Rozdział 3.
Język C#

Jacek Matulewski

Język C# został zaprojektowany i jest rozwijany przez grupę programistów Microsoftu pod kierunkiem Andersa Hejlsberga. Jest to osoba, która wcześniej była odpowiedzialna za kompilator Turbo Pascala, środowisko Delphi i bibliotekę VCL w firmie Borland, a od 1996 roku, już w Microsofcie, za projekt J++, a potem C# i platformę .NET. Podstawowa składnia języka C# jest podobna do znanej z C++. Jednak ponieważ C# dynamicznie się rozwija, w jego wersjach 3.0 i 4.0 pojawiły się zupełnie nowe słowa kluczowe. Natomiast idea samej platformy uruchomieniowej .NET niewątpliwie przypomina założenia wirtualnej maszyny Javy.

Technologia .NET została zaprojektowana, aby zrealizować kilka podstawowych celów. Pierwszym z nich jest przenaszalność. Microsoft od kilku lat rozszerza ilość urządzeń, na których działają jego systemy. Poza komputerami typu PC, są to wszelkiego typu urządzenia kontrolowane przez systemy Windows Embedded, telefony komórkowe i urządzenia typu Pocket PC z Windows Mobile i Windows Phone czy wreszcie konsole Xbox i Zune. Na większości z tych urządzeń działa okrojona wersja platformy .NET Compact, jak również XNA — odmiana platformy .NET zoptymalizowana do wyświetlania grafiki komputerowej. Jeżeli do tej panoramy włączymy także technologię Silverlight, tj. platformę .NET w postaci pluginu przeglądarki, to zrozumiemy, że mamy do czynienia z technologią, której nie można ignorować[1]. A przecież wszystkie wersje platformy .NET możemy programować, używając jednego języka — C#.

Co ciekawe, idea przenaszalności programów przygotowanych dla platformy .NET realizowana jest także wysiłkiem społeczności *open source*. Dzięki temu, że Microsoft udostępnił pełną specyfikację platformy .NET, stale rozwijane są jej wersje otwarte, w szczególności platforma *Mono*, która działa m.in. w systemach Linux.

Platforma .NET rozwijana jest dopiero od końca lat dziewięćdziesiątych ubiegłego wieku (w 2001 udostępniona została jej pierwsza wersja publiczna), a już można ją uważać za w pełni dojrzałą. Więcej — wyznacza nowe kierunki rozwoju informatyki i penetruje mało zbadane komercyjnie jej obszary. Mam tu na myśli technologię LINQ (wersja 3.5) czy programowanie współbieżne (wersja 4.0).

Platforma .NET

Środowisko uruchomieniowe

Biblioteki platformy .NET tworzą środowisko, w którym uruchamiane są aplikacje .NET. Jego oficjalna nazwa to *Common Language Runtime* (w skrócie CLR). Powstaje w ten sposób warstwa pośrednia między aplikacją a systemem Windows, która z jednej strony chroni system przed niebezpiecznymi zachowaniami aplikacji, przejmuje kontrolę nad zarządzaniem pamięcią i wątkami, a z drugiej — udostępnia aplikacjom bibliotekę komponentów (nazwaną *Windows Forms*), dodatkowe ujednolicone mechanizmy dostępu do zasobów systemu, w tym szczególnie do baz danych (ADO.NET), ułatwia komunikację między komputerami itp. Ciekawa z

[1] Wspólny element systemów kontrolujących wszystkie te urządzenia, a więc platforma .NET ofiarująca jednolity interfejs programistyczny, nie wyczerpuje dążeń do unifikacji wizjonerów Microsoftu. Chcieliby oni, aby także interfejs wszystkich tych urządzeń był podobny. W praktyce oznacza to systemy Windows na wszelkiego rozmiaru ekranach: od urządzeń mobilnych, poprzez monitory komputerów PC, po inteligentne telewizory i interaktywne stoły (Microsoft Surface).

punktu widzenia programisty jest dostępność wielu kompilatorów dla platformy .NET. Obecnie aplikacje .NET możemy tworzyć w C++, Visual Basicu, C#, F#, Delphi, a nawet Perlu lub JavaScripcie.

W wersji 4.0 platformy .NET do środowiska uruchomieniowego CLR dołączona została warstwa DLR (ang. *Dynamic Language Runtime*) odpowiedzialna za obsługę języków dynamicznych, takich jak Python, Ruby, ale również nowych dynamicznych elementów C# 4.0, który jednak pozostaje językiem ze statycznymi typami (podrozdział „Typy dynamiczne" w tym rozdziale).

Kod pośredni i podwójna kompilacja

Podobnie jak w przypadku Javy, także tutaj kod źródłowy nie jest kompilowany bezpośrednio do kodu maszynowego. Kompilacja jest dwustopniowa. W pierwszej fazie zostaje on skompilowany do kodu pośredniego (ang. *Intermediate Language*, w skrócie *IL*), wspólnego dla wszystkich języków programowania platformy .NET i dla wszystkich środowisk uruchomieniowych .NET, bez względu na system operacyjny i procesor. Kod pośredni MSIL jest odpowiednikiem *bytecode* z Javy. Druga faza to kompilacja kodu pośredniego za pomocą kompilatorów *Just-In-Time* (skrót *JIT*). Nie musi się ona odbywać bezpośrednio po pierwszym etapie ani nawet na tym samym komputerze. Komputer przeprowadza ją w momencie uruchamiania programu.

Nieco zamieszania może powodować fakt, że wynikiem pierwszej kompilacji kodu źródłowego C# — oraz innych języków .NET — jest plik *.exe*. Mimo tego samego rozszerzenia i nagłówka rozpoczynającego się od „MZ", struktura tego pliku jest zupełnie inna niż tradycyjnych plików wykonywalnych zawierających kod maszynowy. Zawiera on bowiem kod pośredni, nazywany również kodem zarządzanym (ang. *managed code*)[2], ze względu na stopień kontroli, jaki nad wykonaniem tego kodu i wykorzystywaną przez niego pamięcią sprawuje platforma .NET. Z punktu widzenia użytkownika, jeżeli tylko platforma .NET jest zainstalowana w systemie, nie ma to większego znaczenia — po prostu uruchamia on program, klikając dwukrotnie plik *.exe*, jednak z punktu widzenia systemu różnica jest ogromna.

Skróty, które warto poznać

Jak każda nowa technologia, także .NET wprowadza wiele nowych nazw i skrótów. Aby ułatwić czytelnikowi posługiwanie się nowym żargonem związanym z .NET, co jest warunkiem swobodnego korzystania ze źródeł informacji na temat .NET dostępnych w internecie, przedstawiam najczęściej spotykane skróty.

CLR (ang. *Common Language Runtime*) — czyli środowisko uruchomieniowe aplikacji platformy .NET, warstwa pośrednia między systemem Windows a aplikacją.

CLS (ang. *Common Language Specification*) — wymagania stawiane językom i ich kompilatorom, aby mogły tworzyć aplikacje dla platformy .NET.

CTS (ang. *Common Type System*) — podzbiór specyfikacji *CLS* określający jednolity system typów istniejących w środowisku .NET (tabela 2.1), które powinny być dostępne w każdym kompilatorze.

CAS (ang. *Code Access Security*) — podsystem środowiska *CLR* odpowiedzialny za dopilnowanie, aby ze względu na bezpieczeństwo systemu aplikacje nie wykraczały poza wyznaczone im ramy działania, tzn. aby np. aplikacja uruchamiana z sieci nie miała możliwości zapisu na lokalnych dyskach itp.[3].

JIT (ang. *Just-In-Time*) — kompilator, który kod pośredni (*IL*) kompiluje do kodu maszynowego, co pozwala na wykonanie poleceń znacznie szybciej niż przy bezpośredniej interpretacji *IL*. Kompilator *JIT* jest elementem środowiska uruchomieniowego (*CLR*), więc jego wykorzystanie nie ogranicza przenośności kodu pośredniego.

Podstawowe typy danych

Poniżej znajduje się krótki przegląd języka C# zbierający podstawowe informacje na jego temat w takiej postaci, żeby łatwo można było się do nich odwołać, czytając kolejne rozdziały.

---

[2] Można się o tym przekonać, stosując prosty program *ildasm.exe* dołączony do Microsoft .NET Framework SDK.

[3] Analogicznie jak w Java 2, w nowszej wersji .NET Framework 1.1 użytkownik może kontrolować politykę bezpieczeństwa (aplikacja *CasPol.exe*).

Deklaracja i zmiana wartości zmiennej

Zaczniemy od elementarza. Podobnie jak we wszystkich językach z rodziny C/C++/Java, także w C# deklarujemy zmienną w dowolnym miejscu metody, podając jej typ i nazwę:

```
typ nazwa_zmiennej;
```

np.

```
int i;
long l;
string s;
float f;
double d;
```

Zmienne zadeklarowane w ten sposób nie mogą być użyte, zanim nie zostanie im przypisana jakaś wartość (w przeciwnym razie pojawi się błąd kompilatora).

Zmiennej może być nadana wartość za pomocą operatora przypisania `=`:

```
i = 1;
l = 1L;
s = "Helion";
f = 1.0f;
d = 1.0;
```

Obie instrukcje mogą być połączone i wówczas deklaracji towarzyszy inicjacja zmiennej. Najczęściej używana forma deklaracji zmiennej wygląda wobec tego następująco:

```
typ nazwa_zmiennej = wartość_inicjująca;
```

np.

```
int i = 1;
long l = 1L;
string s = "Helion";
float f = 1.0f;
double d = 1.0;
```

Warto pamiętać, że koprocesor matematyczny wbudowany we wszystkie obecne procesory jest zoptymalizowany dla typów `int` i `double`. Niewiele zatem zyskamy na szybkości, korzystając z typów `byte` lub `float`.

Typy liczbowe oraz znakowy

W C/C++ wymienione powyżej typy danych nazywane byłyby typami prostymi — dla odróżnienia od klas. Z kolei w Javie istniały typy proste, ale dodatkowo zdefiniowane były odpowiadające im tzw. klasy opakowujące, dostarczające metody pomocnicze i dodatkowe informacje o typie. Natomiast projektanci C# poszli o krok dalej — tzw. typy proste są strukturami, a słowa kluczowe znane z C, C++ i Javy, takie jak `int` lub `double`, są tylko aliasami nazw tych struktur.

W efekcie stała typu `int`, np. 1, jest instancją struktury `System.Int32` i pozwala na dostęp do wszystkich metod tej klasy. Nie powinien zatem dziwić następujący kod:

```
int i=10;
string s=i.ToString();
```

lub wręcz

```
string s=10.ToString();
```

Dostępne typy danych „prostych” zebrane zostały w tabeli 3.1.

Tabela 3.1. „Proste” typy danych dostępne w C# oraz ich zakresy

Nazwa typu (słowo kluczowe, alias klasy)	Klasa z obszaru nazw System	Liczba zajmowanych bitów	Możliwe wartości (zakres)	Domyślna wartość
`bool`	`Boolean`		`true`, `false`	`false`
`sbyte`	`SByte`	8 bitów ze znakiem	od –128 do 127	0
`byte`	`Byte`	8 bitów bez znaku	od 0 do 255	0
`short`	`Int16`	2 bajty = 16 bitów ze znakiem	od –32 768 do 32 767	0
`int`	`Int32`	4 bajty = 32 bity ze znakiem	od –2 147 483 648 do 2 147 483 647	0
`long`	`Int64`	8 bajtów = 64 bity ze znakiem	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	0L
`ushort`	`UInt16`	2 bajty = 16 bitów bez znaku	maks. 65 535	0
`uint`	`UInt32`	4 bajty = 32 bity bez znaku	maks. 4 294 967 295	0
`ulong`	`UInt64`	8 bajtów = 64 bity bez znaku	maks. 18 446 744 073 709 551 615	0L
`char`	`Char`	2 bajty = 16 bitów (zob. komentarz)	Znaki Unicode od U+0000 do U+FFFF (od 0 do 65 535)	‘\0’
`float`	`Single`	4 bajty (zapis zmiennoprzecinkowy)	Wartość może być dodatnia lub ujemna; najmniejsza bezwzględna wartość to $1{,}4 \cdot 10^{-45}$, największa to ok. $3{,}403 \cdot 10^{38}$	0.0F
`double`	`Double`	8 bajtów (zapis zmiennoprzecinkowy)	Najmniejsza wartość to $4{,}9 \cdot 10^{-324}$, największa to ok. $1{,}798 \cdot 10^{308}$	0.0D
`decimal`	`Decimal`	16 bajtów (większa precyzja, ale mniejszy zakres niż `double`)	Najmniejsza wartość to 10^{-28}, największa to $7{,}9 \cdot 10^{28}$	0.0M

Zakres większości typów „prostych” można sprawdzić za pomocą statycznych pól `MinValue` i `MaxValue`, np. `double.MaxValue`, natomiast ich rozmiar zwraca operator `sizeof`.

Wartość domyślną typu można sprawdzić za pomocą słowa kluczowego `default`, np. `default(int)`.

Warto zwrócić uwagę na to, że C#, podobnie jak Java, posiada typ znakowy, który jest dwubajtowy i dzięki temu liczba znaków, które mogą być adresowane zmiennymi tego typu, obejmuje całą tablicę Unicode, tj. 2^{16} = 65 536 znaków. W ten sposób rozwiązany został problem liter narodowych.

Do inicjacji liczb wykorzystywane są stałe liczbowe, czyli wszystkie te wyrażenia, których wartość znana jest już w momencie kompilacji. Poza stałymi o typowej postaci, jak 1, 1.0 lub 1E0, można wykorzystać dodatkowe stałe wzbogacone o litery określające typ stałej (tabela 3.2).

Tabela 3.2. Stałe liczbowe w C#

Zapis w kodzie C#	Typ	Opis
1	`int`	Liczba całkowita ze znakiem
1U	`uint`	Liczba całkowita bez znaku
1L	`long`	Liczba całkowita 64-bitowa
1F	`float`	Liczba

1M	`decimal`	Liczba całkowita ze znakiem o zwiększonej precyzji (dzięki zapisowi podobnemu do liczb zmiennoprzecinkowych)
1E0, 1.0E0, 1.0	`double`	Liczba zmiennoprzecinkowa
01		Liczba ósemkowa
0x01		Liczba szesnastkowa

Określanie typu zmiennej przy inicjacji (pseudotyp var)

Aby ułatwić odczytywanie danych z nieznanych źródeł, do C# 3.0 wprowadzono słowo kluczowe `var`. Pozwala ono definiować zmienne o typie ustalanym przez kompilator na podstawie wartości użytej do ich inicjacji (ang. *implicitly-typed variables*). Inicjacja musi jednak nastąpić w tej samej linii kodu, co deklaracja. Nie są to jednak zmienne o dynamicznym typie (por. typ `Variant` w Delphi, wskaźnik na `void` w C++ czy wreszcie typ `dynamic` w C# 4.0); typ `var` w C# 3.0 po inicjacji respektuje kontrolę typu i np. do zmiennej typu `var` zainicjowanej stałą typu `int` nie można już przypisać łańcucha. Zmienne tego typu mogą być jedynie zmiennymi lokalnymi, a więc mogą być deklarowane i inicjowane tylko wewnątrz metody.

Możliwe jest zatem deklarowanie m.in. następujących zmiennych typów prostych:

```
var i = 5;
var l = 5L;
var s = "Helion";
var f = 1.0f;
var d = 1.0;
```

W tym przypadku zmienna `i` będzie typu `int` (`System.Int32`), `l` typu `long` (`System.Int64`), `s` typu `string`, `f` typu `float`, a `d` — `double`. Można się o tym łatwo przekonać, odczytując typ zmiennych poleceniem `i.GetType().FullName`. Typu `var` można również używać do kontrolek bibliotek platformy .NET, kolekcji, typów parametrycznych, klas i struktur definiowanych przez programistę. W pełni odkryjemy jego zalety, gdy będziemy chcieli przechować dane odczytane za pomocą zapytania LINQ (niżej w tym rozdziale) — typ zwracany przez zapytania często jest trudny do określenia.

W C# są dwa rodzaje zmiennych: wartościowe i referencyjne. Różnica między nimi jest bardzo istotna i zostanie omówiona w dalszej części tego rozdziału.

