

Anna Kempa

Tomasz Staś

Wstęp do programowania w C#

Łatwy podręcznik dla początkujących

Aktualna wersja podręcznika na stronie <http://c-sharp.ue.katowice.pl>



Katowice, kwiecień 2014

Wersja 1.1

Anna Kempa, Tomasz Staś
Katedra Inżynierii Wiedzy
Wydział Informatyki i Komunikacji
Uniwersytet Ekonomiczny w Katowicach
e-mail: c-sharp@ue.katowice.pl

Anna Kempa	rozdziały 2, 5, 6, 7, podrozdział 1.1, dodatki, redakcja całości
Tomasz Staś	rozdziały 3, 4, podrozdział 1.2 oraz opracowanie strony internetowej

Copyright © Anna Kempa, Tomasz Staś
Wydrukowano z pliku w formacie *pdf* dostarczonym przez autorów

Wszelkie prawa zastrzeżone. Książka w wersji elektronicznej jest dostępna bezpłatnie na stronie <http://c-sharp.ue.katowice.pl>. Zabronione jest jej drukowanie czy powielanie w celach komercyjnych bez odpowiedniej zgody i umowy z autorami.

ISBN 978-83-62652-60-0

Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego
44-100 Gliwice, ul. Pszczyńska 44
tel. 506132960, www.pkjs.pl
Gliwice 2014

Spis treści

Wstęp.....	7
Dla kogo jest ta książka?	7
Jak czytać tę książkę?.....	8
1 Przywitanie ze światem	11
1.1 Świat wcale nie taki nowy	11
1.1.1 Zmienna jaką znamy z matematyki	11
1.1.2 Stałe i literały też nieobce.....	13
1.1.3 Algorytm – kolejne pojęcie jakie znasz.....	13
1.1.4 Instrukcje	13
1.1.5 Funkcje	14
1.1.6 Klasy i obiekty – o tym wiesz od dziecka	14
1.2 Uruchomienie pierwszej aplikacji w Visual Studio Express	15
1.2.1 Instalacja Visual Studio Express	15
1.2.2 Pierwsze uruchomienie.....	17
2 Arytmetyka w programowaniu.....	25
2.1 Zmienne i ich typy.....	25
2.1.1 Typ wartościowy i typ referencyjny	25
2.1.2 Deklaracja zmiennych	27
2.2 Wyrażenia algebraiczne.....	29
2.2.1 Działania bez konwersji typów	29
2.2.2 Modyfikacja typów dla literałów.....	30
2.2.3 Konwersja typów.....	33
2.2.4 Funkcje matematyczne	35
2.2.5 Arytmetyka++	36
2.3 Wyrażenia logiczne	39
2.3.1 Operatory koniunkcji.....	40
2.3.2 Operatory alternatywy	43
2.3.3 Złożone wyrażenia logiczne	44
2.4 Proste operacje na tekstach i znaki specjalne	45
2.5 Kolejność wykonywania działań.....	49
2.6 Prezentacja wyników.....	51

2.7	Zadania do samodzielnego rozwiązania.....	54
3	Sterowanie działaniem programu	57
3.1	Instrukcje warunkowe	57
3.1.1	Instrukcja warunkowa <i>if</i>	58
3.1.2	Instrukcja warunkowa <i>if..else</i>	60
3.1.3	Zagnieżdżanie instrukcji warunkowych	62
3.1.4	Operator warunkowy	64
3.1.5	Instrukcja warunkowa <i>switch..case</i>	65
3.2	Instrukcje cykliczne – pętle	69
3.2.1	Pętla <i>for</i>	69
3.2.2	Pętla <i>while</i> i <i>do..while</i>	76
3.2.3	Polecenia <i>break</i> i <i>continue</i>	82
3.3	Zadania	85
3.3.1	Zadania z rozwiązaniami	85
3.3.2	Zadania do samodzielnego rozwiązania	88
4	Operacje na typach referencyjnych – tablice i typ <i>string</i>	91
4.1	Tablice	92
4.1.1	Tablice jednowymiarowe	92
4.1.2	Tablice dwuwymiarowe	104
4.1.3	Tablice postrzępione	108
4.1.4	Wybrane metody klasy <i>Array</i>	112
4.2	Operacje na tekstach	114
4.2.1	Tekst jako tablica znaków	115
4.2.2	Wybrane metody klasy <i>String</i>	116
4.3	Zadania do samodzielnego rozwiązania	121
5	Metody	125
5.1	Metody statyczne	125
5.2	Definicja metody	126
5.3	Przekazywanie argumentów przez wartość	131
5.4	Przekazywanie argumentów przez referencję	132
5.5	Lista argumentów o zmiennej długości	134
5.6	Przekazywanie i zwracanie tablic	135
5.7	Argumenty domyślne	138
5.8	Metody przeciążone	139

5.9	Rekurencja.....	143
5.10	Zadania do samodzielnego rozwiązania.....	148
6	Wprowadzenie do tworzenia klas.....	151
6.1	Klasa a obiekt.....	151
6.2	Budowa klasy	152
6.2.1	Opis klasy	153
6.2.2	Elementy klasy (ciało klasy)	154
6.3	Użycie zdefiniowanej klasy.....	158
6.3.1	Deklaracja obiektu.....	158
6.3.2	Wywołanie metody dla obiektu.....	159
6.3.3	Zmiana wartości pól obiektu	160
6.3.4	Właściwości.....	161
6.3.5	Składniki statyczne.....	164
6.3.6	Tablice obiektów	166
6.4	Typ referencyjny w kolejnej odsłonie	168
6.5	Struktury a klasy.....	173
6.6	Cechy programowania obiektowego	176
6.7	Zadania do samodzielnego rozwiązania.....	180
7	Poprawianie błędów w programie	183
7.1	Poprawianie błędów zgłaszanych na etapie kompilacji	183
7.2	Poprawianie błędów występujących po uruchomieniu programu	189
	Zakończenie, czyli początek.....	197
	Dodatki – podręczny bryk	198
	Literatura	207

„Naucz mnie sztuki stawiania małych kroków”

Antoine de Saint-Exupéry

Wstęp

Wielu młodych ludzi, rozważając studiowanie kierunku informatycznego pyta, czy programowanie jest trudne. Niektórzy utożsamiają słowo „trudne” z „pracochłonne” lub „uciążliwe”. W istocie nauka programowania wymaga pewnego wysiłku. Ale wysiłku wymaga większość cennych umiejętności, jakie podczas życia nabywamy. Począwszy od tych najbardziej podstawowych, zdobywanych w pierwszych latach życia, jak np. nauka chodzenia. Gdyby dzieci były nastawione jedynie na szybką i bezbolesną naukę – zapewne nigdy nie nauczyłyby się chodzić. Na szczęście w ich psychice nie ma takich ograniczeń. Pomimo pierwszych niepowodzeń i wielu bolesnych upadków instynktownie i wytrwale dążą do celu, jakim jest możliwie szybkie pokonywanie odległości i dalsze badanie świata.

Kiedy nauka męczy, a kiedy sprawia przyjemność? Na ogół męczy nas nauka tego, co tak naprawdę nas nie interesuje. W szkole nie wszystkie przedmioty interesują nas w równym stopniu. Nauka męczy nas także wówczas, gdy nie przynosi szybkich, bezpośrednio widocznych efektów. Wiele osób, które w swoich dziedzinach – nauki, sztuki, sportu osiągnęły sukces twierdzi, że „najtrudniejsze były początki”. Później, pomimo faktu, iż konieczny wysiłek był na ogół większy – wzrastał wraz z poziomem wyzwań – było łatwiej. Każdy z nas w końcu dostrzega, że „trudne” wcale nie znaczy „bardziej pracochłonne”. Trudne to coś, czego nie znamy. Obce. Dotąd dla nas nieprzeniknione jak gruby mur. Ta bariera istnieje jednak jedynie w naszym umyśle. Dlatego też później, po osiągnięciu pewnych umiejętności, jest nam łatwiej, mimo że zazwyczaj mamy jeszcze więcej pracy. Wówczas już widzimy pierwsze efekty włożonego wysiłku i jednocześnie zaczynamy coraz bardziej lubić to, czego się uczymy.

