub ustawia czas ostatniego dostępu;
- **LastAccessTimeUtc** - pobiera lub ustawia czas ostatniego dostępu w formacie UTC;
- **LastWriteTime** - pobiera lub ustawia czas ostatniego zapisu;
- **LastWriteTimeUtc** - pobiera lub ustawia czas ostatniego zapisu w formacie UTC;
- **Name** - dla plików pobiera nazwę pliku; dla katalogów pobiera ostatnią nazwę w hierarchii (jeśli została podana hierarchia), w przeciwnym wypadku zwraca nazwę katalogu.

Klasa *File*

Klasa *File* jest klasą statyczną i jako taka pozwala na wykonywanie operacji na plikach poprzez następujące swoje metody:

- **AppendAllText** - dołącza podany jako argument łańcuch do wskazanego pliku, tworząc ten plik, jeśli nie istnieje;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **AppendText** - tworzy obiekt klasy *StreamWriter* skojarzony z istniejącym plikiem dla dołączania (ustawia pozycję na końcu strumienia) tekstu kodowanego w standardzie UTF-8;
- **Copy** - kopiuje istniejący plik do nowego pliku;
- **Create** - tworzy nowy (lub nadpisuje istniejący) plik z podanej ścieżki;
- **CreateText** - tworzy nowy (lub nadpisuje istniejący) plik z podanej ścieżki i otwiera go do zapisu tekstu kodowanego w standardzie UTF-8 (tworzy obiekt klasy *StreamWriter* skojarzony z utworzonym plikiem);
- **Decrypt** - deszyfruje plik zaszyfrowany metodą *Encrypt*;
- **Delete** - usuwa istniejący plik;
- **Encrypt** - szyfruje istniejący plik (dla danego konta) - zaszyfrowany w ten sposób plik powinien być deszyfrowany metodą *Decrypt* na tym samym koncie;
- **Exists** - sprawdza, czy wskazany plik istnieje;
- **GetAccessControl** - pobiera listę ACL (w postaci obiektu klasy *FileSecurity*) pliku ze wskazanej ścieżki;
- **GetAttributes** - pobiera atrybuty pliku (w postaci obiektu klasy *FileAttributes*) dla pliku z zadanej ścieżki;
- **GetCreationTime** - pobiera datę i czas utworzenia pliku;
- **GetCreationTimeUtc** - pobiera datę i czas utworzenia pliku (w formacie UTC);
- **GetLastAccessTime** - pobiera datę i czas ostatniego dostępu do pliku;
- **GetLastAccessTimeUtc** - pobiera datę i czas (w formacie UTC) ostatniego dostępu;
- **GetLastWriteTime** - pobiera datę i czas ostatniego zapisu;
- **GetLastWriteTimeUtc** - pobiera datę i czas (w formacie UTC) ostatniego zapisu;
- **Move** - przesuwa istniejący plik w nowe miejsce, opcjonalnie zmieniające jego nazwę;
- **Open** - otwiera plik i tworzy skojarzony z nim obiekt klasy *FileStream*;
- **OpenRead** - otwiera istniejący plik do odczytu i tworzy skojarzony z nim obiekt klasy *StreamReader*;
- **OpenText** - otwiera istniejący plik tekstowy kodowany w standardzie UTF-8 do odczytu i tworzy skojarzony z nim obiekt klasy *StreamReader*;
- **OpenWrite** - otwiera istniejący plik do zapisu i tworzy skojarzony z nim obiekt klasy *FileStream*;
- **ReadAllBytes** - otwiera plik binarny, odczytuje całą jego zawartość do tablicy bitowej, a następnie zamyka plik;
- **ReadAllLines** - otwiera plik tekstowy, odczytuje z niego wszystkie linie do tablicy łańcuchów, a następnie zamyka plik;
- **ReadAllText** - otwiera plik tekstowy, odczytuje z niego wszystkie linie do pojedynczego łańcucha, a następnie zamyka plik;
- **Replace** - zastępuje zawartość wskazanego pliku zawartością innego pliku, usuwając jego pierwotną zawartość i tworząc kopię zastępowanego pliku;
- **SetAccessControl** - dla pliku we wskazanej ścieżce ustawia listę ACL na listę ACL z zadanego obiektu *FileSecurity*;
- **SetAttributes** - ustawia atrybuty pliku z zadanej ścieżki na atrybuty z podanego obiektu klasy *FileAttributes*;
- **SetCreationTime** - ustawia datę i czas utworzenia pliku;
- **SetCreationTimeUtc** - ustawia datę i czas utworzenia pliku (w formacie UTC);

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **SetLastAccessTime** - ustawia datę i czas ostatniego dostępu do pliku;
- **SetLastAccessTimeUtc** - ustawia datę i czas (w formacie UTC) ostatniego dostępu;
- **SetLastWriteTime** - ustawia datę i czas ostatniego zapisu;
- **SetLastWriteTimeUtc** - ustawia datę i czas (w formacie UTC) ostatniego zapisu;
- **WriteAllBytes** - tworzy nowy plik binarny i zapisuje do niego zawartość podanej tablicy bitowej, a następnie zamyka plik;
- **WriteAllLines** - tworzy nowy plik tekstowy i zapisuje do niego zawartość wszystkich linii z podanej tablicy łańcuchów, a następnie zamyka plik;
- **WriteAllText** - tworzy nowy plik tekstowy i zapisuje do niego zawartość wszystkich linii z podanego (pojedynczego) łańcucha, a następnie zamyka plik.

Klasa *FileInfo*

Klasa *FileInfo* część swoich składowych dziediczy po klasie *FileSystemInfo*. Poza tym implementuje następujące, specyficzne dla siebie własności

- **Directory** - pobiera instancję reprezentującą katalog zawierający plik;
- **DirectoryName** - pobiera pełną ścieżkę do katalogu zawierającego plik;
- **IsReadOnly** - pobiera lub ustawia wartość decydującą o tym, czy plik jest wyłącznie do odczytu;
- **Length** - pobiera aktualny rozmiar pliku (w bajtach);
- oraz metody
- **AppendText** - tworzy strumień klasy *StreamWriter* pozwalający na dołączanie tekstu do pliku podłączonego do bieżącej instancji obiektu;
- **CopyTo** - kopiuje istniejący plik (podłączony do bieżącego obiektu) do nowego pliku;
- **Create** - działa analogicznie jak dla klasy *File*;
- **CreateText** - działa analogicznie jak dla klasy *File*;
- **Decrypt** - działa analogicznie jak dla klasy *File*;
- **Delete** - działa analogicznie jak dla klasy *File*;
- **Encrypt** - działa analogicznie jak dla klasy *File*;
- **GetAccessControl** - działa analogicznie jak dla klasy *File*;
- **MoveTo** - działa analogicznie jak dla klasy *File*;
- **Open** - działa analogicznie jak dla klasy *File*;
- **OpenRead** - działa analogicznie jak dla klasy *File*;
- **OpenText** - działa analogicznie jak dla klasy *File*;
- **OpenWrite** - działa analogicznie jak dla klasy *File*;
- **Replace** - działa analogicznie jak dla klasy *File*;
- **SetAccessControl** - działa analogicznie jak dla klasy *File*.

Klasa *Directory*

Klasa *Directory* jest klasą statyczną. Pozwala ona na wykonywanie operacji na katalogach poprzez następujące swoje metody:

- **CreateDirectory** - tworzy wszystkie katalogi w podanej ścieżce;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Delete** - usuwa istniejący katalog;
- **Exists** - sprawdza, czy wskazana ścieżka odpowiada istniejącemu katalogowi;
- **GetAccessControl** - pobiera listę ACL (w postaci obiektu klasy *FileSecurity*) katalogu ze wskazanej ścieżki;
- **GetCreationTime** - pobiera datę i czas utworzenia katalogu;
- **GetCreationTimeUtc** - pobiera datę i czas utworzenia katalogu (w formacie UTC);
- **GetCurrentDirectory** - pobiera bieżący katalog roboczy aplikacji;
- **GetDirectories** - pobiera nazwy (tablicę łańcuchów) podkatalogów wskazanego katalogu;
- **GetDirectoryRoot** - pobiera katalog główny dla podanej ścieżki (łańcuch znaków);
- **GetFiles** - pobiera nazwy (tablicę łańcuchów) plików znajdujących się we wskazanym katalogu;
- **GetFileSystemEntries** - pobiera nazwy (tablicę łańcuchów) (wszystkich) plików oraz podkatalogów znajdujących się we wskazanym katalogu;
- **GetLastAccessTime** - pobiera datę i czas ostatniego dostępu do katalogu;
- **GetLastAccessTimeUtc** - pobiera datę i czas (w formacie UTC) ostatniego dostępu;
- **GetLastWriteTime** - pobiera datę i czas ostatniego zapisu;
- **GetLastWriteTimeUtc** - pobiera datę i czas (w formacie UTC) ostatniego zapisu;
- **GetLogicalDrives** - pobiera nazwy dysków logicznych z komputera (w postaci tablicy łańcuchów);
- **GetParent** - pobiera katalog nadzędny (o jeden poziom w górę) dla zadanej ścieżki (w postaci łańcucha);
- **Move** - przesuwa istniejący plik lub katalog w nowe miejsce;
- **SetAccessControl** - dla katalogu we wskazanej ścieżce ustawia listę ACL na listę ACL z zadanego obiektu *FileSecurity*;
- **SetCreationTime** - ustawia datę i czas utworzenia katalogu;
- **SetCreationTimeUtc** - ustawia datę i czas utworzenia katalogu (w formacie UTC);
- **SetCurrentDirectory** - ustawia bieżący katalog roboczy aplikacji;
- **SetLastAccessTime** - ustawia datę i czas ostatniego dostępu do katalogu;
- **SetLastAccessTimeUtc** - ustawia datę i czas (w formacie UTC) ostatniego dostępu;
- **SetLastWriteTime** - ustawia datę i czas ostatniego zapisu;
- **SetLastWriteTimeUtc** - ustawia datę i czas (w formacie UTC) ostatniego zapisu.

Klasa *DirectoryInfo*

Klasa *DirectoryInfo* część swoich składowych dziediczy po klasie *FileSystemInfo*. Poza tym implementuje następujące, specyficzne dla siebie własności

- **Parent** - katalog (obiekt klasy *DirectoryInfo*) zawierający podany podkatalog;
- **Root** - pobiera korzeń ścieżki (obiekt klasy *DirectoryInfo*)

oraz metody

- **Create** - tworzy katalog;
- **CreateSubdirectory** - tworzy podkatalog w ścieżce powiązanej z bieżącym obiektem klasy *DirectoryInfo*;
- **Delete** - usuwa katalog wraz z zawartością;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **GetAccessControl** - pobiera listę ACL (w postaci obiektu klasy *FileSecurity*) katalogu powiązanego z bieżącym obiektem;
- **GetDirectories** - pobiera podkatalogi (tablicę obiektów klasy *DirectoryInfo*) z katalogu powiązanego z bieżącym obiektem (wszystkie lub spełniające podane kryteria);
- **GetFiles** - pobiera pliki (tablicę obiektów klasy *FileInfo*) z katalogu powiązanego z bieżącym obiektem (wszystkie lub spełniające podane kryteria);
- **GetFileSystemInfos** - pobiera pliki i katalogi (tablicę obiektów klasy *FileSystemInfo*) z katalogu powiązanego z bieżącym obiektem (wszystkie lub spełniające podane kryteria);
- **MoveTo** - przesuwa katalog (powiązany z bieżącym obiektem) i jego zawartość do nowej ścieżki;
- **Refresh** - odświeża stan bieżącego obiektu;
- **SetAccessControl** - dla katalogu powiązanego z bieżącym obiektem ustawia listę ACL na listę ACL z zadanego obiektu klasy *FileSecurity*.

Klasa *DriveInfo*

Klasa *DriveInfo* jest posiada metodę statyczną **GetDrives**, która zwraca tablicę obiektów reprezentujących dyski logiczne na komputerze. Posiada ona też następujące własności:

- **AvailableFreeSpace** - pobiera rozmiar dostępnego miejsca na dysku logicznym (może być ograniczony przez „quotę”);
- **DriveFormat** - pobiera nazwę systemu plików (łańcuch znaków, np. NTFS lub FAT32);
- **DriveType** - pobiera typ dysku w postaci wartości typu wyliczeniowego **DriveType**; **IsReady** - pobiera wartość sygnalizującą gotowość dysku;
- **Name** - pobiera nazwę woluminu;
- **RootDirectory** - pobiera katalog główny dysku (obiekt klasy *DirectoryInfo*);
- **TotalFreeSpace** - pobiera rozmiar wolnego miejsca na dysku logicznym;
- **TotalSize** - pobiera rozmiar dysku logicznego;
- **VolumeLabel** - pobiera lub ustawia etykietę dysku logicznego.

Wspomniany typ wyliczeniowy **DriveType** używany przez tę klasę może przyjmować następujące wartości: *Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram*.

Klasa *FileSystemWatcher*

Klasa *FileSystemWatcher* służy do nasłuchiwanego na zmiany mające miejsce w systemie plików. Nasłuchiwanie może być wykonywane w sposób blokowany (podobnie jak w przypadku użycia metody *select()* w języku C do oczekiwania na gotowość do odbioru lub wysłania danych przez deskryptor, co powinno być przećwiczone wcześniej na przedmiocie *Sieci Komputerowe*). Może też być wykonywane w sposób asynchroniczny, poprzez zarejestrowanie metod, które mają być wykonywane w momencie zajścia określonych zmian w systemie plików (jest to realizowane przez zdarzenia). Klasa *FileSystemWatcher* posiada między innymi następujące metody:

- **BeginInit** - rozpoczyna inicjalizację obiektu przez inny komponent;
- **EndInit** - kończy inicjalizację obiektu przez inny komponent;
- **OnChanged** - wywołuje zdarzenie *OnChanged*;
- **OnCreated** - wywołuje zdarzenie *OnCreated*;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **OnDeleted** - wywołuje zdarzenie *OnDeleted*;
- **OnError** - wywołuje zdarzenie *OnError*;
- **OnRenamed** - wywołuje zdarzenie *OnRenamed*;
- **WaitForChanged** - oczekuje (w sposób synchroniczny) na zajście określonej zmiany w systemie plików

oraz własności:

- **CanRaiseEvents** - pobiera wartość informującą o tym, czy komponent może wywoływać zdarzenia;
- **EnableRaisingEvents** - pobiera lub ustawia wartość odpowiedzialną za to, czy komponent jest włączony;
- **Filter** - pobiera lub ustawia maskę (filtr) plików, których zmiany mają być notyfikowane;
- **IncludeSubdirectories** - pobiera lub ustawia wartość decydującą o tym, czy katalog ma być przeszukiwany rekurencyjnie w poszukiwaniu zmian;
- **InternalBufferSize** - pobiera lub ustawia rozmiar wewnętrznego bufora;
- **NotifyFilter** - pobiera lub ustawia typ zmian, które mają być brane pod uwagę przy „śledzeniu” zmian w systemie plików;
- **Path** - pobiera lub ustawia ścieżkę do katalogu, którego zawartość ma być „śledzona”;
- **SynchronizingObject** - pobiera lub ustawia obiekt używany do „szeregowania” zdarzeń wywoływanych przez zmiany w systemie plików.

Klasy implementujące binarne strumienie

Najpopularniejszym mechanizmem pozwalającym na blokowy dostęp do binarnych danych poprzez fizyczne urządzenia (np. dyski, karty sieciowe) są strumienie. Klasy implementujące binarne strumienie zlokalizowane są w przestrzeni *System.IO*. Zaliczyć do nich możemy następujące klasy:

- **Stream** - abstrakcyjna klasa dostarczająca mechanizmów do sekwencyjnego dostępu do bajtów, jest klasą bazową dla poniżej przedstawionych klas;
- **FileStream** - klasa implementująca strumienie związane z plikami, pozwala ona na przeprowadzanie zarówno synchronicznych, jak i asynchronicznych operacji odczytu i zapisu;
- **MemoryStream** - klasa tworząca strumienie przechowujące dane w pamięci;
- **BufferedStream** - zapieczętowana klasa dodająca mechanizmy buforowania odczytu i zapisu dla innego strumienia;
- **NetworkStream** - klasa implementująca strumień danych dla połączeń sieciowych.

Składowe abstrakcyjnej klasy *Stream*

Klasa *Stream* jest klasą abstrakcyjną, z której dziedziczą pozostałe klasy strumieni. Zawiera ona szereg składowych abstrakcyjnych składowych, które powinny być zaimplementowane w klasach pochodnych. W szczególności klasa ta posiada następujące własności:

- **CanRead** - pobiera wartość informującą o tym, czy strumień pozwala na operacje odczytu;
- **CanSeek** - pobiera wartość informującą o tym, czy strumień pozwala na operacje zmiany pozycji;
- **CanTimeout** - pobiera wartość informującą o tym, czy strumień obsługuje przeterminowania;
- **CanWrite** - pobiera wartość informującą o tym, czy strumień pozwala na operacje zapisu;
- **Length** - pobiera długość strumienia;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Position** - pobiera wartość bieżącej pozycji w strumieniu;
- **ReadTimeout** - pobiera lub ustawia czas przeterminowanie dla odczytu;
- **WriteTimeout** - pobiera lub ustawia czas przeterminowanie dla zapisu

z których wszystkie, poza przeterminowaniami, są abstrakcyjne. Klasa ta posiada również metody

- **BeginRead** - rozpoczyna asynchroniczną operację odczytu;
- **BeginWrite** - rozpoczyna asynchroniczną operację zapisu;
- **Close** - zamyka strumień i zwalnia związane z nim zasoby (deskryptory plików, gniazda itp.);
- **EndRead** - oczekuje na zakończenie asynchronicznej operacji odczytu;
- **EndWrite** - kończy asynchroniczną operację zapisu;
- **Flush** - czyści wszystkie bufora strumienia, co powinno spowodować m.in. zapis buforowanych danych do właściwego urządzenia (pliku na dysku, wysłania pakietu siecią itp.);
- **Read** - czyta sekwencję bajtów ze strumienia poczynając od aktualnej pozycji w strumieniu i zwiększa pozycję strumienia o ilość odczytanych bajtów (zwraca ilość odczytanych bajtów);
- **ReadByte** - odczytuje bajt ze strumienia (i zwraca go po rzutowaniu na tym *Int32*) oraz zwiększa pozycję strumienia o jeden bajt (lub zwraca wartość -1 w przypadku osiągnięcia końca strumienia);
- **Seek** - ustawia pozycję w bieżącym strumieniu;
- **SetLength** - ustawia długość strumienia;
- **Synchronized** - opakowuje strumień w mechanizmy synchronizujące dostęp do strumienia pomiędzy wątkami;
- **Write** - zapisuje sekwencję bajtów do strumienia poczynając od aktualnej pozycji w strumieniu i zwiększa pozycję strumienia o ilość zapisanych bajtów (zwraca ilość zapisanych bajtów);
- **WriteByte** - zapisuje bajt do strumienia i zwiększa pozycję strumienia o jeden bajt;
- Klasa *Stream* posiada również statyczne pole do wyłącznego odczytu **Null**, które przechowuje referencję do strumienia nie powiązanego z żadnym źródłem danych²⁹.

Klasy pochodne klasy *Stream*

Klasy pochodne klasy *Stream* posiadają często specyficzne składowe, których przykłady zostały podane poniżej.

Składowe klasy *FileStream*

Klasa *FileStream* posiada własności

- **IsAsync** - pobiera wartość informującą o tym, czy strumień został otwarty w sposób synchroniczny, czy asynchroniczny;
- **SafeFileHandle** - pobiera obiekt skojarzony z (systemowym) uchwytem pliku (odpowiednik UNIXowego deskryptora), który używany jest przez strumień

oraz metody

- **GetAccessControl** - pobiera obiekt klasy *FileSecurity* z listą ACL dla pliku z którym powiązany jest strumień;

²⁹ Operacje zapisu do takiego strumienia oraz odczytu z tego strumienia można porównać do zapisu i odczytu z urządzenia **/dev/null** w systemach Unix-o-podobnych.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Lock** - blokuje dostęp do pliku powiązanego ze strumieniem zapobiegając zmianom jego zawartości (przez inne procesy), jednocześnie zezwalając na jego odczyt (można podać początek i długość blokowanego obszaru);
- **Unlock** - przywraca dostęp (innym procesom) do poprzednio zablokowanych obszarów pliku powiązanego ze strumieniem lub ich fragmentów (można podać początek i długość odblokowywanego obszaru);
- **SetAccessControl** - ustawia listę ACL dla pliku z którym powiązany jest strumień zgodnie z podanym obiektem klasy *FileSecurity*.

*Składowe klasy **MemoryStream***

Klasa *MemoryStream* posiada własność **Capacity** służącą do pobierania i ustawiania ilości bajtów zaalokowanych dla strumienia (rozmiar bufora) oraz następujące metody:

- **GetBuffer** - zwraca tablicę bajtów (bez znaku) z której został utworzony strumień;
- **ToArray** - zapisuje zawartość strumienia do tablicy bajtowej (pomija niewykorzystywanie bajty);
- **WriteTo** - zapisuje zawartość bieżącego strumienia pamięci do innego strumienia.

*Składowe klasy **NetworkStream***

Klasa *NetworkStream* posiada metody

- **DataAvailable** - pobiera wartość informującą o tym, czy można odczytywać dane ze strumienia (czy strumień jest gotowy do odczytu z niego danych);
- **Readable** - pobiera lub ustawia wartość informującą o tym, czy strumienia można używać do odczytu;
- **Socket** - pobiera gniazdo używane przez strumień;
- **Writeable** - pobiera lub ustawia wartość informującą o tym, czy strumienia można używać do zapisu.

Klasy implementujące formatowane dane („reader”-y i „writer”-y)

Języku C# posiada też mechanizmy pozwalające na obsługę formatowanych strumieni wejścia i wyjścia zwanych „reader”-ami i „writer”-ami³⁰. Są one dostarczane poprzez następujące klasy

- **TextReader, TextWriter** - abstrakcyjne klasy dostarczające mechanizmów do sekwencyjnego odpowiednio: odczytu oraz zapisu ciągów znaków;
- **StringReader, StringWriter** - klasy pochodne odpowiednio klas *TextReader* oraz *TextWriter* implementujące odpowiednio odczyt i zapis łańcuchów znaków;
- **StreamReader, StreamWriter** - klasy pochodne odpowiednio klas *TextReader* oraz *TextWriter* implementujące odpowiednio odczyt i zapis znaków we wskazanym kodowaniu z/do strumieni bajtów;
- **BinaryReader, BinaryWriter** - klasy implementujące odpowiednio odczyt i zapis danych jako binarnych wartości we wskazanym kodowaniu

i ich klasy pochodne.

³⁰ Przykładowo klasa *System.Console* używa takich strumieni do obsługi wejścia, wyjścia i wyjścia diagnostycznego. Można je pobrać przez właściwości odpowiednio *In*, *Out* i *Error* oraz ustawić przez metody odpowiednio *SetIn*, *SetOut* i *SetError*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Inne mechanizmy związane z wejściem-wyjściem

C# ma również wbudowaną obsługę bardziej zaawansowanych mechanizmów związanych z wejściem-wyjściem. Wspomnimy tutaj o dwóch z nich.

Izolowane zasobniki

W C# istnieje możliwość zarządzania izolowanymi zasobnikami danych. W tym celu stworzone zostały następujące trzy klasy:

- **IsolatedStorage** - abstrakcyjna klasa dostarczająca mechanizmów zarządzania wydzielonym (izolowanym) obszarem;
- **IsolatedStorageFile** - zapieczętowana klasa pochodna klasy *IsolatedStorage* implementująca wydzielony (izolowany) obszar zawierający pliki i katalogi;
- **IsolatedStorageFileStream** - klasa zapewniająca dostęp do pliku znajdującego się w wydzielonym (izolowanym) obszarze.