Operatory

Operacje na typach liczbowych i znakowym umożliwiają m.in. operatory. W C# można definiować operatory w projektowanych przez siebie klasach (tym zajmiemy się niżej, ale jeszcze w tym rozdziale). Dostępne w C# operatory wymienione zostały w tabeli 3.3 w kolejności malejącego priorytetu.

Tabela 3.3. Predefiniowane operatory C# (kolejność odpowiada priorytetowi)

Grupa operatorów	Operator	Opis
Podstawowe (ang. *primary*)	`x.y`	Dostęp do pola, metody, własności i zdarzenia
	`f()`, `f(x)`	Wywołanie metody
	`a[n]`	Odwołanie do elementu tablicy lub indeksatora
	`x++`, `x--`	Zwiększenie lub zmniejszenie wartości o 1 **po** wykonaniu innej instrukcji (np. po wykonaniu `int x=2; int y=x++;` otrzymamy `y=2`, `x=3`)
	`new`	Tworzenie obiektu, składnia: *`Klasa referencja=new Klasa(arg_konstr);`*

Grupa operatorów	Operator	Opis
	`typeof`	Zwraca zmienną typu `System.Type` opisująca typ lub klasę podaną jako argument: `typeof(int).ToString()` równy jest `System.Int32`
	`checked`, `unchecked`	Operatory kontrolujące zgłaszanie wyjątku przekroczenia zakresu dla operacji na liczbach całkowitych `int i = unchecked(int.MaxValue + 1);`
Jednoargumentowe (ang. *unary*)	`+x`, `-x`	Operatory zmiany znaku liczby (np. –3)
	`!`	Negacja logiczna (np. `!true` to `false`)
	`~`	Negacja bitowa (zdefiniowana dla typów `int`, `uint`, `long` i `ulong`): `~01001010 = 10110101`
	`++x`, `--x`	Zwiększenie lub zmniejszenie wartości o 1 **przed** wykonaniem innej instrukcji (np. po wykonaniu `int x=2; int y=++x;` otrzymamy `y=3`, `x=3`)
	`(`*typ*`)x`	Jawne rzutowanie (konwersja) zmiennej `x` na typ *typ*. Uwaga! `(int)0.99 = 0`
Mnożenia (ang. *multiplicative*)	`*`, `/`, `%`	Operator dzielenia zwraca wynik zgodny z typem argumentów: `5/2=2`, `5/2.0=2.5`, `5f/2=2.5`. Operator reszty z dzielenia: `5%2=1`
Dodawania (ang. *additive*)	`+`, `-`	Dodawanie i odejmowanie. Typ wyniku zależy od typu argumentów, więc: `uint i=1; uint j=3; uint k=i-j;` spowoduje, że `k=4294967294`
Przesunięcia bitów (ang. *shift*)	`<<`, `>>`	`1<<1 = 2`, bo jedynka została przesunięta z ostatniej pozycji na przedostatnią i zostało dostawione zero
Porównania wartości i typów (ang. *relational and type testing*)	`<`, `>`, `<=`, `>=`	Porównanie wartości: `1>2 = false`
	`is`	Porównanie typów: `1 is int = true`
	`as`	Rzutowanie równoznaczne z: *`wyrażenie is typ ? (typ)expression : (typ)`*`null`
Równości (ang. *equality*)	`==`, `!=`	`0.5==1/2f` jest prawdziwe, `0.5!=1/2`, czyli `0.5!=0` jest prawdziwe
Operacje na bitach	`&`	Bitowe `AND` (i), tzn. bit jest zapalany tylko wtedy, jeżeli oba odpowiednie bity argumentów są zapalone `74&15=10` (`01001010 & 00001111 = 00001010`)
	`^`	Bitowe `XOR` (rozłączne lub) — bit jest równy 1 tylko wtedy, gdy odpowiednie bity argumentów są różne `74^15=69` (`01001010 ^ 00001111 = 01000101`)
	`|`	Bitowe `OR` (lub) — bit jest zapalony, jeżeli przynajmniej jeden odpowiedni bit argumentów jest zapalony `74|15=79` (`01001010 | 00001111 = 01001111`)
Operacje na wartościach logicznych (`bool`)	`&&`	Logiczne `AND` (`true` jedynie wtedy, gdy oba argumenty są `true`)
	`||`	Logiczne `OR` (`true`, gdy przynajmniej jeden argument jest `true`)

Grupa operatorów	Operator	Opis
Warunkowy (ang. *conditional*)	`?:`	*`warunek`*`?`*`wartość_jeżeli_true`*`:`*`wartość_jeżeli_false`*, gdzie *`warunek`* musi mieć wartość typu `bool`, np. `x>y?x:y` zwraca większą wartość spośród `x` i `y`
Przypisania (ang. *assignment*)	`=` oraz `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	Inicjacja i zmiana wartości zmiennych, np. polecenia `int x=1;` `x+=2;` nadadzą zmiennej `x` wartość 3

Operatory bitowe (`~`, `&`, `^` i `|`) nie są przeciążone dla typu `byte`. Aby wobec tego uzyskać wyniki z przykładów umieszczonych w tabeli 3.3, należy wykonać odpowiednie rzutowania na ten typ, np. `byte x=74, z=(byte)(~x); MessageBox.Show(""+z);` lub `byte x=74, y=15, z=(byte)(x&y); MessageBox.Show(""+z);`.

Konwersje typów podstawowych

Poza jawną konwersją, którą programista może uzyskać, korzystając z odpowiedniego operatora (tabela 3.3), w kodzie C# można stosować także konwersję niejawną. Ze względów bezpieczeństwa jest ona jednak bardziej ograniczona niż w C++. Niejawna konwersja dla liczb jest zawsze możliwa z typu całkowitego na typ zmiennoprzecinkowy (generalnie do typu o większej precyzji) oraz z typu o mniejszym zakresie na typ o większym zakresie (kodowany przez większą liczbę bitów). Konwersja w drugą stronę, a więc wiążąca się z utratą informacji (precyzji), wymaga jawnego użycia operatora konwersji, a więc świadomej decyzji programisty. Dlatego przy próbie przypisania wartości typu `double` do zmiennej typu `int` pojawi się błąd kompilatora ostrzegający przed możliwą utratą precyzji (jego treść to: „Cannot implicitly convert 'double' to 'int'”):

```
double x=1; //to jest OK
int i=1.0; //tu pojawi się błąd
```

W przypadku operatorów dwuargumentowych, które są użyte dla argumentów różnych typów (np. `1.0+1`), niejawna konwersja jest również wykonywana tylko zgodnie z powyższymi zasadami. W takiej sytuacji typ zwracanej przez operator wartości zależy — oczywiście — od typów użytych argumentów:

```
1f+1 //double
1f/2 //double, wartość 0.5
1/2 //int, wartość 0
```

Wybierany jest typ o większej precyzji lub większym zakresie.

Pośród operatorów arytmetycznych tylko operacja dzielenia może powodować problemy związane z typami i zasadami ich konwersji. W szczególności w przypadku dzielenia liczb całkowitych wynikiem może być liczba wymierna, ale ze względu na konwencję, w której wynik działania operatora ma typ odpowiadający typowi argumentu o największej precyzji i zakresie, wynikiem dzielenia dwóch liczb o typach całkowitych jest zaokrąglona w dół liczba całkowita. Oznacza to, że wartość operatora `/` jest w takim przypadku przybliżona. Od wielu lat jest to jedno z podstawowych źródeł błędów logicznych w programach.

Należy także zwrócić uwagę, że jawna konwersja liczb zmiennoprzecinkowych na całkowite wiąże się z obcięciem części po przecinku, a nie poprawnym matematycznie zaokrągleniem. Zatem wyrażenie `(int)0.99` ma wartość 0, a nie spodziewane 1. Dokładnie tak samo wykonywane jest zaokrąglenie w przypadku dzielenia liczb całkowitych.

Operatory is i as

W C# możliwe jest przypisanie referencji, np. typu `Button` (przycisk) do zmiennej typu `Object`. Klasa `Object` jest bowiem (niebezpośrednio) klasą bazową klasy `Button`, co oznacza, że możliwe jest bezpieczne rzutowanie na uboższą klasę. W każdej chwili można jednak sprawdzić, z jakiego typu obiektem mamy tak naprawdę do czynienia. Pozwala na to operator `is`, który zwraca wartość logiczną będącą odpowiedzią na pytanie typu: „Czy obiekt jest przyciskiem?”. Oto przykład:

```
Object o = new Button();
if (o is Button) MessageBox.Show("Obiekt jest przyciskiem");
else MessageBox.Show("Obiekt nie jest przyciskiem");
```

Poza tym na rzecz każdego obiektu w platformie .NET możemy wywołać metodę `GetType`, która zwraca typ tego obiektu.

```
MessageBox.Show(o.GetType().FullName);
```

Po zweryfikowaniu typu obiektu można jego referencję bez obaw zrzutować na właściwy:

```
Button b = null;
if (o is Button) b = (Button)o;
```

Jednak zamiast takiej konstrukcji wygodniej skorzystać z operatora `as`:

```
Button b = o as Button;
```

Jest on równoznaczny z konstrukcją: `o is Button ? (Button)o : (Button)null`. Operator `as`, podobnie jak jego pierwowzór z Object Pascala, może być używany tylko do zmiennych referencyjnych. To ograniczenie nie dotyczy operatora `is`.

Łańcuchy

Problem łańcuchów został w C# rozwiązany w sposób konsekwentny. Po pierwsze, łańcuchy są implementowane w klasie `System.String`[4] (alias `string`), w której zdefiniowane zostały wszystkie potrzebne w praktyce metody i własności. Pozwalają one m.in. na porównywanie łańcuchów, analizę ich zawartości oraz modyfikacje poszczególnych znaków lub fragmentów. Najważniejsze metody zostały zebrane w tabeli 3.4. Dostęp do metod klasy `String` możliwy jest zarówno wtedy, gdy dysponujemy zmienną typu `string`, jak i na rzecz stałych łańcuchowych:

```
string s="Wydawnictwo Helion";
int dlugosc=s.Length;
```

lub po prostu:

```
int dlugosc="Wydawnictwo Helion".Length;
```

Po drugie, łańcuchy w C# bazują na zestawie znaków Unicode, co oznacza, że każdy znak kodowany jest dwoma bajtami. Nie istnieje zatem problem dostępności znaków narodowych. I wreszcie po trzecie, do dyspozycji mamy przeciążony operator `+` służący do łatwego łączenia łańcuchów.

Tabela 3.4. Najczęściej używane metody klasy String

Funkcja *wartość* nazwa(*argumenty*)	Opis	Przykład użycia
indeksator []	Zwraca znak na wskazanej pozycji; pozycja pierwszego znaku to 0	`"Helion"[0]`
boolean Equals(*string*)	Porównuje łańcuch z podanym w argumencie	`"Helion".Equals("HELP")`
int IndexOf(*char*), *int* IndexOf(*String*)	Pierwsze położenie litery lub łańcucha; –1, gdy nie zostanie znaleziony	`"Lalalalala".IndexOf('a')`
int LastIndexOf(*char*), *int* LastIndexOf(*String*)	Jw., ale ostatnie wystąpienie litery lub łańcucha	`"Lalalalala".LastIndexOf('a')`
int Length	Zwraca długość łańcucha,	`"Helion".Length`

[4] Jest to typ referencyjny, ale jego operatory przypisania `=` i porównania `==` działają, tak jak dla typu wartościowego. Kopiowanie oznacza zatem w istocie klonowanie, a porównanie dotyczy wartości, a nie adresu.

	własność	
string Replace(*string*,*string*), *string* Replace(*string*,*string*)	Zamiana wskazanego fragmentu przez inny	"HELP".Replace("P","ION")
string Substring(*int*,*int*)	Fragment łańcucha od pozycji w pierwszym argumencie o długości podanej w drugim	"Wydawnictwo".Substring(5,3)
string Remove(*int*,*int*)	Usuwa wskazany fragment	"Helion".Remove(1,2)
string Insert(*int*,*string*)	Wstawia łańcuch przed literą o podanej pozycji	"Helion".Insert(2,"123")
string ToLower() *string* ToUpper()	Zmienia wielkość wszystkich liter	"Helion". ToLower()
string Trim() oraz TrimStart, TrimEnd	Usuwa spacje z przodu i z tyłu łańcucha	" Helion ".Trim()
string PadLeft(*int*,*char*) *string* PadRight(*int*,*char*)	Uzupełnia łańcuch znakiem podanym w drugim argumencie, aż do osiągnięcia długości zadanej w pierwszym argumencie	"Helion".PadLeft(10,' ')
bool EndsWith(*string*), *bool* StartsWith(*string*)	Sprawdza, czy łańcuch rozpoczyna się lub kończy podanym fragmentem	"csc.exe".EndsWith("exe")

Ciągi definiujące łańcuchy mogą zawierać sekwencje specjalne rozpoczynające się od znaku *backslash* \ (identycznie jak w C/C++). Często wykorzystywany jest znak końca linii \n oraz znak cudzysłowu \". Ten ostatni nie kończy łańcucha i jest traktowany dokładnie tak samo jak inne znaki. Sekwencja kasująca poprzedni znak to \b. Wreszcie sekwencje pozwalające definiować znaki Unicode (także spoza dostępnego na klawiaturze zestawu ASCII) zaczynają się od \u i obejmują cztery kolejne znaki alfanumeryczne[5], np. \u0048. Oto przykład łańcucha zdefiniowanego w ten sposób:

```
string helion="Wydawnictwo \" \u0048 \u0065 \u006c \u0069 \u006f \u006e \"";
```

Wartość łańcucha helion to „Wydawnictwo " H E L I O N "" (spacje w jego definicji zostały umieszczone tylko po to, żeby wyraźniej pokazać sekwencje dla każdego znaku Unicode).

Ze względu na wykorzystanie znaku \ do rozpoczynania sekwencji kodujących znaki specjalne również dla wprowadzenia jego samego konieczna jest specjalna sekwencja \\. Stąd biorą się podwójne znaki *backslash* w C/C++, Javie i C# w przypadku zapisanych w łańcuchach ścieżek dostępu do plików w systemie Windows:

```
string nazwapliku="\\Windows\\BubbleBreaker.exe";
```

Wartość tego łańcucha to w rzeczywistości *\Windows\BubbleBreaker.exe*. W C# tę samą wartość można uzyskać, stawiając przed łańcuchem znak @ i pomijając podwójne znaki *backslash*, np.:

```
string nazwapliku=@"\Windows\BubbleBreaker.exe";
```

W łańcuchu poprzedzanym znakiem @ kompilator interpretuje wszystkie znaki dosłownie, ignorując ewentualne sekwencje specjalne. Uwzględniane są nawet znaki końca linii w kodzie, jeżeli występują między cudzysłowami. Same cudzysłowy uzyskuje się powtórzonym znakiem "" (podczas gdy bez znaku @ należy użyć sekwencji \").

[5] Przypominam, że w .NET typ char jest dwubajtowy (tabela typów prostych), trzeba go zatem kodować aż czterocyfrowymi liczbami szesnastkowymi. W „stacjonarnym" systemie Windows właściwe kody znaków Unicode można znaleźć w aplikacji Tablica znaków w menu *Start/Akcesoria/Narzędzia systemowe*.

String vs StringBuilder

Wspomniałem wyżej, że choć łańcuch (`string`) jest typem referencyjnym, to jego operator przypisania `=` zdefiniowany jest w taki sposób, że powoduje tak naprawdę klonowanie obiektu. To dotyczy również jego metod służących do manipulacji zawartością łańcucha. Wszystkie te, które zwracają wartość typu `string`, a więc m.in. `Insert`, `Remove` czy `Replace`, nie operują na bieżącej instancji łańcucha, a tworzą i zwracają nowy łańcuch. Zatem jeżeli wykonujemy dużo operacji na łańcuchach, np. wewnątrz pętli, koszt wszystkich kopiowań może być spory. Z tego powodu w momencie, w którym chcemy wielokrotnie zmieniać raz utworzony łańcuch, zamiast klasy `string` powinniśmy użyć klasy `StringBuilder`.