Dla kogo jest ta książka?

Nie wiemy dokładnie, jak wiele rzeczy w życiu zarzucamy zbyt pochopnie, uznając je za trudne dla nas, ale niewątpliwie trochę ich jest. Celem tego podręcznika jest pomoc w zbudowaniu pomostu pomiędzy tym punktem w czasie, w którym programowanie wydaje się jedynie trudne a momentem, gdy zaczyna sprawiać przyjemność. W chwili, gdy polubimy programowanie, do dalszej nauki potrzebny nam będzie inny rodzaj podręczników. Najczęściej będzie to po prostu dokumentacja techniczna do danego programistycznego narzędzia, opis projektu, algorytmu. Tę książkę dedykujemy wszystkim początkującym – osobom rozpoczynającym naukę programowania:

- Uczniom szkół średnich, którzy rozważają studiowanie informatyki i zadają sobie pytania: Czy sobie poradzę? Czy to jest dla mnie?

- Studentom, którzy już wybrali kierunek informatyczny, ale sami jeszcze jak dotychczas nie programowali. W szkołach wyższych przedmioty poświęcone nauce programowania nierzadko rozpoczynają się od podstaw, jednak zajęcia realizowane są w tempie, które dla osób początkujących może się wydać zbyt szybkie. Studenci niemający wstępnego przygotowania w tym zakresie mogą mieć więcej trudności,
- Także – wszystkim tym, którzy chcą rozpocząć przygodę z programowaniem, niezależnie od tego, czy i gdzie się uczą bądź planują się uczyć.

Długoletnie doświadczenie w nauczaniu programowania pozwoliło autorom dobrze rozeznaczyć potrzeby początkujących. Głównym celem, jaki przyświeca autorom jest pomoc w początkowym etapie nauki, w pokonaniu barier i obaw przed nowym wyzwaniem. Z tego powodu wybór konkretnego języka programowania był na nieco dalszym planie. Oczywiście musieliśmy wybrać jakiś język i jak tytuł podręcznika wskazuje jest to język C#. Nie jest jednak celem autorów zaprezentowanie czytelnikom wszystkich możliwości tego języka. Pominięcie niektórych zagadnień dotyczących języka jak i środowiska uruchomieniowego pozwoli skupić się na wejściu w świat programowania komputera częściowo w oderwaniu od niuansów samego narzędzia. Takie podejście jest dopuszczalne na samym początku nauki, później jednak, gdy Czytelnik będzie zbliżał się do profesjonalnego poziomu – już nie. Poznane tu podstawowe konstrukcje językowe pozwolą Czytelnikowi na dalsze rozwijanie umiejętności – zarówno w ramach omawianego języka (C#) jak i kilku innych, takich jak C++, Java, PHP, a także w ramach języków składniowo mniej podobnych – takich jak Pascal czy Delphi.

Jak czytać tę książkę?

Podręcznik jest dostępny zarówno w postaci tradycyjnej, papierowej książki, jak i w formie elektronicznej. Wersję elektroniczną można pobrać w postaci pliku pdf bezpłatnie na stronie <http://c-sharp.ue.katowice.pl>. Kody wszystkich użytych programów są dostępne w elektronicznej wersji książki.

Jak wspomniano, w książce pominięto wiele aspektów języka C# (mniej istotnych z punktu widzenia wstępnego etapu nauki), pewne jednak rozszerzenia będą się pojawiać jako dodatkowe odnośniki. Podręcznik ma także liczne linki wewnętrzne (do innych części podręcznika), co może pomóc Czytelnikowi w trakcie przeglądania wersji elektronicznej. Nie ma jednak potrzeby korzystania z każdego napotkanego linku, mogłoby to wręcz utrudniać naukę. Podczas pierwszego czytania zalecamy, aby linki dotyczące tej części podręcznika, której Czytelnik jeszcze nie przeczytał, traktować jedynie jako zapowiedzi rozwinięcia danego tematu i nie czytać treści, do których odsyłają. Wyjątek stanowi jedynie sekcja *Dodatki*, do której warto zaglądać. Natomiast nie ma przeciwwskazań do korzystania z linków, które wskazują na treści już przeczytane, zwłaszcza jeśli Czytelnik odczuwa, że przyda się mu przypomnienie danej kwestii. Do tego typu podręczników, po pierwszym przeczytaniu,

zagląda się wielokrotnie. Wówczas Czytelnik już zna lepiej zawartość całej książki, a ponadto wie, które zagadnienia rozumie, a które wymagają dalszego zgłębienia, przez co łatwiej jest mu samodzielnie podejmować decyzje o korzystaniu z proponowanych odniesień (linków).

Zrywamy z pewną konwencją podręcznikową. W wielu podręcznikach, także dla początkujących, w pierwszych rozdziałach traktujących o podstawowych elementach językach, takich jak zmienne, operatory, umieszcza się jednocześnie wszystkie informacje związane z danym tematem, np. wszystkie typy wbudowane, wszystkie operatory, słowa kluczowe. Wymienione elementy języka to wprawdzie rodzaj „alfabetu”, ale nie wszystkie „litery” są od razu potrzebne. Jako autorzy podręcznika wprowadzającego łagodnie Czytelnika w temat, dbaliśmy przede wszystkim o stopniowe ukazywanie kolejnych elementów języka. Kompendium języka i jego „alfabet” znajduje się na końcu podręcznika, w sekcji *Dodatki*, w postaci zbliżonej do „tablic informatycznych”. W poszczególnych rozdziałach znajdują się liczne odnośniki do *Dodatków* (oraz innych części podręcznika). W samych *Dodatkach* są odnośniki „na zewnątrz”, do dokumentacji MSDN¹ (dostępnej w Internecie). Pełna wiedza o języku jest dostępna w oficjalnej dokumentacji, w podręcznikach dla zaawansowanych, na licznych forach programistycznych i w tych zasobach programista szuka sobie potrzebnych informacji sam. W tym podręczniku pomagamy mu rozpocząć tę drogę samodzielnego poszukiwań.

Wyjaśnianie poszczególnych konstrukcji języka odbywa się w tym podręczniku, podobnie jak w wielu innych, przy pomocy dokładnego omówienia licznych przykładów. Pod koniec większości rozdziałów znajdują się zadania do samodzielnego rozwiązania, przy niektórych podano wskazówki. Dobór odpowiednich przykładów jest kluczowy. W naszej książce przykłady są proste i na wstępnym etapie mogą wydawać się mało barwne. Podobnie jak dźwięki wydawane przez osoby, które uczą się grać na jakimś instrumencie muzycznym. Jeszcze nie słyhać melodii... Mamy jednak nadzieję, że Czytelnik przetrwa dzielnie z nami ten pracowity rozruch i z każdym nowym rozdziałem będzie dostrzegał coraz więcej barw, a tworzone przez niego programy będą coraz bardziej złożone i ciekawe.

W pierwszym rozdziale wprowadzamy w świat programowania jeszcze bez programowania, wstępnie omawiając pojęcia używane w programowaniu przy pomocy wiedzy znanej z wcześniejszej edukacji. W rozdziale drugim będziemy poznawać podstawowe zagadnienia, które można by porównać do arytmetyki w matematyce, m.in. wyrażenia arytmetyczne, logiczne oraz operacje na tekstach. Rozdział trzeci omawia instrukcje sterujące, dzięki którym programista może wpływać na kolejność wykonywania poleceń w programie. Poznanie instrukcji sterujących pozwala wykorzystać jedną z kolekcji danych, jaką jest tablica i zbiorowo przetwarzać dane – temu tematowi poświęcony został rozdział czwarty. Do tego miejsca kolejność czytania rozdziałów wydaje się być obowiązkowa dla wszystkich początkujących.

¹ MSDN - Microsoft Developer Network. Biblioteka MSDN to scentralizowana baza oficjalnych dokumentów dla programistów i deweloperów, zawierająca dokumentację techniczną <http://msdn.microsoft.com/library>.