Kompresja i dekompresja danych

C# zawiera też klasy wspierające kompresję i dekompresję danych (choć w jedynie dwóch formatach i o ograniczonych możliwościach³¹):

- **DeflateStream** - klasa dostarczająca mechanizmów do kompresji i dekompresji z użyciem algorytmu *Deflate* (jeden z algorytmów używanych w formacie ZIP łączący algorytm słownikowy LZ77 z kodowaniem Huffmana);
- **GZipStream** - klasa dostarczająca mechanizmów do kompresji i dekompresji zgodnie ze standardem *GZip* (opublikowanym w RFC 1952) (w szczególności pozwala na sprawdzanie sum CRC).

Przykłady

Przykład na użycie klasy **Path**

Poniższy przykład pokazuje w jaki sposób można używać klasy **Path**. Warto zwrócić uwagę na to, że wyłącznie przy wywołaniu metody *GetTempFileName()* zachodzi sytuacja, w której rzeczywiście musimy pracować ze ścieżką z rzeczywistego systemu plików. Ten przykład pokazuje między innymi, że różne wywołania metody *GetRandomFileName()* nie muszą dać różnych wyników. Ponieważ generator liczb pseudolosowych używany przez tę metodę używa jako ziarna bieżącego czasu, istnieje duża szansa, że w wyniku kilku wywołań tej funkcji występujących bezpośrednio po sobie otrzymamy te same ścieżki.

```
using System;
using System.IO;

namespace KlasaPath
{
    class Program
```

³¹ Zarówno pod względem maksymalnego limitu kompresowanych danych, jak i wariantów używanych algorytmów.



Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
{
    static void Main(string[] args)
    {
        string sciezka = @"C:\WINDOWS\system32\config\system.LOG";

        Console.WriteLine("Rozpatrywana ścieżka: \">{0}\", sciezka);
        Console.WriteLine("Path.GetDirectoryName(): \">{0}\",
                           Path.GetDirectoryName(sciezka));
        Console.WriteLine("Path.GetExtension(): \">{0}\",
                           Path.GetExtension(sciezka));
        Console.WriteLine("Path.GetFileName(): \">{0}\",
                           Path.GetFileName(sciezka));
        Console.WriteLine("Path.GetFileNameWithoutExtension(): \">{0}\",
                           Path.GetFileNameWithoutExtension(sciezka));
        Console.WriteLine("Path.GetFullPath(): \">{0}\",
                           Path.GetFullPath(sciezka));
        Console.WriteLine("Path.GetPathRoot(): \">{0}\",
                           Path.GetPathRoot(sciezka));
        Console.WriteLine("Path.HasExtension(): {0}",
                           Path.HasExtension(sciezka));
        Console.WriteLine("Path.IsPathRooted(): {0}",
                           Path.IsPathRooted(sciezka));
        Console.WriteLine();

        Console.WriteLine("Path.GetTempPath(): \n\t{0}\",
                           Path.GetTempPath());
        Console.WriteLine("Path.GetRandomFileName(): \n\t{0}\n\t{0}\",
                           Path.GetRandomFileName(), Path.GetRandomFileName());
        Console.WriteLine("Path.GetRandomFileName(): \n\t{0}\n\t{0}\",
                           Path.GetRandomFileName(), Path.GetRandomFileName());
        Console.WriteLine();

        string plikTymczasowy = Path.GetTempFileName();
        Console.Write("Tworzenie pliku tymczasowego przez ");
        Console.WriteLine("Path.GetTempPath(): \n\t{0}\",
                           plikTymczasowy);
        if (File.Exists(plikTymczasowy))
        {
            Console.Write("Usuwanie pliku tymczasowego ... ");
            File.Delete(plikTymczasowy);
            if (!File.Exists(plikTymczasowy))
                Console.WriteLine("usunięty");
        }
    }
}
```

Przykład na użycie klasy **File**

Poniższy przykład pokazuje w jaki sposób można używać klasy **File** do przeprowadzania operacji na plikach.

```
using System;
using System.IO;

namespace KlasaFile
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

{
  class Program
  {
    static void Main(string[] args)
    {
      string nazwaPliku = "plik.txt";
      if (File.Exists(nazwaPliku))
      {
        Console.WriteLine("Plik \"{0}\" istnieje", nazwaPliku);
      }
      else
      {
        Console.WriteLine("Tworzę plik \"{0}\" ...", nazwaPliku);
        File.Create(nazwaPliku);
        //Thread.Sleep(1000);
        if (!File.Exists(nazwaPliku))
        {
          Console.WriteLine("Błąd utworzenia pliku \"{0}\",",
                            nazwaPliku);
        }
        Console.WriteLine("Kończę ...", nazwaPliku);
        return;
      }
      File.AppendAllText(nazwaPliku, "Witaj świecie");
      Console.WriteLine("Data utworzenia pliku: {0}",
                        File.GetCreationTime(nazwaPliku));
      Console.WriteLine("Data ostatniego zapisu: {0}",
                        File.GetLastWriteTime(nazwaPliku));
      Console.WriteLine("Data ostatniego dostępu: {0}",
                        File.GetLastAccessTime(nazwaPliku));

      FileAttributes fa = File.GetAttributes(nazwaPliku);
      Console.WriteLine("Atrybuty: {0}", fa);

      Console.WriteLine("Czy posiada atrybut archiwalny: {0}",
                        FileAttributes.Archive & fa);

      Console.WriteLine("Czy zaszyfrowany: {0}",
                        FileAttributes.Encrypted & fa);

      Console.WriteLine("Szyfrowanie ...");
      File.Encrypt(nazwaPliku);

      Console.WriteLine("Czy zaszyfrowany: {0}",
                        FileAttributes.Encrypted & File.GetAttributes(nazwaPliku));

      Console.WriteLine("Zawartość: {0}",
                        File.ReadAllText(nazwaPliku));

      Console.WriteLine("Odszyfrowywanie ...");
      File.Decrypt(nazwaPliku);

      Console.WriteLine("Czy zaszyfrowany: {0}",
                        FileAttributes.Encrypted & File.GetAttributes(nazwaPliku));

      Console.WriteLine("Zawartość: {0}",

```



Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

        File.ReadAllText(nazwaPliku)) ;

    Console.WriteLine("Usuwanie ...");
    File.Delete(nazwaPliku);
}
}
}
}
```

Przykład na użycie klas **DirectoryInfo** i **FileInfo**

Poniższy przykład pokazuje w jaki sposób można używać klas **DirectoryInfo** i **FileInfo**.

```

using System;
using System.IO;

public class WyliczanieRozmiaruKatalogu
{
    public static long PobierzRozmiar(DirectoryInfo katalog)
    {
        long rozmiar = 0;

        FileInfo[] pliki = katalog.GetFiles();
        foreach (FileInfo plik in pliki)
        {
            rozmiar += plik.Length;
        }

        DirectoryInfo[] katalogi = katalog.GetDirectories();
        foreach (DirectoryInfo podkatalog in katalogi)
        {
            rozmiar += PobierzRozmiar(podkatalog);
        }
        return (rozmiar);
    }
    public static void Main(string[] args)
    {
        if (args.Length != 1)
        {
            DirectoryInfo katalog = new DirectoryInfo(
                Environment.CurrentDirectory);
            Console.WriteLine("Katalog {0} zajmuje {1} bajtów.",
                katalog, PobierzRozmiar(katalog));
        }
        else
        {
            DirectoryInfo katalog = new DirectoryInfo(args[0]);
            Console.WriteLine("Katalog {0} zajmuje {1} bajtów.",
                katalog, PobierzRozmiar(katalog));
        }
    }
}
```

Przykład na użycie klasy **DriveInfo**

Poniższy przykład pokazuje w jaki sposób można używać klasy **DriveInfo**.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

using System;
using System.IO;

namespace KlasaDriveInfo
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] dyski = DriveInfo.GetDrives();

            foreach (DriveInfo dysk in dyski)
            {
                Console.WriteLine("Dysk {0}", dysk.Name);
                Console.WriteLine(" * rodzaj urządzenia: {0}", dysk.DriveType);
                if (dysk.IsReady == true)
                {
                    Console.WriteLine(" * etykieta dysku: {0}",
                                      dysk.VolumeLabel);
                    Console.WriteLine(" * system plików: {0}",
                                      dysk.DriveFormat);
                    Console.WriteLine(
                        " * przestrzeń dostępna dla bieżącego użytkownika:{0, 15} bytes",
                        dysk.AvailableFreeSpace);

                    Console.WriteLine(
                        " * łączna dostępna wolna przestrzeń: {0, 15} bytes",
                        dysk.TotalFreeSpace);

                    Console.WriteLine(
                        " * całkowita pojemność dysku: {0, 15} bytes ",
                        dysk.TotalSize);
                }
            }
        }
    }
}
  
```

Przykład na użycie klasy **FileWatcher**

Poniższy przykład pokazuje w jaki sposób można używać klasy **FileWatcher** w wariantie nasłuchiwanego zmiany w systemie plików w sposób asynchroniczny.

```

using System;
using System.IO;
using System.Diagnostics;
using System.Security.Permissions;

namespace KlasaFileSystemWatcher
{
    class Program
    {
        static void Main(string[] args)
        {
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    Run();
}

[PermissionSet(SecurityAction.Demand, Name = "FullTrust")]
public static void Run()
{
    FileSystemWatcher watcher = new FileSystemWatcher();
    watcher.Path =
        Environment.GetFolderPath(Environment.SpecialFolder.Desktop);

    watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
        | NotifyFilters.FileName | NotifyFilters.DirectoryName;
    watcher.Filter = "*.*"; // domyślne ustawienie

    // dodanie uchwytów oczekiwania
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    watcher.Created += new FileSystemEventHandler(OnChanged);
    watcher.Deleted += new FileSystemEventHandler(OnChanged);
    watcher.Renamed += new RenamedEventHandler(OnRenamed);

    // włączenie nasłuchiwanie na zdarzenia
    // (bez tego możliwe jest wyłącznie nasłuchiwanie
    // w trybie synchronicznym)
    watcher.EnableRaisingEvents = true;

    Console.WriteLine("Wciśnij klawisz Enter aby zakończyć program.");
    Console.Read();
}

// definiowanie metod obsługi zdarzeń
private static void OnChanged(object source, FileSystemEventArgs e)
{
    // metoda jest uruchamiana w przypadku zdarzeń:
    // zmiana, stworzenie lub usunięcie pliku
    string zmiana;
    switch (e.ChangeType)
    {
        case WatcherChangeTypes.Changed:
            zmiana = "Zmodyfikowano";
            break;
        case WatcherChangeTypes.Created:
            zmiana = "Utworzono";
            break;
        case WatcherChangeTypes.Deleted:
            zmiana = "Usunięto";
            break;
        default:
            zmiana = string.Empty;
            break;
    }
    Console.WriteLine(zmiana + " plik " + e.FullPath);
}

private static void OnRenamed(object source, RenamedEventArgs e)
{
    // metoda jest uruchamiana w przypadku zmiany nazwy pliku
}

```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
Console.WriteLine("Zmieniono nazwę pliku z {0} na {1}",  
    e.OldFullPath, e.FullPath);  
}  
}  
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

17. Bezpieczeństwo: Code Access Security

W systemach Windowsowych został zaimplementowany szereg mechanizmów bezpieczeństwa. Mechanizmy rozpatrywane w ramach niniejszych materiałów zostały podzielone na trzy grupy.

Pierwszą z nich stanowią zabezpieczenia na poziomie kodu (ang. *Code Access Security*, w skrócie CAS). Przydzielają one uprawnienia z zależnością od własności kodu, np. posiadanych przez niego podpisów lub jego pochodzenia. Przykładowo zwykle programy pochodzące z sieci lokalnej (Intranet) są uznawane za bardziej „bezpieczne” i „zaufane” niż pochodzące z sieci globalnej (Internetu) i jako takie będą posiadały większe uprawnienia przy uruchomieniu, a jeszcze większe uprawnienia dostaną programu uruchamianego lokalnie z komputera.

Drugą grupę uprawnień będą stanowiły zabezpieczenia na poziomie użytkownika. Uprawnienia te przypisywane są konkretnym użytkownikom. Użytkownicy muszą przejść proces autentykacji i autoryzacji.

Osobną grupę stanowią mechanizmy kryptograficzne, takie jak szyfrowanie, deszyfrowanie i podpisywanie danych.

Code Access Security w .NET Frameworku o wersjach niższych niż 4.0

Możemy wyróżnić trzy rodzaje mechanizmów CAS:

- z wersji 1.0 i 1.1 Frameworka,
- z wersji Frameworka od 2.0 do 3.5,
- z wersji 4.0 Frameworka i wyższych.

Są one niekompatybilne ze sobą, ale mogą jednocześnie funkcjonować w systemie niezależnie od siebie. Wymagają wówczas osobnych narzędzi konfiguracyjnych. Warto o tym pamiętać, gdyż np. konfiguracja CAS przeprowadzona dla wersji 2.0 .NET Frameworka będzie stosowana przy uruchamianiu programów z wersjami 3.0 i 3.5 .NET Frameworka, ale nie dla programów uruchomionych z wersjami 1.1 lub 4.0.

Mechanizmy CAS zaimplementowane w wersjach od 1.0 do 3.5 są podobne i zostaną opisane razem w tym podrozdziale. Natomiast w wersji 4.0 zostały wprowadzone istotne zmiany, które zostaną przedstawione osobno.

Elementy CAS

W systemie bezpieczeństwa CAS wyróżnić można następujące pięć rodzajów elementów:

- **Identyfikacja** (ang. *Evidence*).
- **Uprawnienia** (ang. *Permission*)
- **Zbiory uprawnień** (ang. *Permission Set*)
- **Grupy kodu** (ang. *Code Group*)
- **Polityka bezpieczeństwa** (ang. *Security Policy*)

Narzędzia do konfigurowania CAS

Do konfiguracji CAS w .NET Frameworku służą dwa narzędzia. Jednym z nich jest okienkowa aplikacja *.NET Framework Configuration Tool* (*Mscorcfg.msc*), przydatna do przeprowadzania ręcznej konfiguracji.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Drugim z nich jest aplikacja konsolowa *Code Access Security Policy Tool* (`caspol.msc`), którą można w wygodny sposób wywoływać ze skryptów lub innych programów używanych do konfiguracji CAS.

Rodzaje identyfikacji (**Evidence**)

Wyróżnić możemy dwa rodzaje identyfikacji: identyfikację maszyny (ang. *Host Evidence*) dostarczaną przez maszynę z której pobierany jest kodu oraz identyfikację kodu (ang. *Assembly Evidence*), będącą integralną częścią kodu tworzoną i dostarczaną przez użytkownika lub wytwórcę/dostawcę oprogramowania.

Do przechowywania informacji o identyfikacji służy klasa ***Evidence***. Obiekty tej klasy mogą zawierać następujące rodzaje identyfikacji:

- ***Application directory*** - katalog w którym położone jest assembly;
- ***Hash*** -- podpis (funkcją skrótu) jednoznacznie identyfikujący konkretną wersję assembly;
- ***Publisher*** - podpis cyfrowy wydawcy, jednoznacznie identyfikujący twórcę oprogramowania (tylko dla podpisanego assembly);
- ***Site*** - domena z której pobrane zostało assembly;
- ***Strong Name*** - podpis elektroniczny (silny) jednoznacznie identyfikujący assembly wraz z jego przestrzenią nazw (tylko dla podpisanego assembly);
- ***URL*** - ścieżka URL z której pobrane zostało assembly;
- ***Zone*** - strefa w której zostało uruchomione assembly (np. *Internet* lub *LocalIntranet*).

Domyślne uprawnienia (**Permissions**)

Uprawnienia są przechowywane w obiektach klasy ***Permissions***. .NET Framework dostarcza następujących domyślnych rodzajów uprawnień:

- ***Directory Services*** - nadawanie uprawnień dostępu do Active Directory;
- ***DNS*** - zezwalanie lub ograniczanie praw do wysyłania zapytań do serwera DNS;
- ***Environment Variables*** - nadaje uprawnienia dostępu do zmiennych środowiskowych (wszystkich lub wybranych) (do ich odczytu i modyfikacji);
- ***Event Log*** - kontroluje dostęp do logów;
- ***File Dialog*** - decyduje czy assembly ma uprawnienia do otwierania *Open dialog box* oraz *Save dialog box* (do żadnego, do wybranego lub obu);
- ***File IO*** - ogranicza dostęp do plików i folderów (dostęp nieograniczony lub lista ścieżek z uprawnieniami (*Read, Write, Append, Path Discovery*));
- ***Isolated Storage File*** - nadaje uprawnienia dostępu do izolowanego systemu plików, ustala poziom izolacji i rozmiar quoty;
- ***Message Queue*** - pozwala na dostęp do kolejek komunikatów (może być ograniczony przez ścieżkę i rodzaj dostępu);
- ***Performance Counter*** - ustala czy assembly ma prawo odczytu lub/i zapisu liczników;
- ***Printing*** - ustala zakres możliwości drukowania;
- ***Reflection*** - odpowiada za to, czy assembly może odczytywać informacje o składnikach i ich typach z innych assembly;
- ***Registry*** - ustala dostęp do kluczy rejestru (nieograniczony lub do wybranych kluczy (rodzaj dostępu: *Read, Write, Delete*));

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Security** - pozwala ustawić dostęp do różnych mechanizmów i własności CAS (np. wywoływanie kodu niezarządzalnego, dostęp do uprawnień, kontrola wątków);
- **Service Controller** - ustala do jakie usługi assembly może przeglądać lub kontrolować;
- **Socket Access** - kontroluje uprawnienia do inicjalizowania połączeń TCP/IP (można ograniczać dostęp do adresów, portów, protokołów);
- **SQL Client** - ustala czy assembly ma prawo do łączenia się z SQL Serverem (i czy może używać pustego hasła);
- **User Interface** - ustala uprawnienia do otwierania nowych okien oraz dostęp do schowka;
- **Web Access** - ustala czy assembly ma dostęp do stron WWW (a jeśli tak, to do których);
- **X509 Store** - nadaje assembly uprawnienia dostępu do bazy certyfikatów X509 (ustala czy może ono dodawać, usuwać, otwierać certyfikaty).

Domyślne zbiory uprawnień (*Permission Sets*)

Uprawnienia zbierane są w zestawy nazywane zbiorami uprawnień. .NET Framework dostarcza następujące domyślne zbiory uprawnień:

- **FullTrust** - zwalnia assembly ze sprawdzania uprawnień CAS;
- **SkipVerification** - umożliwia assembly pominięcie sprawdzania uprawnień (co może zwiększyć wydajność kosztem zmniejszenia bezpieczeństwa);
- **Execution** - nadaje assembly uprawnienia wyłącznie do uruchomienia;
- **Nothing** - nie nadaje assembly żadnych uprawnień (w szczególności do uruchomienia);
- **LocalIntranet** - nadaje rozbudowany zbiór uprawnień do assembly (ale np. pozwala na dostęp do systemu plików tylko przez okienka dialogowe);
- **Internet** - nadaje ograniczony zbiór uprawnień do assembly;
- **Everything** - nadaje wszystkie uprawnienia (ale nie wyłącza sprawdzania uprawnień CAS).

Przykłady domyślnych zbiorów uprawnień

LocalIntranet

- Environment Variables
- File Dialog
- Isolated Storage File
- Reflection
- Security
- User Interface
- DNS
- Printing

Internet

- File Dialog
- Isolated Storage File
- Security
- User Interface
- Printing

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Grupy kodu (*Code Groups*) i polityka bezpieczeństwa (*Security Policy*)

Grupa kodu (ang. *Code Group*) jest mechanizmem autoryzacji kojarzącym assembly ze zbiorami uprawnień.

Polityka bezpieczeństwa (ang. *Security Policy*) jest logicznym powiązaniem grup kodu ze zbiorami uprawnień.

Wielowarstwość polityki bezpieczeństwa

Do wersji 3.5 .NET Frameworka polityka bezpieczeństwa była wielowarstwowa. Zwykle składała się z następujących czterech warstw, z których pierwsze trzy były konfigurowalne przez narzędzia .NET Frameworka (istniała też możliwość konfiguracji z poziomu kodu), a ostatnia tylko z poziomu kodu.

- Warstwa instytucji (ang. *Enterprise*),
- Warstwa komputera (ang. *Machine*),
- Warstwa użytkownika (ang. *User*),
- Warstwa domeny aplikacji.

Przestrzenie nazw związane z systemem bezpieczeństwa w .NET Framework

Klasy związane z bezpieczeństwem zostały umieszczone w przestrzeni `System.Security` oraz jej podprzestrzeniach. Dokładniej, klasy związane z CAS zostały rozłożone następująco:

- `System.Security` - klasy odpowiadające za podstawową strukturę polityki bezpieczeństwa, w tym klasy bazowe dla pozostałych klas (również dla uprawnień), a także wyjątki i większość interfejsów. Przykładowe klasy: `CodeAccessPermission`, `PermissionSet`, `SecureString`, `SecurityContext`, `SecurityManager`.
- `System.Security.Policy` - klasy zarządzające grupami kodu (ang. *code groups*), warunkami przynależności (ang. *membership conditions*) i identyfikacją (ang. *evidence*). Przykładowe klasy: `CodeConnectAccess`, `CodeGroup`, `Evidence`, `FileDialogGroup`, `PolicyLevel`, `Site`, `StrongName`.
- `System.Security.Permissions` - klasy kontrolujące dostęp do operacji i zasobów. Przykładowe klasy: `CodeAccessSecurityAttribute`, `FileDialogPermission`, `FileDialogPermissionAttribute`.

Przykłady na żądanie uprawnień

Żądanie konkretnych uprawnień przez artykuły

```
[FileIOPermission(SecurityAction.Demand, Read = @"C:\tajne")]
[FileDialogPermission(SecurityAction.Demand, Unrestricted = true)]
```

Utworzenie grupy uprawnień i żądanie przyznania jej uprawnień

```
PermissionSet ps = new PermissionSet(PermissionState.None);
ps.AddPermission(new FileDialogPermission(PermissionState.Unrestricted));
ps.AddPermission(new FileIOPermission(
    FileIOPermissionAccess.Read, @"C:\tajne"));
ps.Demand();
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Żądanie uprawnień z zadanej grupy przez artybuty

```
[PermissionSet(SecurityAction.Demand, Name = "FullTrust")]
```

Zmiany w .NET Frameworku 4.0 w stosunku do wcześniejszych wersji

W wersji 4.0 .NET Framework wprowadzono istotne zmiany w CAS w stosunku do wersji poprzednich. W szczególności:

- Usunięto *.NET Framework Configuration Tool* (`Mscorcfg.msc`).
- Pozostawiono *Code Access Security Policy Tool* (`caspol.exe`) jako (jedyne) narzędzie do zarządzania CAS (poza kodem).
- Czterowarstwowa hierarchia bezpieczeństwa uległa „spłaszczeniu”.
- Zalecono rezygnację z używania „polityki bezpieczeństwa” CAS na rzecz mechanizmów takich jak *Windows Software Restriction Policies* (SRP) oraz *AppLocker* pod systemami Windows 7 oraz Windows Serwer 2008.
- Wprowadzono *security transparency* (*Security Transparency Code*).
- Usunięto grupy kodu (*Code Groups*) i polityki bezpieczeństwa (*Security Policy*).
- Pozostawiono m.in. identyfikacje (*Evidence*) i uprawnienia (*Permissions*) (w szczególności grupy uprawnień).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

18. Bezpieczeństwo: Role Based Security

W modelu bezpieczeństwa bazującym na rolach uprawnienia przypisywane są nie programom, ale użytkownikom, którzy te programy uruchamiają.

Najważniejsze przestrzenie i klasy

Klasy związane z systemem RBS zostały umieszczone w przestrzeni System.Security oraz jej podprzestrzeniach.

W przestrzeni System.Security.Principal zostały umieszczone między innymi klasa **WindowsIdentity** reprezentująca konta użytkowników oraz klasa **WindowsPrincipal** odpowiadająca za przyporządkowanie do grup przynależności użytkowników (*user's group memberships*).