Rozważmy dwa łańcuchy: jeden typu `string`, a drugi — `StringBuilder`. Wykonajmy na nich dwie operacje: dołączmy do nich inny łańcuch i zastąpmy ich fragment innym. Realizuje to poniższy kod:

```
//string
string s = "abc---";
s += "xyz";
s = s.Replace("---", " ijk ");
MessageBox.Show(s);


//StringBuilder
StringBuilder sb = new StringBuilder("abc---");
sb.Append("xyz");
sb.Replace("---", " ijk ");
MessageBox.Show(sb.ToString());
```

Efekt wykonania obu zbiorów instrukcji jest taki sam — pojawi się komunikat o treści „abc ijk xyz”. Przyjrzyjmy się im jednak uważniej. W przypadku typu `string` działanie operatora `+=` oznacza utworzenie nowej zmiennej typu `string`, której adres[6] jest zapisywany w tej samej referencji, co pierwotny łańcuch. To oznacza kopiowanie. Natomiast zastępowana wartość łańcucha, a więc obiekt typu referencyjnego, pozostaje bez referencji, co oznacza, że zajmie się nim *garbage collector*. To jednak nie dzieje się natychmiast. Analogicznie działają metody `Replace`, `Insert` czy `Remove`. Również one nie modyfikują obiektu typu `string`, na rzecz którego zostały wykonane, a tworzą nowe łańcuchy. To oznacza, że ich wynik musi być skopiowany (użycie operatora `=`), bo bez tego nie zostanie zachowany.

Odpowiednikiem operatora `+=` w klasie `StringBuilder` jest metoda `Append`. Dodaje ona do bieżącej zawartości łańcucha łańcuch wskazany w argumencie. I robi to bez kopiowania — modyfikuje bieżącą instancję obiektu. Analogicznie działają metody `Replace`, `Insert` i `Remove`. Korzystanie z klasy `StringBuilder` jest zatem wydajniejsze.

Z drugiej strony, nie ma co popadać w przesadę. Jeżeli manipulacji łańcuchem nie jest więcej niż kilka czy kilkanaście, to różnica czasu wykonywania operacji w przypadku łańcucha typu `string` i `StringBuilder` jest niezauważalna, nawet na tak mało wydajnych urządzeniach jak Pocket PC. I dlatego w tej książce właściwie zawsze korzystam z typu `string`.

Typ wyliczeniowy

Typ wyliczeniowy jest implementowany przez klasę `System.Enum`, ale jest to klasa szczególna. W tym przypadku alias `enum` nie jest prostym zastąpieniem nazwy klasy[7].

Typ wyliczeniowy jest stosowany do definiowania grupy powiązanych ze sobą stałych o całkowitych wartościach. Może być zdefiniowany jedynie w przestrzeni nazw lub jako pole klasy, a więc jego definicja nie

---

[6] Używam tu i poniżej sformułowania „adres”, choć powinienem używać słowa „referencja”. Tego ostatniego zwykło się jednak używać na określenie zmiennej typu referencyjnego. Gdybym był purystą terminologicznym, zamiast „adres obiektu jest zapisywany do referencji”, powiedziałbym „referencja do obiektu jest zapisywana do zmiennej referencyjnej”. Wydaje się to jednak mniej zrozumiałe.

[7] Zastąpienie słowa kluczowego `enum` w definicji typu wyliczeniowego nazwą klasy `System.Enum` doprowadzi do błędu kompilacji.

może znaleźć się wewnątrz metody (stosuje się tu zasady dotyczące miejsca, w którym może być zdefiniowana klasa):

```
public enum DniTygodnia:
   byte{niedziela=1,poniedzialek,wtorek,sroda,czwartek,piatek,sobota};
```

Domyślnie wartość pierwszego wymienionego literału ustalana jest na 0, a każdy kolejny otrzymuje wartości o jeden większe. W powyższym przykładzie zamiast domyślnej wartości pierwszego elementu użyta została wartość 1.

Konkretny typ wyliczeniowy związany jest zawsze z jakimś typem liczbowym całkowitym. W naszym przykładzie jest nim `byte` (`System.Byte`). Zaskakujące może być jednak to, że wyrażenie `DniTygodnia.poniedziałek` nie jest wcale typu `byte`, a typu `DniTygodnia` (zdefiniowanego typu wyliczeniowego). Gdy chcemy go wykorzystać jako stałą, należy dodać jawne rzutowanie na typ `byte`, tj. `(byte) DniTygodnia.poniedziałek`, np.:

```
byte nrDniaTygodnia=(byte)DniTygodnia.poniedzialek;
```

Ale też prawdziwym zadaniem typu wyliczeniowego nie jest tworzenie zbioru stałych. Jest nim ograniczanie możliwych wartości zmiennych. W odróżnieniu od zmiennej typu `byte` zmienna typu `DniTygodnia` może przyjmować tylko i wyłącznie wartości określone w typie wyliczeniowym. Zatem jeżeli np. definiujemy metodę, której argumentem ma być dzień tygodnia, i chcemy uniknąć przypadkowych wartości prowadzących do błędu, powinniśmy jako argumentu użyć właśnie typu wyliczeniowego.

Definiowanie typu wyliczeniowego jest w istocie rozszerzaniem klasy `System.Enum`. W tworzonej w ten sposób klasie potomnej definiowane są publiczne pola statyczne typu zadeklarowanego w definicji. Takiego dziedziczenia nie można jednak wykonać jawnie, bo `System.Enum` jest tzw. klasą specjalną[8].

Kolejna cecha szczególna typu wyliczeniowego mogąca wywołać konsternację u czytelnika, który uważnie przeczytał pierwszą część rozdziału, to fakt, że pomimo iż typ ten jest implementowany przez klasę, jest typu wartościowego. W istocie zdefiniowany przez nas typ `DniTygodnia` zachowuje się bardziej jak struktura niż klasa. Elementy określone w definicji typu wyliczeniowego (`poniedzialek`, `wtorek` itd.) są obiektami stałymi typu `DniTygodnia`, a wykonanie polecenia `DniTygodnia pn=DniTygodnia.poniedzialek;` lub `byte nrDniaTygodnia=(byte)DniTygodnia.poniedzialek;` oznacza utworzenie nowego obiektu i skopiowanie do niego wartości z oryginału, a nie utworzenie nowej referencji. Można więc w skrócie powiedzieć, że typ wyliczeniowy należy do grupy typów wartościowych.

Leniwe inicjowanie zmiennych

W .NET 4, w przestrzeni System pojawiła się nowa klasa o nazwie `Lazy`. Implementuje ona wzorzec leniwej inicjacji. We wzorcu tym zmienna opakowywana typem `Lazy` nie jest rzeczywiście inicjowana, aż do momentu jej pierwszego użycia. Służy to — oczywiście — optymalizowaniu użycia procesora i pamięci, szczególnie w przypadku zmiennych typów referencyjnych, które są deklarowane „na zapas” i ewentualnie inicjowane (tworzone są obiekty) np. w instrukcji warunkowej. Użycie nowego „wrappera” jest bardzo proste:

```
Lazy<int> li = new Lazy<int>(()=>1); //deklaracja zmiennej i wskazanie funkcji
MessageBox.Show(li.IsValueCreated.ToString()); //jeszcze niezainicjowana
MessageBox.Show("Odwołanie do zmiennej, li=" + li.Value); //leniwa inicjacja
MessageBox.Show(li.IsValueCreated.ToString()); //już zainicjowana
```

W pierwszej linii powyższego przykładu jako argumentu konstruktora klasy `Lazy` używam funkcji zwracającej wartość 1. Jest to prosta fabryka, która służy do inicjowania zmiennej. Powyższy przykład jest — oczywiście — zbyt prosty, ale jego zadaniem jest tylko prezentacja idei leniwej inicjacji. W szczególności lepiej byłoby, gdyby

---

[8] Inne tego typu klasy to: `System.Value` będąca prototypem struktur, `System.Delegate` — klasa bazowa delegacji, które zostaną omówione, i `System.Array` — prototyp tablic. Po klasach specjalnych nie można dziedziczyć jawnie, a jedynie przez wykorzystanie odpowiednich konstrukcji (słów kluczowych `struct`, `delegate` dla struktur i delegacji oraz operatora `[]` dla tablic).

leniwym typem była klasa, a nie struktura. Wówczas zadaniem fabryki jest tworzenie instancji owej klasy. Na poniższym listingu prezentuję to na przykładzie przycisku:

```
Lazy<Button> lb = new Lazy<Button>(() =>
{
   Button b = new Button();
   b.Parent = this;
   b.Top = 100;
   b.Left = 100;
   b.Text = "Leniwy przycisk";
   return b;
   });
MessageBox.Show(lb.IsValueCreated.ToString());
MessageBox.Show("Odwołanie do zmiennej, etykieta przycisku: \"" + lb.Value.Text + "\"");
MessageBox.Show(lb.IsValueCreated.ToString());
```

Metody

W C# nie jest możliwe definiowanie funkcji niebędących metodami jakiejś klasy. Funkcja może być wprawdzie statyczną składową klasy, ale tak czy inaczej zawsze jest metodą (tj. właśnie funkcją składową klasy zdefiniowaną w obrębie tej klasy). Na szczęście, nie utrudni to nam nauki, zmuszając od razu do skoku na głębokie wody programowania obiektowego — pierwsze metody zdefiniujemy po prostu w obrębie istniejącej klasy okna `Form1`.

Utwórzmy nowy projekt typu *Windows Forms Application*. Na formie umieśćmy przycisk i kliknijmy go dwukrotnie, aby utworzyć domyślną metodę zdarzeniową. Przy okazji przeniesieni zostaniemy do edytora kodu. Obok powstałej przed chwilą metody zdefiniujmy teraz samodzielnie własną metodę. Zacznijmy od prostej nic nierobiącej metody wyróżnionej na listingu 3.1. Jej kod umieśćmy gdziekolwiek wewnątrz klasy `Form1`, ale — oczywiście — nie wewnątrz innej metody lub konstruktora. Przykładowa metoda `Metoda` nie przyjmuje żadnych argumentów i nie zwraca wartości. Możemy ją wywołać, tak jak pokazano na listingu 3.1, wewnątrz metody `button1_Click`.

Listing 3.1. Przykład metody i jej wywołanie

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace MetodyCS
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```

```
            InitializeComponent();
        }

        void Metoda() //głowa metody, sygnatura
        {
            MessageBox.Show("Hello World!"); //ciało metody
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Metoda();
        }
    }
}
```

Słowo kluczowe `void` użyte do zaznaczenia, że funkcja nie zwraca żadnej wartości, zostało „pożyczone” z języka C++. W przeciwieństwie do Pascala i Delphi, pusta lista argumentów nie zwalnia z używania nawiasów przy wywołaniu metody.

Komentarze w listingu 3.1 wskazują dwie części metody. Część pierwsza to głowa metody zawierającą zwracany przez nią typ wartości, nazwę metody i listę jej parametrów (w tym przykładzie lista ta jest pusta). Elementy te, a szczególnie nazwa metody i jej parametry, składają się na sygnaturę metody, po której jest ona rozpoznawana wśród innych metod klasy.

Przeciążanie metod

Skomplikujmy nieco nasz przykład, definiując drugą metodę o tej samej nazwie. Różnić się jednak będzie obecnością parametru. Będzie nim łańcuch określający treść wyświetlanego komunikatu. Zdefiniujmy zatem obok pierwszej metody `Metoda` drugą, tyle że z parametrem (listing 3.2).

Listing 3.2. Metody przeciążone i ich wywołanie

```
void Metoda()
{
   MessageBox.Show("Hello World!");
}

void Metoda(string tekst)
{
   MessageBox.Show(tekst);
}

private void button1_Click(object sender, EventArgs e)
{
   Metoda();
   Metoda("Witaj świecie!");
}
```

Język C# umożliwia definiowanie wielu metod o tych samych nazwach, pod warunkiem że różnią się parametrami (dzięki temu mają również inne sygnatury). Nazywa się to przeciążaniem metody (ang. *overload*). Niemożliwe jest natomiast definiowanie dwóch metod różniących się jedynie zwracanymi wartościami.

W formalnym języku informatyki rozróżnia się dwa pojęcia, które w tym opisie w zasadzie utożsamiam. Mam na myśli parametry i argumenty metod (funkcji). Parametry (lub inaczej parametry formalne) można rozumieć

jako zmienne zdefiniowane w sygnaturze metody, a dokładniej w jej liście argumentów. Tam wskazywane są typ i nazwa poszczególnych parametrów. Podczas definiowania ciała metody nie wiemy, jaka będzie wartość poszczególnych parametrów. Ta znana jest dopiero w trakcie działania programu, w momencie wywołania metody. Wówczas z parametrem kojarzony jest argument (inaczej parametr aktualny), tj. wartość zmiennej wartościowej lub referencja do obiektu zmiennej referencyjnej, które wstawiamy w instrukcji wywołania metody, na odpowiedniej pozycji wewnątrz nawiasów znajdujących się za nazwą metody. Tym samym w czasie działania metody parametr przyjmuje wartość argumentu. Jeżeli parametry są typu wartościowego, zmiana ich wartości podczas działania metody nie zostaje uwzględniona w wartości argumentu po jej zakończeniu — zmieniana jest bowiem tylko lokalna kopia argumentu[9]. W przypadku parametrów typu referencyjnego z argumentu kopiowana jest tylko referencja do obiektu (bez jego klonowania), a więc wszelkie modyfikacje obiektu wewnątrz metody są widoczne także po jej zakończeniu. Do zagadnienia można podejść bardziej abstrakcyjnie. Bo czymże jest parametr, jeżeli nie miejscem w pamięci zarezerwowanym przez metodę, a argument — wartością zapisywaną w tym miejscu w momencie wywołania metody. Ja jednak, jak wcześniej zastrzegałem, będę pojęć parametr i argument używał zamiennie.

Domyślne wartości argumentów metod — argumenty opcjonalne (nowość języka C# 4.0)

Od wersji 4.0 języka C# możliwe jest wreszcie ustalanie domyślnych wartości parametrów metod. Pokazuję to w listingu 3.3, w którym widoczna jest zmodyfikowana metoda `Metoda`. Dodałem do niej drugi parametr, który ustala tytuł okna komunikatu.

Listing 3.3. Metoda z domyślną wartością argumentu

```
void Metoda(string tekst, string tytuł = "Komunikat")
{
   MessageBox.Show(tekst,tytuł);
}
```

Dzięki temu przy wywołaniu tej metody drugi argument jest opcjonalny — jeżeli nie wystąpi w liście argumentów w instrukcji wywołania metody, jej drugi parametr przyjmie wartość domyślną. Rozwiązanie to, znane od zawsze w C++, pozwala ograniczyć ilość przeciążonych wersji metod. Parametry z domyślną wartością (inaczej argumenty opcjonalne) muszą być jednak zawsze definiowane jako ostatnie.

W naszym przykładzie możliwe jest teraz wywołanie metody `Metoda` zarówno z jednym, jak i z dwoma argumentami:

```
Metoda("Witaj świecie!");
Metoda("Witaj świecie!","Powitanie");
```

Gdybyśmy, zamiast modyfikować istniejącą wersję metody, zdefiniowali jej trzecią wersję, powstałaby niejednoznaczność. Wywołanie metody z jednym argumentem pasowałoby zarówno do jej jednoargumentowej wersji, jak i do wersji dwuargumentowej z pominiętym drugim argumentem. W takim przypadku wywoływana jest wersja, której sygnatura dokładnie pasuje do podanych argumentów, a więc jednoargumentowa.

Zwróćmy uwagę, że nawet nasze dwie wersje przeciążone metody `Metoda`: bezargumentowa i dwuargumentowa, też są w zasadzie niepotrzebne. Bezargumentowej można by się pozbyć, a w zamian dodać wartość domyślną do pierwszego argumentu wersji dwuargumentowej.

Argumenty nazwane (nowość języka C# 4.0)

Kolejną nowością języka C# w wersji 4.0 jest możliwość identyfikacji parametrów nie za pomocą ich kolejności, a przy użyciu ich nazw np.:

```
Metoda(tytuł: "Powitanie", tekst: "Witaj świecie!");
Metoda(tekst: "Witaj świecie!", tytuł: "Powitanie");
```

---

[9] Wyjątkiem jest sytuacja, w której do parametru dodajemy modyfikator `ref` lub `out` (punkt „Zwracanie wartości przez argument metody", w tym rozdziale).