Począwszy od rozdziału piątego Czytelnik może podjąć decyzję, czy chce skorzystać z jeszcze jednego „schodka”, jaki proponujemy w postaci rozdziału piątego (o metodach), czy też woli od razu przejść do rozdziału szóstego – omawiającego kompleksowo elementy programowania obiektowego. Obie drogi mają swoje zalety i wady. Rozdział piąty mieści się jeszcze w ramach koncepcji nauki od szczegółu do ogółu, jaka jest konieczna w poprzednich wstępnych rozdziałach. Na przykładzie metod statycznych można przećwiczyć niemal wszystkie aspekty związane z używaniem metod (statycznych i niestatycznych), jednocześnie nie borykając się z wieloma innymi elementami, które pojawiają się wraz z budową własnych klas. Wybór rozdziału szóstego pozwala z kolei przyspieszyć wejście w paradygmat obiektowy i ujrzeć poznane wcześniej zagadnienia z punktu widzenia całości programu w tym paradygmacie. Rozdział ostatni poświęcony jest tematyce poprawiania błędów. Opisuje m.in. *debugger* – narzędzie służące do analizy programu w celu odnalezienia i identyfikacji zawartych w nim błędów. Umieszczamy ten rozdział na końcu, ponieważ nie wszystkie z opisywanych możliwości tego narzędzia są potrzebne przy analizie programów z pierwszych rozdziałów podręcznika. Zachęcamy jednak, aby po napotkaniu problemów podczas pisania programu przejść do rozdziału siódmego i przeanalizować zaprezentowane przykłady śledzenia programu.

Na zakończenie jeszcze kilka słów o dalszym „życiu” tego podręcznika. Podręcznik (w wersji elektronicznej) jest dostępny bezpłatnie, jako jego autorzy korzystający wcześniej wielokrotnie z różnych darmowych zasobów programistycznych (programów, książek, porad na forach) spłacamy niejako „dług” i zawieramy część swojego doświadczenia (nie tylko w zakresie programowania, ale także tego, jak uczyć) w postaci tej książki. Bardzo liczymy na żywy oddźwięk ze strony Czytelników i pomoc w jej doskonaleniu – przede wszystkim poprzez wskazanie wszelkich zauważonych błędów. Prosimy także o uwagi dotyczące aspektów merytorycznego i dydaktycznego, np. gdy opis któregoś z zagadnień wydaje się mało czytelny albo gdy Czytelnik dostrzega w danym miejscu konieczność dodania przykładów, itp. W wersji elektronicznej, na każdej stronie książki jest u dołu link obok tekstu *Zgłoś uwagę do tej strony*, który otwiera stronę do wpisywania uwag. Można też wpisać w przeglądarce internetowej adres <http://c-sharp.ue.katowice.pl>, a następnie w opcji *Formularz błędów* podać numer strony, do której uwaga ma się odnosić. Jeśli uwaga ma dotyczyć całej książki lub większego jej fragmentu, można wpisać jako numer strony cyfrę 0.

Obecna wersja podręcznika została uzupełniona i poprawiona na podstawie uwag zgłoszonych do wersji poprzedniej. Bardzo dziękujemy za przekazane uwagi studentom oraz naszym współpracownikom z Uczelni. Szczególne podziękowania pragniemy złożyć dr Bognie Zacny oraz dr. Krzysztofowi Michalikowi, których uwagi w największy sposób przyczyniły się do ulepszenia zawartości niniejszego podręcznika.

1 Przywitanie ze światem

Naukę programowania rozpoczniemy od napisania programu, który wyświetli na ekranie tekst „Witaj świecie!”. Nasze przywitanie będzie składało się z dwóch części. Pierwsza wprowadza w świat programowania bez samego programowania. To pomost z rejonów, jakie są Czytelnikowi znane z wcześniejszej edukacji oraz szeroko rozumianej intuicji do miejsca, jakie czeka go na kartach tego podręcznika. Kolejna część rozdziału to już faktyczny krok naprzód, którym jest uruchomienie pierwszego programu.

Celem rozdziału jest przygotowanie warsztatu do dalszej pracy. Ale nie wszystko co dotyczy środowiska oraz procesu generowania programu jest na obecnym etapie nam potrzebne. Witając się ze światem nowo poznanej dziedziny wcale nie musimy wiedzieć, co jak działa już u progu tego świata. Należy pozwolić rzeczom się wydarzać, oswajać się z klimatem, a nie czekać gorączkowo na „oświecenie”. Nie pozwólmy, aby takie naturalne zjawisko podczas nauki – jak **niezrozumienie czegoś** – budziło nasz dyskomfort czy bezradność. Będziemy mieli dość okazji w całym podręczniku by przekonać się, że to tylko stan przejściowy.

1.1 Świat wcale nie taki nowy

Bez względu na to jakiego języka programowania się uczymy, napotykamy te same lub podobne pojęcia, takie jak zmienna, stała, algorytm, instrukcja, obiekt, klasa. Zanim przywitamy się z właściwym światem programowania i rozpoczniemy pisanie programów, wyjaśnimy te pojęcia w sposób przystępny, bazując na przykładach i analogiach znanych Czytelnikowi ze szkoły średniej, a nawet z codziennego życia.

1.1.1 Zmienna jaką znamy z matematyki

W matematyce szkolnej przez zmienną rozumiemy wielkość, która może przyjmować wiele różnych wartości liczbowych. Dokładniej, może ona przyjmować wszystkie wartości należące do pewnego zbioru, który nazywamy zakresem zmienności zmiennej. Tak więc mówimy o zmiennej rzeczywistej, wymiernej, całkowitej, dodatniej, ujemnej, należącej do określonego przedziału liczbowego, itd. Zmienne oznaczamy symbolicznie, najczęściej literami alfabetu łacińskiego, czasami z dodaniem wskaźników liczbowych: a , c , x , ale także $a1$, $z1$, czy $b12$, $z14$. W programowaniu zmienne definiujemy w pełnej analogii do zmiennych matematycznych. Jest jednak pomiędzy tymi definicjami zmiennych, w matematyce i programowaniu, zasadnicza różnica. Zmienne matematyczne są wielkościami całkowicie abstrakcyjnymi, istniejącymi jedynie w przestrzeniach matematycznych (w naszej wyobraźni) i jako takie nie podlegają żadnym ograniczeniom. Natomiast w programowaniu każdej

zmiennej odpowiadać będzie jej możliwy zapis w fizycznej pamięci komputera – co na zbiór możliwych wartości tych zmiennych nakłada liczne ograniczenia.

Z tych powodów zmienne w informatyce/programowaniu najlepiej jest interpretować jako *miejsce w pamięci* komputera. Czytelnik zetknie się z tym zagadnieniem jeszcze wielokrotnie w czasie dalszej nauki. Inaczej też niż w matematyce interpretujemy w językach programowania wiele zapisów, wyglądających formalnie, z zewnątrz, jak zwykłe reguły matematyczne. Tak jest np. z interpretacją znaku „=”, znanego powszechnie w szkolnej matematyce jako znak równości. W językach programowania znak równości pojawia się jednak często w innym kontekście niż w matematyce. Jednym z najbardziej charakterystycznych i pouczających przykładów obrazujących właściwe rozumienie powyższych treści jest tzw. instrukcja przypisania², znana także jako instrukcja podstawienia. Formalnie, instrukcja przypisania może wyglądać jak zwykłe równanie matematyczne, np. tak: $x = x + 1$. Jednak ten zapis nie oznacza równania, nie mówi, iż „lewa strona równa się prawej stronie” jak interpretowałby ten zapis matematyk. Cała instrukcja podstawienia (przypisania) mówi co następuje: Wyznacz aktualną wartość wyrażenia po prawej stronie znaku „=” i tę obliczoną wartość wstaw do zmiennej umieszczonej po lewej stronie znaku „=”. Realizując tę powyższą instrukcję $x = x + 1$ komputer wykona po kolei następujące czynności: Odszuka w pamięci aktualną wartość zmiennej „x”. Wartość tę powiększy o jeden ($x + 1$). Wynik dodawania zapisze do zmiennej z lewej strony znaku „=”, czyli wpisze ją do tej samej komórki pamięci związanej ze zmienną „x” – jako jej nową wartość (niszcząc, czyli nadpisując wartość poprzednią). Przykład ten wyraźnie uwidacznia, iż w programowaniu znane nam dobrze z matematyki pojęcie zmiennej najlepiej interpretować jako określone miejsce w pamięci komputera, do którego to będą zapisywane wszystkie kolejne wartości danej zmiennej w trakcie realizacji programu.