Przestrzeń System.Security.Permissions zawiera klasy **PrincipalPermission** oraz **PrincipalPermissionAttribute** (zwyczaj opisywane zbiorczo jako *PrincipalPermission*), które pozwalają na sprawdzenie lub wymuszenie tego, że osoba uruchamiająca kod została zautentykowana, posiada określony identyfikator lub należy do zadanej roli.

Klasa WindowsIdentity

Tworzenie nowych obiektów klasy WindowsIdentity

Nowe obiekty klasy *WindowsIdentity* można tworzyć na dwa sposoby.

Pierwszy z nich polega na użyciu jednego z przeciążonych konstruktorów, które pozwalają na utworzenie instancji tej klasy dla użytkownika o zadanej nazwie, tokenie lub identyfikacji, którym dodatkowo można przy tworzeniu ustawić typ konta oraz status i stan autentykacji.

Drugim jest użycie jednej z następujących statycznych metod tej klasy:

- *GetAnonymous* - zwraca obiekt reprezentujący anonimowego, niezautentyfikowanego użytkownika;
- *GetCurrent* - zwraca obiekt reprezentujący bieżącego użytkownika.

Właściwości

Do pobierania informacji o koncie służą następujące właściwości klasy *WindowsIdentity*³²:

- *AuthenticationType* – pobiera typ autentykacji użyty do identyfikacji użytkownika;
- *Groups* – pobiera kolekcję grup do których należy użytkownik;
- *IsAnonymous* – określa, czy konto użytkownika jest traktowane jako konto anonimowe;
- *IsAuthenticated* – określa, czy użytkownik przeszedł autentykację;
- *IsGuest* – określa, czy konto użytkownika jest traktowane jako konto gościa;
- *IsSystem* – określa, czy konto użytkownika jest traktowane jako konto systemowe;
- *Name* – zwraca nazwę (nazwę logowania) użytkownika w systemie;
- *Token* – zwraca token konta użytkownika.

³² Można ich użyć wyłącznie do odczytu.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykłady

Pobranie *Identity* bieżącego użytkownika i jego *Principal-a*

```
WindowsIdentity currentIdentity =
    WindowsIdentity.GetCurrent();
WindowsPrincipal currentPrincipal =
    new WindowsPrincipal(currentIdentity);
```

Alternatywna metoda pobrania *Principal-a* bieżącego użytkownika

```
AppDomain.CurrentDomain.SetPrincipalPolicy(
    PrincipalPolicy.WindowsPrincipal);
WindowsPrincipal currentPrincipal =
    (WindowsPrincipal)Thread.CurrentPrincipal;
```

Klasa PrincipalPermission

Klasa *PrincipalPermission* pozwala na sprawdzenie aktualnych uprawnień dla bieżącej roli użytkownika. Można za jej pomocą wymusić posiadanie konkretnych uprawnień.

Przykłady

Żądanie konkretnego użytkownika przez atrybut

```
[PrincipalPermission (SecurityAction.Demand, Name="jb")]
```

Żądanie konkretnej roli przez atrybut

```
[PrincipalPermission (SecurityAction.Demand, Role="Administratorzy")]
```

Żądanie konkretnego użytkownika przez wywołanie metody

```
new PrincipalPermission ("jb", null).Demand();
```

Principal dla pary użytkowników

```
PrincipalPermission jb =
    new PrincipalPermission ("jb", null);
PrincipalPermission mat =
    new PrincipalPermission ("mat001", null);
PrincipalPermission users =
    (PrincipalPermission) jb.Union (mat);
Console.WriteLine (jb.IsSubsetOf (users));
```

Implementacja własnych klas

Implementacja własnej klasy identyfikacji

Programista może tworzyć własne klasy identyfikacji (*Identity*). Klasy takie powinny spełniać następujące wymagania.



Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- dziedziczyć po interfejsie *IIdentity*;
- implementować przynajmniej jeden konstruktor;
- implementować własności *AuthenticationType*, *IsAuthenticated* i *Name*.

Implementacja własnej klasy identyfikującej rolę

Programista może tworzyć własne klasy identyfikujące role użytkownika lub grupy (*Principal*). Klasy takie powinny spełniać następujące wymagania:

- dziedziczyć po interfejsie *IPrincipal*;
- implementować przynajmniej jeden konstruktor;
- implementować metodę *IsInRole*;
- dodatkowo może implementować własność *Roles*, metody *IsInAllRoles* i *IsInAnyRole*.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

19. Kryptografia

Ten rozdział ma za zadanie wskazanie mechanizmów kryptograficznych zaimplementowanych w .NET Framework. Zakłada się przy tym, że czytelnik zna pojęcia związane z kryptografią, takie jak szyfrowanie, deszyfrowanie, podpiswanie, funkcja jednokierunkowa (hash-ująca) i klucz. Ponadto powinien być mu znany podział na symetryczne algorytmy szyfrowania (w których ten sam klucz (lub te same klucze) używany jest zarówno do szyfrowania, jak i deszyfrowania danych) oraz algorytmy asymetryczne (używające różnych kluczy do szyfrowania i deszyfrowania). (W szczególności powinno to pociągać znajomość pojęć kluczy prywatnych i publicznych oraz jawnych i tajnych.)

Większość z wymienianych w tym rozdziale klas znajduje się w przestrzeni **System.Security.Cryptography**, a ich użycie wymaga dodania referencji do assembly **System.Security**.

Po więcej informacji na temat zaimplementowanych w .NET Framework i języku C# mechanizmów kryptograficznych odsyłamy czytelnika do rozdziału 20 książki [14], z której zaczerpniętych zostało wiele z zamieszczonych poniżej przykładów.

Kryptografia symetryczna

Używanie mechanizmów wbudowanych w system

Przypomnijmy, że w systemy windows-owe wbudowano mechanizmy pozwalające na szyfrowanie danych. Używają one symetrycznego szyfrowania, w którym klucz jest tworzony na podstawie danych użytkownika (między innymi jego identyfikatora i **hasła**), przez co jedynie właściciel może odczytać zawartość zaszyfrowanych w ten sposób informacji. Z rozdziału 16 wiadomo, że można zaszyfrować w ten sposób pliki (używając w tym celu klasy *File* lub *FileInfo*) i został podany został przykład przeprowadzenie takiego szyfrowania

```
File.WriteAllText("plik.txt", "Jakieś dane");
File.Encrypt("plik.txt");
// ...
string s = File.ReadAllText("plik.txt");
Console.WriteLine(s);
// ...
File.Decrypt("plik.txt");
```

Tego rodzaju szyfrowanie można też przeprowadzić bez konieczności zapisu danych do pliku. W tym celu można użyć klasy **DataProtectionScope**, z wspomnianej wcześniej przestrzeni i assembly, np.:

```
byte[] daneWejsciowe = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
DataProtectionScope scope =
  DataProtectionScope.CurrentUser;
byte[] zaszyfrowane =
  ProtectedData.Protect(daneWejsciowe, null, scope);
byte[] odszyfrowane =
  ProtectedData.Unprotect(zaszyfrowane, null, scope);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Algorytmy

W .NET Frameworku zostały zaimplementowane następujące symetryczne algorytmy szyfrowania:

- **DES (Data Encryption Standard)** - dawniej uznawany za oficjalny standard i bardzo popularny, obecnie ze względu na krótki klucz oraz wzrost wydajności obliczeniowej komputerów uznawany za mało bezpieczny i nie zalecany; używa on kluczy długości 56 bajtów;
- **Triple DES** (lub 3DES) - algorytm używający trzykrotnego szyfrowania algorytmem DES w celu zapewnienia większego bezpieczeństwa niż typowy algorytm DES (pierwsze i ostatnie szyfrowanie opiera się na tym samym kluczu); w tym szyfrowaniu długość klucza wynosi 156 bajtów, z których tylko 112 jest efektywnie wykorzystywane;
- **RC2 (Rivest Cipher lub Ron's Code)** - algorytm o zmiennej długości klucza opracowany w celu zastąpienia algorytmu DES;
- **Rijndael/AES (Advanced Encryption Standard)** - kolejny standard „rządowy” (m.in. po DES-ie), jako pierwszy w pełni zaimplementowalny w .NET Frameworku (w kodzie zarządzalnym); umożliwia on używanie kluczy długości 128, 160, 192, 234 i 256 bajtów.

Klasy

Następujące klasy implementują wspomniane algorytmy (wytluszczone zostały klasy, których obiekty służą do przeprowadzania operacji kryptograficznych):

- **SymmetricAlgorithm** - klasa abstrakcyjna, z której dziedziczą wszystkie implementacje algorytmów symetrycznych;
- **Rijndael** - klasa abstrakcyjna, z której dziedziczą wszystkie implementacje algorytmu Rijndael;
- **RijndaelManaged** - klasa implementująca zarządzalną wersję algorytmu Rijndael;
- **Aes** - klasa abstrakcyjna, z której dziedziczą wszystkie implementacje algorytmu AES;
- **AesManaged** - klasa implementująca zarządzalną wersję algorytmu AES;
- **AesCryptoServiceProvider** - klasa implementująca asymetryczne operacje szyfrowania i deszyfrowania algorytmu AES używając algorytmu CAPI;
- **RC2;**
- **RC2CryptoServiceProvider;**
- **DES;**
- **DESCryptoServiceProvider;**
- **TripleDES;**
- **TripleDESCryptoServiceProvider.**

Przykłady

Szyfrowanie AES (przykład z książki C# 4.0 in a Nutshell)

```
byte[] key = { 145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50 };
byte[] iv = { 15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7 };

byte[] data = { 1, 2, 3, 4, 5 };
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
using (Stream f = File.Create("encrypted.bin"))
using (Stream c = new CryptoStream(f, encryptor, CryptoStreamMode.Write))
  c.Write(data, 0, data.Length);
```

Deszyfrowanie AES (przykład z książki C# 4.0 in a Nutshell)

```
byte[] key = { 145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50 };
byte[] iv = { 15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7 };

byte[] decrypted = new byte[5];

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
using (Stream f = File.OpenRead("encrypted.bin"))
using (Stream c = new CryptoStream(f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte()) > -1; )
    Console.WriteLine(b + " ");
```

Kryptografia asymetryczna

Algorytmy

W .NET Frameworku zostały zaimplementowane następujące asymetryczne algorytmy szyfrowania bazujące na złożoności problemu rozkładu dużych liczb całkowitych na czynniki pierwsze:

- **DSA (Digital Signature Algorithm),**
- **RSA (Rivest-Shamir-Adleman)**

oraz następujące algorytmy oparte na krzywych eliptycznych:

- **ECDH (Diffie-Hellman, Elliptic Curve Diffie-Hellman),**
- **ECDSA (Elliptic Curve Digital Signature Algorithm).**

Klasy

Następujące klasy implementują wspomniane algorytmy:

- **AsymmetricAlgorithm** - klasa abstrakcyjna, z której dziedziczą wszystkie implementacje, algorytmów asymetrycznych,
- **DSACryptoServiceProvider**,
- **RSACryptoServiceProvider**,
- **ECDiffieHellman**,
- **ECDiffieHellmanCng**,
- **ECDiffieHellmanCngPublicKey**,
- **ECDsaCng**.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Przykłady

Szyfrowanie danych algorytmem RSA

```
byte[] dane = { 1, 2, 3, 4, 5 };

using (var rsa = new RSACryptoServiceProvider(2048))
{
    byte[] zaszyfrowane = rsa.Encrypt(dane, true);
    byte[] odszyfrowane = rsa.Decrypt(zaszyfrowane, true);
}
```

Export kluczy do pliku

```
using (var rsa = new RSACryptoServiceProvider())
{
    File.WriteAllText("PublicKeyOnly.xml",
                      rsa.ToXmlString(false));
    File.WriteAllText("PublicPrivate.xml",
                      rsa.ToXmlString(true));
}
```

(Klucze można też wyexportować do tablicy bajtów metodą *ExportCspBlob*.)

Szyfrowanie z użyciem klucza publicznego i odszyfrowywanie z użyciem prywatnego

```
byte[] dane = Encoding.UTF8.GetBytes("Tekst do zaszyfrowania");

string publicKeyOnly = File.ReadAllText("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText("PublicPrivate.xml");

byte[] zaszyfrowane, odszyfrowane;

using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
    rsaPublicOnly.FromXmlString(publicKeyOnly);
    zaszyfrowane = rsaPublicOnly.Encrypt(data, true);
}

using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
    rsaPublicPrivate.FromXmlString(publicPrivate);
    odszyfrowane = rsaPublicPrivate.Decrypt(zaszyfrowane, true);
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Funkcje jednokierunkowe

Algorytmy

Mamy też dostęp do wielu algorytmów podpisu. W szczególności zaimplementowane zostały następujące „bezkłuczowe” algorytmy funkcji skrótów:

- **MD5** - algorytm *Message Digest* z podpisem o długości 128 bitów;
- **MD160** - algorytm *Message Digest* z podpisem o długości 160 bitów;
- **SHA1** - algorytm *Secure Hash Algorithm 1* z podpisem o długości 160 bitów;
- **SHA256** - algorytm *Secure Hash Algorithm 256* z podpisem o długości 256 bitów;
- **SHA384** - algorytm *Secure Hash Algorithm 384* z podpisem o długości 384 bitów;
- **SHA512** - algorytm *Secure Hash Algorithm 512* z podpisem o długości 512 bitów

oraz następujące „kluczowe” algorytmy funkcji skrótów:

- **HMACSHA1** - algorytm *Hash-based Message Authentication Code* używający algorytmu *SHA1* i generujący skrót o długości 20 bajtów;
- **MACTripleDES** - algorytm *Message Authentication Code* używający algorytmu *TripleDES* i generujący skrót o długości 8 bajtów.

Klasy

Następujące klasy (odpowiednio abstrakcyjne i instancyjne) implementują powyżej wymienione algorytmy:

- **MD5, MD5CryptoServiceProvider;**
- **RIPemd160, RIPEMD160Managed;**
- **SHA1, SHA1CryptoServiceProvider;**
- **SHA256, SHA256Managed;**
- **SHA384, SHA384Managed;**
- **SHA512, SHA512Managed;**
- **HMACSHA1;**
- **MACTripleDES.**

Przykłady

Użycie funkcji skrótu MD5 na pliku

```
byte[] hash;
using (Stream fs = File.OpenRead("plik.txt"))
    hash = MD5.Create().ComputeHash(fs);
```

Użycie funkcji skrótu SHA256 na ciągu znaków

```
byte[] haslo = System.Text.Encoding.UTF8.GetBytes(
    "M0jeCha$1o");
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
byte[] hash = SHA256.Create().ComputeHash(data);
```

Inny wariant użycia funkcji skrótu MD5 na pliku

```
MD5 hash = new MD5CryptoServiceProvider();
FileStream plik = new FileStream("plik.txt",
    FileMode.Open, FileAccess.Read);
BinaryReader reader = new BinaryReader(plik);
hash.ComputeHash(reader.ReadBytes((int)plik.Length));
Console.WriteLine(Convert.ToString(hash.Hash));
```

Podpisy

Klas używanych do szyfrowania można również używać do podpisywania informacji, np.:

```
byte[] dane = Encoding.UTF8.GetBytes("Wiadomość do podpisu");
byte[] kluczPubliczny;
byte[] podpis;
object fSkrotu = SHA1.Create();

using (var publicPrivate = new RSACryptoServiceProvider())
{
    podpis = publicPrivate.SignData(dane, fSkrotu);
    kluczPubliczny = publicPrivate.ExportCspBlob(false);
}

using (var publicOnly = new RSACryptoServiceProvider())
{
    publicOnly.ImportCspBlob(kluczPubliczny);
    Console.WriteLine(publicOnly.VerifyData(dane, fSkrotu, podpis)); // true

    dane[0] = 0;
    Console.WriteLine(publicOnly.VerifyData(dane, fSkrotu, podpis)); // false

    podpis = publicOnly.SignData(dane, fSkrotu); // błąd uruchomienia
                                                // (konieczny klucz prywatny)
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

20. Integracja kodu zarządzanego i niezarządzanego (Interoperability)

Ten rozdział poświęcony jest mechanizmom pozwalającym na integrację kodu zarządzanego (np. skompilowanych programów napisanych w języku C#) z niezarządzanym (np. natywnymi bibliotekami (COM) DLL).

Wybrane różnice pomiędzy kodem zarządzanym i niezarządzanym

Na początku przedstawione zostały wybrane zagadnienia dotyczące różnic pomiędzy kodem zarządzanym i niezarządzanym.

Podstawowe podobieństwa i różnice pomiędzy bibliotekami

Należy pamiętać o tym, że istnieją zarówno podobieństwa, jak i różnice pomiędzy bibliotekami niezarządzalnymi (w natywnym kodzie maszynowym systemu Windows) i bibliotekami w kodzie zarządzanym (w języku pośrednim).

Każda biblioteka, aby mogła zostać użyta, musi być albo zarejestrowana w systemie operacyjnym albo musi znajdować się w ścisłe określonym miejscu, spełniającym pewne szczególne warunki albo musieć być w specjalny sposób wskazana (np. programy zwykle mogą się odwoływać do bibliotek zlokalizowanych w bieżącym katalogu, programy działające w trybie deweloperskim mogą używać bibliotek zarządzalnych zlokalizowanych w katalogu wskazanym przez zmienną DEVPATH).

W przypadku bibliotek niezarządzalnych (np. dynamiczne biblioteki COM) każda biblioteka może być zarejestrowana w tylko jednej wersji. Podczas instalacji nowego oprogramowania nowe wersje bibliotek zastępują stare, co może powodować błędy działania wcześniej zainstalowanego oprogramowania, które korzystało z wcześniejszych wersji bibliotek.

W przypadku bibliotek zarządzalnych (.NET Framework) powyższy problem nie istnieje. Każda biblioteka może być zainstalowana i zarejestrowana (w GAC – *Global Assembly Cache*) w wielu wariantach – różniącących się nie tylko numerami wersji, ale też np. ustawieniami regionalnymi).

Różnice w implementacji metod

Chcąc integrować kod zarządzany i niezarządzany należy pamiętać o różnicach w implementacji w każdym z tych podejść.

W szczególności komponenty COM nie wspierają przeciążania parametrów, zatem kod zarządzalny, który ma być używany w kodzie niezarządzalnym, nie może zawierać przeciążonych metod.

Z kolei język C# w wersjach wcześniejszych niż 4.0 nie posiadał wsparcia dla opcjonalnych parametrów. Z tego powodu przy używaniu metod obiektów COM należało podawać pełną listę argumentów (nawet tych, które nie były używane). Często metody takie posiadały kilkanaście lub kilkadziesiąt parametrów, z których większość była opcjonalna. Aby podnieść czytelność wywołań takich metod wprowadzono specjalny typ argumentu **Type.Missing**, który można było podawać zamiast „sztucznych” parametrów.

Od wersji 4.0 .NET Framework nie trzeba podawać nieużywanych parametrów. Podawane parametry można wówczas specyfikować po nazwie (w postaci *nazwa: wartość*).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Obsługa wyjątków

W .NET Frameworku wyjątki reprezentowane są przez klasy dziedziczące po klasie *Exception*. Natomiast w kodzie niezarządzalnym wyjątki reprezentowane są przez kody błędów. Zwróćmy uwagę na to, że wyjątki które zostaną wyrzucone w kodzie niezarządzalnym nie będą reprezentowane w kodzie zarządzalnym przez obiekty dziedziczące po klasie *Exception* (ponieważ w ogóle nie mogą być reprezentowane w przestrzeni CTS (*Common Type System*)). Z tego powodu we wcześniejszych wersjach .NET Frameworka przechwycenie wyjątków z kodu niezarządzalnego było możliwe wyłącznie z pustych (bezargumentowych) klauzulach *catch* i nie było możliwe sprawdzenie informacji o przyczynie wygenerowania wyjątku (nie była przekazywana informacja o kodzie błędu).

W wersjach .NET Frameworka począwszy od 2.0 problem ten rozwiązyany został przez opakowywanie kodów błędów w klasie *Exception* (dokładniej przez obiekt *RuntimeWrappedException* z przestrzeni *System.Runtime.CompilerServices*). Kod błędu można pobrać z wyjątku poprzez własność *WrappedException*.

```
try
{
    // Wystąpienie jakiegoś wyjątku
}
catch (Exception ex)
{
    // Framework przed 2.0:
    //     przechwytywanie tylko wyjątków z CLS
    // Framework od 2.0:
    //     domyślnie zarówno CLS jak i nie CLS
}
catch
{
    // Wszystkie wyjątki
}
```

Nadal jednak można przywrócić poprzedni sposób obsługi wyjątków poprzez atrybut *RuntimeCompatibility*:

```
using System.Runtime.CompilerServices;
[assembly: RuntimeCompatibility(
    WrapNonExceptionThrows = false)]
```

Ograniczenia komponentów COM

Poniżej zestawione zostały ograniczenia komponentów COM (niezarządzalnych), które należy brać pod uwagę planując integrację tworzonego kodu zarządzalnego z komponentami COM:

- nie posiadają one wsparcia dla statycznych i współdzielonych elementów (co uniemożliwia używanie tego typu elementów (pół metod i właściwości)),
- nie pozwalają na przekazywanie argumentów w konstruktorach (co pociąga konieczność używania wyłącznie bezargumentowych konstruktorów)
- elementy dziedziczone z klasy bazowej nie są rozpoznawane (co ogranicza możliwość używania dziedziczenia),

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- mają ograniczoną użyteczność (nie można z nich korzystać) w systemach bez rejestrów (innych niż Windows) (co powoduje ograniczenia w ich przenośności).

Ograniczenia w assembly .NET używanych w komponentach COM

W związku z powyższymi ograniczeniami stosowny kod zarządzalny musi być tworzony zgodnie z poniższymi wytycznymi:

- Wszystkie klasy muszą używać domyślnego konstruktora bezargumentowego.
- Każdy typ, który ma być użyty w COM musi być publiczny.
- Wszystkie elementy klasy przeznaczone do użycia w COM muszą być publiczne.
- Nie można korzystać z klas abstrakcyjnych.

Najpopularniejsze narzędzia

Lista narzędzi

Poniżej podane zostało zestawienie najpopularniejszych narzędzi służących do komplilacji, konwersji, rejestracji i analizy kodu zarówno zarządzalnego, jak i niezarządzalnego:

- **TlbImp.exe** – „Type Library Importer” (służy do „konwersji” niezarządzalnych bibliotek COM do zarządzalnych bibliotek .NET-owych);
- **TlbExp.exe** - „Type Library Exporter” (służy do „konwersji” zarządzalnych bibliotek .NET-owych do niezarządzalnych bibliotek COM);
- **RegAsm.exe** – „Assembly Registration Tool” – służy do rejestrowania zarządzalnych bibliotek w GAC (Global Assembly Cache)
- **RegSvr32.exe** - narzędzie do rejestracji i wyrejestrowywania OLE (niezarządzalnych kontrolek);
- **RegEdit.exe** - „Registry Editor” (edytor rejestru – narzędzie służące do podglądu i edycji rejestru systemu);
- **Ildasm.exe** – „Intermediate Language Disassembler” (narzędzie służące do analizy skompilowanego kodu zarządzalnego);
- **Rc.exe** - „Resource Compiler” (kompilator zasobów);
- **Csc.exe** - „C# Compiler” (kompilator języka C# - dostarczany jako element .NET Framework).

Przy czym w powyższym zestawianiu przez sformułowanie „konwersja” należy rozumieć raczej „dostosowanie” do użycia w zadanym środowisku niż właściwą konwersję.

Przykłady użycia

Przykład rejestracji COM DLL przez RegSvr32.exe

```
regsvr32 MyDLL.dll
```

Przykłady użycia TlbImp.exe

- import z **MyDLL.tlb** do **MyDLL.dll**

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
tlbimp MyDLL.tlb
```

- import z **MyCOM.dll** do **MyNET.dll**

```
tlbimp MyCOM.dll /out:MyNET.dll
```

- import z **MyCOM.dll** do **MyCOM.dll**

```
tlbimp MyCOM.dll
```

- translacja na wariant bez *out* i *retval*

```
tlbimp L1.dll /transform:dispret /out:L2.dll
```

Przykład użycia **TlbExp.exe**

```
csc /t:library MyApp.cs
tlbexp MyApp.dll /out:mytypelib.tlb
echo IDR_TYPELIB1 typelib "mytypelib.tlb" > myresource.rc
rc /v myresource.rc
csc /t:library MyApp.cs /win32res:myresource.res
```

Przykłady rejestrowania .NET DLL dla COM przez **RegAsm.exe**

```
regasm MyDLL.dll
```

```
regasm MyDLL.dll /regfile:MyDLL.reg
```

```
regasm MyDLL.dll /tlb:MyDLL.tlb
```

Ukrywanie zawartości assembly poprzez atrybut **ComVisible**

Do ukrywania klas kodu zarządzalnego i ich składowych, aby nie były one widoczne z kodu niezarządzalnego służy atrybut **ComVisible** klasy **ComVisibleAttribute** z przestrzeni **System.Runtime.InteropServices**.