Najważniejszą, moim zdaniem, zaletą tego nowego rozwiązania jest czytelność kodu. Bez sięgania do definicji metody możemy ustalić, jakim parametrom przypisywane są które argumenty. Tym samym trudniej też popełnić błąd w przypadku metod o dużej liczbie parametrów tego samego typu.

Wartości zwracane przez metody

Metoda może zwracać wartość. Na listingu 3.4 pokazuję prosty przykład metody obliczającej kwadrat liczby podanej w argumencie. Pojawiło się w niej nowe słowo kluczowe `return`. To właśnie następująca po nim wartość zostanie zwrócona przez metodę. Ponadto słowo `return` kończy działanie metody, a więc wszelkie polecenia występującej po instrukcji `return` nie są wykonywane[10].

Listing 3.4. Metoda zwracająca wartość i przykład jej użycia

```
private int Kwadrat(byte arg)
{
   return arg * arg;
}

private void button2_Click(object sender, EventArgs e)
{
   int wynik = Kwadrat(2);
   MessageBox.Show(wynik.ToString());
}
```

Zwracanie wartości przez argument metody

Problem pojawia się wtedy, gdy metoda oblicza nie jedną, ale kilka wartości. Dobrym przykładem jest metoda obliczająca dwa pierwiastki równania kwadratowego. Jak wówczas zwrócić wyniki do miejsca jej wywołania? Możliwości są co najmniej dwie. Po pierwsze, możemy zdefiniować strukturę, której polami będą wyniki działania metody i którą metoda będzie zwracać. Jeżeli wyniki są tego samego typu, może to być nawet zwykła tablica lub kolekcja. Drugim sposobem jest zwracanie wartości przez argumenty. Jak pisałem wcześniej, jeżeli argumenty są typu referencyjnego, zmiany stanu obiektu przeprowadzone wewnątrz metody widoczne są po jej zakończeniu. Inaczej jest w przypadku zmiennych typu wartościowego — do parametru kopiowana jest wartość argumentu, po tym ich wartości są od siebie niezależne. Język C# umożliwia jednak potraktowanie parametru typu wartościowego (np. zmiennej liczbowej) jak referencji. Wówczas wszystkie zmiany argumentów dokonane wewnątrz metody będą nadal aktualne po jej zakończeniu. W listingu 3.5 zawarłem przykład, w którym argumenty metody `zakresDouble` pobierane są przez wartości, natomiast w listingu 3.6 metoda ta pobiera już referencje. W obu przykładach metoda `button1_Click` jest domyślną metodą zdarzeniową przycisku, który należy umieścić na formie. Z niej wywoływana jest metoda `zakresDouble`.

Listing 3.5. Przekazywanie argumentów do metody przez wartości

```
private void zakresDouble(double min,double max)
{
    min = double.MinValue;
    max = double.MaxValue;
    MessageBox.Show("Liczby double mogą należeć do przedziału ("+min+","+max+")");
}

private void button1_Click(object sender, EventArgs e)
{
```

[10] Słowo kluczowe `return` może się również pojawić w metodzie niezwracającej wartości. Wówczas oznacza wyłącznie miejsce, w którym działanie metody ma się zakończyć.

```
        double min = 0, max = 0;
        zakresDouble(min, max);
        MessageBox.Show("Liczby double mogą należeć do przedziału ("
                        + min + "," + max + ")");
    }
```

Listing 3.6. Przekazywanie argumentów do metody przez referencje

```
    private void zakresDouble(ref double min,ref double max)
    {
        min = double.MinValue;
        max = double.MaxValue;
        MessageBox.Show("Liczby double mogą należeć do przedziału ("+min+","+max+")");
    }

    private void button1_Click(object sender, EventArgs e)
    {
        double min = 0, max = 0;
        zakresDouble(ref min, ref max);
        MessageBox.Show("Liczby double mogą należeć do przedziału (" + min + "," + max +
")");
    }
```

W obu przykładach komunikaty wyświetlane metodą `MessageBox.Show` z wnętrza metody `zakresDouble` będą — oczywiście — takie same. Jednak te, które wyświetlane są po powrocie do metody `button1_Click`, będą się różniły. W pierwszym przypadku zobaczymy zera (zgodnie z inicjacją zmiennych `min` i `max` w metodzie `button1_Click`). W drugim po dodaniu słów kluczowych `ref` w definicji metody i miejscu jej wywołania zobaczymy w komunikacie wartości ustalone w metodzie `zakresDouble`.

Oprócz słowa kluczowego `ref` można użyć w tym samym kontekście słowa kluczowego `out`. Jedyna różnica polega na tym, że `out` wymusza zmianę wartości, a `ref` tylko ją dopuszcza.

Delegacje i zdarzenia

Dwa słowa kluczowe, na które chciałbym jeszcze zwrócić uwagę, to `delegate` i `event`. Dzięki nim możliwe jest w C# uzyskanie zmiennych przechowujących odniesienia do metod bez jawnego używania wskaźników do metod.

Kod z listingu 3.7 należy umieścić wewnątrz jakiejś klasy, choćby `Form1`. Zadeklarowana jest w nim delegacja `Uchwyt` i metoda `Kwadrat`. Delegacja `Uchwyt` jest typem, którego zmienne mogą przechowywać referencję do metod o określonej w nim sygnaturze (konkretnie do metod pobierających argument typu `byte` i zwracających wartość typu `int`). Metoda `Kwadrat`, której definicja jest również widoczna na listingu 3.7, pasuje do delegacji `Uchwyt`, co oznacza, że odwołanie do niej mogłoby być przechowywane w zmiennej typu `Uchwyt`.

Listing 3.7. Definicja delegacji i zgodnej z nią metody

```
    delegate int Uchwyt(byte arg);

    private int Kwadrat(byte arg)
    {
        return arg*arg;
    }
```

Warto to powtórzyć: `Uchwyt` nie jest zmienną, która może przechowywać adres metody, jest typem takiej zmiennej. Zatem zanim przypiszemy referencję do metody, musimy utworzyć zmienną typu `Uchwyt`. Poniższy kod — należy umieścić go w jakiejś metodzie lub konstruktorze klasy — tworzy taką zmienną i zapisuje do niej

odwołanie do metody `Kwadrat`, a następnie pokazuje, że wywołanie metody w tradycyjny sposób i poprzez delegację wygląda podobnie i daje identyczne efekty (listing 3.8).

Listing 3.8. Definiowanie uchwytu do metody

```
Uchwyt UchwytMetodyKwadrat=new Uchwyt(Kwadrat);


int i = Kwadrat(2);
int j = UchwytMetodyKwadrat(2);
MessageBox.Show("i="+i+", j="+j);
```

Zmiennymi typu określonego przez delegacje (w naszym przypadku typu `Uchwyt`) są również zdarzenia. Zdarzenia można deklarować jedynie jako pola klasy:

```
event Uchwyt ZdarzenieKwadrat;
```

i wówczas są one także zmiennymi składowymi, które mogą przechowywać referencje do metod. Inicjacja zdarzenia odbywa się podobnie jak inicjacja instancji delegacji widocznej na listingu 3.8:

```
ZdarzenieKwadrat+=new Uchwyt(Kwadrat);
int k=ZdarzenieKwadrat(2);
```

Należy sobie jednak zdawać sprawę z zasadniczej różnicy między referencją do metody `UchwytMetodyKwadrat` zdefiniowaną na listingu 3.8 a zdarzeniem `ZdarzenieKwadrat`. Pierwsza może przechowywać referencję tylko do jednej metody, a zdarzenie przechowuje cały zbiór metod. Dlatego w drugim przypadku użyłem operatora `+=`, który dodaje metodę do tego zbioru.

Przykład użycia delegacji i zdarzeń zamieszczono w rozdziale 8., gdzie wykorzystamy te konstrukcje przy projektowaniu komponentów, co jest ich najbardziej typowym zastosowaniem.

Wyrażenia lambda

Od wersji 3.0 języka C# można również stosować tzw. wyrażenia lambda korzystające z nowego operatora `=>`, które mają zastąpić definiowanie anonimowych metod wprowadzonych do C# 2.0. Ogólna składnia wyrażeń lambda to:

```
(parametry) => wartość
(parametry) => {instrukcja;}
```

A oto przykłady:

```
(int n) => n + 1
(x, y) => x == y
n => { MessageBox.Show(n.ToString()); }
```

W pierwszym przypadku mamy do czynienia z odpowiednikiem definicji metody przyjmującej jako argument wielkość typu `int` o nazwie `n` i zwracającej jej wartość zwiększoną o 1. Powyższe wyrażenie może być przypisane do delegacji opisującej typ metody z jednym argumentem typu `int` i zwracającej wartość typu `int`:

```
delegate int DInc(int n);
```

i dalej używane jak normalnie zadeklarowana metoda. Drugie wyrażenie porównuje dwie wartości o niezadeklarowanych typach i zwraca wartość `true`, jeżeli są równe, a `false` w przeciwnym wypadku. Może być zatem przypisane do delegacji zadeklarowanej np. jako:

```
delegate bool DIsEqual(double x, double y);
```

Wreszcie trzecie wyrażenie przyjmuje argument dowolnego typu i pokazuje go w oknie komunikatu. Na listingu 3.9 pokazuję, w jaki sposób można definiować wyrażenia lambda, przypisywać je do delegacji i wykorzystywać.

Listing 3.9. Definicje wyrażeń lambda

```
delegate int DInc(int n);
delegate bool DIsEqual(double x, double y);
delegate void DShow(int n);
```

```
private void button8_Click(object sender, EventArgs e)
{
   DInc Inc = (int n) => n + 1;
   MessageBox.Show("Inc(1)=" + Inc(1));

   DIsEqual IsEqual = (x, y) => x == y;
   int a = 10;
   int b = 20;
   MessageBox.Show("Czy równe a=" + a + " i b=" + b + "? " + (IsEqual(a, b) ? "Tak" :
"Nie"));
   MessageBox.Show("Czy równe a=" + a + " i a=" + a + "? " + (IsEqual(a, a) ? "Tak" :
"Nie"));

   DShow Show = n => { MessageBox.Show(n.ToString()); };
   Show(10);
}
```

Jednak prawdziwe korzyści i cel wprowadzenia wyrażeń lambda do C# 3.0 ujawnią się w rozdziale 16. i następnych. Przedstawię wówczas zbiór rozszerzeń (to specjalny typ metody, który poznamy jeszcze w tym rozdziale) wywoływanych na rzecz kolekcji, których argumentami są delegacje. Dla przykładu klasa `Array`, tak jak i wszystkie pozostałe kolekcje, wyposażona została w metodę `Min` zwracającą najmniejszy element z tablicy, na rzecz której została wywołana. Musimy jej tylko podpowiedzieć, jak ma mierzyć „małość" elementów — i właśnie delegacja metody, która pomiar „małości" wykona, powinna być podana jako argument metody `Min` (listing 3.10).

Listing 3.10. Praktyczne wykorzystanie wyrażenia lambda

```
string[] slowa = { "czereśnia", "jabłko", "borówka", "wiśnia", "jagoda" };
int dlugoscNajkrotszego = slowa.Min(slowo => slowo.Length);
```

W powyższym kodzie wyrażenie `slowo=>slowo.Length` (z kontekstu wynika, że przyjmuje ono zmienną typu `string`, a zwraca jej długość) jest użyte jako argument metody `Min` tablicy łańcuchów. Metoda `Min` wywołuje metodę podaną w argumencie dla każdego elementu tablicy i zwraca jej wartość dla tego elementu, dla którego jest ona najmniejsza. Mówiąc prościej, zwraca długość najkrótszego łańcucha w tablicy.

Dzięki wyrażeniu lambda uniknęliśmy mniej czytelnej konstrukcji zawierającej metodę anonimową:

```
int dlugoscNajkrotszego = slowa.Min(delegate(string slowo) { return slowo.Length; });
```

Typy wartościowe i referencyjne

C# jest językiem, w którym obiektowość wprowadzona jest w pełni konsekwentnie. Nie ma w nim typów prostych, znanych z C++ czy Javy, a wszystkie zadeklarowane przez nas zmienne są w rzeczywistości obiektami. Należy jednak zwrócić uwagę na to, że nie wszystkie obiekty są instancjami klas — w C# mamy bowiem do dyspozycji również struktury. Poza kilkoma ważnymi ograniczeniami[11], definiuje się je identycznie z klasami — z poziomu kodu różnią się one słowem kluczowym `struct` zastępującym słowo `class`. Jest jednak bardzo istotna różnica między nimi w sposobie użycia. Klasy związane są bowiem ze zmiennymi referencyjnymi, struktury — ze zmiennymi typów wartościowych. Co to oznacza? W przypadku typów referencyjnych obiekt będący instancją klasy może powstać wyłącznie na stercie w wyniku użycia operatora `new`, np.:

[11] Struktury nie mogą być rozszerzane ani same nie mogą dziedziczyć po innych klasach lub strukturach. Mogą natomiast implementować interfejsy. Pola struktury, poza polami statycznymi, nie mogą być inicjowane w miejscu definicji (pola inicjowane są przez kompilator wartościami domyślnymi). Programista nie może też samodzielnie zdefiniować dla struktury bezargumentowego konstruktora, ten jest przygotowywany automatycznie przez kompilator.

```
Button b = new Button();
```

W przypadku typów wartościowych obiekty będące instancjami struktur mogą powstać wyłącznie na stosie. Programista nie ma tu żadnej dowolności. Zmienne wartościowe mogą być inicjowane za pomocą operatora `new`, choć w praktyce częściej korzysta się do ich inicjacji ze stałych, np.:

```
Int32 i1 = new Int32();
int i2 = new int();
int i = 1;
```

Jednak to nie miejsce utworzenia jest najważniejszą różnicą między strukturami i klasami, której programiści C# powinni być świadomi. Zasadnicza różnica ujawnia się dopiero przy próbie ich kopiowania. Przyjrzyjmy się dwóm poleceniom:

```
Button a = b;
int j = i;
```

Pierwsze z nich prowadzi do utworzenia nowej referencji do wcześniej zdefiniowanego obiektu `Button`. W drugim tworzona jest nowa zmienna `j` typu `int`. Jednak w pierwszym przypadku nie powstaje nowy obiekt — obie referencje, `a` i `b`, wskazują na tę samą instancję klasy `Button`. Innymi słowy, zmiana wartości `b` oznaczać będzie zmianę wartości `a`. Natomiast w drugim przypadku zmienna `j` i zmienna `i` są związane z zupełnie niezależnymi instancjami struktury `int`. Oznacza to, że zmiana wartości obiektu `i` nie pociągnie za sobą zmiany wartości `j`.

Zmienne wartościowe można w pełni utożsamiać z odpowiadającymi im instancjami struktur. Struktury (typy wartościowe) imitują w ten sposób zachowanie zmiennych prostych.

Kolejna różnica między obiektami będącymi instancjami klas i struktur dotyczy sposobu ich zwalniania. W przypadku struktur sprawa jest w miarę prosta. Obiekt tego typu usuwany jest w momencie wyjścia poza zakres nawiasu `{}`, w którym został zdefiniowany. Jeżeli zatem zdefiniujemy zmienną `i` typu `int` w metodzie, to w momencie zakończenia tej metody pamięć zajmowana przez tą zmienną zostanie zwolniona. Wartość tej zmiennej może być — oczywiście — skopiowana i przekazana np. przez wartość zwracaną przez metodę, ale wówczas tworzone są kopie obiektu, które nie przedłużają życia oryginału. Inaczej sprawa wygląda w przypadku obiektów — instancji klas. Za ich zwalnianie z pamięci odpowiedzialny jest odśmiecacz (ang. *garbage collector*). Zasada jego działania jest prosta. Cyklicznie przeszukuje pamięć sterty w poszukiwaniu obiektów, które nie są wskazywane przez żadną referencję. Jeżeli taki obiekt zostanie znaleziony, jest usuwany. Użytkownik nie ma na to w zasadzie żadnego wpływu (nie ma operatora `delete`), ale też nie musi się tym martwić. Dzięki temu nie ma niebezpieczeństwa wycieku pamięci.