W programowaniu o zmiennej powiemy, że ma kilka atrybutów – m.in. identyfikator (nazwę), typ oraz może posiadać wartość. Typ zmiennej określa „charakter” danej (np. czy liczbowa, czy tekstowa, logiczna), w tym jej rozmiar. W uproszczeniu możemy powiedzieć, że typ wyznacza zakres możliwych wartości. W programowaniu zakres możliwych wartości ma też znaczenie „techniczne” – mianowicie zmienne muszą być umieszczone w pamięci komputera i należy dla nich rezerwować odpowiednie miejsce.

Zatrzymujemy się na zmiennych z ważnego powodu. Wprawdzie rezygnujemy w tym podręczniku z osobnego wykładu na temat, czym jest programowanie, kod maszynowy i rodzaje translacji, ale umieszczamy niezbędne elementy teoretyczne przy okazji omawiania różnych aspektów. Program realizuje zaplanowany sposób postępowania z danymi (algorytm), który w skończonej liczbie kroków prowadzi do otrzymania oczekiwanego wyniku. Inaczej mówiąc program jest sekwencją kroków, które zmieniają stan maszyny.

² Operator przypisania używany w takiej instrukcji w językach C, C#, C++ to znak równości (=), w Pascalu to dwukropek ze znakiem równości :=

Zmienne biorą w tym ważny udział. Niemala część tego podręcznika dotyczy sposobu postępowania ze zmiennymi. Wyjaśniamy jak organizować „zmiennosc” ich wartości, tak aby program wyprowadził oczekiwany przez nas wynik.

1.1.2 Stałe i literały też nieobce

Stałe znane z matematyki to wielkości, które nie mogą się zmieniać. W programowaniu jest podobnie. Wśród wielkości stałych można wyróżnić dwa główne rodzaje – stałe mające swój identyfikator (nazwę) oraz tzw. literały. W matematyce także występują oba rodzaje stałych wielkości, ale niekoniecznie tak są nazywane. W przykładowym wzorze na obwód koła $O = 2\pi r$, cyfra „2” byłaby literałem, liczba π stałą, a promień r – zmienną. Matematyka ma swoją tradycję nazewnictwa używanych wielkości, co pomaga czytać wzór bez konieczności szczegółowego opisu wszystkich symboli. Wiele nazw matematycznych stałych to litery greckiego alfabetu i w ten sposób łatwo się wyróżniają w zapisie. W programowaniu też jest trochę tradycji odnośnie nazewnictwa, ale nie zwalnia to programisty z jawnej deklaracji występujących symboli. I tak, stałe oznacza się specjalnym słowem kluczowym „const”. W rozdziale drugim pokażemy wykorzystanie stałych i literałów na przykładach.

1.1.3 Algorytm – kolejne pojęcie jakie znasz

Algorytm to sposób postępowania, który prowadzi do rozwiązania określonego problemu. Słowo „algorytm” choć najczęściej spotykane w kontekście matematyki i informatyki, równie dobrze może być stosowane w codziennym życiu. Często przytaczanym przykładem algorytmu jest przepis kulinarny, który zawiera opis „wejść” (potrzebnych składników), wykaz czynności potrzebnych do wykonania, a jego końcowym efektem (rozwiązaniem) jest dana potrawa. Algorytm jest opisem sposobu działania, a zatem – pewną abstrakcją. Aby uzyskać wynik potrzebna jest jego realizacja (implementacja). W programowaniu oznacza to napisanie programu, który będzie wykonywał wszystkie czynności przewidziane przez algorytm. Czynności te opisane są przez instrukcje.

1.1.4 Instrukcje

Każdy program jest budowany jako zbiór instrukcji. Prezentowany wcześniej zapis $x = x + 1$; jest instrukcją przypisania. Po jej wykonaniu zmienna x zwiększy swoją wartość o jeden. Istnieją także inne rodzaje instrukcji (wywołania, powrotu, instrukcje sterujące), ale te poznamy w kolejnym rozdziale. Instrukcja to polecenie wykonania określonego działania. W wielu podręcznikach, także i tu, termin „instrukcja” bywa używany zamiennie z terminem „polecenie”. W językach takich jak C#, który reprezentuje tzw. grupę języków imperatywnych, instrukcja jest podstawową jednostką kodowania algorytmu na język maszyny³. Instrukcja w języku C# musi być zakończona średnikiem. Ale instrukcja to nie to

³ W programowaniu można wyodrębnić dwa główne podejścia – paradygmat imperatywny i deklaratywny. W paradygmacie imperatywnym w kodzie programu określa się dokładnie „jak” program ma dojść do rozwiązania (poprzez sekwencje instrukcji). W paradygmacie deklaratywnym należy odpowiednio sformułować

samo co linia kodu. W trakcie analizy przykładów będziemy zwracać uwagę Czytelnika na to, kiedy średnik należy stawiać, a kiedy nie.

1.1.5 Funkcje

Zadania nierzadko mogą być dość złożone i „upchanie całości do jednego worka” nie będzie czytelne. Przykładowo w przepisie na pizzę można wyodrębnić przepis na ciasto, które stanowi spód pizzy oraz przepis na zawartość wierzchniej warstwy. Każdy z tych dwóch elementów ma swoje osobne wejście i wyjście. A cały algorytm musi posiadać polecenie połączenia obu części w odpowiednim momencie. W programowaniu także można łączyć instrukcje w grupy, które prowadzą do rozwiązania danego elementu składowego jakiejś większej całości. Takie składowe fragmenty programu nazywane są funkcjami, procedurami lub metodami. Paradygmat programowania, który akcentuje podział kodu programu na procedury nazywany jest programowaniem proceduralnym. Język C# jest językiem obiektowym. Paradygmat obiektowy nie unieważnia jednak podziału programu na funkcje, akcentuje obiekty, które łączą dane i funkcje.

W matematyce szkolnej funkcja dla każdego elementu ze zbioru argumentów (dziedziny) wyznacza jeden element ze zbioru wartości funkcji (przeciwdziedziny). Funkcja (procedura) w programowaniu jest czymś podobnym, ale nie podlega w pełni rygorom definicji obowiązujących w matematyce⁴. Można np. wykonać funkcję, która nic nie wyznacza, albo wyznacza obiekt złożony (posiadający faktycznie więcej niż jedną wartość). W pierwszym przykładzie jaki przedstawimy w tym podręczniku będą występowały dwie funkcje (metody). Główna *Main()* oraz metoda *WriteLine()*, której zadaniem jest wypisanie tekstu na ekranie. Wywołania funkcji można zagnieżdżać, można tworzyć wiele wariantów funkcji o tej samej nazwie, wszystkie te elementy zostaną objaśnione w dalszej części podręcznika. Na obecnym etapie wystarczy, jeżeli funkcja (metoda) będzie przez Czytelnika rozumiana jako wydzielona część programu, która ma konkretną pracę do wykonania.

1.1.6 Klasy i obiekty – o tym wiesz od dziecka

W programie występują jednak nie tylko funkcje i zawarte w nich instrukcje opisujące czynności prowadzące do rozwiązania. Występują także dane. W programowaniu proceduralnym dane i opis zachowania tych danych (jak się mają zmieniać) nie są ze sobą bezpośrednio powiązane. Programista sam musi pilnować tych powiązań i dbać o to, aby np. funkcja obliczająca objętość bryły nie była wywoływana dla figur płaskich. Jawne połączenie

problem („co” ma być zrobione). Znane języki obiektowe (jak C++, C# , Java) zachowały imperatywny charakter (który został rozszerzony o obsługę obiektów). Niemniej współcześnie niektóre z języków obiektowych (w tym C#) zostały wzbogacone o elementy deklaratywne. W tym podręczniku jednak nie omawiamy deklaratywnych możliwości języka C#. Zainteresowanych czytelników odsyłamy do strony <http://msdn.microsoft.com/en-us/library/bb669144.aspx>.