Można za jego pomocą domyślnie ukryć całą zawartość assembly.

```
[assembly: ComVisible(false)]
```

Można też ukrywać

```
[ComVisible(false)]
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

i „pokazywać”

[ComVisible(true)]

poszczególne klasy i ich elementy.

Przykład

```
using System.Runtime.InteropServices;
[assembly: ComVisible(false)]

[ComVisible(false)]
class MyClass
{
    public MyClass()
    {
        //...
    }

    [ComVisible(false)]
    public int MyMethod(string parameter)
    {
        return 0;
    }

    public bool MyOtherMethod()
    {
        return true;
    }

    [ComVisible(false)]
    public int MyProperty
    {
        get
        {
            return MyProperty;
        }
    }
}
```

Platform Invoke

Platform Invoke (P/Invoke) jest usługą pozwalającą na wywoływanie z kodu zarządzanego funkcji zaimplementowanych w bibliotekach dynamicznych (DLL).

Ogólna koncepcja

Wywoływanie funkcji przez *Platform Invoke* odbywa się w następujących etapach:

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- Stworzenie nowej statycznej/współdzielonej zewnętrznej metody (modyfikatory **static** i **external**) o nazwie funkcji, która ma być wywołana.
- Oznaczenie tej metody atrybutem **DllImport**, przy którym należy wskazać bibliotekę DLL, która ma być użyta.
- Wywołanie metody (z kodu zarządzalnego).

Zalety enkapsulacji

Zwykle zaleca się przeprowadzenie enkapsulacji funkcji wywoływanych z niezarządzalnej biblioteki DLL poprzez umieszczenie ich w specjalnie w tym celu utworzonej klasie. Rozwiązanie to ma następujące zalety:

- Użytkownicy stworzonej w ten sposób klasy nie muszą wiedzieć o innym kodzie, który jest przez nią używany.
- Zwalnia to programistę z obowiązku pamiętania nazw i parametrów wywołań API. Raz utworzone metody można wywoływać tak jak wszystkie inne metody .NET.
- To podejście jest bardziej „odporne na błędy”. Drobne literówki w wywołaniach *Platform Invoke* powodują zwykle poważne błędy.

Zalecany schemat użycia

Biorąc pod uwagę powyższe argumenty można zakreślić następujący schemat zalecanego użycia *Platform Invoke*:

- Wskazanie funkcji w DLL (w najprostszym przypadku podanie nazwy funkcji oraz nazwy biblioteki DLL, w której jest ona zawarta).
- Stworzenie klasy do przechowywania funkcji DLL (można użyć istniejącej klasy, stworzyć osobne klasy dla poszczególnych funkcji lub jedną zbiorczą klasę dla wszystkich powiązanych niezarządzalnych funkcji).
- Utworzenie prototypu(ów) w kodzie zarządzalnym (używając **DllImportAttribute** wskazujemy stosowną funkcję; prototyp musi mieć modyfikatory **static** i **external**).
- Wywołanie funkcji z DLL (funkcje wywoływanie są z kodu zarządzalnego tak, jak pozostałe funkcje; tym niemniej w szczególnych przypadkach mogą być przekazywane struktury oraz mogą być implementowane funkcje powrotu (o tym będzie mowa w dalszej części tego rozdziału)).

Przykłady

Poniżej podany został jeden z najprostszych przykładów użycia Platform Invoke. W programie tym wyświetlane jest (poprzez metodę *MessageBox()* z biblioteki systemowej *user32.dll*) systemowe okienko dialogowe o zawartości „Witaj świecie” i nagłówku „Przykład na wywołanie Platform Invoke”:

```
using System;
using System.Runtime.InteropServices;

public class Win32
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern int MessageBox(int hWnd,
        String text, String caption, uint type);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

}

public class HelloWorld
{
    public static void Main()
    {
        Win32.MessageBox(0, "Witaj świecie",
                        "Przykład na wywołanie Platform Invoke", 0);
    }
}
  
```

Następujący przykład zaczerpnięty z książki [15] jest nieco bardziej złożony. Używane są w nim dwie metody z tej samej biblioteki: *GetForegroundWindow()* pobierająca wskaźnik do bieżącego (aktywnego) okna oraz *GetWindowText()* pobierająca tekst nagłówka z okna, na które wskazuje przekazany w argumencie wskaźnik. Ponadto utworzona została metoda *GetScreenDemo()*, która z używając powyższych funkcji w konsoli tekstowej wypisuje nagłówek okna aktywnej konsoli tekstowej:

```

using System;
using System.Text;
using System.Runtime.InteropServices;

class WindowHeader
{
    private const Int32 BufferSize = 256;

    [DllImport("user32.dll")]
    private static extern IntPtr GetForegroundWindow();

    [DllImport("user32.dll")]
    private static extern Int32 GetWindowText(IntPtr hWnd,
                                             StringBuilder textView, Int32 counter);

    public static void GetScreenDemo()
    {
        StringBuilder DemoBuilder = new StringBuilder(BufferSize);
        IntPtr DemoHandle = GetForegroundWindow();
        if (GetWindowText(DemoHandle, DemoBuilder, BufferSize) > 0)
        {
            Console.WriteLine(DemoBuilder.ToString());
        }
    }
}

public class WindowHeaderExample
{
    public static void Main()
    {
        WindowHeader.GetScreenDemo();
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Konwersja typów

Przy wywoływaniu kodu niezarządzalnego z kodu zarządzalnego oraz na odwrót często zachodzi potrzeba przekazywania danych pomiędzy nimi, a to z kolei powoduje konieczność dokonywania translacji (konwersji typów) pomiędzy nimi.

Wyróżnić możemy dwa przypadki konwersji danych

- W kodzie zarządzalnym. Może ona zostać przeprowadzana np. przez klasę **TypeConverter**.
- Dla kodu niezarządzalnego: przez atrybut **MarshalAs**. Zaletą tego rozwiązania jest to, że jest ono wspierane przez *Microsoft IntelliSense*.

W poniższych rozważaniach zajmiemy się tym drugim mechanizmem, jako istotniejszym i bardziej specyficzny. Polega on na wskazaniu sposobu ułożenia informacji w przekazywanych danych i będziemy go nazywać **marshalingiem**. Materiał ten został opracowany na podstawie [15] i wszystkie podane w nim przykłady zostały przytoczone (czasami z pewnymi drobnymi zmianami) z tej pozycji.

Przykłady marshalingu danych

Poniższy przykład pokazuje użycie atrybutu *MarshalAs*, aby wymusić sposób ułożenia danych w wybranych polach klasy *MarshalAsDemo*.

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

namespace NetForComDemoCS
{
    class MarshalAsDemo
    {
        [MarshalAs(UnmanagedType.LPStr)]
        public String FirstName;
        public String LastName;
        [MarshalAs(UnmanagedType.Bool)]
        public Boolean IsCurrentlyWorking;
    }
}
```

Atrybut *MarshalAs* można stosować też dla własności oraz metod. Następny przykład (przytoczony z [15]) pokazuje, w jaki sposób można przekształcić własność (tu *LastName*) na metodę pobierającą wartość (tu pobierającą argument typu *string* o nazwie *firstName*).

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
namespace NetForComDemoCS
{
    class MarshalAsDemo
    {
        [MarshalAs(UnmanagedType.LPStr)]
        public String FirstName;
        public String LastName(

```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    [MarshalAs(UnmanagedType.LPStr)] String firstName) { }
    [MarshalAs(UnmanagedType.Bool)]
    public Boolean IsCurrentlyWorking;
}
}
  
```

Kontrola sposobu ułożenia danych w pamięci

Sposób ułożenia w pamięci danych można wskazywać poprzez obiekty klasy **StructLayoutAttribute**. Klasa ta posiada następującą budowę:

- Konstruktor, któremu można w argumencie przekazać wariant rozkładu w pamięci (będzie do później opisane dokładniej).
- Pola (wszystkie publiczne)
 - **CharSet** - określa domyślny sposób układania w pamięciłańców (String) (**LPWSTR** lub **LPSTR**; domyślnie **LPSTR**).
 - **Pack** - określa ułożenie w pamięci pól klasy lub struktury.
 - **Size** - określa całkowity rozmiar struktury lub klasy.
- Właściwości (wszystkie publiczne)
 - **TypeId** - w klasie pochodnej określa identyfikator dla **Attribute**.
 - **Value** - zwraca wartość (typu **LayoutKind**) określającą wariant ułożenia w pamięci.
- Metody (wszystkie publiczne): **Equals()**, **GetCustomAttribute()**, **GetCustomAttributes()**, **GetHashCode()**, **GetType()**, **IsDefaultAttribute()**, **IsDefined()**, **Match()**, **ReferenceEquals()**, **ToString()**.

Dopuszczalnymi argumentami konstruktora **StructLayoutAttribute** mogą być albo poniższe elementy typu wyliczeniowego **LayoutKind** albo odpowiadające im wartości typu **Int16**:

- **LayoutKind.Auto** - przerzucenie kontroli nad ułożeniem danych na CLR (domyślny dla typów referencyjnych).
- **LayoutKind.Sequential** - wymusza na CLR zachowanie układu danych zgodnego z zaimplementowanym przez programistę (domyślny dla typów opartych na wartościach).
- **LayoutKind.Explicit** - pozwala na bezpośrednie wskazanie CLR ułożenia danych przez podanie przesunięć adresów pamięci poszczególnych pól.

Przykład przekazywania struktur

Poniższy przykład przedstawia typowy sposób użycia atrybutu **StructLayout**.

```

using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point {
    public int x;
    public int y;
}

[StructLayout(LayoutKind.Explicit)]
public struct Rect {
    [FieldOffset(0)] public int left;
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
[FieldOffset(4)] public int top;
[FieldOffset(8)] public int right;
[FieldOffset(12)] public int bottom;
}
```

Używanie funkcji powrotu

Pierwotnie funkcje powrotu były implementowane poprzez wskaźniki. Podejście to dawało duże możliwości, ale wiązały się też z nim niebezpieczeństw.

W .NET Frameworku funkcje powrotu przekazuje się przez delegacje.

Funkcje powrotu z kodu niezarządzalnego implementuje się (przez delegacje) w następujący sposób:

- Tworzy się obiekt delegacji o tej samej sygnaturze, co funkcja powrotu.
- Podstawia się delegację jako funkcję powrotu i wywołuje się funkcję (kodu niezarządzalnego).

Poniższy przykład z pracy [15] pokazuje w jaki sposób można używać funkcji powrotu.

```
using System;
using System.Text;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
namespace NetForComDemoCS
{
    public class UnmanagedCallbackDemo
    {
        public delegate Boolean DemoCallback(IntPtr hWnd, Int32 lParam);
        private const String UserReference = "user32.dll";
        private const Int32 BufferSize = 100;

        [DllImport(UserReference)]
        public static extern Int32 EnumWindows(DemoCallback callback,
                                                Int32 param);

        [DllImport(UserReference)]
        public static extern Int32 GetWindowText(IntPtr hWnd,
                                                StringBuilder lpString,
                                                Int32 nMaxCount);

        public static Boolean DisplayWindowInfo(IntPtr hWnd, Int32 lParam)
        {
            StringBuilder DemoBuilder = new StringBuilder(BufferSize);
            if (GetWindowText(hWnd, DemoBuilder, BufferSize) != 0)
            {
                Console.WriteLine("Demo Output: " + DemoBuilder.ToString());
            }
            return true;
        }
        public static void RunDemo()
        {
            EnumWindows(DisplayWindowInfo, 0);
            Console.WriteLine("Beginning process... ");
            Console.ReadLine();
        }
    }
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Pobieranie kodów błędów

Poniższy przykład pokazuje jak w kodzie zarządzalnym można pobrać kod błędu, który wystąpił w wywołanym z niego kodzie niezarządzalnym:

```

using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

namespace NetForComDemoCS
{
    public class UnmanagedErrorDemo
    {
        private const String KernelReference = "kernel32.dll";
        private const String UserReference = "user32.dll";
        private const Int32 MessageSize = 255;

        [DllImport(KernelReference)]
        private static extern Int32 FormatMessage(Int32 dwFlags,
            Int32 lpSource, Int32 intdwMessageId, Int32 dwLanguageId,
            ref String lpBuffer, Int32 nSize, Int32 Arguments);

        [DllImport(UserReference, SetLastError = true)]
        private static extern Int32 MessageBox(IntPtr hWnd,
            String pText, String pCaption, Int32 uType);

        public static void ThrowMessageBoxException()
        {
            IntPtr ProblemCauser = (IntPtr)(-100);
            MessageBox(ProblemCauser, "This won't work",
                "Caption - This won't work", 0);
            Int32 ErrorCode = Marshal.GetLastWin32Error();
            Console.WriteLine("Error Code: " + ErrorCode.ToString());
            Console.WriteLine("Real Error: " +
                GetLastErrorMessage(ErrorCode));
        }

        public static String GetLastErrorMessage(Int32 errorValue)
        {
            // This order doesn't matter but should be kept
            // for logical consistency
            Int32 FORMAT_MESSAGE_ALLOCATE_BUFFER = 0x00000100;
            Int32 FORMAT_MESSAGE_IGNORE_INSERTS = 0x00000200;
            Int32 FORMAT_MESSAGE_FROM_SYSTEM = 0x00001000;

            String lpMsgBuf = String.Empty;

            Int32 dwFlags = FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS;
            Int32 ReturnValue = FormatMessage(dwFlags, 0, errorValue,
                0, ref lpMsgBuf, MessageSize, 0);

            if (ReturnValue == 0)
            {

```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        return null;
    }
    else
    {
        return lpMsgBuf;
    }
}
```

Klasa Marshal

Kolejnym użytecznym mechanizmem pozwalającym na wykonywanie w kodzie zarządzalnym operacji związanych z kodem niezarządzalnym jest klasa ***Marshal*** z przestrzeni nazw ***System.Runtime.InteropServices***. Zawiera ona zbiór metod przeznaczonych do

- alokacji niezarządzalnej pamięci,
 - kopiowania bloków niezarządzalnej pamięci,
 - konwersji niezarządzalnych typów

oraz innych zastosowań. Jest ona też między innymi wrapperem dla pewnych błędów.

Poniżej zaprezentowane zostały (zaczerpnięte książki [15]) przykładowe sposoby użycia tej klasy.

```
using System;
using System.Text;
using System.Runtime.InteropServices;

public struct Point
{
    public Int32 x, y;
}

public sealed class App
{
    static void Main()
    {
        // Demonstrate the use of public static fields of the Marshal class.
        Console.WriteLine("SystemDefaultCharSize={0}, SystemMaxDBCSCharSize={1}",
                          Marshal.SystemDefaultCharSize, Marshal.SystemMaxDBCSCharSize);

        // Demonstrate the use of the SizeOf method of the Marshal class.
        Console.WriteLine("Number of bytes needed by a Point object: {0}",
                          Marshal.SizeOf(typeof(Point)));
        Point p = new Point();
        Console.WriteLine("Number of bytes needed by a Point object: {0}",
                          Marshal.SizeOf(p));
        // Demonstrate how to call GlobalAlloc and
        // GlobalFree using the Marshal class.
        IntPtr hglobal = Marshal.AllocHGlobal(100);
        Marshal.FreeHGlobal(hglobal);

        // Demonstrate how to use the Marshal class to get the Win32 error
        // code when a Win32 method fails.
        Boolean f = CloseHandle(new IntPtr(-1));
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

if (!f)
{
    Console.WriteLine("CloseHandle call failed with an error code of: {0}",
        Marshal.GetLastWin32Error());
}
}