Nullable

Jedną z różnic między typem wartościowym a referencyjnym jest to, że tylko ten drugi może mieć wartość `null`. Czasem możemy jednak chcieć zdefiniować np. liczbę całkowitą typu `int`, która może przyjmować dowolne wartości dodatnie, ujemne i zero, ale oprócz nich także wartość `null`, której możemy np. użyć do zasygnalizowania sytuacji, gdy jakieś obliczenia nie zostały jeszcze wykonane. Możemy wówczas skorzystać z „opakowania” w postaci struktury `Nullable<int>`. Oto przykład:

```
Nullable<int> ni = 1;
int i1;
if (ni.HasValue)
{
   i1 = ni.Value;
}
else
{
   i1 = default(int);
}
int i2 = ni.GetValueOrDefault();
MessageBox.Show("i1=" + i1 + ", i2=" + i2);
```

Zdefiniowaliśmy zmienną typu `Nullable<int>` i zainicjowaliśmy ją wartością 1. Widać więc od razu, że istnieje niejawna konwersja z typu `int` na typ `Nullable<int>` (lub, ogólniej, z typu `T` na `Nullable<T>`). Równie dobrze do inicjacji moglibyśmy użyć słowa kluczowego `null`. O tym, czy zmienna ma wartość, czy jest pusta (tj. właśnie równa `null`), możemy się przekonać, sprawdzając jej własność `HasValue`. Jeżeli zmienna nie jest pusta, jej wartość możemy odczytać, korzystając z własności tylko do odczytu `Value`. Udostępnia ona „opakowaną" przez typ `Nullable<int>` zasadniczą zmienną typu `int`.

Obie czynności, a więc sprawdzenie, czy zmienna ma wartość, i ewentualne jej odczytanie, realizuje metoda `GetValueOrDefault`. W wersji bez argumentu, jeżeli zmienna równa jest `null`, metoda zwraca domyślną wartość typu (uzyskiwaną słowem kluczowym `default`). Wersja z argumentem pozwala na ustalenie własnej, zwracanej przez metodę wartości — oczywiście, w przypadku braku wartości odpytywanej zmiennej.

W praktyce, zamiast korzystać z własności `HasValue`, zazwyczaj stosuje się porównanie do wartości `null`, np.:

```
int i3 = (ni == null) ? -1 : ni.Value;
```

Tę konstrukcję można zresztą skrócić, jeżeli użyjemy operatora `??`, np.:

```
int i4 = ni ?? -1;
```

Powyższa instrukcja jest w pełni równoważna tej poprzedniej.

Warto również wiedzieć, że zamiast jawnego używania typu `Nullable<int>` możemy skorzystać z wygodnego aliasu w postaci znaku zapytania za typem wartościowym:

```
int? ni=1;
```

Pudełkowanie

Zmienne typów wartościowych można zamknąć w pudełku także w inny sposób, korzystając przy tym z jego klasy bazowej `object`, będącej skądinąd typem referencyjnym. Proces ten nazywa się *boxing*, co chyba można przetłumaczyć jako pudełkowanie. Nie potrzeba do tego żadnych specjalnych operatorów. Wystarczy zwykłe przypisanie:

```
int i = 1;
object o = i;
MessageBox.Show(o.GetType().ToString()); //zwraca System.Int32
```

Wartość zmiennej całkowitej, a więc zmiennej typu wartościowego, zamkniętej w zmiennej typu `object`, która jest typu referencyjnego, kopiowana jest ze sterty na stos. Nowa zmienna zachowuje informacje o typie oryginalnej zmiennej, o czym można się przekonać, korzystając z metody `GetType`.

Oryginalną wartość można wydobyć z pudełka, stosując *unboxing* (nie odważę się tłumaczyć tego terminu). W praktyce oznacza to rzutowanie na typ `int`. Musi to być jednak rzutowanie jawne. W istocie jest to jednak kopiowanie wartości ze stosu z powrotem na stertę.

```
int j = (int)o;
MessageBox.Show(j.ToString());
```

Do czego używać pudełkowania? W .NET 1.0 i 1.1 było ono fundamentem kolekcji, które potrafiły przechowywać referencje typu `object`. Bez pudełkowania przechowywanie instancji struktur w nieparametrycznych kolekcjach wiązałoby się z utratą informacji o obiekcie, co z kolei niweczyłoby cały pomysł.

Typy dynamiczne (nowość języka C# 4.0)

C# jest językiem bezpiecznym. Dotychczas wiązało się to m.in. ze ścisłą kontrolą typów. W C# 4.0 ta zasada została nieco nagięta. Możliwe jest bowiem definiowanie zmiennych, których typ nie jest ustalony w trakcie kompilacji. Co więcej, nie jest kontrolowany — może ulec zmianie przy każdym przypisaniu nowej wartości. Zmienne te nazywane są zmiennymi dynamicznymi, a do ich deklaracji używane jest nowe słowo kluczowe `dynamic`. Osobom znającym Delphi natychmiast powinien przypomnieć się typ `Variant`. Także użytkownicy środowisk Matlab i IDL, w których kontrola typów w ogóle jest ograniczona do minimum, powinni poczuć się

jak w domu. Od razu należy jednak zastrzec, że używanie zmiennych dynamicznych nie powinno stać się regułą. Ich nadużywanie jest zdecydowanie szkodliwe. Powinny być używane tylko w wyjątkowych sytuacjach.

Słowo kluczowe `dynamic` pozwala na definiowanie zmiennych lokalnych, pól, własności, indeksatorów czy wartości pobieranych i zwracanych przez metody. We wszystkich tych przypadkach w instrukcjach przypisania zawieszana jest statyczna kontrola typów. Skompiluje się zatem fragment kodu widoczny na listingu 3.11. Co więcej, jego wykonanie nie spowoduje wyjątku.

Listing 3.11. Dynamiczna zmiana typu zmiennej

```
dynamic o; //nie działa IntelliSense
o = 5; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 5L; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = "Helion"; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 1.0f; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 1.0; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
```

Zwróćmy uwagę, że typ, jaki posiada zmienna `o`, ustalany jest w momencie realizacji instrukcji przypisania, tzn. dopiero w momencie działania programu. Różni je to od zmiennych zadeklarowanych z użyciem słowa kluczowego `var`, których typ znany jest już w momencie kompilacji. Kolejna różnica względem `var` to fakt, że typ zmiennej ustalony w momencie pierwszego przypisania (inicjacji) nie musi być zachowany. Następne przypisanie może go zmienić, czego dowodzą komunikaty wyświetlane podczas wykonywania kodu z listingu 3.11.

Jak wspomniałem, możliwe jest definiowanie nie tylko lokalnych zmiennych dynamicznych, ale również takich pól i własności. Również metoda może zwracać obiekt typu `dynamic`. Oznacza to np., że w zależności od decyzji użytkownika podjętej już w momencie działania programu metoda zwraca liczbę 5 typu `int`, napis „Helion” lub referencję do przycisku (listing 3.12).

Listing 3.12. Typ dynamic może być użyty we wszystkich kontekstach

```
public partial class Form1 : Form
{
   ...

   dynamic obiekt; //pole

   dynamic Obiekt //własność
   {
      get
      {
         return zwrocObiekt();
      }
      set
      {
         obiekt = value;
      }
   }

   dynamic zwrocObiekt(int ktoryObiekt = 0) //wartość zwracana przez metodę
   {
      dynamic wartosc;
      switch(ktoryObiekt)
      {
         case 0: wartosc = 5; break;
```

```
            case 1: wartosc = 5L; break;
            case 2: wartosc = "Helion"; break;
            case 3: wartosc = 1.0f; break;
            case 4: wartosc = 1.0; break;
            case 5: wartosc = button1; break;
            default: wartosc = obiekt; break;
         }
         return wartosc;
      }

      private void button34_Click(object sender, EventArgs e)
      {
         try
         {
            dynamic o = zwrocObiekt(comboBox1.SelectedIndex);
            MessageBox.Show("Obiekt: " + o.ToString() + ", typ: " + o.GetType().FullName);
            o.Metoda(); //tu pojawi się wyjątek
         }
         catch (Exception exc)
         {
            MessageBox.Show("Błąd: " + exc.Message);
         }
      }
   }
```

Warto sobie uświadomić, że podobny efekt można było uzyskać już dawniej. Umożliwia to typ `object` — klasa bazowa wszystkich typów — który może być traktowany jak odpowiednik wskaźnika na `void` w C++. Korzystając z typu `object`, na który można zrzutować dowolny typ platformy .NET, możemy również napisać kod widoczny na listingu 3.13, który jest przecież podobny do tego z listingu 3.11. I choć rzeczywiście efekt wykonania obu fragmentów kodu jest bardzo podobny, ich działanie jest zupełnie inne. Obiekt typu `dynamic` rzeczywiście zmienia swój typ. Do zmiennej typu `object`, która jest typu referencyjnego, zapisywane są referencje kolejnych obiektów (stare obiekty usuwane są przez *garbage collector*). Przy tym w przypadku obiektów typu wartościowego dochodzi proces pudełkowania, opisany wyżej.

Listing 3.13. Użycie typu object może dać podobne efekty jak zastosowanie typu dynamic

```
object o;
o = 5; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 5L; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = "Helion"; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 1.0f; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
o = 1.0; MessageBox.Show(o.ToString() + ", " + o.GetType().FullName);
```

Możliwość swobodnej i wielokrotnej zmiany typu zmiennej dynamicznej nie jest jednak zaletą. To raczej cena, jaką trzeba zapłacić za korzyści płynące z braku kontroli typów. Jedną z nich jest łatwość, z jaką na rzecz dynamicznych obiektów można wywoływać metody, odczytywać lub przypisywać ich własności itd. Wszystko to — oczywiście — pod warunkiem że programista rzeczywiście wie, z jakimi obiektami ma do czynienia. Tak naprawdę, to na niego przerzucona jest teraz kontrola typów. Załóżmy dla przykładu, że dysponujemy biblioteką DLL, która nie jest ładowana statycznie i w ten sposób kompilator nie może rozpoznać automatycznie jej zawartości. Przed pojawieniem się platformy .NET 4.0 i C# 4.0 zmuszeni bylibyśmy do korzystania z dynamicznego rozpoznawania typów (mechanizm *reflection* platformy .NET). Jeżeli jednak jesteśmy pewni, że biblioteka zawiera klasę o nazwie `Klasa`, a klasa ta zawiera metodę o nazwie `Metoda` o konkretnej sygnaturze, to korzystając z obiektu typu `dynamic`, bez weryfikowania tych faktów możemy utworzyć instancję tej klasy i wywołać jej metodę. Jeżeli klasy lub metody jednak nie będzie tam, gdzie się ich spodziewamy, nic bardzo

strasznego się nie stanie — po prostu zostanie zgłoszony wyjątek. W końcu implementacja dynamicznego typu również oparta jest na mechanizmie *reflection*.

Klasyczny scenariusz z dynamicznym rozpoznawaniem zawartości biblioteki DLL za pomocą mechanizmu refleksji opisany jest w rozdziale 30. Nie jest on zresztą tak wydumany, jak w pierwszej chwili mogłoby się wydawać. Z podobnym problemem mamy do czynienia, gdy np. korzystamy z obiektów COM (zob. rozdział 35). To tego typu sytuacje były przecież powodem wprowadzenia typu `Variant` do Delphi.

Późne wiązanie typu zmiennej wiąże się z brakiem wsparcia IntelliSense podczas edycji kodu. Możliwe jest jednak instalowanie dodatkowych narzędzi (np. ReSharper), dzięki którym pewne informacje staną się dostępne.

Należy wyraźnie rozróżniać sytuacje, w których typ jest trudny do ustalenia, jak np. typ kolekcji zwracanej przez zapytanie LINQ, od sytuacji, w których ustalenie typu w ogóle nie jest możliwe przed uruchomieniem aplikacji. W sytuacjach pierwszego rodzaju należy bezwzględnie korzystać ze słowa kluczowego `var`. Warto powtórzyć jeszcze raz, że różni się on od `dynamic` tym, że typ zmiennej zadeklarowanej z użyciem słowa kluczowego `var` jest ustalany już w trakcie kompilacji na podstawie wartości przypisanej zmiennej podczas inicjacji, a następnie nie ulega zmianie. W przypadku słowa kluczowego `dynamic` typ nie jest znany także po kompilacji. Jest ustalany dopiero w momencie działania programu. Co więcej, nie musi być także zachowany.

W praktyce, choć nadal mam wątpliwości co do zasadności wprowadzania typu dynamicznego (i idących za tym zmian w platformie .NET[12]), z pewnością jego obecność pozwoli uniknąć tworzenia długiego kodu, w którym, korzystając z refleksji dającej możliwości odczytywania typów obiektów i ich składowych metod, własności i pól, próbowaliśmy poradzić sobie z obiektami, jakich typ ustalany był dopiero w trakcie działania programu, co często ma miejsce w przypadku prób korzystania z kodu niezarządzanego, w tym z technologii COM. Nie mam jednak wątpliwości, że należy go unikać we wszystkich pozostałych sytuacjach.

Sterowanie przepływem

Umiejętność programowania była dotychczas kojarzona głównie ze znajomością instrukcji sterujących języka. Obecnie środek ciężkości przesuwa się w kierunku poznawania bogatych bibliotek klas związanych z Javą, Delphi lub C#. Mimo to, nadal nie da się uniknąć nauki instrukcji warunkowych czy pętli.

Instrukcja warunkowa if..else

Składnia tej instrukcji jest następująca: `if(warunek) instrukcja;`. Oznacza ona tyle, że `instrukcja` jest wykonywana jedynie wtedy, gdy `warunek` ma wartość równą `true`. Składnia rozszerzona to `if(warunek) instrukcja; else alternatywna_instrukcja;`. Tu `alternatywna_instrukcja` jest realizowana, gdy `warunek` nie jest spełniony. W listingu 3.14 zawarłem przykłady użycia tej instrukcji.

Listing 3.14. Przykład wykorzystania instrukcji if i if..else

```
Random r=new Random();
int n=r.Next(8);
MessageBox.Show(""+n);

//Składnia podstawowa if
if (n<6) MessageBox.Show("Wylosowana liczba jest mniejsza od 6.");

//Składnia rozszerzona if..else
if (n<=4) MessageBox.Show("Wylosowana liczba jest mniejsza lub równa 4.");
else MessageBox.Show("Wylosowana liczba jest większa od 4.");
```

[12] Warto w tym kontekście przeszukać MSDN z hasłem DLR (ang. *Dynamic Language Runtime*).

Instrukcja wyboru switch

Listing 3.15 zawiera przykładowy kod korzystający z instrukcji wyboru `switch`.

Listing 3.15. Przykład wykorzystania instrukcji switch

```
Random r=new Random();
int n=r.Next(8);
string opis;
switch(n)
{
   case 1: opis="niedziela"; break;
   case 2: opis="poniedziałek"; break;
   case 3: opis="wtorek"; break;
   case 4: opis="środa"; break;
   case 5: opis="czwartek"; break;
   case 6: opis="piątek"; break;
   case 7: opis="sobota"; break;
   default: opis="błąd!"; break;
}
MessageBox.Show("Dzień tygodnia: "+n+", "+opis);
```

W pierwszej linii tworzymy obiekt — generator liczb pseudolosowych (konstruktor domyślny klasy `Random` inicjuje generator za pomocą zależnego od czasu ziarna). Następnie losowo wybraną liczbę z zakresu [0, 8) przypisujemy do zmiennej całkowitej `n`. Deklarujemy także łańcuch `opis`. Załóżmy, że `n` jest numerem dnia tygodnia. Dzięki instrukcji `switch` możemy w łatwy sposób przypisać do łańcucha `opis` nazwę dnia tygodnia w zależności od wartości liczby `n`. Każdy z opisanych w konstrukcji `switch` przypadków rozpoczyna się od słowa kluczowego `case` i obowiązkowo kończy się poleceniem `break`. Między nimi mogą znaleźć się dowolne instrukcje C#.