⁴ Są języki, które w znacznie bliższy sposób odpowiadają idei matematycznej funkcji – to tzw. języki funkcyjne (np. Lisp, Haskell, Ocaml i bazujący na nim F#).

danych i ich zachowania bardzo ułatwia programowanie. I nie jest to koncepcja nam obca. Wszystko co poznajemy, począwszy od dzieciństwa, przyswajamy w określonych kontekstach. Wokół siebie widzimy mnóstwo obiektów, które grupujemy w klasy niekoniecznie zdając sobie z tego sprawę. Małe dziecko widząc kilka różnych psów (pierwszych w życiu) wytwarza w swoim umyśle odpowiednią „klasę” dla tych czworonogów i widząc kolejnego, wcześniej nieznanego, powie „to pies”.

Poznawany przez nas świat to zbiór obiektów, które są ze sobą powiązane i mogą ze sobą oddziaływać. A obiekty te mogą podlegać określonym dla nich zachowaniom. Języki programowania, które realizują paradygmat obiektowy opisują model danego fragmentu rzeczywistości uwzględniając wymienione powiązania: danych obiektu z jego zachowaniem oraz powiązania obiektów między sobą.

Mówiliśmy, że dla zmiennej przypisujemy typ, czyli zakres możliwych wartości (i rozmiar). W programowaniu obiektowym możemy definiować własny typ (klasę), który składa się zarówno ze zmiennych (danych) jak i funkcji (metod), które będą na tych danych operować. Pozwala to uwzględnić kolejny, przybliżający rzeczywistość wymiar. W rzeczywistości jaka nas otacza występują obiekty wraz ze swoimi cechami i funkcjami. Istnieje np. samochód i wiemy, że charakteryzują ten obiekt takie atrybuty jak wielkość, pojemność silnika, marka, cena i wiele innych. Wiemy, że samochód może jechać, być naprawiany, być myty. Wiemy jednocześnie, że nie służy do prania ubrań czy latania.

Język C# jest językiem obiektowym. Począwszy od pierwszego uruchomionego programu nie da się o tym zapomnieć. Wprawdzie dopiero pod koniec podręcznika będziemy tworzyć własne klasy, ale od samego początku będziemy wykorzystywać istniejące klasy.

Nasza nauka będzie trochę przypominała program nauki biologii, zaczynający się od organizmów jednokomórkowych, a później omawiający organizmy bardziej złożone. Te bardziej złożone składają się jednak z podobnych komórek. Dlatego etap poznania „pojedynczej komórki” – prostych zmiennych, prostych algorytmów, instrukcji sterujących – jest konieczny.

1.2 Uruchomienie pierwszej aplikacji w Visual Studio Express

W poprzednim podrozdziale mówiliśmy o programowaniu bez programowania. Od tego miejsca i aż do końca podręcznika uczyć będziemy się głównie na przykładach. Najpierw jednak trzeba zainstalować narzędzie. Zaraz później będzie można napisać pierwszy program.

1.2.1 Instalacja Visual Studio Express

Tworzenie programów z wykorzystaniem języka C# umożliwia między innymi pakiet Microsoft Visual Studio. Obecnie aktualna wersja tego pakietu nosi numer 2013. Należy

oczekiwać, że firma Microsoft w dalszym ciągu będzie rozwijać to środowisko programistyczne i oferować kolejne jego wersje.

Pakiet Visual Studio jest przykładem zintegrowanego środowiska programistycznego (ang. Integrated Development Environment, IDE). W jego skład wchodzi między innymi edytor tekstu, **kompilator** (program umożliwiający automatyczne tłumaczenie kodu⁵), debugger (program umożliwiający analizowanie programu w celu znalezienia i identyfikacji błędów). Visual Studio występuje w wielu odmianach, różniących się dostępnymi funkcjami oraz (co oczywiste) ceną. Na potrzeby niniejszej książki wystarczająca jest darmowa wersja Visual Studio Express.

Wersję instalacyjną pakietu programistycznego najlepiej pobrać bezpośrednio ze strony firmy Microsoft (<http://www.microsoft.com/visualstudio/>). Wśród dostępnych odmian oprogramowania Visual Studio Express 2013 należy wybrać opisaną jako „Windows Desktop”).



⁵ W przypadku platformy .NET w wyniku kompilacji kod napisany przez programistę tłumaczony jest na tzw. język pośredni, a z pośredniego na kod maszynowy (wykonywany przez komputer). Po przeczytaniu niniejszego podręcznika zalecamy zapoznanie się z materiałami omawiającymi platformę .NET).



Spośród dostępnych opcji instalacji można dokonać wyboru pomiędzy zainstalowaniem i pobraniem obrazu ISO. Instalacja wiąże się z koniecznością ściągnięcia z Internetu niedużego pliku (ok. 1 MB), reszta niezbędnych zasobów będzie automatycznie pobierana ze stron Microsoftu w trakcie procesu instalacji oprogramowania. Ten wybór zaleca się użytkownikom mającym stały i w miarę szybki dostęp do Internetu.

Wybór opcji „Obraz ISO DVD5” wiąże się z pobraniem pliku o rozmiarze prawie 800 MB. Pobrany plik jest obrazem płyty DVD, który należy nagrać na płytę (jako plik obrazu). Przygotowana w ten sposób płyta umożliwi instalację pakietu Visual Studio bez konieczności dostępu do Internetu.

Proces instalacji Visual Studio Express 2013 for Windows Desktop przebiega w sposób standardowy. Na wstępie należy wskazać ścieżkę do folderu na dysku twardym, w którym zainstalowane zostanie oprogramowanie. Zgodnie z wymaganiami sprzętowymi powinno być minimum 5 GB wolnego miejsca na dysku twardym. Konieczne jest również zaakceptowanie warunków licencyjnych („I agree to the Licence terms and conditions”), opcjonalnie można wyrazić zgodę na przystąpienie do programu poprawy funkcjonowania pakietu („Join the Customer Experience Improvement Program”).

Po kilku lub kilkunastu minutach (zależnie od wybranej opcji instalacji, szybkości Internetu oraz mocy obliczeniowej komputera) pakiet **Visual Studio Express 2013 for Windows Desktop** zostanie zainstalowany.

1.2.2 Pierwsze uruchomienie

Po zainstalowaniu pakietu programistycznego na liście programów (Start->Wszystkie programy) pojawi się katalog Visual Studio 2013. Aby uruchomić aplikację należy wybrać pozycję opisaną jako „VS Express 2013 for Desktop”.



W efekcie zostanie uruchomione oprogramowanie Visual Studio. W pierwszym oknie, które się ukaże możliwe jest zapoznanie się z nowościami oraz uruchomienie samouczka. Aby korzystać z przykładów zamieszczonych w niniejszej książce należy rozpocząć nowy projekt („New Project...”). Projektem w Visual Studio nazywana jest (w dużym uproszczeniu) po prostu tworzona przez programistę aplikacja.



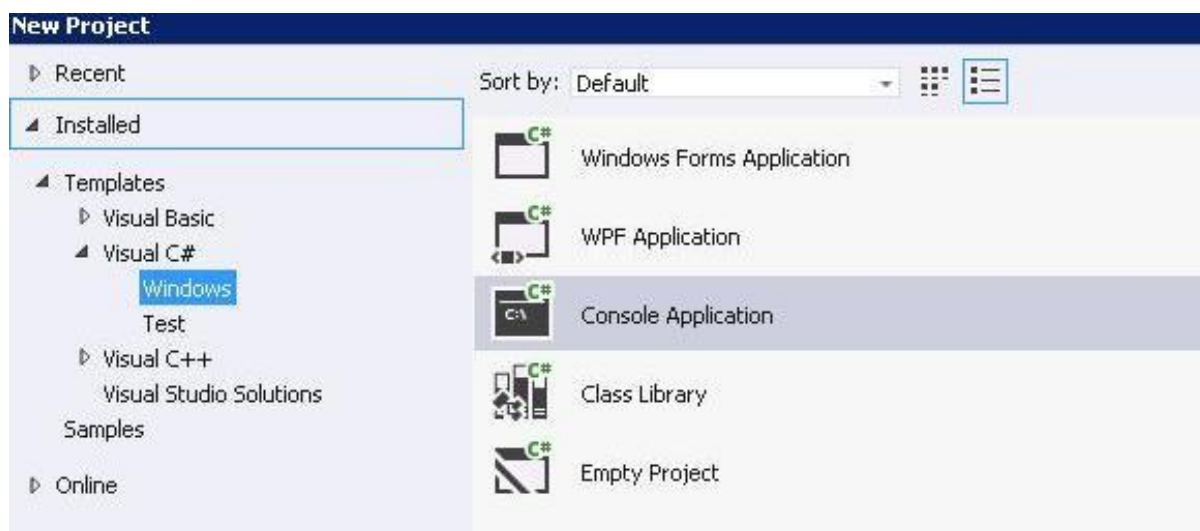
Start

New Project...