// This is a platform invoke prototype. SetLastError is true, which allows
// the GetLastWin32Error method of the Marshal class to work correctly.
[DllImport("Kernel32", ExactSpelling = true, SetLastError = true)]
static extern Boolean CloseHandle(IntPtr h);
}

```

Podsumowanie

Używanie kodu niezarządzalnego wewnętrz kodu zarządzalnego może wiązać się z szeregiem problemów. Wśród najistotniejszych z nich można wymienić następujące cztery kwestie:

- **Wydajność (performance).** Zasadniczo kod, który nie jest zarządzalny powinien działać przynajmniej tak samo szybko jak kod zarządzalny (a nawet szybciej). Spadek wydajności może pojawiać się przy konieczności przekazywania danych pomiędzy kodem zarządzalnym i nie zarządzalnym. Ponadto w kodzie nie zarządzalnym łatwiej mogą pojawić się luki takie jak np. wycieki pamięci.
- **Bezpieczeństwo typów (type safety).** Niezarządzalny kod czasami nie wspiera bezpieczeństwa typów (nie jest „type safe”) (przez co jest mniej czytelny, może powodować problemy związane z niewłaściwym rzutowaniem typów itp.). Na dodatek nie ma gwarancji, że definicje z typów bibliotecznych są właściwe.
- **Mechanizmy bezpieczeństwa kodu (code security).** Brak zgodności z modelem bezpieczeństwa z .NET Frameworka.
- **System wersjonowania (versioning).** Systemy wersjonowania kodu niezarządzalnego i zarządzalnego różnią się w istotny sposób. W szczególności kod zarządzalny może posiadać jednocześnie zarejestrowanych wiele wersji tej samej biblioteki (a czasem nawet o tym samym numerze wersji, ale różniących się np. wariantem ustawień regionalnych), podczas gdy w kodzie niezarządzalnym może być jednocześnie zainstalowana tylko jedna wersja każdej biblioteki.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

21. Refleksje

Niniejszy rozdział poświęcony jest dokładniejszemu przedstawieniu budowy asemblacji oraz wyjaśnieniu roli refleksji i pokazaniu przykładów ich używania.

Assembly

Przypomnijmy, że przez *assembly* lub *asemblacje* nazywamy programy (np. napisane w języku C# lub innych językach wspieranych przez .NET Framework) skompilowane do postaci binarnej. Przykładami asemblacji są pliki uruchomieniowe oraz biblioteki dynamiczne. Przypomnijmy też, że pliki takie są samoopisywalne, co oznacza, że każdy taki plik zawiera informacje o znajdujących się w nim typach (klasach i strukturach). Część asemblacji zawierająca takie informacje nazywana jest **manifestem**.

Podstawowa charakterystyka Assembly

Asemblacje mogą różnić się między sobą w wielu aspektach, między innymi sposobem użycia, budową, czy formatami. Wszystkie one jednak posiadają następujące cechy wspólne:

- Asemblacja zawiera manifest i metadane, które opisują jej budowę.
- Każda asemblacja posiada czteroczęściowy numer wersji składający się z następujących członów oddzielonych kropkami
 - numer główny (ang. *major*),
 - numer poboczny (ang. *minor*),
 - numer budowy (ang. *build*),
 - numer rewizji (ang. *revision*)*(major.minor.build.revision)*.
- Asemblacje posiadają wsparcie dla mechanizmów konfiguracyjnych (przedstawionych w rozdziale 10).
- Asemblacje w postaci bibliotek mogą być w formatach **.exe** i **.dll**.
- Asemblacja jest niezależna od języka, w którym została napisana.

Budowa Assembly jednoplikowego

Assembly może składać się z jednego pliku. Wówczas taki plik zawiera w sobie następujące elementy:

- manifest (metadane assembly);
- metadane typów;
- kod w języku pośrednim (IL/CIL/MSIL);
- opcjonalne zasoby (np. załączona ikonka pliku).

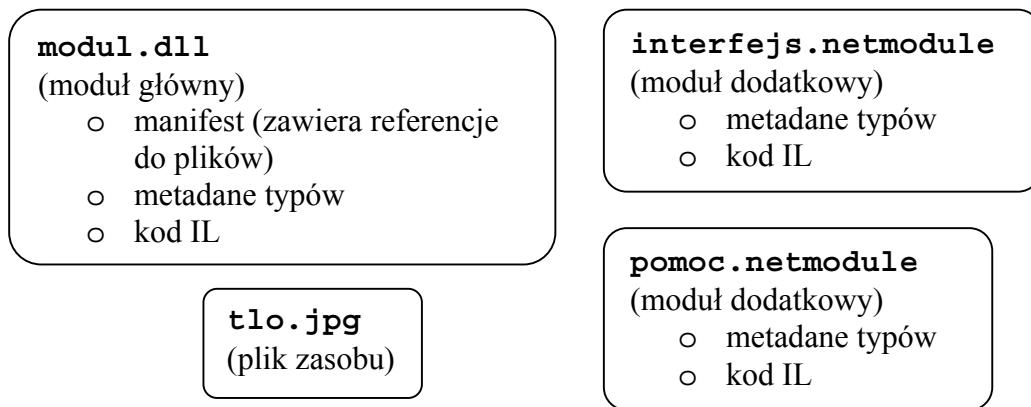
Podstawowe informacje o Assembly wieloplikowych

Assembly wieloplikowe jest zbiorem plików, które zostały zainstalowane z tym samym (czteroczęściowym) numerem wersji. Zawiera ono jeden **moduł podstawowy** zwany też **modułem głównym**. Moduł ten zawiera **manifest assembly** oraz informacje o pozostałych plikach. Oprócz modułu

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

głównego assembly wieloplikowe zawiera **moduły dodatkowe**, które są logicznie powiązane z informacjami z manifestu z głównego modułu. Moduły dodatkowe mają rozszerzenie **.netmodule**. Poza tym assembly wieloplikowe może zawierać dodatkowo pliki zasobów.

Poniższy przykład przedstawia przykładową zawartość assembly wieloplikowego



Należy przy tym zaznaczyć, że swój „typ” posiadają wszystkie „elementy programistyczne” umieszczane w programie. Można w śród nich wyróżnić nie tylko *klasy* i typy wartościowe (ang. *value types*, w szczególności typy wyliczeniowe i struktury), ale również interfejsy.

Refleksje

Rola refleksji

Refleksje są mechanizmem pozwalającym na wykonywanie następujących dwóch rodzajów operacji:

- Określanie typów i ich składowych asemblacji (poprzez pobieranie metadanych z assembly). W skrajnych przypadkach umożliwia to deasembację programów, czyli odtworzenie ich kodu źródłowego. Oczywiście odbywa się to z dokładnością do nazw zmiennych lokalnych i innych podobnych elementów.
- Dynamiczne generowanie kodu (tworzenie asemblacji).

Klasa Type

Podstawowym narzędziem pozwalającym na pobranie informacji o danym typie jest klasa **Type**.

Wybrane metody

Klasa ta posiada szereg metod pozwalających na pobranie informacji o składowych typu. Wśród nich są między innymi:

- **FindInterfaces** – zwraca tablicę obiektów reprezentujących listę interfejsów, które zostały zaimplementowane w typie lub w nim odziedziczone;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **FindMembers** – zwraca tablicę obiektów (typu *MemberInfo*) opisujących elementy składowe typu (struktury, klasy, interfejsu);
- **GetArrayRank** – zwraca liczbę wymiarów tablicy;
- **GetConstructor** – pobiera zadany konstruktor;
- **GetConstructors** – pobiera tablicę konstruktorów;
- **GetDefaultMembers** – pobiera tablicę składowych typu z ustawionym atrybutem *DefaultMemberAttribute*;
- **GetEvent** – pobiera zadane zdarzenie;
- **GetEvents** – pobiera tablicę zdarzeń;
- **GetField** – pobiera zadane pole;
- **GetFields** – pobiera tablicę pól;
- **GetInterface** – pobiera zadany interfejs (implementowany lub dziedziczący w zadanym typie);
- **GetInterfaceMap** – zwraca mapowanie interfejsu dla zadanego interfejsu;
- **GetInterfaces** – pobiera tablicę interfejsów (implementowanych lub odziedziczonych);
- **GetMember** - pobiera zadany składowych (własność, metodę, pole, zdarzenie itp.) typu;
- **GetMembers** – pobiera tablicę składowych;
- **GetMethod** – pobiera zadaną metodę;
- **GetMethodImpl** – pobiera zadaną implementację zadanej metody (virtualnej);
- **GetMethods** - pobiera tablicę metod;
- **GetNestedType** – pobiera zadany typ zagnieżdżony (w danym typie);
- **GetNestedTypes** - pobiera tablicę typów zagnieżdżonych;
- **GetProperties** – pobiera tablicę własności;
- **GetProperty** – pobiera zadaną własność;
- **GetPropertyImpl** – pobiera zadaną implementację zadanej własności (virtualnej);
- **GetType** – pobiera „typ” typu;
- **GetTypeArray** – pobiera typy obiektów przechowywanych w tablicy (dla typu obiektowego);
- **GetTypeFromHandle** – pobiera typ wskazywany przez uchwyt (w przypadku typu uchwytu (ang. *handle*));
- **InvokeMember** – wywołuje zadany element typu (np. metodę lub zdarzenie);
- **IsAssignableFrom** – określa, czy instancja zadanego typu może być przypisana z instancji podanego typu;
- **IsInstanceOfType** – określa czy podany w argumencie obiekt jest instancją bieżącego typu;
- **IsSubclassOf** – określa, czy klasa reprezentowana przez (bieżący) typ dziedziczy z klasy podanej typu podanego w argumencie;
- **MakeArrayType** – zwraca typ obiektu reprezentującego tablicę obiektów bieżącego typu;
- **MemberwiseClone** – tworzy płytka kopię bieżącego obiektu (typu);
- **ReflectionOnlyGetType** - zwraca typ o zadanej nazwie, zwracany typ może być wyłącznie używany do pobrania z niego informacji przez refleksje (nie można go „uruchamiać”);

Wybrane własności

Klasa **Type** posiada też szereg własności, w tym między innymi:

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **Assembly** - zwraca assembly, w którym typ został zadeklarowany (dla typów generycznych assembly, w którym typ generyczny został zdefiniowany);
- **AssemblyQualifiedName** - zwraca nazwę kwalifikowaną typu, zawierającą nazwę assembly z którego typ został załadowany;
- **Attributes** - pobiera atrybuty powiązane z typem;
- **BaseType** - pobiera typ bazowy, z którego bieżący typ bezpośrednio dziedziczy;
- **DeclaringMethod** - zwraca metodę bazową reprezentującą deklarowaną metodę jeśli typ odpowiada typowi parametru metody generycznej;
- **DeclaringType** - pobiera typ który deklaruje bieżący typ zagnieżdzony lub parametr typu generycznego;
- **FullName** zwraca pełną nazwę kwalifikowaną typu zawierającą nazwę przestrzeni nazw, ale nie zawierającą nazwy assembly;
- **IsAbstract** - zwraca wartość określającą, czy typ jest typem abstrakcyjnym i musi zostać z niego utworzony typ pochodny;
- **IsArray** - określa, czy typ jest typem tablicowym;
- **IsAutoLayout** określa, czy dla typu został ustawiony atrybut *AutoLayout*;
- **IsClass** - określa, czy typ jest klasą, co jest równoważne temu, że nie jest to typ wartościowy ani typ interfejsu;
- **IsCOMObject** - określa, czy dany typ jest typem **COM** (zwykle zaimportowanym z kodu niezarządzalnego);
- **IsContextful** - określa, czy dany typ może być osadzony w kontekście;
- **IsEnum** - określa, czy dany typ jest typem wyliczeniowym;
- **IsExplicitLayout** - określa, czy dla typu został ustawiony atrybut *ExplicitLayout*;
- **IsGenericParameter** - określa, czy tym reprezentuje parametr typu w definicji generycznego typu lub generycznej metody;
- **IsGenericType** - określa, czy dany typ jest typem generycznym;
- **IsGenericTypeDefinition** - określa, czy dany typ jest typem generycznym, z którego mogą być konstruowane inne typy;
- **IsImport** - określa, czy dla typu został ustawiony atrybut *ComImportAttribute*, informujący o tym, że typ został importowany z (niezarządzalnej) biblioteki **COM**.
- **IsInterface** - określa, czy dany typ jest interfejsem;
- **IsLayoutSequential** określa, czy dla typu został ustawiony atrybut *SequentialLayout*;
- **IsNested** - określa, czy dany typ jest typem zagnieżdzonym (którego definicja znajduje się wewnątrz definicji innego typu);
- **IsNestedAssembly** - określa, czy dany typ jest typem zagnieżdzonym i jest widoczny tylko wewnątrz własnej asemblacji;
- **IsNestedPrivate** - określa, czy dany typ jest typem zagnieżdzonym i jest zadeklarowany jako typ prywatny;
- **IsNestedPublic** - określa, czy dany typ jest typem zagnieżdzonym i jest zadeklarowany jako typ publiczny;
- **IsNotPublic** - określa, czy dany typ nie jest zadeklarowany jako typ publiczny;
- **IsPointer** - określa, czy dany typ jest wskaźnikiem;

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- **IsPrimitive** - określa, czy dany typ jest jednym z typów pierwotnych;
- **IsPublic** - określa, czy dany typ jest zadeklarowany jako publiczny;
- **IsSealed** - określa, czy dany typ jest zadeklarowany jako zapieczętowany;
- **IsSerializable** - określa, czy dany typ jest serializowalny;
- **IsValueType** - określa, czy dany typ jest typem wartościowym;
- **IsVisible** - określa, czy dany typ jest dostępny z kodu z poza asemblacji;
- **Module** - pobiera moduł (zwykle bibliotekę DLL) w którym typ został zdefiniowany;
- **Name** - pobiera nazwę typu;
- **Namespace** - pobiera przestrzeń nazw typu.

Klasa **ILGenerator**

Klasa **ILGenerator** jest podstawowym narzędziem służącym do generowania kodu wykonywalnego w języku pośrednim. Najczęściej używaną metodą tej klasy jest metoda **Emit**, posiadająca kilkanaście przeciążeń, która generuje i kładzie do strumienia języka pośredniego kompilatora JIT (*just-in-time compiler*) zadaną instrukcję.

Inne klasy związane z refleksami

Wśród wielu innych klas związanych z refleksjami wyróżnić możemy następujące klasy (jeśli nie wymieniono wprost, to są to klasy z przestrzeni *System.Reflections*):

- Klasę abstrakcyjną **MemberInfo** reprezentującą pojedynczą składową typu (np. pole, metodę, konstruktor) oraz jej klasy pochodne takie jak **ConstructorInfo**, **EventInfo**, **FieldInfo**, **MethodInfo**, **PropertyInfo**.
- Inne klasy związane z reprezentacją kodu metod, takie jak **LocalVariableInfo**, **ParameterInfo**, **MethodInfo**, **MethodBase**.
- Typ wyliczeniowy **BindingFlags** reprezentujący modyfikatory dostępu oraz niektóre własności składowych typu.
- Klasy odpowiadające zapisanym na dysku lub znajdującym się w pamięci operacyjnej fragmentom kodu wykonywalnego, jak np. **Assembly** i **Module**.
- Zapieczętowana klasa **Activator** z przestrzeni *System* pozwalająca m.in. na zdalne lub lokalne utworzenie obiektu lub pobranie referencji do istniejącego zdalnego obiektu.

Przykłady

W tej części zaprezentowane zostały przykłady użycia refleksji. Jednym z testowanych w nich typów będzie stworzona przez programistę klasa o poniższej konstrukcji:

```
namespace Refleksje
{
    class Klasa
    {
        public const int Id = 0xFFFF;
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
private static int licznik = 0;

private int liczbaCalkowita;
public int LiczbaCalkowita
{
    get { return liczbaCalkowita; }
    set { liczbaCalkowita = value; }
}

protected double liczbaZmiennoprzecinkowa;
public double LiczbaZmiennoprzecinkowa
{
    get { return liczbaZmiennoprzecinkowa; }
    set { liczbaZmiennoprzecinkowa = value; }
}

string lancuch;
public string Lancuch
{
    set { lancuch = value; }
    private get { return lancuch; }
}

public void Pisz() { Console.WriteLine(lancuch); }

public string ZwrocLancuch() { return lancuch; }

public static int Nastepnik(int i) { return i + 1; }
public int Poprzednik(int i) { return i - 1; }
public Klasa()
{
    lancuch = String.Empty;
    liczbaCalkowita = 0;
    liczbaZmiennoprzecinkowa = 0;
}
public Klasa(string s)
{
    lancuch = s;
    liczbaCalkowita = 0;
    liczbaZmiennoprzecinkowa = 0;
}
}
```

Ponadto w dwóch przykładach użyta zostanie biblioteka klas otrzymana w wyniku komplikacji poniższego kodu.

```
using System;

namespace TestoweAssembly
{
    public class MojaKlasa
    {
        public static int silnia(int n)
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    int i, s = 1;
    for (i = 1; i <= n; i++)
    {
        s *= i;
    }
    return s;
}

public class InnaKlasa
{
    public const int Id = 0xFFFF;

    private static int licznik;

    private int liczbaCalkowita;
    public int LiczbaCalkowita
    {
        get { return liczbaCalkowita; }
        set { liczbaCalkowita = value; }
    }

    protected double liczbaZmiennoprzecinkowa;
    public double LiczbaZmiennoprzecinkowa
    {
        get { return liczbaZmiennoprzecinkowa; }
        set { liczbaZmiennoprzecinkowa = value; }
    }

    string lancuch;
    public string Lancuch
    {
        set { lancuch = value; }
        //private get { return lancuch; }
    }

    public void Pisz() { Console.WriteLine(lancuch); }

    public string ZwrocLancuch() { return lancuch; }

    public static int Nastepnik(int i) { return i + 1; }
    public int Poprzednik(int i) { return i - 1; }
    public InnaKlasa()
    {
        lancuch = String.Empty;
        liczbaCalkowita = 0;
        liczbaZmiennoprzecinkowa = 0;
    }
    public InnaKlasa(string s)
    {
        lancuch = s;
        liczbaCalkowita = 0;
        liczbaZmiennoprzecinkowa = 0;
    }
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

Przykłady będą wykorzystywać przestrzeń nazw **System.Reflection** oraz w wybranych miejscach przestrzeni **System.Reflection.Emit** (do generowania kodu w języku pośrednim).

Odczyt wybranych właściwości typu

Informacje o typie mogą być odczytywane przez odpowiednie jego właściwości. Poniższy przykład przedstawia metodę, która dla zadanego w argumencie typu wypisuje wybrane informacje:

```
static void WyswietlInformacje(Type t)
{
    Console.WriteLine();
    Console.WriteLine("Name: {0}", t.Name);
    Console.WriteLine("FullName: {0}", t.FullName);
    Console.WriteLine("BaseType: {0}", t.BaseType);
    Console.WriteLine("Assembly: {0}", t.Assembly);
    Console.WriteLine("IsPublic: {0}", t.IsPublic);
    Console.WriteLine("IsPrimitive: {0}", t.IsPrimitive);
    Console.WriteLine("IsClass: {0}", t.IsClass);
    Console.WriteLine("IsValueType: {0}", t.IsValueType);
    Console.WriteLine("Wciśnij jakiś klawisz ...");
    Console.ReadKey();
}
```

Metoda ta będzie użyta w następnych przykładach.

Pobieranie typów danych

Najprostszymi sposobami na pobranie typu zadanego wyrażenia jest użycie na nim operatora **typeof** lub wywołanie z niego metody **GetType()**. Należy zwrócić uwagę na to, że w pierwszym z wymienionych przypadków typ jest ustalany podczas komplikacji programu, a w drugim w trakcie jego działania. W poniższym kodzie pokazano przykłady pobierania różnych typów z użyciem obu wymienionych metod.

```
static void InformacjeOTypach()
{
    WyswietlInformacje(typeof(int));
    WyswietlInformacje(typeof(double));
    WyswietlInformacje(typeof(string));
    WyswietlInformacje(typeof(int?));

    Type t1 = DateTime.Now.GetType(); // Typ pobrany w trakcie działania
                                      // programu (runtime)
    WyswietlInformacje(t1);

    Type t2 = typeof(DateTime); // typ określony w momencie komplikacji
    WyswietlInformacje(t2);

    Type t3 = typeof(DateTime[]); // 1-wymiarowy typ tablicowy
    WyswietlInformacje(t3);

    Type t4 = typeof(DateTime[,]); // 2-wymiarowy typ tablicowy
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

  WyswietlInformacje(t4);

  Type t5 = typeof(Dictionary<int, int>); // określony (closed) typ generyczny
  WyswietlInformacje(t5);

  Type t6 = typeof(Dictionary<, >);           // nieokreślony (unbound) typ
                                                // generyczny (Unbound generic type)
  WyswietlInformacje(t6);
}
  
```

Pobieranie typu bazowego

Poniżej przedstawiona została metoda, która dla zadanego typu wyszukuje wszystkie typy, po których on dziedziczy.

```

static void TypyBazowe(Type t)
{
  Console.WriteLine();
  Console.WriteLine("Gałąź dziedziczenia:");
  Console.WriteLine("{0} ({1})", t.Name, t.FullName);
  while (t != typeof(object))
  {
    t = t.BaseType;
    Console.WriteLine("{0} ({1})", t.Name, t.FullName);
  }
  Console.WriteLine("Wciśnij jakiś klawisz ...");
  Console.ReadKey();
}
  
```

metodę tę można wywołać na różnych typach, aby przekonać się, że w szczególności typy wartościowe również dziedziczą po klasie **Object**:

```

static void InformacjeODziedziczeniu()
{
  TypyBazowe(typeof(string));
  TypyBazowe(typeof(int));
  TypyBazowe(typeof(DateTime));
  TypyBazowe(typeof(System.IO.FileStream));
}
  
```

Pobieranie typu z assembly (biblioteki)

Typy można też pobierać z istniejących asemblacji, przez które należy tu rozumieć zarówno programy (wykonywalne), jak i biblioteki. Poniższy przykład pokazuje w jaki sposób można pobrać konkretny typ z uruchomionego (i działającego) w pamięci operacyjnej programu oraz z zarejestrowanej w GAC (*Global Assembly Cache*) biblioteki

```

static void WyszukiwanieTypu()
{
  Type t1 = Assembly.GetExecutingAssembly().GetType("Refleksje.Klasa");
  WyswietlInformacje(t1);
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    WyswietlInformacje(typeof(int));
    Type t2 = Type.GetType("System.Int32, mscorelib, Version=2.0.0.0, " +
                           "Culture=neutral, PublicKeyToken=b77a5c561934e089");
    WyswietlInformacje(t2);
}
  
```

Sposobom ładowania assemblacji będzie poświęcony jeden z następujących przykładów.

Pobieranie informacji o składowych typu

Poniższy przykład pokazuje, w jaki sposób można odczytać informacje o wszystkich publicznych składowych klasy, w jaki sposób można pobrać informacje o wszystkich składowych (lub składowych posiadających zadane modyfikatory) oraz w jaki sposób można pobrać informacje o konkretnych rodzajach składowych (tutaj – o metodach)

```

static void PobieranieSkładowych()
{
    //MemberInfo[] members = typeof(Klasa).GetMembers();
    Type t = typeof(Klasa);
    MemberInfo[] members = t.GetMembers(); // pobiera wszystkie
                                         // składowe publiczne
    foreach (MemberInfo m in members)
    {
        Console.WriteLine(m);
    }

    Console.WriteLine("Wciśnij Jakiś klawisz...");
    Console.ReadKey();
    Console.WriteLine();

    BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Public
                       | BindingFlags.Static | BindingFlags.Instance;
    //BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Static;
    members = t.GetMembers(flags);
    foreach (MemberInfo m in members)
    {
        Console.WriteLine(m);
    }

    Console.WriteLine("Wciśnij Jakiś klawisz...");
    Console.ReadKey();
    Console.WriteLine();

    MemberInfo[] methods = t.GetMethods();
    foreach (MemberInfo m in methods)
    {
        Console.WriteLine(m);
    }
    // podobnie można:
    // t.GetMember(); t.GetProperties(); t.GetNestedTypes(), t.GetNestedType()
}
  
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Pobieranie informacji o konkretnych rodzajach składowych typu

Poniższy przykład pokazuje pobranie informacji o wszystkich rodzajach składowych klasy z podziałem na ich rodzaje.

```

static void PobieranieSkladowychRodzajami()
{
    Type t = typeof(Klasa);
    BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Public
        | BindingFlags.Static | BindingFlags.Instance;

    //MemberInfo[] members = typeof(Klasa).GetMembers();
    //MemberInfo[] members = t.GetMembers(flags);
    //foreach (MemberInfo m in members)
    //    Console.WriteLine(m);

    Console.WriteLine("      LISTA PÓŁ:");
    FieldInfo[] fields = t.GetFields(flags);
    foreach (FieldInfo f in fields)
    {
        Console.WriteLine(f);
    }

    Console.WriteLine("      LISTA KONSTRUKTORÓW:");
    ConstructorInfo[] constructors = t.GetConstructors(flags);
    foreach (ConstructorInfo c in constructors)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine("      LISTA METOD:");
    MethodInfo[] methods = t.GetMethods(flags);
    foreach (MethodInfo m in methods)
    {
        Console.WriteLine(m);
    }

    Console.WriteLine("      LISTA WŁASNOŚCI:");
    PropertyInfo[] properties = t.GetProperties(flags);
    foreach (PropertyInfo p in properties)
    {
        Console.WriteLine(p);
    }

    Console.WriteLine("      LISTA ZDARZEŃ:");
    EventInfo[] events = t.GetEvents(flags);
    foreach (EventInfo e in events)
    {
        Console.WriteLine(e);
    }

    Console.WriteLine();
    Console.WriteLine("Wciśnij Jakiś klawisz... ");
    Console.ReadKey();
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Właściwości jako składowe o specyficznej konstrukcji

Należy pamiętać o dość specyficznym sposobie implementacji składowych klasy, jakimi są właściwości. Pod każdą właściwość podpięta jest jedna lub dwie metody odpowiadające blokom **get** i **set**. Metody te można pobrać metodami odpowiednio **GetGetMethod()** i **GetSetMethod()** klasy **PropertyInfo**, co zostało zaprezentowane w poniższym przykładzie.

```
static void PobieranieProperty()
{
    Type t = typeof(Klasa);

    Console.WriteLine("      LISTA WŁASNOŚCI:");
    PropertyInfo[] properties = t.GetProperties();
    foreach ( PropertyInfo p in properties)
    {
        Console.WriteLine(p);
    }
    Console.WriteLine();

    PropertyInfo wlasnosc = t.GetProperty("LiczbaCalkowita");
    MethodInfo metodaGet = wlasnosc.GetGetMethod();
    MethodInfo metodaSet = wlasnosc.GetSetMethod();
    MethodInfo[] obieMetody = wlasnosc.GetAccessors();

    Console.WriteLine("Metoda \"get\" dla właściwości \"LiczbaCalkowita\"");
    Console.WriteLine("  " + metodaGet);
    Console.WriteLine("Metoda \"set\" dla właściwości \"LiczbaCalkowita\"");
    Console.WriteLine("  " + metodaSet);
    Console.WriteLine("Obie metody dla właściwości \"LiczbaCalkowita\"");
    foreach (MethodInfo m in obieMetody)
    {
        Console.WriteLine("  " + m);
    }
}
```

Tworzenie instancji typów

Jak zostało wcześniej wspomniane, instancje klas (a także typów wartościowych) można zdalnie tworzyć poprzez klasę **Activator**. Poniższy przykład pokazuje w jaki sposób można wykonywać te operacje. Można w nim też zobaczyć jak można pobrać z klasy konstruktor i wywołać go zdalnie tworząc instancję klasy.

```
static void TworzenieInstancjiTypow()
{
    int i = (int)Activator.CreateInstance(typeof(int));

    DateTime dt = (DateTime)Activator.CreateInstance(typeof(DateTime),
        2010, 1, 26);

    Console.WriteLine(dt);

    // Pobieranie konstruktora przyjmującego jeden argument typu "string"
    ConstructorInfo ci = typeof(Klasa).GetConstructor(new[] { typeof(string) });
}
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
// tworzenie obiektu z pobrañym w powyższy sposób konstruktorem
object ob = ci.Invoke(new object[] { "Argument..." });
// tak wygladałoby tworzenie obiektu z konstruktorem bezargumentowym
//object ob = ci.Invoke(new object[] { null });

Klasa klasa = ob as Klasa;
klasa.Pisz();
}
```

Tworzenie instancji funkcji

Poniższy przykład pokazuje w jaki sposób można utworzyć instancje funkcji na podstawie metod wydobytých z typu klasy.

```
static void TworzenieInstancjiFunkcji()
{
    Delegate metodaStatyczna = Delegate.CreateDelegate(
        typeof(DelegacjaInt), typeof(Klasa), "Nastepnik");

    Delegate metodaDynamiczna = Delegate.CreateDelegate(
        typeof(DelegacjaInt), new Klasa(), "Poprzednik");

    Console.WriteLine(metodaStatyczna.DynamicInvoke(5));
    Console.WriteLine(metodaDynamiczna.DynamicInvoke(5));

    DelegacjaInt metoda = (DelegacjaInt)metodaStatyczna;
    Console.WriteLine(metoda(10));

    metodaStatyczna = Delegate.CreateDelegate(
        typeof(DelegacjaInt), typeof(Program), "Nastepnik");

    metodaDynamiczna = Delegate.CreateDelegate(
        typeof(DelegacjaInt), new Program(), "Poprzednik");

    Console.WriteLine(metodaStatyczna.DynamicInvoke(5));
    Console.WriteLine(metodaDynamiczna.DynamicInvoke(5));

    metoda = (DelegacjaInt)metodaStatyczna;
    Console.WriteLine(metoda(10));
}
```

Ładowanie assembly

Poniższy przykład przypomina w jaki sposób można pobrać działające assembly oraz pokazuje w jaki sposób można otworzyć asemblację do używania wyłącznie poprzez refleksje. Ponadto pokazane jest, jak można odczytać informację o wszystkich typach zawartych w asemblacji.

```
static void LadowanieAssembly()
{
    Type t1 = Assembly.GetExecutingAssembly().GetType("Refleksje.Klasa");
    WyswietlInformacje(t1);

    // Działa, ale jest nie zalecane (np. niepotrzebne
```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
// wywołania statycznych konstruktorów)
// Assembly a = Assembly.LoadFrom(@".\TestoweAssembly.dll");

// Ładowanie assembly tylko dla refleksji:
Assembly a = Assembly.ReflectionOnlyLoadFrom(@".\TestoweAssembly.dll");
Console.WriteLine(a.ReflectionOnly); // True

foreach (Type t in a.GetTypes())
{
    WyświetlInformacje(t);
    Console.WriteLine();
    BindingFlags flags = BindingFlags.NonPublic | BindingFlags.Public
        | BindingFlags.Static | BindingFlags.Instance;
    MemberInfo[] members = t.GetMembers(flags);
    foreach (MemberInfo m in members)
        Console.WriteLine(m);

    Console.WriteLine("Wciśnij Jakiś klawisz... ");
    Console.ReadKey();
    Console.WriteLine();
}
Console.WriteLine();
}
```

Pobranie ciała metody

Poniższy przykład pokazuje w jaki sposób można pobrać z assemblacji szczegółowe informacje o metodzie, łącznie z jej kodem maszynowym.