Za pomocą `case` określiliśmy czynności dla wartości od 1 do 7. Co pewien czas wylosowana będzie jednak również liczba 0. Wówczas wykorzystana zostanie linia rozpoczynająca się od słowa kluczowego `default`. Jej obecność nie jest w C# obowiązkowa. Jeżeli jednak jej zabraknie, w powyższym kodzie kompilator zgłosi błąd informujący, że w ostatniej linii zmienna `opis` jest wykorzystywana, choć może nie być zainicjowana. W naszym przykładzie taka sytuacja może rzeczywiście mieć miejsce, gdy `n` będzie miało wartość `0`.

Pętle

Pętle są przykładem zagadnienia, o którym można powiedzieć krótko (tabela 3.5) lub mówić bez końca, omawiając poszczególne rodzaje pętli oraz ich typowe i nietypowe zastosowania. Wybieram pierwszy sposób, ponieważ w kodach, które znajdą w następnych rozdziałach, pętle będą wykorzystywane w sposób naturalny i typowy.

Tabela 3.5. Typy pętli dostępne w C#

Typ pętli	Przykład

`for (inicjacja_indeksu;warunek;inkrementacja)instrukcja;`
Typowa pętla:
`for(int i=0;i<ilosc;i++) instrukcja;`
Pętla nieskończona:
`for(;true;) instrukcja;`
	`long silnia(byte arg)`
	`{`
	`  if (arg==0) return 0;`
	`  long wartosc=1;`
	`  for(byte i=1;i<=arg;i++)`
	`  wartosc*=i;`
	`  return wartosc;`
	`}`

`while (warunek) instrukcja;`	`long najwiekszyDzielnik(long arg)`
	`{`
	`  long dzielnik=arg-1;`
	`  while(arg%dzielnik!=0)`
	`  dzielnik--;`
	`  return dzielnik;`
	`}`

`do instrukcja while (warunek);`	`Random r=new Random();`
	`int n=r.Next(8);`
	`//Zadanie komputera to odgadnąć`
	`//liczbę z zakresu od 0 do 7`
	`int z; int licznik=0;`
	`do`
	`{`
	`  licznik++;`
	`  z=r.Next(8);`
	`}`
	`while (n!=z);`
	`MessageBox.Show("Komputer zgadł liczbę "+z+" po "+licznik+" próbach!");`

Wszystkie powyższe pętle są wykonywane, dopóki *warunek* ma wartość `true`. *Instrukcja* może być również blokiem instrukcji otoczonych operatorem zakresu `{ }`.

Różnica między pętlami `while` i `do..while` polega na tym, że w pierwszej *warunek* sprawdzany jest przed wykonaniem *instrukcji*, a w `do..while` dopiero po wykonaniu. Z tego wynika, że stosując pętlę `do..while`, mamy gwarancję, iż *instrukcja* zostanie wykonana przynajmniej raz nawet wtedy, jeżeli *warunek* nigdy nie zostanie spełniony.

Instrukcje wykonywane w każdej iteracji pętli mogą być przerwane za pomocą poleceń `break` i `continue`. Słowo kluczowe `break` przerywa w ogóle działanie pętli i przechodzi do instrukcji znajdującej się bezpośrednio za nią. Natomiast `continue` przerywa działanie bieżącej iteracji i rozpoczyna następną. W przypadku pętli `for` oznacza to odpowiednie zwiększenie indeksu, zgodnie z poleceniem inkrementacji (listing 3.16).

Listing 3.16. Przykład pętli for z instrukcją continue

```
for (int i=-1; i<=1; i++)
   for (int j=-1; j<=1; j++)
   {
      if (i==0 && j==0) continue; //pomijamy przypadki, gdy obie zmienne równe są 0
      MessageBox.Show("i=" + i + ", j=" + j);
   }
```

W powyższym przykładzie pokazanych zostanie tylko osiem komunikatów. Przypadek, gdy oba indeksy są równe `0`, zostanie bowiem pominięty. Gdybyśmy zamiast `continue` wstawili `break`, pętla wykonałaby tylko cztery iteracje.

Istnieje jeszcze jeden rodzaj pętli — `foreach` — który związany jest ściśle z kolekcjami i dlatego zostanie omówiony w dalszej części rozdziału.

Wyjątki

Ważnym elementem nowoczesnego programowania jest korzystanie z wyjątków (ang. *exception*) do sygnalizowania błędów. W C# przez wyjątek można rozumieć obiekt, który przenosi wiadomość o błędzie z miejsca jego wystąpienia, tzn. z miejsca, z którego wyjątek jest zgłaszany, do miejsca jego obsługi.

W poniższym przykładzie wyjątek jest generowany podczas wykonywania operacji dzielenia, ponieważ dzielnik (zmienna `x`) jest równy `0`[13]. Obszar, w którym śledzone jest ewentualne wystąpienie wyjątków, jest otoczony blokiem następującym po słowie kluczowym `try`. Ewentualny wyjątek, który może tam powstać, zostanie przekazany do sekcji `catch` wykonywanej jedynie w takim przypadku. Należy mieć świadomość tego, że po przejściu do sekcji `catch` nie ma już możliwości powrotu do sekcji `try`; pozostała część znajdujących się tam poleceń nie zostanie wykonana, a po wykonaniu poleceń w sekcji `catch` wątek przejdzie do poleceń znajdujących się po konstrukcji `try..catch` (listing 3.17).

Listing 3.17. Przykład konstrukcji try..catch przechwytującej wyjątek dzielenia przez 0

```
try
{
   int x=0;
   int y=1/x;
   MessageBox.Show("To polecenie nie zostanie wykonane!");
}
catch(Exception exc)
{
   MessageBox.Show("Wyjątek: "+exc.Message);
}
```

Po sekcji `try` może znajdować się kilka sekcji `catch` wychwytujących poszczególne typy wyjątków. Nazywa się to filtrowaniem wyjątków. Na listingu 3.18 pokazany jest prosty przykład działania takiego mechanizmu.

Listing 3.18. Przykład wielokrotnej sekcji catch przechwytującej coraz bardziej ogólne klasy wyjątków

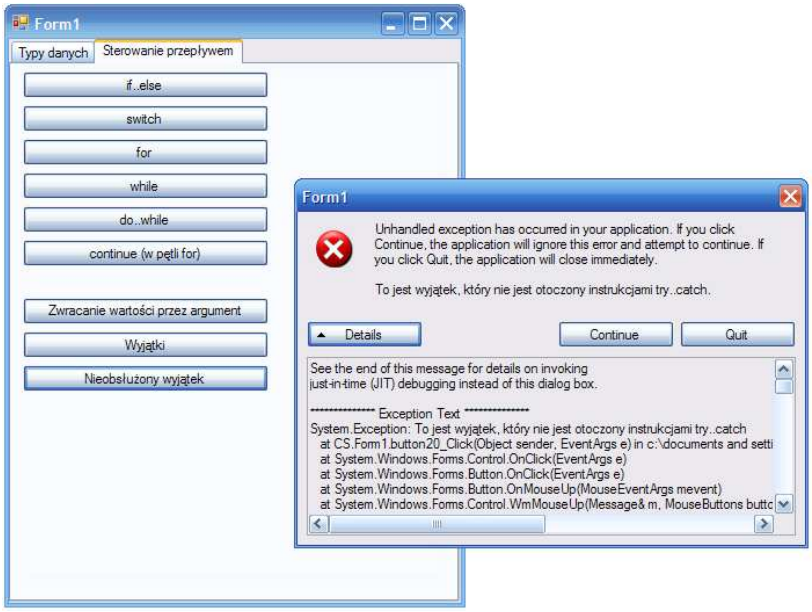
```
try
{
   int x=0;
   int y=1/x;
}
catch(DivideByZeroException exc)
{
   MessageBox.Show("Dzielenie przez zero ("+exc.Message+")");
}
catch(Exception exc)
{
   MessageBox.Show("Inne wyjątki\n"+exc.Message);
}
```

Sekcje `catch` powinny być ułożone od wychwytującej najbardziej szczegółową klasę wyjątku do najbardziej ogólnej, tj. od klasy wielokrotnie dziedziczącej po `Exception` do samego `Exception`. W przeciwnym razie, gdybyśmy na pierwszym miejscu ustawili sekcję `catch` przechwytującą wyjątki typu `Exception`, to ze względu na to, że mogą być na ten typ niejawnie rzutowane wszystkie inne klasy wyjątków (jako ich klasy bazowe), automatycznie wychwytywałaby ona wszystkie wyjątki, nie pozostawiając nic dla kolejnych sekcji `catch`.

---

[13] Konieczne było dzielenie przez zmienną równą 0 zamiast jawnie przez 0, żeby oszukać kompilator. Warto również wiedzieć, że dzielenie przez 0 jest niedopuszczalne w liczbach całkowitych, ale np. w przypadku liczb `double` wyjątek nie zostałby zgłoszony, a zmienna przyjęłaby wartość równą `Infinity` lub `-Infinity`.

Wystąpienie wyjątku nie oznacza zakończenia działania aplikacji (*vide* przycisk *Kontynuuj* w oknie komunikatu na rysunku 3.1). Zakończy się jednak działanie metody, w której on wystąpił[14].



Rysunek 3.1. Nieobsłużony wyjątek przechwycony przez platfomę .NET

Składnia konstrukcji obsługującej wyjątek może być rozszerzona o sekcję `finally`. Polecenia z tej sekcji wykonywane są w każdym przypadku, bez względu na to, czy wystąpił wyjątek, czy nie. Także wówczas, gdy w sekcji `catch` znajdzie się instrukcja `return` lub inne polecenie kończące działanie metody. W razie wystąpienia wyjątku polecenia z tej sekcji wykonywane są po poleceniach z sekcji `catch`. Typowym wykorzystaniem sekcji `finally` jest zwalnianie zasobów, np. zamknięcie pliku otwartego w sekcji `try`. Na listingu 3.19 pokazuję dość banalny przykład wykorzystania sekcji `finally`.

Listing 3.19. Przykład wykorzystania sekcji finally

```
try
{
   int x=0;
   int y=1/x;
}
catch(DivideByZeroException exc)
{
   MessageBox.Show("Dzielenie przez zero ("+exc.Message+")");
   return;
}
catch(Exception exc)
{
   MessageBox.Show("Inne wyjątki\n"+exc.Message);
   return;
}
finally
{
   MessageBox.Show("Kod wykonywany w każdym przypadku");
```

[14] Inaczej jest, gdy tę samą aplikację uruchomimy na Windows Mobile. Wówczas nieobsłużony wyjątek (tzn. nieotoczony konstrukcją `try..catch`) powoduje zamknięcie aplikacji poprzedzone komunikatem o wystąpieniu błędu.

```
}
```

Do zgłaszania wyjątku służy słowo kluczowe `throw` z następującą po nim referencją do obiektu wyjątku. Zatem w momencie, w którym wykryjemy krytyczny błąd programu, należy w kodzie umieścić instrukcję analogiczną do poniższej:

```
throw new Exception("Komunikat opisujący błąd");
```

Dyrektywy preprocesora

Termin **dyrektywy preprocesora** może być mylący dla osób znających dobrze C++, bo w odróżnieniu od C++ w C# nie ma osobnego etapu preprocesora podczas kompilacji kodu, zatem tzw. dyrektywy preprocesora są analizowane równocześnie z analizą kodu języka C#. Wobec tego w C# nie można użyć dyrektyw do tworzenia makr, ich głównym zadaniem jest warunkowa kompilacja kodu.

Kompilacja warunkowa — ostrzeżenia

Poniżej znajduje się przykład wykorzystania dyrektywy do generowania ostrzeżenia, które przypomina, aby przed rozpowszechnieniem programu skompilować go z opcjami usuwającymi informacje przeznaczone dla debuggera i włączoną optymalizacją. Poniższe dyrektywy można umieścić na samym początku pliku z kodem źródłowym:

```
#if DEBUG
   #warning Pamiętaj o skompilowaniu ostatecznej wersji z optymalizacją
#endif
```

Symbol preprocesora `DEBUG`, którego obecność sprawdzamy w powyższym kodzie, może być zdefiniowany przez środowisko Visual Studio, jeżeli zaznaczymy opcję *Define DEBUG constant* na zakładce *Build* własności projektu (ostatnia pozycja w menu *Project*). Dyrektywa `#if` pełni podobną rolę jak instrukcja `if` z języka C# — podczas kompilacji uwzględniany jest kod znajdujący się między `#if` i `#endif`, jeżeli spełniony jest wskazany warunek (np. obecność symbolu preprocesora). Instrukcja ta może mieć również rozbudowaną postać:

```
#if DEBUG
   #warning Kompilacja "debug"
#else
   #warning Kompilacja "release"
#endif
```

Oczywiście, różnica między dyrektywą `#if` a instrukcją `if` jest bardzo istotna. Sprawdzenie warunku znajdującego się za dyrektywą `#if` odbywa się podczas kompilacji, a nie w trakcie działania programu. Zatem jeżeli w konstruktorze umieścimy taką instrukcję:

```
#if DEBUG
   MessageBox.Show("Kompilacja \"debug\"");
#endif
```

to przy każdym uruchomieniu aplikacji zobaczymy odpowiedni komunikat — nie dlatego, że wartość `DEBUG` jest sprawdzana za każdym razem. Została ona sprawdzona raz podczas kompilacji i jeżeli warunek dyrektywy `#if` został spełniony, instrukcja `MessageBox.Show` została „na stałe” umieszczona w kodzie pośrednim.

Definiowanie stałych preprocesora

Przy użyciu dyrektyw preprocesora możliwe jest przygotowanie jednego kodu w taki sposób, żeby łatwe było jego kompilowanie w różnych warunkach. Jak wspomniałem, typowym celem dyrektyw preprocesora jest kompilacja warunkowa, która umożliwia np. przygotowanie kodu do kompilacji przez różne kompilatory i na odmiennych platformach.

Dla przykładu można wyobrazić sobie klasę, która korzysta z przycisku z biblioteki `System.Windows.Forms` lub `System.Web.UI.WebControls`, w zależności od obecności symbolu preprocesora o nazwie `Web`. Od jego obecności zależy to, która przestrzeń nazw zostanie zadeklarowana:

```
#define Web
#if Web
   using System.Web.UI.WebControls;
#else
   using System.Windows.Forms;
#endif
```

W pierwszej linii za pomocą dyrektywy `#define` zdefiniowany jest symbol preprocesora `Web`. Jego obecność powoduje, że zadeklarowana jest przestrzeń nazw `System.Web.UI.WebControls`. Usunięcie lub oznaczenie tej linii komentarzem spowoduje zmianę deklarowanej przestrzeni nazw na `System.Windows.Forms`. W analogiczny sposób od obecności symbolu preprocesora można uzależnić wybór całych grup instrukcji języka C#. I to jest właśnie cel kompilacji warunkowej: aby zawartością jednej linii kontrolować przebieg procesu kompilacji. Bez tej możliwości konieczne byłoby utrzymywanie dwóch równoległych wersji kodu lub ciągłe komentowanie i usuwanie komentarzy całych bloków instrukcji.

Definiowanie symboli preprocesora może mieć miejsce jedynie przed pierwszym poleceniem języka C#, co w praktyce oznacza, że dyrektywy `#define` powinny znaleźć się przed blokiem poleceń `using`. Ponadto stałe preprocesora są widoczne jedynie w obrębie pliku, w którym zostały zdefiniowane. Jeżeli potrzebujemy stałych globalnych, należy je dopisać do pola *Conditional compilation symbols* w opcjach projektu (ostatnia pozycja w menu *Project*) na zakładce *Build*.

Bloki

Do wykorzystania symboli preprocesora i warunkowej kompilacji dochodzi raczej w większych projektach, w których unika się utrzymywania wielu wersji kodu nawet przy tworzeniu kilku wersji programu. W mniejszych i prostszych projektach, które nie przekraczają kilkuset linii kodu źródłowego, najważniejszym zastosowaniem dyrektyw jest zazwyczaj budowanie bloków kodu — za pomocą dyrektyw `#region` i `#endregion` (listing 3.20).