Open Project...

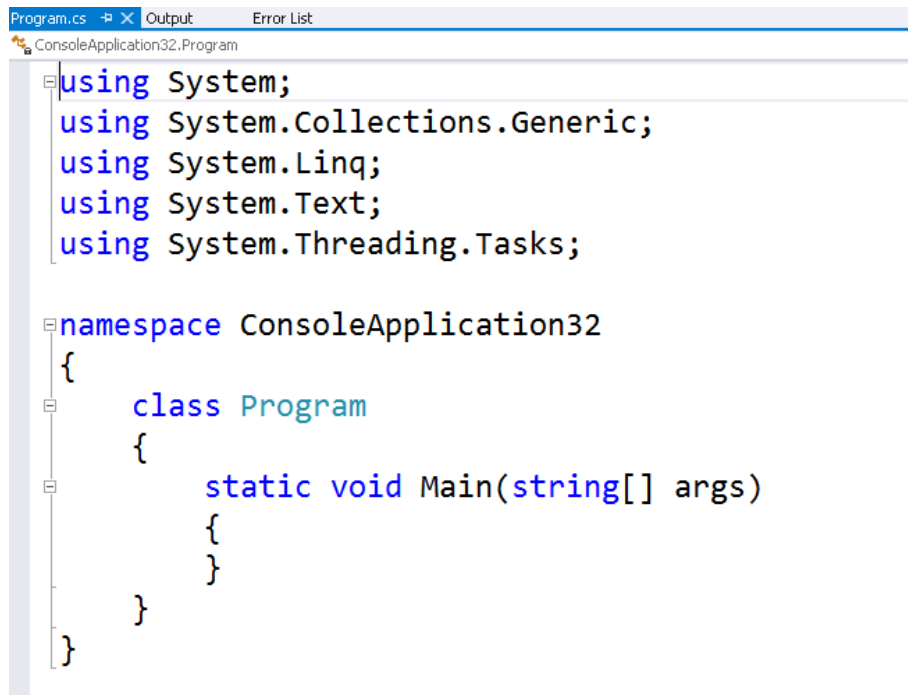
Open from Source Control...

Środowisko programistyczne Visual Studio Express umożliwia tworzenie aplikacji w kilku językach programowania (Visual Basic, C++, C#). Niniejsza książka przedstawia podstawy programowania w języku C#, dlatego też należy wybrać ten język z dostępnej listy. Wszystkie przykłady zawarte w tej książce odnoszą się do aplikacji konsolowej, czyli takiej, której efekty działania można obserwować w oknie konsoli systemu operacyjnego Windows. Dlatego nowo tworzony projekt aplikacji powinien być typu „Console Application”.



Tworząc nowy projekt należy w dolnej części ekranu wprowadzić nazwę projektu („Name:”), oraz ścieżkę do folderu na dysku twardym, w którym wszystkie pliki wchodzące w skład tego projektu będą zapisywane („Location:”).

Wybór rodzaju tworzonej aplikacji („Console Application”) spowoduje pojawienie się w nowym projekcie kilku niezbędnych linii kodu, które umożliwią komputerowi przetworzenie i uruchomienie pisanego przez programistę programu:



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication32
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

Pokrótkie omówimy te linie. Na samym początku są instrukcje zaczynające się od słowa *using*, po którym wpisane są nazwy klas, jakie będą dostępne w bieżącym programie. Zaraz poniżej jest polecenie zaczynające się od słowa *namespace* (przestrzeń nazw). Dopiero w ostatnim rozdziale wyjaśnimy, czym jest przestrzeń nazw i trochę więcej napiszemy o poleceniu *using*. Teraz możemy te zagadnienia ominąć. Prosimy tylko przyjąć, że te elementy muszą występować w programie, pełnią ważną rolę, którą bliżej poznamy później (w [podrozdziale 6.6](#)). Zakres jaki obejmuje dana instrukcja jest oznaczony nawiasami klamrowymi „{ }”.

W kolejnej linii (po nawiasie klamrowym otwierającym) jest słowo kluczowe *class* i nazwa klasy (tu *Program*). Język C# jest językiem obiektowym, program napisany w tym języku zbudowany jest z pewnych komponentów i nie można pisać kodu „luzem” (niektóre języki na to pozwalają). Nawet najprostszy program musi mieć przynajmniej jedną klasę.

Wewnątrz klasy zawarta jest definicja metody *Main()* – jest to obowiązkowa metoda, od niej zaczyna się wykonywanie programu. W najbliższych rozdziałach (2 – 4) kod tworzonych programów będziemy umieszczać w metodzie *Main()*.

Przykładowe kody programów w tym podręczniku będą się pojawiać w dwóch formach:

- kompletnego przykładu, który wystarczy wpisać⁶ do środowiska i uruchomić, oraz
- fragmentu kodu, który stanowi nierzadko alternatywne rozwiązanie dla jakiejś części omawianego programu.

Kompletne przykłady są numerowane w całej książce, a ponadto mają zachowaną strukturę – i tak w rozdziałach 2 – 4 numerowane przykłady są objęte deklaracją metody *Main()*, czyli są umieszczone wewnątrz zapisu (między klamrami):

```
static void Main(string[] args)
{
    // Tu jest kod przykładu
}
```

W rozdziale piątym dodatkowo znajdują się definicje innych metod. A w rozdziale szóstym – definicje klasy.

Wpisując (lub kopiując) przykłady z podręcznika należy pilnować struktury kodu. Zawartość przykładu – całość tekstu wewnątrz metody *Main()* (w klamrach) – wpisać należy do wnętrza deklaracji metody *Main()* w edytorze środowiska programistycznego. Należy jednocześnie zachować ostrożność, aby edytując kod źródłowy aplikacji nie usuwać żadnego znaku, który pojawił się w szablonie projektu aplikacji konsolowej.

Edytor kodu źródłowego dostępny w pakiecie Visual Studio posiada kilka istotnych cech, które bardzo ułatwiają pracę programiście. Przede wszystkim, w edytorze występuje tzw. kolorowanie składni, to znaczy, że poszczególne elementy kodu źródłowego będą wyróżniane kolorem (np. słowa kluczowe, ciągi znaków, komentarze itp.).

Bardzo dużym udogodnieniem jest zaznaczanie par nawiasów. Wszystkie występujące w kodzie źródłowym nawiasy (klamrowe, kwadratowe, okrągłe) po otwarciu muszą być zamykane. Edytor tekstu Visual Studio pozwala sprawdzić, gdzie znajduje drugi nawias z danej pary nawiasów. Wystarczy umieścić kursor przed nawiasem otwierającym lub za nawiasem zamykającym, aby zaznaczona w kolorze szarym została para odpowiadających sobie nawiasów.

```
{
```

```
}
```

To drobne, jak się wydaje, udogodnienie w praktyce pozwala programiście zaoszczędzić bardzo dużo czasu.

⁶ Na początku nauki zalecamy, aby przepisywać kod przykładu, a nie kopiować.

Dla komputera (a dokładnie kompilatora) ważna jest treść kodu źródłowego, nie ma znaczenia sposób jego zapisu. Warto jednak od samego początku wyrobić w sobie nawyk czytelnego zapisu kodu źródłowego. Pozwoli to zaoszczędzić czas podczas edycji kodu. Często zdarza się, że programista musi poprawić lub rozbudować aplikację, którą napisał „kiedyś tam”, czasami nawet wiele lat temu. Odpowiedni wygląd kodu źródłowego wyraźnie skraca czas przypominania sobie, co dzieje się w danym fragmencie programu. Jeszcze większe znaczenie ma sposób zapisu kodu programu w sytuacji, gdy pracuje nad nim większa liczba osób.