```
static void PobieranieCialaMetody()
{
    //Assembly a = Assembly.ReflectionOnlyLoadFrom(@".\TestoweAssembly.dll");
    Assembly a = Assembly.LoadFrom(@".\TestoweAssembly.dll");
    //Console.WriteLine(a.ReflectionOnly); // True
    Type typ = a.GetType("TestoweAssembly.MojaKlasa");

    MethodInfo metoda = typ.GetMethod("silnia");
    MethodBody cialoMetody = metoda.GetMethodBody();

    Console.WriteLine("Cialo metody" + metoda);
    Console.WriteLine();
    Console.WriteLine("Rozmiar stosu: {0}", cialoMetody.MaxStackSize);
    Console.WriteLine();
    Console.WriteLine("Zmienne lokalne:");
    foreach (LocalVariableInfo zmienna in cialoMetody.LocalVariables)
    {
        Console.WriteLine(" {0} ({1}): ", zmienna.LocalIndex, zmienna.LocalType);
    }
    Console.WriteLine();

    Console.WriteLine("Kod maszynowy:");
    byte[] kodMaszynowy = cialoMetody.GetILAsByteArray();
    for (int i = 0; i < kodMaszynowy.Length; i++)
    {
        Console.Write(" {0:X2}", kodMaszynowy[i]);
    }
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

    if ((i & 0x0F) == 0x07) Console.WriteLine("  ");
    if ((i & 0x0F) == 0x0F) Console.WriteLine();
}
Console.WriteLine();
}
  
```

W zasadzie w wielu przypadkach możliwe jest odtworzenie (z dokładnością do nazw zmiennych lokalnych i innych podobnych konstrukcji) kodu źródłowego metody. Jednak jest to dość skomplikowane i w niniejszych materiałach nie będziemy się tym zajmować.

Pobranie atrybutów assembly

Poniższy przykład pokazuje w jaki sposób można pobrać informacje o atrybutach asemblacji.

```

static void PobieranieAtrybutow()
{
    Assembly asm = Assembly.GetExecutingAssembly();
    foreach (Attribute attr in asm.GetCustomAttributes(false))
    {
        if (attr.GetType() == typeof(AssemblyCopyrightAttribute))
        {
            Console.WriteLine("Prawa autorskie:");
            Console.WriteLine("  " +
((AssemblyCopyrightAttribute)attr).Copyright);
            Console.WriteLine();
        }
        if (attr.GetType() == typeof(AssemblyCompanyAttribute))
        {
            Console.WriteLine("Instytucja:");
            Console.WriteLine("  " +
((AssemblyCompanyAttribute)attr).Company);
            Console.WriteLine();
        }
        if (attr.GetType() == typeof(AssemblyDescriptionAttribute))
        {
            Console.WriteLine("Opis:");
            Console.WriteLine("  " +
((AssemblyDescriptionAttribute)attr).Description);
            Console.WriteLine();
        }
    }
}
  
```

Dynamiczne generowanie metody

Poniższy kod pokazuje w jaki sposób poprzez klasę **ILGenerator** można wygenerować metodę. W tym przypadku metoda ta wypisuje tylko komunikat na standardowe wyjście i kończy działanie, przy czym komunikat jest generowany przez klasę **ILGenerator** metodą *EmitWriteLine*.

```

static void DynamiczneGenerowanieMetody()
{
    var dynamicznaMetoda = new DynamicMethod("Metoda", null, null,
                                              typeof(Program));
    ILGenerator generator = dynamicznaMetoda.GetILGenerator();
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
generator.EmitWriteLine("Metoda działa !");
generator.Emit(OpCodes.Ret);
dynamicznaMetoda.Invoke(null, null);
}
```

W drugim wariantie tego przykładu instrukcje skoku do metody wypisywania komunikatu są bezpośrednio podawane.

```
static void DynamiczneGenerowanieMetody2()
{
    var dynamicznaMetoda = new DynamicMethod("Metoda", null, null,
                                                typeof(Program));
    ILGenerator generator = dynamicznaMetoda.GetILGenerator();
    //generator.EmitWriteLine("Metoda działa !");
    MethodInfo writeLineStr = typeof(Console).GetMethod("WriteLine",
                                                          new Type[] { typeof(string) });
    // wywoływanie instrukcji assemblerowych
    generator.Emit(OpCodes.Ldstr, "Tak też działa !"); // ładowanie łańcucha
    generator.Emit(OpCodes.Call, writeLineStr);           // wywołanie metody
    generator.Emit(OpCodes.Ret);                          // rozkaz powrotu
                                                       // z podprogramu (metody)
    dynamicznaMetoda.Invoke(null, null);
}
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

22. Obsługa poczty

Standardowa biblioteka języka C# implementuje między innymi funkcjonalność serwera SMTP, dzięki czemu w języku C# można w prosty sposób zaimplementować wysyłanie listów elektronicznych. Poniższy rozdział poświęcony jest pokazaniu przykładów takich implementacji, bez zagłębiania się w opis wykorzystywanych klas oraz sposób działania używanych mechanizmów. Warto nadmienić, że wśród bibliotek dołączonych do Visual Studio nie ma biblioteki implementującej mechanizmy klienta IMAP ani POP (POP3), więc nie można w prosty sposób zaimplementować odbierania listów od serwera pocztowego.

Wysyłanie listu

Do zaimplementowania wysyłania listu elektronicznego w najprostszym przypadku wystarczą dwie klasy: **MailMessage** oraz **SmtpClient**. Obiekty pierwszej z tych klas reprezentują list przeznaczony do wysłania, natomiast obiekty drugiej klasy implementują połączenia z serwerem SMTP (wraz z wykonywanymi w obrębie tych połączeń operacjami). Schemat ich użycia pokazany został na poniższym przykładzie.

```
using (MailMessage list = new MailMessage("ciech2@mat.uni.torun.pl",
                                             "ciech1@mat.umk.pl", "Tytuł listu 0",
                                             "Zawartość\nlistu"))
{
    SmtpClient client = new SmtpClient("158.75.2.81");
    client.Port = 587;
    client.Send(list); //Błąd jeśli wymagana jest autoryzacja
}
```

Zwróćmy uwagę, że obiekt listu jest zarówno tworzony, jak i inicjalizowany wywołaniem konstruktora, co zwykle daje się zaimplementować jedynie w najprostszych przypadkach. Ponadto, jeśli serwer wymaga autoryzacji, to wysłanie listu zakończy się niepowodzeniem.

Wysyłanie listu z autoryzacją

W nieco bardziej rozbudowanym przykładzie podawany jest login i hasło do serwera SMTP.

```
using (MailMessage list = new MailMessage("uzytkownik-testowy2@wp.pl",
                                             "ciech1@mat.umk.pl", "Tytuł listu 1",
                                             "Zawartość\nlistu"))
{
    list.DeliveryNotificationOptions = DeliveryNotificationOptions.OnSuccess;

    NetworkCredential nc = new NetworkCredential(
        "uzytkownik-testowy2", "uzytkownik-testowy");

    SmtpClient client = new SmtpClient("smtp.wp.pl");
    client.Port = 587;
    client.Credentials = nc;
    //client.DeliveryMethod = SmtpDeliveryMethod.Network;
    client.Send(list);
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

}

Pozwala to na wysłanie listu do serwera wymagającego autoryzacji (oczywiście w rzeczywistym użyciu zdecydowanie nie należy wpisywać hasła otwartym tekstem).

Edytowanie nagłówka wysyłanego listu

W przypadku listów o bardziej złożonej strukturze znacznie wygodniejszym rozwiązaniem od przekazywania wszystkich elementów listu w konstruktorze jest przekazywanie ich przez odpowiednie własności, co zostało pokazane na poniższym przykładzie.

```

MailAddress nadawca = new MailAddress("uzytkownik-testowy2@wp.pl",
                                         "Użytkownik Testowy");
MailAddress odbiorca = new MailAddress("ciech2@mat.umk.pl",
                                         "Konto testowe na WMII");
MailMessage list = new MailMessage(nadawca, odbiorca);
list.Subject = "Tytuł listu 2";
list.Body = "Treść\nwysyłanego\nciastu";
list.ReplyTo = new MailAddress("ciech1@mat.uni.torun.pl");
list.Priority = MailPriority.High;

NetworkCredential nc = new NetworkCredential(
    "uzytkownik-testowy2", "uzytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;

client.Send(list);
  
```

Warto zauważyć na nim również użycie obiektów klasy **MailAddress** do przechowywania informacji o adresach pocztowych.

Wysyłanie listu pod kilka adresów

Przy wysyłaniu listu można podać więcej niż jeden adres odbiorcy. Jest to realizowane w ten sposób, że własność **To** przyjmuje obiekty będące kolekcjami adresów (obiektami z klasy **MailAddressCollection**), co zostało pokazane na poniższym przykładzie.

```

MailMessage list = new MailMessage();

// Źle:
//MailAddressCollection odbiorcy = new MailAddressCollection();
//odbiorcy.Add(new MailAddress("jb@mat.umk.pl", "Jerzy Białkowski"));
//odbiorcy.Add(new MailAddress("jb@mat.uni.torun.pl", "Jerzy Białkowski"));
//list.To = odbiorcy;

list.From = new MailAddress("uzytkownik-testowy2@wp.pl",
                            "Użytkownik Testowy");

list.To.Add(new MailAddress("jb@mat.umk.pl", "Jerzy Białkowski"));
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
list.To.Add(new MailAddress("jb@mat.uni.torun.pl", "Jerzy Białkowski"));

list.Subject = "Tytuł listu 3";
list.Body = "Treść\nwysyłanego\nlistu";
list.ReplyTo = new MailAddress("jb@mat.uni.torun.pl");

NetworkCredential nc = new NetworkCredential(
    "użytkownik-testowy2", "użytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;

client.Send(list);
```

Wysyłanie oraz kopii listu

W podobny sposób można podać wiele adresów, pod które ma zostać wysłana kopia lub ukryta kopia listu, np.:

```
MailMessage list = new MailMessage();

list.From = new MailAddress("użytkownik-testowy2@wp.pl",
                           "Użytkownik Testowy");

list.To.Add(new MailAddress("jb@mat.umk.pl", "Jerzy Białkowski"));
list.CC.Add("jb@mat.uni.torun.pl");
list.Bcc.Add("użytkownik-testowy2@wp.pl");

list.Subject = "Tytuł listu 4";
list.Body = "Treść\nwysyłanego\nlistu";
list.ReplyTo = new MailAddress("jb@mat.uni.torun.pl");

NetworkCredential nc = new NetworkCredential(
    "użytkownik-testowy2", "użytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;

client.Send(list);
```

Dołączanie załącznika

Do wysyłanego listu można dołączyć załącznik. Na poniższym przykładzie jako załącznik dołączany jest plik z lokalnego systemu plików.

```
MailAddress nadawca = new MailAddress("użytkownik-testowy2@wp.pl",
                                         "Użytkownik Testowy");
MailAddress odbiorca = new MailAddress("jb@mat.umk.pl",
                                         "Jerzy Białkowski");
MailMessage list = new MailMessage(nadawca, odbiorca);
list.Subject = "Tytuł listu 5";
list.Body = "Treść\nwysyłanego\nlistu";
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

list.ReplyTo = new MailAddress("jb@mat.uni.torun.pl");
list.Attachments.Add(new Attachment(@"C:\temp\plik1.txt",
                                     new ContentType("text/plain; charset=cp1250")));
list.Attachments.Add(new Attachment(str, "plik2-przez-strumien.txt",
                                     MediaTypeNames.Text.Plain));
NetworkCredential nc = new NetworkCredential(
    "uzytkownik-testowy2", "uzytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;
client.Send(list);
  
```

Wysyłanie listu w formacie HTML

W celu wysłania listu w formacie HTML wystarczy własność **IsBodyHtml** wysyłanego listu ustawić wartość **true**.

```

MailAddress nadawca = new MailAddress("uzytkownik-testowy2@wp.pl",
                                         "Użytkownik Testowy");
MailAddress odbiorca = new MailAddress("jb@mat.umk.pl",
                                         "Jerzy Białkowski");
MailMessage list = new MailMessage(nadawca, odbiorca);
list.Subject = "Tytuł listu 6";
list.Body =
"<html><body><h1>List w postaci HTML</h1><br>(jak podano powyżej)</body></html>";
list.IsBodyHtml = true;
NetworkCredential nc = new NetworkCredential(
    "uzytkownik-testowy2", "uzytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;
client.Send(list);
  
```

Wysyłanie listu z alternatywnymi formatami zawartości

Można też wysyłać listy zawierające alternatywne formaty zawartości. W poniższym przykładzie wysyłany jest list posiadający ciało zarówno w formacie HTML (z dodatkowym obrazkiem) jak i w postaci czystego tekstu.

```

MailAddress nadawca = new MailAddress("uzytkownik-testowy2@wp.pl",
                                         "Użytkownik Testowy");
MailAddress odbiorca = new MailAddress("jb@mat.umk.pl",
                                         "Jerzy Białkowski");
MailMessage list = new MailMessage(nadawca, odbiorca);
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

list.CC.Add("uzytkownik-testowy2@wp.pl");
list.Subject = "Tytuł listu 7";

//list w formacie HTML
string htmlBody = "<html><body><h1>Wariant HTML wiadomości, z
obrazkiem:</h1><br><img src=\"cid:Logo\"></body></html>";
AlternateView avHtml =
    AlternateView.CreateAlternateViewFromString(htmlBody, null,
                                                MediaTypeNames.Text.Html);
//dodanie obrazka
LinkedResource logo = new LinkedResource(@"C:\temp\logo.jpg",
                                            MediaTypeNames.Image.Jpeg);
logo.ContentId = "Logo";
avHtml.LinkedResources.Add(logo);

//czysty tekst
string textBody = "Wariant tekstowy wiadomości";
AlternateView avText =
    AlternateView.CreateAlternateViewFromString(textBody, null,
                                                MediaTypeNames.Text.Plain);

list.AlternateViews.Add(avHtml);
list.AlternateViews.Add(avText);

NetworkCredential nc = new NetworkCredential(
    "uzytkownik-testowy2", "uzytkownik-testowy");
SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;

client.Send(list);
  
```

Wysyłanie listu w szyfrowanym połączeniu

Aby wysłać list w bezpiecznym połączeniu wystarczy ustawić własność **EnableSsl** połączenia na **true**.

```

MailMessage list = new MailMessage("uzytkownik-testowy2@wp.pl",
                                    "jb@mat.umk.pl", "Tytuł listu 8", "Zawartość\nlistu\n(SSL)");

NetworkCredential nc = new NetworkCredential(
    "uzytkownik-testowy2", "uzytkownik-testowy");

SmtpClient client = new SmtpClient("smtp.wp.pl");
client.Port = 587;
client.Credentials = nc;
client.EnableSsl = true;

client.Send(list);
  
```

Wysyłanie listu w sposób asynchroniczne

Często warto jest zaimplementować wysyłanie listu w sposób asynchroniczny, aby niepotrzebnie nie blokować aplikacji w oczekiwaniu na nawiązanie połączenia oraz zaakceptowanie wysłanego listu. Przy

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

wysyłaniu listu w taki sposób można podpiąć pod zdarzenie **SendCompleted** metodę, która zostanie wywołana po zakończeniu operacji wysłania listu, jak to widać na poniższym przykładzie:

```

static void AsynchroniczneWyslanieListu()
{
    MailMessage list = new MailMessage("uzytkownik-testowy2@wp.pl",
        "jb@mat.umk.pl", "Tytuł listu 9", "Zawartość\nlistu\n(SSL)");
    //list.DeliveryNotificationOptions = DeliveryNotificationOptions.OnSuccess;

    NetworkCredential nc = new NetworkCredential(
        "uzytkownik-testowy2", "uzytkownik-testowy");

    SmtpClient client = new SmtpClient("smtp.wp.pl");
    client.Port = 587;
    client.Credentials = nc;

    client.SendCompleted += new SendCompletedEventHandler(onCompleted);

    client.SendAsync(list, null);

    Console.WriteLine("Wysyłanie listu - wcisnij klawisz ENTER aby przerwać
...");
    if (Console.ReadKey().Key == ConsoleKey.Enter)
    {
        client.SendAsyncCancel();
        Thread.Sleep(100);
    }
}

static void onCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Cancelled)
        Console.WriteLine("Anulowano wysłanie listu (!)");
    else if (e.Error != null)
        Console.WriteLine("Wystąpił błąd: " + e.Error.ToString());
    else
        Console.WriteLine("List wysłany ...");
}
  
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

23. Programowanie obłokowe w Windows Azure

Ta część materiałów w dużej mierze bazuje na szkoleniu Microsoft IT Academy nr 112 „Cloud Computing” [9] i przytacza obszerne jego fragmenty. Ponadto część wykładu poświęcona implementacji wykorzystuje materiały do samodzielnej nauki [21]. Poniżej przedstawiony jest jedynie szkic wykładu.Więcej informacji można znaleźć w materiałach ze wspomnianego szkolenia.

Wykład podzielony jest na dwie części. W pierwszej z nich (dłuższej) przedstawione są koncepcja programowania obłokowego oraz teoretyczne informacje dotyczące chmury obliczeniowej Windows Azure. W drugiej (krótszej) zaprezentowane zostaną elementy implementacyjne. Istotna część nauki tworzenia usług działających w chmurze Windows Azure przeniesiona została na laboratorium.

Koncepcja programowania obłokowego

W tej części poruszone zostaną następujące zagadnienia:

- Początki programowania obłokowego.
- Koncepcja i definicje programowania obłokowego.
- Podstawowe charakterystyki programowania obłokowego.
- Przykłady problemów, które w chmurze obliczeniowej nie występują.
- Porównanie kosztów na utworzenie (zakup) oraz utrzymanie (działanie) infrastruktury IT w standardowym modelu.
- Główne obawy w stosunku do programowania obłokowego.
- Podstawowe modele usług
 - On premise;
 - **IaaS** (*Infrastructure as a Service*);
 - **PaaS** (*Platform as a Service*);
 - **SaaS** (*Software as a Service*).
- Podział kontroli w powyższych modelach (nad danymi, aplikacją, maszyną wirtualną, serwerem, magazynem danych i siecią).
- Zalety dużych centrów danych (DC – *Data Center*):
 - Porównanie wybranych typowych kosztów (sieci, pamięci, administracji).
 - Zależność kosztów od lokalizacji na przykładzie cen prądu.
- Rozmiary i lokalizacje centrów (w przypadku Windows Azure).
- Porównanie generacji centrów danych.
- Infrastruktura centrów danych.

Konstrukcja Windows Azure

W tej części poruszone zostaną następujące zagadnienia:

- Kategorie usług
 - Application Services
 - Software Services

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- Platform Services
- Infrastructure Services
- Główne usługi platformy
 - Platforma uruchomieniowa i obliczeniowa.
 - Magazyn danych.
 - Relacyjne bazy danych.
- Rozwiązania dostępne w ramach platformy z podziałem na
 - Application Services
 - Frameworks
 - Security
 - Connectivity
 - Data
 - Compute
 - Storage
- Mechanizmy stosowane w ramach platformy z podziałem na powyższe elementy
- Opłaty
- Porównanie instancji obliczeniowych
- Wybrane oferty (pakiety)
- Ograniczenia platformy Windows Azure
- Piramida opóźnień
- Windows Azure AppFabric Caching
- Content Delivery Network
- „Mechanizmy obronne” stosowane w Windows Azure
- SLA
- Podsumowanie: czym jest Windows Azure.
- Konstrukcja maszyny wirtualnej
- Podsumowanie: jaka jest rola Windows Azure, a jaka programisty.

Tworzenie oprogramowania działającego pod Windows Azure

W tej części poruszone zostaną następujące zagadnienia:

- Co działa, a co nie działa w Windows Azure w stosunku do klasycznego programowania.
- Podobieństwa i różnice pomiędzy Windows Azure a emulatorem Windows Azure.
- Instalacja potrzebnego oprogramowania.
- Rejestracja i używanie Windows Azure.
- Sposoby wgrywania oprogramowania (i problemy, które mogą przy tym wystąpić).

Większość zadań implementacyjnych jest przeniesiona na laboratorium.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

24. Tworzenie aplikacji dla systemu Windows 8 w Metro New UI

Na wykładzie poruszone zostaną następujące zagadnienia:

- Koncepcja systemu Windows 8 (użytkownik, argumenty emocjonalne, "kupowanie oczami")
- Elementy interfejsu użytkownika (Store App):
 - proste, znane elementy (np. ikonki)
 - typografia
 - kompozycja (kafelki)
 - założenia programistyczne
 - *Pride in craftsmanship*
 - *Be fast and fluid*
 - *Authentically digital*
 - *Do more with less*
 - *Win as one*
- Nawigacja: hierarchiczność i semantyczny zoom.
- Sposoby wydawania poleceń
 - powierzchnia aplikacji, panel (charm), AppBar,
 - skróty klawiszowe, mysz, menu kontekstowe itp.
 - polecenia dotyku,
 - strefy optymalnej interakcji i czytania.
- Do jakich zastosowań nie nadają się aplikacje pisane w Modern UI/Window Store App/Metro
- Aplikacje w stylu metro
 - zwykły użytkownik może uruchamiać aplikacje nabyte (podpisane) przez AppStore
 - jeśli dodatkowo posiada konto developerskie w AppStore lub licencję developerską, to może uruchamiać nie podpisane programy (np. tworzone przez siebie)
 - w wariancie Windows 8 Enterprise możliwość instalacji "zaufanych" aplikacji (zgodnie z polityką grupową bezpieczeństwa (Group Policy))
 - możliwość podpisania przez "sideloading product activation key"
- Przechowywanie danych lokalnie i w chmurze.
- „Szybkie i płynne” („*fast and fluid*”) - operacje zajmujące więcej niż 50 ms muszą być wykonywane asynchronicznie.
- Nowe mechanizmy asynchronicznego wykonywania operacji (np. async / await)
- WinRT Interop
- Cykl życia aplikacji
 - działająca
 - wstrzymana
 - wyłączona (lub nieuruchomiona)
- W jakich okolicznościach aplikacja może zostać wyłączona.
- W jakich okolicznościach aplikacja może zostać zatrzymana.
- Konieczność zaimplementowania przywrócenia stanu aplikacji

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- Zadania w tle (Background Tasks).
- Przydzielanie zasobów dla zadań działających w tle (ograniczenia w dostępie do procesora i transferze przez sieć).
- Krytyczne zadania:
 - notyfikacje PUSH
 - Control Channel
- Globalna pula zasobów
- Kafelki
- Powiadomienia:
 - Live Tiles
 - notyfikacje Toast
 - szablony komunikatów Toast
- Koncepcja sklepu AppStore

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

25. Windows Communication Foundation

W niniejszych materiałach przedstawiona zostanie jedynie ogólna koncepcja *Windows Communication Foundation*. Główny ciężar zapoznania z tą technologią zostanie przeniesiony na laboratorium.

Idea WCF

Windows Communication Foundation jest API (interfejsem aplikacji, ang. *Application Process Interface*) służącym do komunikacji pomiędzy programami działającymi w modelu „usług” (ang. „*service-oriented model*”). Można go scharakteryzować w następujący sposób:

- Pełni rolę podobną jak **RPC** (ang. *Remote Procedure Call*) w C lub **RMI** (ang. *Remote Method Invocation*) w Javie.
- Dostarcza interfejs operujący na wysokim poziomie.
- Pozwala na wykorzystywanie różnych mechanizmów komunikacji (HTTP, TCP, Message Queues, Named Pipes).
- Używa komunikatów przesypane w postaci XML (są one opakowywane przez **SOAP** (*Simple Object Access Protocol*)). Pozwala to na uniezależnienie komunikacji od platformy sprzętowej.