Listing 3.20. Przykład bloku instrukcji zdefiniowanego w kodzie C#

```
#region Metody związane z obsługą przycisków
private void button1_Click(object sender, EventArgs e)
{
    //pierwsza metoda
}

private void button2_Click(object sender, EventArgs e)
{
    //druga metoda
}

private void button1_Click(object sender, EventArgs e)
{
    //trzecia metoda
}
#endregion
```

Nie są to dyrektywy, które wpływają na działanie kompilatora. Ten całkowicie je ignoruje. Mają natomiast wpływ na edytor. Definiują blok kodu źródłowego, który może być zwinięty do jednej linii w podobny sposób, jak można zwinąć każdą metodę czy klasę. Jeżeli edytor znajdzie parę dyrektyw `#region` i `#endregion`, to natychmiast w linii, w której znajduje się pierwsza z nich, pojawi się kwadracik z minusem. Kliknięcie go

zwinie zaznaczony w ten sposób blok. Zwijanie nieedytowanych partii kodu bardzo poprawia jego czytelność, dlatego zachęcam do grupowania np. pól, metod prywatnych, metod zdarzeniowych itd. w oddzielnych blokach.

Atrybuty

Atrybuty są kolejnym sposobem wpływania na proces kompilacji. Stanowią źródło dodatkowych informacji dla kompilatora. Przykładowo atrybuty umieszczone w pliku *Properties\AssemblyInfo.cs* dostarczają kompilatorowi informacje o opisie i wersji pliku wykonywalnego lub biblioteki, którą umieszcza w skompilowanym kodzie pośrednim. Z kolei atrybuty znajdujące się bezpośrednio przed definicją metod pozwalają na poinformowanie kompilatora o własnościach lub ograniczeniach tej metody. Dobrym przykładem jest wspomniany w poprzednim rozdziale atrybut `MTAThread` znajdujący się przed metodą `Main` aplikacji *Kolory*. Z kolei atrybut `DllImport` pozwala deklarować metodę, której definicja znajduje się w niezarządzanej bibliotece DLL. Za pomocą atrybutów można także zaznaczyć metodę jako wychodzącą z użycia (atrybut `Obsolete`). Jeżeli umieścimy ten atrybut przed metodą:

```
[Obsolete("Metoda przestarzała. Lepiej użyj metody NowaMetoda",false)]
```

to w momencie, gdy kompilator napotka na jej wywołanie, pokaże komunikat informujący, że używamy metody zaznaczonej jako przestarzała. Jeżeli drugi argument jest równy `false` lub nie ma go w ogóle, komunikat będzie ostrzeżeniem. Gdy drugi argument to `true`, pojawi się błąd.

Atrybuty wykorzystywane są również do określania grup, do których trafiają własności i zdarzenia projektowanych w oknie własności komponentów oraz ich opisy widoczne na dole tego okna. Warto również zwrócić uwagę na atrybut `Conditional` wykorzystywany w kompilacji warunkowej i wiele innych.

Kolekcje

Tablice to istotny element niemal każdego języka programowania, ale oprócz tablic istnieją także inne struktury danych: listy, kolejki, drzewa i różne ich podtypy. Wszystkie one, łącznie z tablicami, zostały nazwane w C# kolekcjami. Co więcej, większość typowych kolekcji (a także kilka dodatkowych) została zaimplementowana w platformie .NET i są dostępne w przestrzeni nazw `System.Collections`. Wyjątkiem jest tablica, której definicja znajduje się bezpośrednio w przestrzeni `System`. W wersji 2.0 platformy .NET dodane zostały dwie nowe przestrzenie nazw: `System.Collections.Generic` oraz `System.Collections.Specialized`. Pierwsza zawiera kolekcje zdefiniowane za pomocą klas sparametryzowanych, druga — ich wyspecjalizowane wersje. Warto zwrócić szczególną uwagę na często używaną listę zaimplementowaną w klasie `List<>` oraz na `SortedList<>` — słownik, w którym elementy są automatycznie sortowane.

„Zwykłe” tablice

Definicja tablicy w dowolnym języku programowania jest zawsze taka sama: tablica to struktura, która przechowuje elementy tego samego typu. Inną sprawą jest jej implementacja. W C# zrobiono to w taki sposób, aby niemożliwe było popełnienie często występującego w C/C++ błędu, w którym dochodzi do odczytu lub zapisu danych „za” zadeklarowanym obszarem tablicy. W C# indeksy są kontrolowane przez klasę implementującą tablicę. Bo o ile w C/C++ tablica to po prostu fragment pamięci z adresem zapamiętanym we wskaźniku do pierwszego elementu, to w C# jest ona instancją klasy `System.Array`. Oznacza to, że tablica jest obiektem. Dzięki temu możliwe jest łatwe kontrolowanie tego, czy indeksy nie są mniejsze od zera lub czy nie są większe od rozmiaru tablicy.

Pomimo że implementacja jest całkiem inna, praktyczne zasady dotyczące tworzenia tablic w C# są niemal identyczne z zasadami rządzącymi **dynamicznie** tworzonymi tablicami w C++[15]. Możemy zadeklarować tablicę, co oznacza utworzenie referencji do tablicy, oraz ją zdefiniować, czyli zarezerwować pamięć, a następnie zainicjować, czyli zapełnić wartościami.

---

[15] Dynamicznie, czyli za pomocą operatora `new`. Deklarowanie tablic w stylu `int i[3];`, czyli tak jak tablic na stosie w C++, w C# nie jest możliwe.

Oto poprawna składnia deklaracji referencji do tablicy z elementami typu `int`:

```
int[] i;
```

A oto przykład deklaracji referencji wraz z utworzeniem obiektu tablicy (rezerwowana jest pamięć na stercie):

```
int[] i=new int[3];
```

Po utworzenia obiektu tablicy (tj. w istocie instancji klasy `System.Array`) jest ona automatycznie inicjowana wartościami domyślnymi dla danego typu (tabela 3.1), czyli w przypadku typu `int` — zerami. Możemy także samodzielnie zainicjować elementy tablicy:

```
int[] i=new int[3] {1,2,4};
```

W tablicy zawierającej typy referencyjne domyślne wartości elementów tablicy to `null`. Konieczna jest wobec tego jawna inicjacja elementów przez wywołanie operatora `new` dla każdego elementu tablicy i utworzenie odpowiedniej grupy obiektów[16]. Rozważmy następujące polecenia:

(1)

```
Button[] b;
MessageBox.Show(""+b[0].Text);
```

(2)

```
Button[] b=new Button[3];
MessageBox.Show(""+b[0].Text);
```

(3)

```
Button[] b=new Button[3] {new Button(),new Button(),new Button()};
MessageBox.Show(""+b[0].Text);
```

Polecenia z grupy (1) w ogóle się nie skompilują, ponieważ w drugiej linii próbujemy odwołać się do zmiennej `b` (referencji do tablicy), która nie została zainicjowana. Polecenia (2) skompilują się i program można uruchomić, ale podczas działania zgłoszony zostanie wyjątek `System.NullReferenceException`. Dlaczego? Tablica `b` jest, co prawda, utworzona (obiekt tablicy powstał) i zainicjowana, ale domyślne wartości referencji, a to one są tu elementami przechowywanymi w tablicy, to `null`. Nie powstał — oczywiście — żaden obiekt klasy `Button`, więc elementy tablicy `b` nie mogą przechowywać żadnych adresów, a co za tym idzie, próba odczytania własności `Text` musi skończyć się błędem. Dopiero w poleceniach (3) tablica zostaje zapełniona referencjami zawierającymi adresy do istniejących obiektów i metoda `MessageBox.Show` pokaże pustą etykietę pierwszego przycisku.

Obiekty, do których odnoszą się referencje przechowywane w tablicy, nie muszą być tworzone w tej samej linii, co deklaracja tablicy. Można do tego równie dobrze wykorzystać pętlę. Pokazuję to na listingu 3.21.

Listing 3.21. Przykład tworzenia grupy przycisków i gromadzenia ich referencji w tablicy

```
Button[] b = new Button[3];
for (int i = 0; i < b.Length; i++)
{
    b[i] = new Button();
    b[i].Text = "" + i;
    b[i].Top = i * 30;
    //b[i].Parent = this;
    b[i].Parent = tabPage3;
}
```

Elementy tablicy są indeksowane od 0. Zatem ostatni element ma indeks *rozmiar*–1. W każdej chwili rozmiar tablicy można odczytać z własności `Length`. Rozmiaru tablicy nie można zmieniać — jeżeli potrzebujemy takiej możliwości, musimy wybrać inną kolekcję, np. opisaną niżej listę `List<>`. Powyższy przykład pokazuje

---

[16] Nie należy mylić w tym przypadku wykorzystania operatora `new` do utworzenia tablicy z użyciem operatora `new` do utworzenia jej elementów.

także, że dostęp do elementów tablicy jest identyczny z dostępem w C/C++, tj. za pomocą operatora `[]` z indeksem elementu wewnątrz.

Nieco inaczej niż w C++ definiuje się natomiast tablice wielowymiarowe. Nie jest to tablica tablic, a nadal jeden obiekt klasy `System.Array`, którego elementy są odpowiednio poukładane. Oto przykład:

```
int[,] i2=new int[2,3] {{0,1,2},{3,4,5}};
```

Wielkość tej tablicy, odczytana za pomocą własności `i2.Length`, równa jest 6, ale dostęp do elementów możliwy jest jedynie przez umieszczenie dwóch indeksów wewnątrz nawiasów kwadratowych. Dla przykładu ostatni element w tablicy to `i2[1,2]`.

Do inicjacji tablicy dwuwymiarowej można wykorzystać podwójną pętlę:

```
int[,] i2=new int[2,3];
for(int i=0; i<2; i++)
   for(int j=0; j<3; j++)
      i2[i,j]=3*i+j;
```

Pętla foreach

Po zdefiniowaniu i zainicjowaniu tablic (np. poleceniami z listingu 3.21) możemy wykonywać na ich elementach dowolne operacje. Często wykorzystywane są do tego pętle `for` o budowie identycznej z budową choćby tej na listingu 3.21. W niektórych przypadkach, gdy pętla `for` przebiega po wszystkich elementach tablicy, wygodniej skorzystać z nowego typu pętli, a mianowicie `foreach` (listing 3.22).

Listing 3.22. Przykład wykorzystania pętli foreach

```
foreach(Button bi in b)
{
   bi.Text=""+0;
   MessageBox.Show(bi.Text);
}
```

Pętla taka nie nadaje się jednak do inicjowania lub zmiany wartości obiektów. Na listingu 3.22 referencja `bi` w tego typu pętli jest zadeklarowana *implicite* jako tylko do odczytu. A to oznacza, że niemożliwe jest również modyfikowanie elementów tablic o typach wartościowych. Zatem następujące polecenie nie będzie mogło być skompilowane:

```
foreach(int i in i2) i=1; //błąd
```

Nic nie stoi natomiast na przeszkodzie, aby w pętli `foreach` modyfikować obiekty typów referencyjnych. Przykładowo na listingu 3.22 obiekt, którego adres przechowujemy w referencji `bi` (w każdej iteracji inny), jest modyfikowany — jak widać, zmieniana jest jego etykieta. Tu nie ma jednak sprzeczności, ponieważ to nie obiekt przycisku jest tylko do odczytu, ale referencja `bi` do niego. Gdybyśmy natomiast próbowali wykonać pętlę widoczną na listingu 3.23, zobaczylibyśmy podczas kompilacji błąd z komunikatem informującym, że próbujemy zmienić wartość samej referencji, przypisując jej adres nowego obiektu, czego w pętli `foreach` robić nie wolno. Bez problemu możemy za to zmieniać stan obiektu, na który referencja `bi` wskazuje.

Listing 3.23. Poniższa pętla jest niepoprawna — zmienna bi nie może być modyfikowana

```
foreach(Button bi in b)
{
   bi=new Button(); //błąd
   bi.Text=""+0;
   MessageBox.Show(bi.Text);
}
```

Pętla `foreach` działa także w odniesieniu do tablic wielowymiarowych, np. po wykonaniu poleceń:

```
int[,] i2=new int[2,3] {{0,1,2},{3,4,5}};
foreach(int i in i2) MessageBox.Show(""+i);
```

zobaczymy sześć komunikatów z cyframi od 0 do 5.

Od C# 3.0 w pętli `foreach` do deklaracji elementu tablicy możemy użyć typu `var` lub w ogóle pominąć typ. Kompilator wywnioskuje typ elementu na podstawie kolekcji:

```
foreach(var i in i2) MessageBox.Show(""+i);
foreach(i in i2) MessageBox.Show(""+i);
```

Sortowanie

Typy liczbowe i znakowy implementują interfejs `IComparable` (z ang. *comparable* to porównywalny, ale nie w sensie „podobnej wielkości”, a raczej współmierny, oznaczający obiekty, w przypadku których jest sens, żeby je ze sobą porównywać). Interfejs ten zostanie dokładniej omówiony niżej. Tu tylko wspomnę, że wymaga on od implementującego go typu zdefiniowania metody `CompareTo` umożliwiającej porównywanie wartości bieżącego egzemplarza obiektu z obiektem wskazanym w argumencie. Jeżeli w tablicy znajdują się obiekty typu implementującego ten interfejs, mogą być sortowane. Bez problemu można zatem posortować np. tablicę liczb całkowitych, rzeczywistych lub łańcuchów. Służy do tego metoda statyczna `Array.Sort` (listing 3.24).

Listing 3.24. Sortowanie tabeli liczb pseudolosowych

```
int[] losy = new int[30];
Random r = new Random();
for (int indeks = 0; indeks < losy.Length; indeks++)
    losy[indeks] = r.Next(100);

string s = "Przed sortowaniem:\n";
foreach (int los in losy) s += los.ToString() + "\t";
MessageBox.Show(s);

Array.Sort(losy);

s = "Po sortowaniu:\n";
foreach (int los in losy) s += los.ToString() + "\t";
MessageBox.Show(s);
```

Próba napisania tego kodu dla zmiennych, które nie implementują tego interfejsu, skończy się zgłoszeniem błędu podczas działania programu `System.InvalidOperationException`. Przykład takiej skazanej na niepowodzenie próby widoczny jest na listingu 3.25.

Listing 3.25. Sortowanie komponentów Button nie jest możliwe

```
//uruchomienie tego kodu skończy się błędem
Button[] przyciski=new Button[100];
Random r=new Random();
for(int indeks=0;indeks<przyciski.Length;indeks++)
{
   przyciski[indeks]=new Button();
   przyciski[indeks].Text=""+r.Next(100);
}
Array.Sort(przyciski);
```

Jeżeli chcielibyśmy sortować przyciski, np. według etykiet, musielibyśmy utworzyć klasę potomną do `Button`, która implementuje interfejs `IComparable` i w której zdefiniowana byłaby metoda `CompareTo` wskazująca, w jaki sposób porównywać takie obiekty. O tym powiem niżej w tym rozdziale.

Na rzecz kolekcji, w tym tablic, można wywołać wiele metod, pozwalających m.in. przeprowadzić uśrednienie, sumowanie czy zliczanie elementów spełniających określony wyrażeniem lambda warunek. O tym opowiem jednak w części trzeciej, przy okazji omawiania technologii LINQ.

Kolekcja List

Przyjrzyjmy się teraz najczęściej wykorzystywanej kolekcji — `System.Collections.Generics.List` — czyli implementacji listy. Należy ona do grupy typów ogólnych (ang. *generic types*), nazywanych też typami parametrycznymi. Ma tę zaletę w porównaniu do zwykłej tablicy (`System.Array`), że liczba elementów może być zmieniana już po utworzeniu listy — można dodawać elementy na koniec, wstawiać do środka i usuwać dowolnie wybrany element. Jednocześnie możliwy jest dostęp do dowolnego elementu, tak samo jak w przypadku tablicy. W poniższym przykładzie (listing 3.26) utworzymy kolekcję tego typu składającą się ze stu losowo wybranych liczb całkowitych. Następnie do jej końca dodamy dziesięć liczb o wartości 0, a do środka wstawimy kolejnych pięć o wartości –1. Wreszcie na pierwszej pozycji ustawimy liczbę 1. Na koniec usuniemy wszystkie te liczby, które są większe od 20, i posortujemy resztę.