Aby poprawić czytelność kodu źródłowego należy przede wszystkim stosować komentarze. Komentarz to nic innego jak dowolny tekst, umieszczony w kodzie źródłowym, który jednak nie wpływa na działanie programu, jest przez kompilator pomijany. W większości edytorów programistycznych istnieje możliwość stosowania komentarzy obejmujących jedną linię lub wiele kolejnych linii. W przypadku komentarza ograniczonego do pojedynczej linii kodu źródłowego jako komentarz potraktowany zostanie tekst umieszczony po podwójnym znaku ukośnika //. Można więc napisać linię programu, która w całości będzie tylko komentarzem lub umieścić komentarz z prawej strony polecenia, które zostanie przez komputer wykonane. Oba sposoby użycia jednoliniowego komentarza pokazuje poniższy przykład:

```
static void Main(string[] args)
{
    // poniższa linia wyświetli tekst

    Console.WriteLine("Witaj świecie"); // ta linia wyświetla tekst
}
```

Jeżeli komentarz ma obejmować kilka linii można utworzyć tak zwany komentarz blokowy, który powinien zostać umieszczony pomiędzy znakami /* oraz */.

```
/* To
   jest
   komentarz
   blokowy
*/
```

Wpisywanie w edytorze środowiska Visual Studio Express kolejnych linii komentarza (potwierdzanych klawiszem *Enter*) powoduje, że nowe linie automatycznie rozpoczynają się od znaku gwiazdki (jak na rysunku niżej). Znaki gwiazdki (pojedynczej gwiazdki bez ukośnika) dla linii środkowych komentarza blokowego są umieszczane przez środowisko jedynie dla zwiększenia czytelności.

```

/* To
 * jest
 * komentarz
 * blokowy
 */

```

Komentarze umieszczać należy przede wszystkim w tych miejscach programu, które są złożone (skomplikowane), a same nazwy użytych zmiennych czy metod nie wystarczają do szybkiego odczytania funkcjonalności danego fragmentu programu. Nie powinno się natomiast umieszczać komentarzy tam gdzie, kod źródłowy jest wystarczająco czytelny (wręcz oczywisty). Przykład linii ze zbędnym komentarzem:

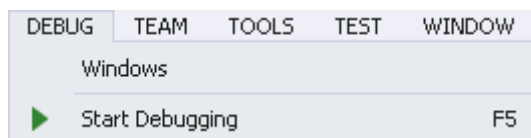
```
liczbaMaszyn = 10;    // Liczba maszyn równa się 10
```

Drugim istotnym elementem zwiększającym czytelność kodu źródłowego jest stosowanie wcięć poszczególnych wierszy. Edytor Visual Studio w sposób automatyczny sugeruje wcięcia i nie należy go poprawiać w tym względzie. Zdarza się, że początkujący programiści (kierując się osobliwie pojmowanym poczuciem estetyki) usuwają wcięcia tak, aby wszystkie linie programu rozpoczynały się w tym samym miejscu (tj. w tej samej kolumnie). W efekcie bardzo pogarsza się czytelność kodu źródłowego, a tym samym wydłuża się czas poszukiwania ewentualnych błędów i miejsc późniejszych zmian. Edytor kodu źródłowego Visual Studio posiada wiele innych udogodnień, z którymi Czytelnik będzie miał możliwość zapoznania się w czasie lektury książki⁷.

Po napisaniu przez programistę kodu źródłowego można przejść do uruchomienia aplikacji. W Visual Studio czynność ta sprowadza się do kliknięcia myszką ikony zielonej strzałki z napisem *Start* (która znajduje się w górnym menu) lub naciśnięcia klawisza funkcyjnego *F5*.



Uruchomienie kodu napisanej aplikacji nastąpi również po wybraniu z menu „Debug” opcji „Start Debugging”.



Mozemy już przystąpić do napisania naszej pierwszej aplikacji. Najpierw wpiszemy wewnątrz metody *Main()* dwie linie, np. jak w przykładzie:

⁷ W środowisku programistycznym MS Visual Studio jest możliwość automatycznego stosowania wcięć w trakcie pisania kodu. Można także wymusić wyrównanie wcięć dla całego dokumentu (lub wskazanego fragmentu) – opcja *Edit / Advanced / Format Document*.

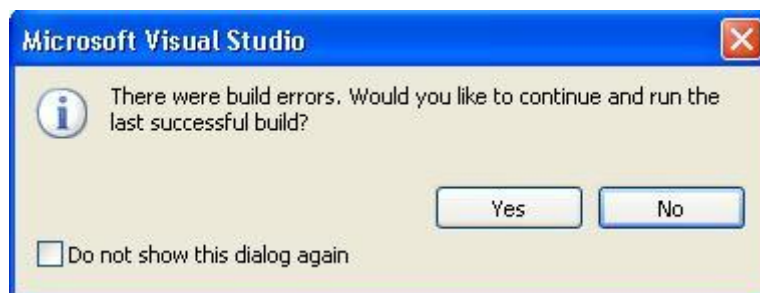
Przykład 1.1.

```
static void Main(string[] args)
{
    Console.WriteLine("Witaj świecie!");
    Console.ReadKey();
}
```

Po wpisaniu kodu programu można go uruchomić, klikając ikonkę z zieloną strzałką lub naciskając klawisz *F5*. Jeżeli kod programu został prawidłowo wpisany, to po jego uruchomieniu zobaczymy napis „Witaj świecie!”. Wyświetlenie tego napisu zrealizowała metoda *Console.WriteLine()*. Druga z metod, *Console.ReadKey()*, pełni rolę pomocniczą – zatrzymania na ekranie okienka z konsolą, tak aby można było odczytać komunikat (dopiero wciśnięcie dowolnego klawisza zamknie to okienko). Proszę zauważyć, że obie te linie kończą się średnikiem, co też jest obowiązkowe.

Jeżeli Czytelnikowi udało się uruchomić powyższy program, to dobrze, ale jeśli się nie udało, to też dobrze, a może nawet lepiej. Bo właśnie chcemy zasymulować sytuację błędną i jeśli ktoś ma błąd, to już jest gotów.

W programie wykonamy celowo błąd, aby wiedzieć jak reagować w przypadku błędów zgłaszanych przez kompilator języka. Przykładowo, usuńmy średnik w dowolnej linii. Pojawi się wówczas okno z komunikatem, jaki przedstawia rysunek:



W pojawiającym się oknie należy wybrać odpowiedź „No”, a następnie przeczytać informacje o błędzie w oknie *Error List*, np.: „Program.cs (12,30): error CS1002: ; expected”. Kliknięcie w linię z opisem błędu spowoduje umiejscowienie kursora w linii, w której jest błąd. Treść błędu „; expected” oznacza, że oczekiwany jest średnik we wskazanej linii programu. Nie zawsze jednak komunikat o błędzie pokazuje dokładnie tę linię, w której jest faktyczna przyczyna błędu. Niezamknięcie klamry „}” lub brak zamknięcia cudzysłowu może skutkować komunikatem o błędzie w innym miejscu (szerzej o tym w [podrozdziale 7.1](#)).

Podsumowując kwestię błędów w programie – jeżeli kod źródłowy nie będzie zawierał błędów, aplikacja zostanie uruchomiona w oknie konsoli (ponieważ w niniejszej książce tworzone są projekty aplikacji konsolowych). Częściej jednak zdarza się, że kod źródłowy zawiera błędy (zwłaszcza podczas jego pierwszego uruchomienia). W tym przypadku uruchomienie programu zostanie zatrzymane, a w dolnej części ekranu pojawi się informacja

o rodzaju napotkanego błędu lub błędów, oraz o numerze linii, w której (wg kompilatora) błąd występuje – co pokazaliśmy podczas symulacji sytuacji błędnej. Wbrew intuicji początkującego programisty, pojawiająca się informacja o błędzie jest powodem do radości. W czasie nauki programowania niejednokrotnie pojawią się sytuacje, w których informacji o błędzie nie będzie, a mimo tego program nie będzie działał poprawnie. Dlatego warto się uśmiechać za każdym razem, gdy kompilator poinformuje o znalezionym błędzie w kodzie programu. W [rozdziale 7](#) zostanie przedstawiony debugger, czyli element Visual Studio umożliwiający identyfikowanie błędów i analizowanie programu.