Rozróżnienie ze względu na sposób udostępniania (uruchamiania)

Ze względu na sposób uruchamiania (w związku z tym udostępniania „usług”) programy dostarczające usługi podzielić możemy na cztery grupy:

- Samodzielne (ang. *self-hosting*) aplikacje – są to „typowe” aplikacje, które po uruchomieniu nasłuchują
- Usługi windows-owe (Windows Services – była o nich mowa w rozdziale 9.)
- Uruchamiane jako serwisy WWW pod serwerem IIS (Internet Information Services).
- Uruchamiane jako serwisy WWW przez WAS (Windows Activation Service) (dostępne od Vista i IIS 7.0).

Punkty końcowe

Istotną rolę w komunikacji przez WCF pełnią tzw. **punkty końcowe** (ang. *Endpoints*), pomiędzy którymi ustanawiane są połączenia. Często mówi się o punktach końcowych, że mają one strukturę „ABC” od angielskich nazw ich składowych:

- Adress (adres),
- Binding (dowiązanie),
- Contract (kontrakt).

Adresem jest w tym wypadku adresem maszyny (np. w przypadku protokołów TCP-IP adres IP oraz numer portu). *Dowiązaniem* jest mechanizm komunikacji (np. HTTP, Message Queues, Named Pipes). Natomiast *kontrakt* opisuje nazwy i sposób wywoływania udostępnianych metod. Najogólniej mówiąc ma on postać interfejsu, którego metody są opatrzone stosownymi atrybutami.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Więcej informacji o tej technologii pojawi się w trakcie ćwiczeń praktycznych na laboratorium. Osobom chcącym dogłębniej zapoznać się z zagadnieniami związanymi z komunikacją poprzez WCF polecam książki [5] i [17].

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

26. Elementy programowania w zespole (i Team Foundation Server)

Ta część materiałów w całości bazuje na szkoleniu Microsoft IT Academy nr 111 „Programowanie zespołowe” [8] i przytacza obszerne jego fragmenty. Poniżej przedstawiony jest jedynie szkic wykładu. Więcej informacji można znaleźć w materiałach ze wspomnianego szkolenia.

Modele i fazy tworzenia oprogramowania.

Zakładam, że modelom oraz fazom tworzenia oprogramowania (zgodnie z tymi modelami) była poświęcona istotna część przedmiotu *Inżynieria Oprogramowania*, więc nie będę rozwijał tego zagadnienia. Przytoczę jedynie „typowy” cykl tworzenia oprogramowania.

Typowy cykl tworzenia oprogramowania

Przypomnijmy, że „typowy” cykl tworzenia oprogramowania składa się z następujących faz:

- faza strategiczna,
- faza tworzenia,
- faza wdrożenia,
- faza testów produkcyjnych,
- faza utrzymania.

Poniżej przedstawione zostały ich zakresy.

Faza strategiczna

Faza strategiczna jest to czas, w którym, prowadząc rozmowy z klientem lub jego przedstawicielami – próbujemy odpowiedzieć na pytanie, czy realizacja tego przedsięwzięcia jest możliwa, a jeśli tak to jakie metody będą najlepsze.

Faza tworzenia

W **fazie tworzenia**, jak sama nazwa wskazuje tworzone oprogramowanie. Proces ten przebiega według przyjętej metodologii (stanowiącej element tzw. cyklu życia oprogramowania) oraz przy użyciu określonych narzędzi w oparciu o wybrane technologie. Zatem będą podczas tego procesu występować różne procesy wytwórcze oraz testy akceptacyjne.

Faza wdrożenia

Właściwie niezależnie od wielkości tworzonego na potrzeby klienta oprogramowania, w umowie „o tworzenie” zawarty jest zazwyczaj punkt związany z wdrożeniem. **Wdrożenie** jest niczym innym jak zainstalowaniem, skonfigurowaniem (ewentualnie parametryzacją) oprogramowania w środowisku uruchomieniowym klienta.

Faza testów produkcyjnych

Oprogramowanie po stworzeniu i przetestowaniu przez producenta (faza tworzenia) musi zostać uruchomione i przetestowane w środowisku docelowym (zwany produkcyjnym), przy czym zwykle nie na rzeczywistych danych. Jeśli nawet oprogramowanie jest uruchamiane dla rzeczywistych danych, to nie jako

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

jedyny system (wtedy uruchamia się nowe oprogramowanie obok istniejącego). Ta faza zwana jest **fazą testów produkcyjnych**.

Faza utrzymania

Każde oprogramowanie, podobnie jak np. samochód wymaga serwisowania, przy czym w przypadku programowania nie wynika to z faktu jego starzenia się (co nie ma miejsca), a dezaktualizacji (np. w świetle zmieniających się przepisów prawa). Stąd też firma tworząca oprogramowanie zwykle od razu proponuje tzw. jego **utrzymanie** (ang. *maintenance*) na określonych warunkach.

Klasyfikacja narzędzi używanych w pracy nad projektem

W pracy nad projektem programistycznym zwykle używane są następujące rodzaje narzędzi:

- zintegrowane środowiska programistyczne (IDE),
- systemy kontroli wersji,
- systemy śledzenia błędów (kontrolują „cykl życia błędów”),
- narzędzia biurowe,
- narzędzia wspierające zarządzanie projektem.

Zintegrowane środowiska programistyczne

Zintegrowane środowiska programistyczne (IDE – ang. *Integrated Development Environment*) charakteryzują się następującymi cechami:

- edycja kodu tworzonego programu wraz z podświetlaniem składni, coraz częściej udostępniane jest również autouzupełnianie;
- kompilacja/budowania aplikacji oraz tworzenie profili budowa np. *Release* (wersja dla klienta) i *Debug* (wersja z informacją dla debugera),
- zawierają zintegrowany debugger wraz z pracą krokową i podglądem zmiennych, rejestrów i często obliczaniem wyrażeń,
- pozwalają na wizualne programowanie - np. tworzenie czy graficzne projektowanie okien dialogowych menu itp.
- pozwalają na tworzenie kontrolek i zawierają elementy programowania komponentowego,
- pozwalają na projektowanie schematu bazy danych i tworzenie skryptów generujących bazę.

Systemy kontroli wersji

Systemy kontroli wersji (ang. *version/revision control systems*) służą do śledzenia zmian w kodzie źródłowym. Systemy te ułatwiają pracę w grupie, gdyż pomagają w łączeniu zmian dokonywanych przez kilka osób. Pamiętana jest wtedy data zmiany oraz kto ją dokonał. Zachowaniu jest dzięki temu pełna historii modyfikacji, co pozwala łatwo jest wrócić do konkretnej wersji lub określonego momentu czasowego. Dzięki temu, przy zachowaniu odpowiedniej kultury tworzenia oprogramowania, istnieje zawsze możliwość powrotu do konkretnej wersji pliku, czy też całej aplikacji i stwierdzenie np. czemu wystąpił jakiś błąd – jest to cecha bardzo istotna w przypadku utrzymywania oprogramowania przez długi czas, kiedy dostarczamy Klientowi kolejne wersje. Dla przykładu my już pracujemy nad nową, ale Klient użytkuje wersję 1.0 i do niej raportuje błędy, w takiej sytuacji możliwość „cofnięcia” się do wersji 1.0

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

pozwala na stwierdzenie gdzie leży problem. Co więcej, pozwala też na stwierdzenie, czy w obecnej wersji będzie ona nadal występował i jeśli tak, to umożliwia usunięcie go zawsze.

Systemy śledzenia błędów

Systemy śledzenia błędów (BTS – ang. Bug Tracking System) - kontrolują „cykl życia” błędów. Zwykle taki cykl wygląda następująco:

- osoba posiadająca uprawnienia do raportowania wprowadza informację o błędzie, czasami też oznacza jego ważność i priorytet,
- osoba lub system określa osobę, której dotyczy błąd i generowane jest powiadomienie (jabber, email i inne kanały komunikacji),
- osoba, która otrzymała powiadomienie, akceptuje go (błąd) do poprawy lub deleguje ten błąd do innej osoby,
- osoba odpowiedzialna kończy pracę nad błędem na jeden z kilku sposobów – informując, że błąd został poprawiony, odłożony w czasie, nie jest możliwy do poprawy czy, że zgłoszenie błędu było niepoprawne,
- jeśli błąd usunięto, osoba odpowiednio uprawniona osoba weryfikuje, czy błąd faktycznie został usunięty i zwykle zgłoszenie błędu jest zamkane,
- po ostatecznym rozwiązaniu problemu (np. po opublikowaniu albo instalacji u klienta wersji pozbawionej błędu) błąd jest zamknięty.

Narzędzia biurowe

Zakładam, że narzędzia biurowe i ich rola są ogólnie znane i pomijam ich opis.

Narzędzia wspierające zarządzanie projektem

Narzędzia wspierające zarządzanie projektem „wspierają” wykonywanie następujących operacji

- Definiowanie podstawowych informacji o projekcie.
- Opracowanie struktury projektu (WBS lub SPP).
- Tworzenie zależności pomiędzy zadaniami, ograniczenia sztywne i elastyczne zadań.
- Tworzenie wykresów Gantta, PERT innych (wyznaczanie ścieżki krytycznej).
- Definiowanie zasobów i ich zarządzanie (koszty, kalendarze).
- Przypisywanie zasobów do zadań, balansowanie, rozwiązywanie konfliktów.
- Szacowanie kosztów projektu.
- Zapisywanie projektu bazowego, analiza odchyleń i zmian.
- Raportowanie.
- Zarządzanie zasobami w różnych projektach.

Team Foundation Server jako narzędzie wspierające zarządzanie projektami programistycznymi

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Należy zaznaczyć, że Team Foundation Server współpracuje poprawnie jedynie z najbardziej rozbudowaną wersją środowiska Visual Studio – wersją Ultimate. Działać będzie poprawnie wyłącznie pod systemami z rodziny Windows Server (nie będzie działał poprawnie np. pod systemem Windows 7)

Visual Studio Ultimate

Visual Studio **Ultimate** jest najbardziej rozbudowanym wariantem aktualnej wersji zintegrowanego środowiska *Visual Studio*. Zawiera ono zarówno typowe narzędzia deweloperskie (np. kompilatory), jak i te bardziej zaawansowane, jak debugger, profiler, oraz narzędzia do projektowania interfejsów, testów jednostkowych i wiele innych.

Kod może być pisany w jednym z kilku języków: C#, Visual Basic, C++, F#, a za pomocą tego środowiska możliwe jest tworzenie:

- Aplikacji desktopowych - zarówno w oparciu o bibliotekę Windows Forms, jak i wykorzystujących WPF.
- Aplikacji dla przeglądarek - w technologii ASP.NET i Silverlight.
- Aplikacji dla urządzeń mobilnych - aplikacje te mogą pracować pod kontrolą Windows Mobile i Windows Phone
- Rozszerzeń aplikacji - daje to możliwość tworzenia dodatków dla wielu aplikacji i systemów Microsoft, takich jak: Office czy SharePoint.

Wersja Ultimate zawiera zaawansowane mechanizmy służące do testowania i debugowania aplikacji, m.in.:

- *IntelliTrace* - debugger, który pamięta historię wykonania aplikacji i umożliwia cofanie się, by prześledzić kroki wykonania.
- *Microsoft Test Manager* - nowe środowisko do wykonywania testów zawierające: moduły testów obciążeniowych aplikacji, zarządzanie wirtualnym środowiskiem testowym.
- Generator diagramów zgodnych z UML 2.0.

Team Foundation Server

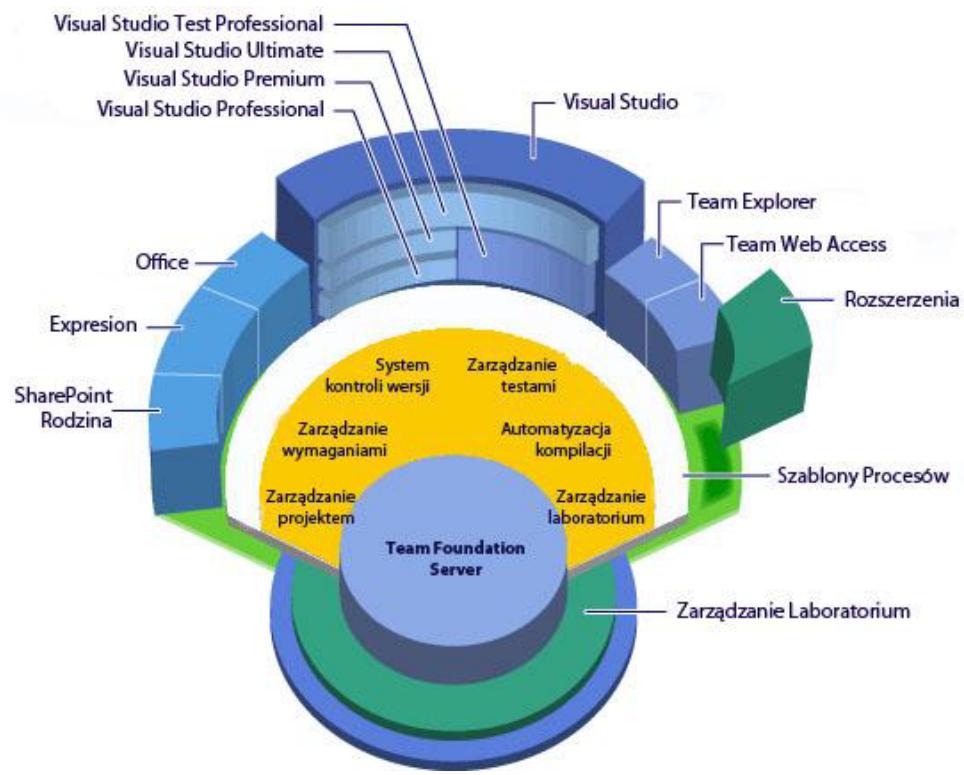
Upraszczając można powiedzieć, iż **Team Foundation Server** jest systemem do zarządzania projektami programistycznymi o rozbudowanych możliwościach, skupia on w sobie: serwer kontroli wersji, wspiera zarządzanie pracą w zespole, wspiera koordynowanie budowania aplikacji na różne platformy, posiada rozbudowany system raportów, wspiera budowę na serwerze oraz wykonywanie testów i wiele innych, np. dla każdego nowego projektu tworzona jest witryna SharePoint, która daje podstawę do budowy portalu projektu. Warto w tym miejscu powiedzieć, że TFS jest narzędziem wspierającym zarządzanie projektami w znaczeniu programistycznym, tzn. nie jest odpowiednikiem MS Project, choć daje możliwość podłączenia się tej aplikacji.

Firma Microsoft określa TFS jako rdzeń systemu zarządzania cyklem życia aplikacji (*Application Lifecycle Management – ALM*).

Integracja narzędzi

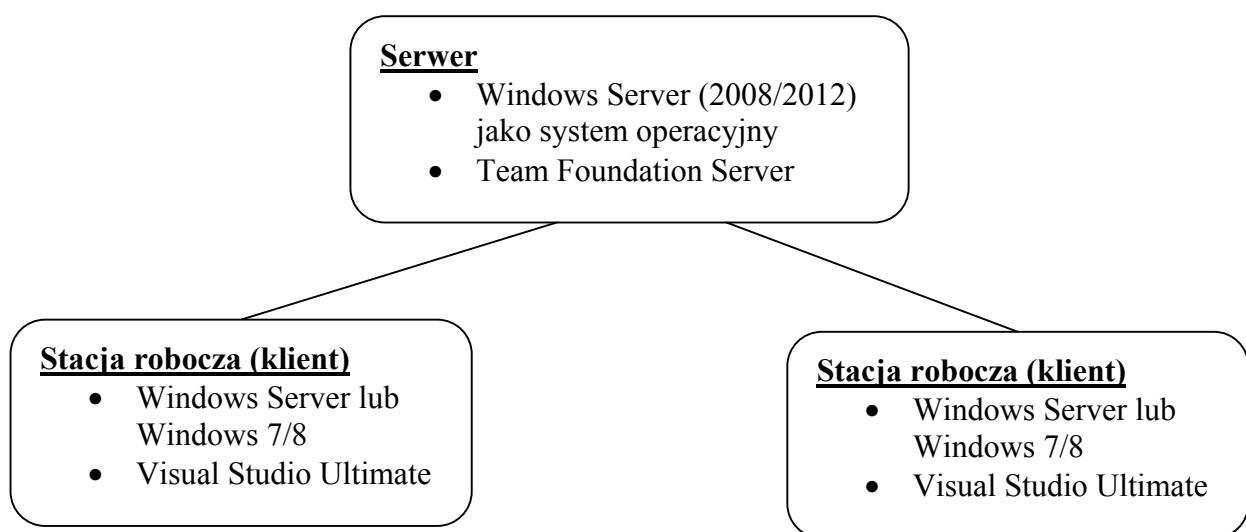
Poniższy rysunek, zaczerpnięty ze materiałów do szkolenia MS ITA 111 do pokazuje powiązania pomiędzy różnymi produktami rodziny VS 2010+TFS 2010.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki



Struktura sieci

Jak zostało wcześniej wspomniane, sam serwer TFS musi być zainstalowany pod systemem operacyjnym z rodziny Windows Server. Samo środowisko klienckie może być zainstalowane pod systemem Windows 7 (lub Windows 8 lub również pod systemem Windows Server).



Elastyczne (ang. agile) podejście do tworzenia oprogramowania

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Elastyczne podejście do tworzenia programowania, nie jest koncepcją zupełnie nową, ale dużą popularność zdobyła ona dopiero w ostatnim okresie czasu (rozumiem przez to 10 do kilkunastu lat). Charakteryzuje się ono następującymi zasadami:

- Priorytetem jest osiągnięcie zadowolenia klienta poprzez wczesne i nieprzerwane dostarczanie mu oprogramowania wysokiej jakości.
- Nie ma się co sprzeciwiać częstym zmianom wymagań.
- Należy dostarczać działające wydania oprogramowania często, raz na kilka tygodni do kilku miesięcy, z tendencją do krótszych okresów.
- Klient i programiści muszą pracować razem codziennie w całym projekcie.
- Projekt musi być oparty o zmotywowanych ludzi.
- Należy dostarczyć zespołowi środowisko pracy i wszelkie wsparcie oraz zaufać im, że wykonają swoją pracę.
- Najbardziej skuteczną i wydajną metodą dostarczania i wymiany informacji w zespole programistycznym jest bezpośrednia rozmowa.
- Działające oprogramowanie jest głównym miernikiem postępu.
- Klient i programiści powinni móc śledzić postęp stale i bez ograniczeń.
- Upraszczanie, jako sztuka maksymalizacji ilości nie wykonanej pracy, jest sprawą zasadniczą.
- Najlepsze architektury, wymagania i projekty powstają w samoorganizujących się zespołach. W regularnych odstępach czasu zespół winien rozważać, jak zwiększyć swoją wydajność i zgodnie z tym zmieniać swoje metody pracy.

Scrum

Koncepcja Scruma, o ile nie pojawiła się na wykładzie z *Inżynierii Oprogramowania*, zostanie przedstawiona na slajdach i omówiona na ich podstawie.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Programowanie Zaawansowanych Aplikacji w C#

Laboratorium

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

1. Laboratorium 1: zapoznanie z IDE Visual Studio

Zadaniem pierwszego laboratorium jest zapoznanie z zintegrowanym środowiskiem programistycznym (ang. IDE) Visual Studio w wersji Ultimate 2010 lub Ultimate 2012. W ramach ćwiczeń student powinien:

- uruchomić środowisko Visual Studio;
- zapoznać się z dostępnymi rodzajami projektów – w szczególności w trakcie zajęć tworzone będą projekty następujących rodzajów:
 - Console Application,
 - WPF Application (lub Windows Forms Application),
 - Class Library,
 - Windows Service,
 - Setup Project;
- utworzyć projekt typu *Console Application* wraz solucją;
- zapoznać się z zawartością okien, które pojawią się po utworzeniu projektu:
 - oknem edytora
 - Solution Explorer,
 - Properties,
 - rozwijalnymi przybornikami;
- wpisać kod przykładowego programu z wykładu („*HelloWorld*”);
- w trakcie wpisywania kodu przećwiczyć działanie systemu podpowiedzi *IntelliSense*, oraz przećwiczyć korzystanie z dokumentacji;
- skompilować i uruchomić ten program ze środowiska Visual Studio;
- dodać do programu dodatkowe instrukcje sterujące i przeprowadzić debugowanie tego programu w środowisku Visual Studio;
- dodać znaczniki komentarzy i wygenerować automatycznie dokumentację dla stworzonego kodu (ze środowiska Visual Studio);
- skompilować program z linii poleceń;
- wygenerować dokumentację dla stworzonego kodu z linii poleceń;
- nauczyć się przełączać wariant generowanego kodu (*Debug – Release*) w środowisku Visual Studio;
- nauczyć się dodawać referencje oraz wyłączyć ukrywanie wygenerowanych plików w *Solution Explorerze*.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

2. Laboratorium 2 i 3: Kolekcje, serializacja, biblioteka

Zadanie 1.

Stwórz klasę kolekcji generycznej dziedziczącą po interfejsie *IEnumerable* o elementach, które mogą być porównywalne (dziedzicząc po interfejsie *IComparable<>*), w której zaimplementowane mają być:

- metody
 - *Add()*,
 - *Remove()*,
 - *RemoveAt()*,
 - *GetEnumerator()*,
- właściwości (property)
 - *Count* - zwraca bieżącą ilość elementów kolekcji,
 - *Max* - zwraca maksymalną wartość elementów kolekcji lub domyślną wartość elementu stosownego typu elementu kolekcji o ile kolekcja jest pusta,
 - *Min* - zwraca minimalną wartość elementów kolekcji lub domyślną wartość elementu stosownego typu elementu kolekcji o ile kolekcja jest pusta,
- iterator ([]), który będzie można użyć zarówno do odczytu jak i przypisywania wartości,
- operatory "<" i ">", które mają sprawdzać relację częściowego porządku polegającą na tym, że kolekcje *A* i *B* są w relacji "<" (czyli "*A < B*") jeśli dla dowolnych elementów *a* z kolekcji *A* oraz *b* z kolekcji *B* element *a* jest mniejszy od elementu *b* (co jest ustalane metodą dziedziczoną z interfejsu *IComparable<>*).

Użyj kolekcji w przykładowym programie (prezentując działanie wskazanych składowych).

Solucję (klasa kolekcji i klasa zawierająca metodę *Main()*) spakuj do archiwum ZIP i prześlij jako pierwsze rozwiązanie.

Wskazówka: 1. Zapoznaj się przykładem implementacji prostej generycznej kolekcji, której elementy można iterować z rozdziału 3.
 2. Typ generyczny musi użyć klauzuli *where* dla wskazania, że implementuje interfejs *IComparable<>*.

Zadanie 2.

Zmodyfikuj rozwiązanie poprzedniego zadania w następujący sposób:

- Przenieś klasę kolekcji do osobnego projektu biblioteki (typu *Class Library*) w obrębie tej samej solucji.
- Zaimplementuj w niej metodę *ISerializable.GetObjectData()* oraz stosowny konstruktor do serializacji (dla stworzonej powyżej kolekcji, kolekcja ta ma teraz dziedziczyć po interfejsie *ISerializable*) w postaci tablicy uporządkowanych elementów.
- W solucji stwórz projekt aplikacji konsolowej z dodaną referencją do projektu biblioteki, który będzie prezentował działanie serializacji i deserializacji kolekcji.

Solucję spakuj do archiwum ZIP i prześlij jako drugie rozwiązanie.