Listing 3.26. Przykład wykorzystania list, który zawiera kilka charakterystycznych dla nich błędów

```
int rozmiar = 30;
Random r = new Random();
List<int> a = new List<int>(new int[rozmiar]);
for (int i = 0; i < rozmiar; i++) a[i] = r.Next(100);
a.AddRange(new int[10]);
int[] i5 = { -1, -1, -1, -1, -1 };
a.InsertRange(rozmiar / 2, i5);
a.Insert(0, 1);

//tu kryją się błędy
for (int i = 0; i < rozmiar; i++)
{
    if (a[i] > 20)
        a.RemoveAt(i);
}

a.Sort();

string s = "Elementy listy:\n";
foreach (object ai in a) s += ai.ToString() + "\t";
MessageBox.Show(s);
```

Z góry uprzedzam, że powyższy kod zawiera błędy, które trudno w pierwszej chwili zauważyć, a które są bardzo typowe dla kolekcji ze zmienną długością.

Błędy, i to aż trzy, kryją się w pętli usuwającej liczby większe niż 20. Pierwszy polega na tym, że po wcześniejszych manipulacjach lista nie ma już wielkości określonej przez zmienną `rozmiar`. W tej chwili jest ona równa `rozmiar`+10+5. Możemy więc albo kontrolować na bieżąco wartość zmiennej `rozmiar`, zmniejszając ją w iteracji, gdy usuwany jest element z listy, albo wymusić sprawdzanie rzeczywistego rozmiaru listy, odczytując własność `List<>.Count`, w tym przypadku `a.Count` (listing 3.27).

Listing 3.27. Pierwsza poprawka

```
//lepiej, ale nadal z błędem
for(int i=0;i<a.Count;i++)
{
   if (a[i]>20)
```

```
            a.RemoveAt(i);
        }
```

Drugi, poważniejszy błąd kryje się w samej pętli. Otóż, usuwając w niektórych jej iteracjach elementy listy, zmniejszamy jej rozmiar. Zatem jeżeli na liście jest więcej niż piętnaście liczb większych od 20, a zapewne tak jest, rozmiar listy zmniejszy się i znajdzie się poniżej wartości określonej przez wielkość zapamiętaną w zmiennej `rozmiar`, a w efekcie kolejne iteracje będą musiały się skończyć zgłoszeniem wyjątku `System.ArgumentOutOfRangeException` (ang. argument poza zakresem). Nie pomoże używanie pętli `foreach`, bo ona również nie sprawdza, czy w wyniku wykonania iteracji zmienił się rozmiar kolekcji ustalony podczas inicjacji pętli. To jednak nie wszystko. W pętli nadal kryje się trzeci błąd logiczny. W razie spełnienia warunku `a[i]>20` usuwany jest element listy na pozycji `i`, co doprowadza do zmniejszenia listy o 1. Pętla przechodzi wówczas do następnej pozycji listy, czyli `i+1`. Ale po usunięciu elementu `i` obecny element `i+1` spada na pozycję `i` i w efekcie nie jest w ogóle testowany. Należałoby więc wymusić ponowne sprawdzenie elementu o tym numerze, obniżając indeks pętli (listing 3.28).

Listing 3.28. Druga poprawka

```
for(int i=0;i<a.Count;i++)
{
   if (a[i]>20)
   {
      a.RemoveAt(i);
      i--;
   }
}
```

Jeżeli ktoś nie lubi ingerencji w indeksy pętli, może wykorzystać pętlę `do..while` widoczną na listingu 3.29.

Listing 3.29. Inne podejście do drugiej poprawki

```
int j=0;
do
{
   if (a[j]>20)
      a.RemoveAt(j);
   else
      j++;
}
while(j<a.Count);
```

Kolekcja SortedList i inne słowniki

Poza `List` w przestrzeni nazw `System.Collections.Generics` dostępne są także inne typowe struktury danych. Pierwsza z nich to kolejka zaimplementowana w klasie `Queue`. Do kolejki można dodać element tylko na końcu, a zdjąć — tylko na początku (FIFO). Inna popularna struktura danych to stos (klasa `Stack`), tj. zbiór, z którego można zdjąć jedynie element ostatnio dołożony (LIFO). Pośród innych kolekcji na uwagę zasługuje `SortedList`, która w odróżnieniu od omówionych wcześniej jest „dwukolumnowa”. Każdy element listy przechowuje bowiem klucz i wartość (własności `Key` i `Value`). Pozwala to m.in. na sortowanie obu wartości według klucza, co często bywa przydatne. Podobnie zbudowane są kolekcje `Dictionary` i `SortedDictionary`. Wszystkie implementują wspólny interfejs `IDictionary` i dlatego często nazywane są słownikami.

Typowym zastosowaniem tej kolekcji są różnego typu listy z kluczem, np. słowniki, listy zmiennych itp. W poniższym przykładzie korzystam z kolekcji `SortedList` z oboma parametrami typu `String`, wykorzystując ją do przechowywania imion i nazwisk osób, które posługują się pseudonimami. Pseudonimy będą pełniły rolę kluczy (ang. *key*), a prawdziwe personalia — wartości (ang. *value*). Wygodną własnością kolekcji `SortedList`

jest to, na co wskazuje jej nazwa, że jest automatycznie sortowana zgodnie z kolejnością alfabetyczną kluczy. Listing 3.30 zawiera deklaracje kolekcji `artysci`, jej inicjacji i prezentacji w oknie komunikatu.

Listing 3.30. Przykład użycia listy parametrycznej SortedList

```
SortedList<string,string> artysci = new SortedList<string,string>();
artysci.Add("Sting","Gordon Matthew Sumner");
artysci.Add("Bolesław Prus","Aleksander Głowacki");
artysci.Add("Pola Negri","Barbara Apolonia Chałupiec");
artysci.Add("John Wayne","Marion Michael Morrison");
artysci.Add("Chico","Leonard Marx");
artysci.Add("Harpo","Arthur Marx");
artysci.Add("Groucho","Julius Marx");
artysci.Add("Bono","Paul Hewson");
artysci.Add("Ronaldo","Luiz Nazario de Lima");
artysci.Add("Madonna","Madonna Louise Veronica Ciccone");
artysci.Add("Gabriela Zapolska","Maria G. Śnieżko-Błocka");

string komunikat="Zawartość listy:\n\n";
foreach(KeyValuePair<string,string> artysta in artysci)
   komunikat+=artysta.Key+" - "+artysta.Value+"\n";
MessageBox.Show(komunikat);
```

Kolejka i stos

Przestrzeń `System.Collections.Generic` zawiera również implementacje dwóch standardowych struktur danych: kolejki (ang. *queue*) i stosu (ang. *stack*). Pierwsza jest strukturą typu FIFO (ang. *first in, first out*) — element włożony jako pierwszy może być pierwszy zdjęty. Kolejkę można sobie wyobrażać jako rurę, do której wkłada się piłki z jej jednego końca i wyjmuje z drugiego. Inaczej jest w przypadku stosu. Jest to struktura typu FILO (ang. *first in, last out*) — element włożony jako pierwszy, dostępny będzie jako ostatni. Nazwa tej struktury dobrze oddaje też to, jak można sobie ją wyobrażać — jako stos np. kartek. To, co położymy na stosie jako pierwsze, będzie przykrywane przez następne elementy i tym samym dostępne, dopiero gdy wszystkie je wcześniej zdejmiemy.

Na listingu 3.31 obrazuję tę prostą różnicę między kolejką i stosem. W obu strukturach umieszczamy liczby od 0 do 9. Następnie kolejno je zdejmujemy. W przypadku kolejki otrzymamy liczby w tej samej kolejności, w jakiej je włożyliśmy. W przypadku stosu ich kolejność będzie odwrócona.

Listing 3.31. Użycie stosu i kolejki

```
private void button33_Click(object sender, EventArgs e)
{
   int rozmiar = 10;
   Queue<int> kolejka = new Queue<int>(rozmiar);
   Stack<int> stos = new Stack<int>(rozmiar);
   for (int i = 0; i < rozmiar; ++i)
   {
      kolejka.Enqueue(i);
      stos.Push(i);
   }

   string s = "Elementy zdjęte z kolejki (" + kolejka.Count + " elementów):\n";
   for (int i = 0; i < rozmiar; ++i) s += kolejka.Dequeue().ToString() + " ";
```

```
    s += "\n\nElementy zdjęte ze stosu (" + stos.Count + " elementów):\n";
    for (int i = 0; i < rozmiar; ++i) s += stos.Pop().ToString() + " ";
    MessageBox.Show(s);
}
```

Tablice jako argumenty metod oraz metody z nieokreśloną liczbą argumentów

C# dopuszcza definiowanie metod, w których liczba argumentów nie jest z góry określona. Możliwość ta nie jest wykorzystywana zbyt często, ale jest np. kluczowa dla technologii *LINQ to XML* omówionej w rozdziale 12. Poniżej przedstawię prosty przykład takiej metody. Zacznijmy jednak od metody z listingu 3.32, której argumentem jest tablica.

Listing 3.32. Funkcja, której argumentem jest tablica liczb całkowitych

```
private int Suma(int[] lista)
{
    MessageBox.Show("Liczba argumentów: "+lista.Length);
    int suma=0;
    foreach(int liczba in lista) suma+=liczba;
    return suma;
}
```

Do takiej metody przekazywana jest tablica liczb typu `int`, tzn. jej wywołanie powinno wyglądać następująco:

```
MessageBox.Show("Suma: "+Suma(new int[] {1,2,3}));
```

Gdy jednak zmienimy sygnaturę metody, dodając przed typem tablicowym słowo kluczowe `params` (listing 3.33), możemy ją wywoływać, podając w argumentach dowolną liczbę elementów typu `int`.

```
MessageBox.Show("Suma: "+Suma(1,2,3));
```

Listing 3.33. Użycie modyfikatora params

```
private int Suma(params int[] lista)
{
    MessageBox.Show("Liczba argumentów: "+lista.Length);
    int suma=0;
    foreach(int liczba in lista) suma+=liczba;
    return suma;
}
```

Kompilator na ich podstawie sam utworzy tablicę. Nie jest to jednak obowiązkowe, możemy nadal użyć jako argumentu tablicy, a nawet kolekcji będącej wynikiem zapytania LINQ.

Co więcej, ponieważ argumentem metody jest tablica, a więc w istocie referencja do niej (tablice, jak już wiemy, są typem referencyjnym), możliwa jest zmiana elementów tychże i w efekcie zwrot wartości przez argumenty, tak jak w przypadku słów kluczowych `ref` i `out`.

Słowo kluczowe yield

Wyobraźmy sobie metodę, której zadaniem jest generowanie pewnego zbioru elementów. Przyjmijmy, że chodzi o zbiór liczb pseudolosowych — to jednak ma naprawdę drugorzędne znaczenie. Najbardziej naturalny przepis na taką metodę to: zadeklarować tablicę liczb całkowitych, utworzyć instancję klasy `Random`, w pętli zapełnić tablicę liczbami losowymi i całość zwrócić przez wartość funkcji. Listing 3.34 zawiera metodę przygotowaną zgodnie z tym przepisem.

Listing 3.34. Tradycyjna metoda zwracająca kolekcję i metoda zdarzeniowa wyświetlająca tę kolekcję

```
int[] metoda(int rozmiar)
```

```
{
    int[] wynik = new int[rozmiar];
    Random r = new Random();
    for (int i = 0; i < rozmiar; ++i)
    {
        wynik[i]=r.Next();
    }
    return wynik;
}

private void button1_Click(object sender, EventArgs e)
{
    int[] ti = metoda(10);

    string s="";
    foreach (int element in ti)
        s += element.ToString() + "\n";
    MessageBox.Show(s);
}
```

Korzystając z tego schematu, można łatwo przygotować funkcję zwracającą np. kolejne potęgi dwójki lub kolekcję przycisków. Zmieni się tylko typ kolekcji i zapełniająca ją pętla. W C# istnieje jednak stosunkowo mało znana alternatywa dla tego schematu — słowo kluczowe `yield` dodane do języka w wersji 2.0. Przyjrzyjmy się metodzie widocznej na listingu 3.35. Realizuje ona to samo zadanie, co metoda z listingu 3.34.

Listing 3.35. Prosty przykład użycia słowa kluczowego yield

```
IEnumerable<int> metoda(int rozmiar)
{
    Random r = new Random();
    for (int i = 0; i < rozmiar; ++i)
    {
        yield return r.Next();
    }
    yield break;
}

private void button1_Click(object sender, EventArgs e)
{
    int[] ti = metoda(10).ToArray();
    ...
```

Zwróćmy uwagę, że zmienił się typ wartości zwracanej przez metodę. Zamiast tablicy jest nim teraz kolekcja policzalnych elementów typu `IEnumerable<int>`. Słowo kluczowe `yield` pojawia się w metodzie w dwóch kontekstach. W pierwszym występuje przed słowem kluczowym `return`. Zwróćmy uwagę, że za słowem `return` nie ma kolekcji, która przecież jest typem wartości zwracanej przez metodę, a jedynie jej elementy. Co więcej, konstrukcja `yield return` znajduje się wewnątrz pętli. Ale właśnie na tym polega innowacja — instrukcja `yield return` nie kończy działania metody, tak jak samo `return`, a jedynie dodaje występujący po niej element do niejawnie zdefiniowanej kolekcji. Kolekcja ta budowana jest dopóty, dopóki wątek nie natrafi na instrukcję `yield break`. Jest to sygnał do przerwania zbierania elementów. W naszym prostym przykładzie instrukcja ta wykonywana jest po zakończeniu pętli.

Kolekcja zwracana przez metodę nie musi mieć z góry ustalonego rozmiaru. Jest tworzona dynamicznie, wobec tego proces jej budowania można przerwać w dowolnym momencie. Na listingach 3.36 i 3.37 prezentuję wersje powyższych metod (korzystającą ze słowa kluczowego `yield` i obywającą się bez niego), w której tworzenie kolekcji zostanie zakończone, jeżeli wylosowana zostanie liczba większa niż podany w argumencie próg.

Listing 3.36. Wersja tradycyjna...

```
IEnumerable<int> metoda (int próg)
{
    List<int> wynik=new List<int>();
    Random r = new Random();
    int los;
    do
    {
        los = r.Next(15);
        wynik.Add(los);
    }
    while (los < próg);
    return wynik.AsEnumerable<int>();
}
```

Listing 3.37. ... i korzystająca ze słowa kluczowego yield

```
IEnumerable<int> metoda(int próg)
{
    Random r = new Random();
    int los;
    do
    {
        los = r.Next(15);
        yield return los;
    }
    while (los < próg);
    yield break;
}
```

Nowa forma inicjacji obiektów i tablic

Jedną ze zmian w C# 3.0, która skraca wyrażenia pobierające dane, jest nowa forma inicjacji pól publicznych tworzonych obiektów. Pozwala ona zainicjować obiekt bez konieczności stosowania dodatkowych instrukcji. Jeżeli zatem, używając operatora `new`, utworzymy obiekt będący instancją struktury `Button`, to za nazwą typu można będzie w nawiasach klamrowych umieścić listę inicjacji jego publicznych pól zgodnie ze schematem:

```
Button b = new Button { Text = "Przycisk", Left=200, Top=30, Parent = tabPage3 };
```

Zastąpi to konstrukcję:

```
Button u = new Button();
b.Text = "Przycisk";
b.Left=200;
b.Top=30;
b.Parent = tabPage3
```

Narażając się na uproszczenie, można więc powiedzieć, że nowa forma inicjacji zastępuje definiowanie konstruktorów inicjujących stan obiektów. Konieczne są jednak publiczne pola lub własności zapewniające dostęp do nich.

Nowy sposób inicjacji dotyczy również tablic. Jedynym warunkiem jest, aby tablice były jednorodne, tj. ich elementy muszą być tego samego typu:

```
var it = new[] {5,4,3,2,1,0};
var lt = new[] {5L,4L,3L,2L,1L,0L};
var st = new[] {"Helion","Onepress","Sensus","Septem","Editio"};
var ft = new[] {1.0f,0.75f,0.5f,0.25f,0.0f};
var dt = new[] {1.0,0.75,0.5,0.25,0.0};
```

Podobna zmiana dotyczy inicjowania kolekcji — analogicznie do powyższego przykładu będzie można inicjować wartości przechowywanych przez nią elementów, np.:

```
List<string> lista = new List<string> { "Helion", "Onepress", "Sensus", "Septem",
"Editio" };
```