Wskazówka: Zapoznaj przykładami z rozdziału 4.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

3. Laboratorium 4: Stworzenie aplikacji w WPF

Stwórz aplikację w Windows Presentation Foundation której implementacja będzie zawierała następujące elementy:

- przynajmniej dwa okna zdefiniowane w XAML-u,
- obrazek osadzony jako tło jednego z okien,
- kontrolki
 - *Button* (przynajmniej 3 instancje),
 - *Label*,
 - *TextBox*,
 - *CheckBox*,
 - *ListBox*,
 - *ComboBox*,
 - *RadioButton* (przynajmniej 3 instancje osadzone i zgrupowane w kontrolce *GroupBox*),
- przynajmniej jedną kontrolkę importowaną z *Windows Forms* (trzeba ją odpowiednio hostować),
- zdefiniowany przynajmniej jeden dynamiczny styl kontrolki, definiujący zmiany w stylu czcionki, poziomie przezroczystości i kolorach,
- zdefiniowany przynajmniej jeden statyczny styl kontrolki, który będzie można użyć w kontrolkach różnych typów i który będzie implementował akcję dla przynajmniej jednego wyzwalacza,
- wielopoziomowe menu rozwijalne osadzone w kontrolce *DockPanel* ze zdefiniowanymi skrótami klawiszowymi i przynajmniej jedną osadzoną ikonką,
- menu kontekstowe,
- obsługę zdarzeń *Closing* (ma być możliwość anulowania tego zdarzenia) i *Click* (dla każdego przycisku i wybranych pól z menu),
- kontrolkę *MessageBox* (ma być możliwość wyboru akcji oraz ma być sprawdzana udzielona odpowiedź).

Uwaga: Poza niniejszymi materiałami potrzebne do wykonania tego zadania wskazówki można znaleźć między innymi w [Część IV, Rozdziały 23-24, 1] (odpowiednio, [Part IV, Chapters 23-24, 10]).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

4. Laboratorium 5: Tworzenie i synchronizacja wątków

1. Zadanie na odczyt faz działania wątków

Napisz program (aplikację konsolową), w której

- wątek główny utworzy wątek potomny;
- zarówno wątek rodzica, jak i potomka mogą być usypane raz lub większą ilość razy na dowolny okres czasu w dowolnych ustalonych przez programistę momentach;
- w wątku potomka mogą być wykonywane (w pętli) duże ilości operacji arytmetycznych symulujących pracę;
- wątek rodzica ma czterokrotnie odczytywać stany wątku potomnego i wypisywać informację o tym stanie na standardowe wyjście (można dowolnie ustalić momenty odczytu tych stanów);
- należy doprowadzić do sytuacji, w której za wątek przodka wypisze cztery różne stany (za każdym razem przy odczycie ma otrzymać inny stan).

2. Zadanie na ustawianie priorytetów wątków

Napisz program (aplikację konsolową), w którym

- wątek główny utworzy przynajmniej dwukrotnie więcej wątków potomnych, niż jest dostępnych rdzeni na maszynie i przydzieli im różne priorytety (od najwyższego do najniższego) oraz przypisze różne nazwy, po czym uruchomi je jednocześnie;
- każdy ze wspomnianych wątków potomnych ma wykonywać pewne operacje arytmetyczne symulujące wykonywanie obliczeń (tej samej liczby obliczeń);
- przy kończeniu działania każdy ze wspomnianych wątków potomnych ma wypisywać linijkę, w której będzie podawał swoją nazwę, swój priorytet oraz rzeczywisty czas swojego działania (można go wyliczyć jako różnicę w wartościach zwracanych przez **DateTime.Now**).

3. Zadanie na przeprowadzenie synchronizacji wątków

- Skopiuj i skompiluj zamieszczony poniżej program, który ma wyliczać liczbę PI metodą Leibnitz'a.
- Uruchom ten program na maszynie posiadającej więcej niż 1 rdzeń i spróbuj wyjaśnić (ustnie) powody otrzymywania istotnie różnych wyników.
- Zmodyfikuj program, aby ograniczyć czas pomiędzy odczytem i zapisem współdzielonej zmiennej.
- Do programu dodaj synchronizację, aby zapewnić poprawność obliczeń.
- Ogranicz ilość obliczeń w sekcji krytycznej w celu lepszego wykorzystania mocy obliczeniowej (i przyśpieszenia obliczeń).
- Wyjaśnij powód różnic pomiędzy różnymi obliczeniami i zmodyfikuj sposób sumowania, aby zwiększyć precyzję obliczeń.

```
using System;
using System.Threading;
```

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```

namespace LiczeniePi
{
    internal class ObliczaniePi
    {
        private double wyliczanaWartosc = 0.0;
        private int ograniczenieMianownika = 1000000;
        bool zsumowaneDodatnie = false;
        bool zsumowaneUjemne = false;

        public ObliczaniePi(int iloscSkladnikow)
        {
            this.graniczenieMianownika = iloscSkladnikow * 2;
            wyliczanaWartosc = 0.0;
            zsumowaneDodatnie = false;
            zsumowaneUjemne = false;
        }

        public void SumowanieDodatnichSkladnikow()
        {
            //for (int i = 1; i < ograniczenieMianownika; i+=4)
            //    wyliczanaWartosc += 4.0/i;
            for (int i = 1; i < ograniczenieMianownika; i += 4)
            {
                double poprzedniaWartosc = wyliczanaWartosc;
                wyliczanaWartosc = poprzedniaWartosc + 4.0 / i;
            }
            zsumowaneDodatnie = true;
            if (zsumowaneUjemne) WypiszWartosc();
        }

        public void SumowanieUjemnychSkladnikow()
        {
            //for (int i = 3; i < ograniczenieMianownika; i+=4)
            //    wyliczanaWartosc -= 4.0/i;
            for (int i = 3; i < ograniczenieMianownika; i += 4)
            {
                double poprzedniaWartosc = wyliczanaWartosc;
                wyliczanaWartosc = poprzedniaWartosc - 4.0 / i;
            }
            zsumowaneUjemne = true;
            if (zsumowaneDodatnie) WypiszWartosc();
        }

        public void WypiszWartosc()
        {
            if (zsumowaneDodatnie && zsumowaneUjemne)
                Console.WriteLine("Wyliczona wartość Pi = {0}", wyliczanaWartosc);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
        }
    }
}

```

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

```
        {
            ObliczaniePi obliczenia = new ObliczaniePi(10000000);
            Thread WatekDodatnich = new Thread(new ThreadStart(
                obliczenia.SumowanieDodatnichSkladnikow));
            Thread WatekUjemnych = new Thread(new ThreadStart(
                obliczenia.SumowanieUjemnychSkladnikow));
            WatekDodatnich.Start();
            WatekUjemnych.Start();
        }
    }
}
```

4. Zadanie na ręczny i automatyczny reset uchwytu oczekiwania na zdarzenie

Skompiluj i przetestuj działanie programów z wykładu obrazujących różnicę pomiędzy ręcznym i automatycznym resetem uchwytu oczekiwania na zdarzenie.

Projekt pn. „Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

5. Laboratorium 6 i 7: Tworzenie usług i instalatora, dzienniki, kontrola usługi

1. Stworzenie projektu usługi

Stwórz usługę (albo „od zera” albo używając stosownego rodzaju projektu), której przypisz ustalone przez siebie

- unikalną nazwę,
- wyświetlaną nazwę,
- opis.

2. Ręczna instalacja i deinstalacja usługi

- Wylistuj usługi zainstalowane w systemie (zarządzanie komputerem → usługi → ...).
- Używając polecenia **installutil.exe** zainstaluj utworzoną w poprzednim kroku usługę.
- Ponownie wylistuj zainstalowane w systemie usługi i wyszukaj tę, która właśnie została dodana.
- Używając polecenia **installutil.exe** odinstaluj stworzoną przez siebie usługę.
- Zweryfikuj, że ta usługa została usunięta z listy usług.

3. Stworzenie wygodniejszego instalatora

- Do solucji (zawierającej projekt usługi) dodaj projekt instalatora (typu *Setup and Deployment*).
- Do projektu instalatora dodaj wyjście projektu usługi.
- Zweryfikuj, że ścieżki instalacji (źródła usług) są poprawnie ustalone.
- Dla tego wyjścia dodaj operacje *install*, *uninstall*, *commit* i *rollback* (pod *Custom Action*).
- Zbuduj projekt instalatora.
- Powtórz operacje z ręcznej instalacji dla otrzymanego instalatora.

4. Dodanie do usługi funkcjonalności

- W projekcie usługi zaimplementuj mechanizmy obsługi utworzonego przez siebie dziennika zdarzeń (zarówno dziennik, jak i wykorzystywane źródło mają być tworzone w usłudze):
- Przy starcie usługi, zarówno dziennik, jak i wykorzystywane źródło mają być tworzone, jeśli nie istnieją.
- Logowane mają być uruchomienie i zatrzymanie usługi.
- W trakcie działania usługi co pół minuty (co 30 sekund) ma być zapisywana informacja do dziennika.

5. Stworzenie aplikacji zarządzającej usługą

- Utwórz projekt aplikacji graficznej (WPF lub Windows Forms).
- W aplikacji dodaj przyciski których wcisnięcie będzie odpowiednio uruchamiać oraz zatrzymywać usługę.
- Dodaj kontrolkę (*Label*, *TextBox* itp.) wypisującą aktualny stan usługi (ma być inicjalizowana w momencie uruchomienia aplikacji oraz uaktualniana przy każdym wcisnięciu przycisku).
- W metodach wywoływanych w reakcji na zdarzenia wcisnąć przycisków dodaj oczekiwanie na zmianę stanu usługi (przed odczytem stanu usługi).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- Uczyń nieaktywnymi przycisk uruchamiania usługi, jeśli jest ona uruchomiona oraz przycisk zatrzymywania usługi w momencie, gdy jest ona zatrzymana.
- Dodaj kontrolkę która ma pokazywać zawartość logu dziennika zdarzeń (przy starcie aplikacji ma być pusta) oraz przyciski do czyszczenia tej kontroli oraz odświeżenia jej zawartości.

6. Połączenie projektów

- Wszystkie projekty połącz w jednąację.
- Zmień typy komplikacji projektów z *Debug* na *Release*.
- W projekcie instalatora mają znaleźć się wyjścia pozostałych projektów.
- Zweryfikuj poprawność miejsca kładzenia binariów w instalatorze.
- Zmień nazwę tworzonej paczki instalatora (plików EXE i/lub MSI).
- Zbuduj instalator i przetestuj jego działanie.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

6. Laboratorium 8 i 9: Procesy, pliki konfiguracyjne, przełączniki śledzenia, watcher

Zadania na procesy.

1. Napisz program, który wylistuje wszystkie procesy w systemie, wypisując
 - ich nazwy,
 - ich identyfikatory,
 - ilość zajmowanej przez nie (każdy z osobna) pamięci,
 - od jakiego czasu są one uruchomione.
2. Napisz program, który uruchomi proces (wybranej) przeglądarki z wybraną stroną oraz
 - po 5 sekundach wypisze listę modułów załadowanych przez tą przeglądarkę,
 - co 2 sekundy będzie monitorować rozmiar pamięci zajmowany przez tę przeglądarkę,
 - po upływie 5 minut od uruchomienia procesu zakończy go.

Zadanie na pliki konfiguracyjne i przełączniki śledzenia, dzienniki zdarzeń oraz watcher na system plików.

Napisz program spełniający następujące wymagania:

- Do programu dołączony ma być plik konfiguracyjny zawierający
 - w sekcji ustawień aplikacji zmienne **Sciezka**, **Czas**, **Dziennik**, **Zrodlo**, reprezentujące odpowiednio ścieżkę do katalogu w systemie plików, limit czasu działania programu, nazwę dziennika zdarzeń, który będzie używany przez program oraz źródło wykorzystywane do zapisywania informacji przez do tego dziennika,
 - w sekcji diagnostycznej przełącznik wielostanowy (śledzenia) **Logowanie**.
- Program ten po uruchomieniu ma nasłuchiwać na zmiany na systemie plików wewnętrz katalogu zadanego przez zmienną **Sciezka** (oraz jego katalogach).
- Zmiany mają być logowane do dziennika zdarzeń o nazwie zadanej przez zmienną **Dziennik**, przy czym nazwa źródła wpisów jest przekazywana przez zmienną **Zrodlo** (jeśli nie istnieją, to należy przy uruchomieniu programu utworzyć ten dzienniki i/lub źródło i skojarzyć źródło z dziennikiem).
- W zależności od wartości przyjmowanej przez przełącznik **Logowanie** do dziennika mają być logowane następujące informacje:
 - o dostępie do plików o ile wartość przełącznika jest ustalona na **Verbose**,
 - o zmianie zawartości plików o ile przełącznik przyjmuje wartość **przynajmniej Info**,
 - o zmianie nazw plików o ile przełącznik przyjmuje wartość **przynajmniej Warning**,
 - o utworzeniu oraz usuwaniu plików o ile przełącznik przyjmuje wartość **przynajmniej Error**,
 - jeśli przełącznik jest wyłączony, to nie mają być logowane żadne informacje.
- Program ma kończyć działanie po upływie liczby sekund zadanej przez zmienną **Czas**.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

7. Laboratorium 10: Liczniki (Performance Counters)

Zadania na liczniki systemowe (*Performance Counters*).

1. Napisz program, który wylistuje wszystkie kategorie liczników.
2. Z otrzymanej listy wybierz nazwy kategorii, dla których napisz program, który wypisze wszystkie instancje liczników z tych kategorii oraz wszystkie liczniki z tych instancji.
3. Napisz aplikację graficzną, która będzie monitorowała wartość odczytaną z wybranego przez siebie licznika (np. stan naładowania baterii, procent wykorzystania pamięci, wskaźnik utylizacji łącza internetowego itp.) i przedstawiała tę wartość w postaci belki (np. *ProgressBar*) lub wykresu (trudniejsze).

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

8. Laboratorium 11: Refleksje i obsługa poczty

Zadania na refleksje.

Stwórz klasę zawierającą metodę, która dla zadanego typu będzie wypisywać dla tego typu **oraz każdego jego typu bazowego** następujące informacje:

- pełną nazwę kwalifikowaną,
- osobno odczytane:
 - nazwę typu,
 - przestrzeń nazw w której jest on umieszczony,
 - nazwę modułu (biblioteki) w którym jest on zdefiniowany,
- informację czy jest to typ wartościowy, klasa, czy interfejs,
- informacje czy jest to typ
 - prymitywny (pierwotny),
 - abstrakcyjny,
 - tablica,
 - typ wyliczeniowy.

Użyj tej metody w aplikacji konsolowej, która będzie odczytywać z linii poleceń nazwę typu (w wariantie z pełną ścieżką) i wywoływać dla tego typu wspomnianą metodę.

Przetestuj działanie tej aplikacji na typach wybranych typów prymitywnych (np. int, double, string), interfejsów, tablic i typów wyliczeniowych.

Stwórz aplikację konsolową, która dla zadanego typu wypisze wszystkie jego składniki (ich nazwy) z podziałem na konstruktory, pola, metody, zdarzenia i własności, przy każdym składniku podając informację, czy jest on publiczny.

Zadanie na pocztę.

Załóż darmowe konto pocztowe w dowolnym serwisie i napisz program, który prześle na nie w bezpiecznym połączeniu list zawierający alternatywne formaty zawartości oraz załącznik.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

9. Laboratorium 12 i 13: Tworzenie i publikowanie usług w Windows Azure

Jeśli w ramach zajęć dostępne będą kody Azure Pass, to utwórz konto i aktywuj darmowy dostęp do chmury Windows Azure z otrzymanym kodem. (w przeciwnym wypadku zadania będą wykonywane na emulatorze).

Pobierz i zainstaluj materiały do kursu [21] lub używać materiałów dostępnych w sieci oraz zrób zadania z lekcji

- Introduction to Cloud Services
- Deploying Cloud Services in Windows Azure

z tych materiałów.

Wskazówki do powyższych laboratoriów zostały szczegółowo opisane, ale na wszelki wypadek podaje następujące wskazówki dotyczące wykonywania zadań z tych lekcji:

Introduction to Cloud Services.

- Przy tworzeniu rozwiązania od podstaw należy dopisać modyfikatory dostępu public dla odpowiednich klas i ich metod (w szczególności konstruktorów) w tworzonej bibliotece.
- Nie należy zwiększać liczby instancji ani dla web-roli ani dla workera (nie ma na to zasobów w subskrypcji).
- Nie zauważałem innych wertych podkreślenia problemów, które mogłyby wystąpić w tym laboratorium. Zadania z tego laboratorium nie mają szczególnych wymagań odnośnie posiadania najnowszej wersji Windows Azure (poszło mi nawet na półtorarocznej wersji), jej kompletności, autoryzacji, synchronizacji czasu, wersji systemu itp. W paru miejscach opis jest nie za dobrze pokopiowany (tzn. część została skopiowana z innego miejsca i nie została uzupełniona), ale jest ich mało i łatwo domyścić się o co chodzi.

Deploying Cloud Services in Windows Azure

W tej lekcji można będzie napotkać na więcej problemów.

- Sugeruję pracę rozpoczęć od zaktualizowania dodatków do Windows Azure, doinstalowania PowerShella i ewentualnego ściągnięcia uaktualnionej wersji tutoriala (być może będzie mniej usterek).
- W pierwszym zadaniu (i wszystkich następnych) należy zweryfikować, czy ustawiana domyślna wersja, nie jest przypadkiem "przestarzała" i nie można już tworzyć z nią nowych serwisów ani uaktualniać istniejących. W takim wypadku parametr "**osVersion**" w znaczniku "**ServiceConfiguration**" należy ustawić (lub zmienić) na "*" (co oznacza autodetekcję).
- Jeśli przy publikowaniu przez *PowerShell* oraz przez *VisualStudio* będzie uporczywie wyskakiwać błąd autoryzacji, to powodem może być przesunięcie czasu. Najprościej jest wówczas zsynchronizować czas systemowy z jakimś serwerem czasu. (Przy ręcznym publikowaniu nie powinno to mieć znaczenia.)

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

- Przy publikowaniu przez *PowerShell* najlepiej mieć jeden zasobnik i jeden serwis - oba mające tę samą nazwę. W przeciwnym wypadku problemy pojawią się już przy próbie odczytania nazwy zasobnika. Można to obejść pomijając automatyczne przeszukiwanie wyjścia polecenia *Get-AzureStorageAccount* i ręczne wydobycie z niego potrzebnych informacji. (Dalej trzeba też trochę pozmieniać i uważnie czytać dodatkowe uwagi i się do nich stosować.)

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

10. Laboratorium 14: Pisanie programów dla Windows 8 w stylu Metro New UI

Zapoznaj się z interfejsem Visual Studio służącym do tworzenia aplikacji typu AppStore. W tym celu aktywuj miesięczną bezpłatną licencję developerską.

Wykonaj zadania laboratoryjne (od 1 do 5) zgodnie z kursem [22]:

- Part 1: Create a "Hello, world" app
- Part 2: Manage app lifecycle and state
- Part 3: Navigation, layout, and views
- Part 4: File access and pickers
- Part 5: Create a blog reader

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

11. Laboratorium 15: WCF i kończenie pisania rozpoczętych wcześniej programów

Pobierz z [23], skompiluj i przetestuj działanie przykładu składającego się pary programów komunikujących się przez WCF w wersji nie wyższej niż dla Frameworka 4.0 (wersja dla Frameworka 4.0 jest dostępna pod adresem [http://msdn.microsoft.com/en-us/library/dd483346\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/dd483346(v=vs.100).aspx), a dla Frameworka 3.5 pod adresem [http://msdn.microsoft.com/en-us/library/vstudio/ms751514\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/vstudio/ms751514(v=vs.90).aspx)). Wspomniany przykład składa się z projektu usługi windowsowej (Windows Service), która dostarcza usługę wykonywania prostych operacji arytmetycznych i konsolowej aplikacji klienckiej.

Przejrzyj kod źródłowy i pliki konfiguracyjne tych programów, zwracając szczególną uwagę na sposób definiowania kontraktu i dowiązania.

Zmodyfikuj kontrakt dodając metodę wykonującą dodatkową operację arytmetyczną i zaimplementuj obsługę tej metody w serwerze i kliencie. Skompiluj programy i przetestuj ich działanie. Należy o tym pamiętać o każdorazowym wyrejestrowywaniu usługi windowsowej przed rozpoczęciem wprowadzania zmian do plików źródłowych.

Z adresu [http://msdn.microsoft.com/en-us/library/dd483346\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd483346(v=vs.110).aspx) pobierz analogiczny przykład dla Frameworka 4.5 i porównaj jego pliki konfiguracyjne z plikami pobranego wcześniej przykładu. Zwróć uwagę na istotne różnice wynikające z braku konieczności podawania ustawień domyślnych wartości.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*”
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Literatura

Bibliografia w języku polskim

I. Książki

Literatura podstawowa

1. John Sharp, *Microsoft Visual C# 2010 Krok po kroku*, APN PROMISE 2010.
[Jako zamiennika można użyć starszej wersji: *Microsoft Visual C# 2008 Krok po kroku*]
2. John Sharp, *Microsoft Visual C# 2012 Krok po kroku*, PROMISE 2013 (w druku).
3. Andrew Troelsen, *Język C# 2010 i platforma .NET 4.0*, Wydawnictwo Naukowe PWN / Apress 2011.
4. Joseph Mayo, *C# dla .NET 3.5. Księga eksperta*, Helion 2010.

Literatura uzupełniająca

5. Maciej Grabek, *WCF od podstaw. Komunikacja sieciowa nowej generacji*, Helion 2012.
6. Jacek Matulewski i inni, *Visual Studio 2010 dla programistów C#*, Helion 2011.

II. Skrypty do szkoleń w Microsoft IT Academy (darmowy dostęp)

7. Michał Włodarczyk, *Object Oriented Programming*, szkolenie ITA-105.
8. Ścibór Sobieski, *Programowanie zespołowe*, szkolenie ITA-111.
9. Piotr Burbacz, *Cloud Computing*, szkolenie ITA-112.

Projekt pn. „*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*” realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Bibliografia w języku angielskim

I. Książki

Literatura podstawowa

10. John Sharp, Microsoft Visual C# 2010 Step by Step, Microsoft Press 2010.
[Jako zamiennika można użyć starszej wersji: Microsoft Visual C# 2008 Step by Step]
11. John Sharp, Microsoft Visual C# 2012 Step by Step, Microsoft Press 2012.
12. Andrew Troelsen, Pro C# 2010 and the .NET 4 Platform, Fifth Edition, Apress 2010.
13. Jeffery Richter, CLR via C#, Third Edition, Microsoft Press 2010.
14. Joseph Albahari and Ben Albahari, C# 4.0 in a Nutshell, Fourth Edition, O'Reilly Media 2010.
15. Tony Northrup, Shawn Wildermuth and Bill Ryan, .NET Framework 2.0 Application Development Foundation, MCSE Exam 70-536, Self-Paced Training Kit, Microsoft Press 2005.

Literatura uzupełniająca

16. Donis Marshall, Parallel Programming with Microsoft Visual Studio 2010 Step by Step, Microsoft Press 2011.
17. Aaron Hamilton-Pearce, Expert WCF 4: Maximizing Windows Communication Foundation, Apress 2011.
18. Raffaele Garofalo, Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel (MVVM) Pattern, Microsoft Press 2011.
19. Dino Esposito, Programming Microsoft® ASP.NET 4, Microsoft Press 2011.
20. Roberto Brunetti, Windows Azure Step by Step, Microsoft Press 2011.

II. Darmowe materiały do samodzielnego nauki - kursy z podziałem na laboratoria

21. Windows Azure Training Kit, <http://www.windowsazure.com/en-us/develop/net/other-resources/training-kit/>
22. Create your first Windows Store app using C# or Visual Basic, <http://msdn.microsoft.com/en-us/library/windows/apps/hh974581.aspx>

III. Dokumentacja w postaci elektronicznej i odsyłacze do strom WWW

23. MSDN Library for Visual Studio (wersja sieciowa lub opcjonalny dodatek do Visual Studio - do wersji 2008 był dostarczany na płycie instalacyjnej, od wersji 2010 można go pobrać i zainstalować przez sieć).
24. Standard ECMA-334, C# Language Specification, 4th edition.
25. C# Language Specification (składnik instalacji Visual Studio).
26. Portale społecznościowe i fora – w obu językach (w szczególności <http://codeguru.pl/>, <http://wss.pl/>, (dawniej <http://ms-groups.pl/>), <http://stackoverflow.com/>, <http://www.codeproject.com/>)