2 Arytmetyka w programowaniu

Arytmetyka to jedna z najstarszych części matematyki, opisująca podstawowe działania na liczbach. W tytule rozdziału to metafora szeroko rozumianych operacji elementarnych w programowaniu. Naukę programowania (która, jak zapowiadano we wstępie, będzie polegała głównie na wykonywaniu przykładów) rozpoczniemy od wykonania podstawowych działań – zarówno na liczbach jak i na innych rodzajach danych. W trakcie analizy przykładów bliżej poznamy wybrane typy danych, możliwe operacje na nich oraz operatory, których możemy użyć budując wyrażenia. Zaczniemy od prostych wyrażeń algebraicznych, poprzedzając krótkim omówieniem zmiennych i ich typów.

2.1 Zmienne i ich typy

Zmienna to konstrukcja programistyczna umożliwiająca przechowywanie danej. W poprzednim rozdziale ([punkt 1.1.1](#)) napisaliśmy, że zmienna ma kilka atrybutów – identyfikator (nazwę), typ oraz może posiadać wartość. Oprócz wymienionych zmienna ma jeszcze jeden ważny atrybut – adres, czyli miejsce przechowywania. Najpierw przedstawimy krótko typy. Nie będziemy ich tu szczegółowo omawiać – są zawarte w tabeli na końcu książki i będą wyjaśniane na konkretnych przykładach w dalszych podrozdziałach. Tu tylko umieszczamy wstęp do tematu typów, a zaczniemy od przedstawienia „typów typów”, bo są dwa główne rodzaje typów – wartościowe i referencyjne. Później pokażemy deklaracje zmiennych.

2.1.1 Typ wartościowy i typ referencyjny

Wyobraźmy sobie wielkie archiwum, w rzędach stoją segmenty pełne szuflad w różnych rozmiarach, a każda szuflada ma określony symbol (nazwę) oraz swój „adres” (np. numer segmentu i numer szuflady). Całe archiwum to będzie dla nas metafora pamięci komputera. Archiwum jest duże i gdy ktoś potrzebuje wykonać pracę na konkretnych dokumentach, może zgromadzić najbardziej potrzebne dokumenty na jakimś dużym biurku. Dzięki temu rozwiązaniu nie trzeba zbyt wiele czasu tracić na chodzenie do określonych segmentów. Niemniej w przypadku niektórych danych trzymanie ich na tym podręcznym biurku mogłoby być kłopotliwe. Powodem może być np. ich duży rozmiar. Wówczas na biurku kładziemy sobie jedynie kartkę z adresem do danego zasobu (dokumentu), a gdy akurat będzie nam potrzebny, odczytamy z kartki adres i pójdziemy do wskazanego segmentu. Miejsce i czas – to kryteria wpływające na organizację pamięci komputera. Nie możemy mieć pod ręką wszystkiego, bo wówczas „zagracymy” sobie warsztat pracy. Chyba każdy z nas doprowadził kiedyś swoje biurko do takiego stanu, gdy za dużo rzeczy było na wierzchu i w końcu nie można było niczego znaleźć. Ale nie możemy też mieć pustego biurka

i po każdą rzecz z osobna chodzić i tracić czas. Gdy pracujemy, mamy przed sobą konkretne zadanie. Dla tego zadania ustalamy, co będzie nam potrzebne pod ręką, a co może być na swoim miejscu, a nam wystarczy znajomość lokalizacji.

W programowaniu w środowisku .NET jest podobnie. Program to konkretne zadanie dla komputera. Programista ustala, jakie rzeczy mają być „na biurku” – kładzione są one na tzw. **stosie** (ang. stack) – a które w segmentach, czyli **stercie** (ang. heap) w wolnej pamięci. Przy czym to ustalenie, co ma być w jakim miejscu odbywa się pośrednio – poprzez wybór typu dla zmiennej. Tylko niektóre dane (o określonym typie) nadają się do tego, aby je czasowo przechowywać na biurku (na stosie). O takich danych mówi się, że mają **typ wartościowy**. Natomiast te dane, które muszą być umieszczone w segmentach (na stercie) mają **typ referencyjny**. Referencja zawiera adres (odniesienie) do faktycznego miejsca, gdzie coś jest przechowywane. Referencję zapisuje się na stosie, a dane, na które referencja wskazuje – zapisuje się na stercie. Wrócimy jeszcze do tego tematu w [podrozdziale 6.4](#).

Napisaliśmy w pierwszym rozdziale, że typ zmiennej określa rodzaj danej (np. czy jest liczbą, tekstem, wartością logiczną itd.) oraz jej rozmiar. Typ określa także, czy wartość zmiennej będzie ona na stosie czy stercie, inaczej mówiąc, czy to jest dana typu wartościowego czy typu referencyjnego⁸. Typy wartościowe to przede wszystkim typy dla pojedynczych danych (niezłożonych), takich jak liczba, znak czy wartość logiczna, ale także struktury oraz typ wyliczeniowy⁹. Typy referencyjne to m.in. tablice, łańcuchy znakowe i klasy.

Każdy język udostępnia listę gotowych typów, są to tzn. **typy wbudowane**. Ich listę zawiera [Tabela 1](#) w *Dodatkach*. W ostatniej kolumnie tabeli jest informacja, czy dany typ wbudowany to typ wartościowy, czy referencyjny. Tylko jeden z tej listy jest typem referencyjnym – typ *string* (dla łańcuchów znakowych). Niemniej dla wygody programisty można się danymi tego typu posługiwać (przy niektórych operacjach) jakby były danymi typu wartościowego, ale o tym więcej powiemy w [podrozdziale 4.2](#).

W bieżącym rozdziale (drugim) bazować będziemy na typach wbudowanych. W poszczególnych częściach tego rozdziału będą (na przykładach) omawiane wybrane typy wbudowane. Natomiast w rozdziale czwartym poznamy zmienną typu referencyjnego w postaci tablicy. W rozdziale szóstym będziemy tworzyć własne klasy, czyli definiować własne typy – a klasy także są typem referencyjnym. I tam dowiemy się, co to jest obiekt. Już teraz możemy wspomnieć, że **każda zmienna w języku C# jest obiektem**, przy czym w

⁸ Może się pojawić pytanie, czy to określenie jest stałe. Czy przykładowo wartość danej typu *int* (typ wartościowy) może znaleźć się na stercie? Owszem jest taka możliwość, którą wykonuje się poprzez tzw. pakowanie (z typu wartościowego na referencyjny) oraz rozpakowanie (z typu referencyjnego na wartościowy). W tym podręczniku nie będziemy pogłębiać tego zagadnienia. Dla zainteresowanych i bardziej zaawansowanych czytelników podajemy stosowny link: <http://msdn.microsoft.com/en-us/library/y2be5wk.aspx>.

⁹ Struktury zostały opisane w [podrozdziale 6.5](#). Typ wyliczeniowy jest opisany na stronie <http://msdn.microsoft.com/en-us/library/sbbt4032.aspx>.

odniesieniu do zmiennych, które mają typ wbudowany częściej używa się terminu „zmienna”, a w odniesieniu do zmiennych pozostałych typów (klas własnych lub bibliotecznych) używa się terminu „obiekt”.

2.1.2 Deklaracja zmiennych

Przedstawimy składnię deklaracji pojedynczej zmiennej. W opisie składni elementy ujęte w nawiasy kwadratowe są nieobowiązkowe.

Składnia 2.1

Typ Nazwa [=wartość];

W deklaracji umieszczamy typ zmiennej, jej nazwę i opcjonalnie możemy przypisać wartość początkową. Takie przypisanie wartości początkowej nazywać będziemy **inicjalizacją**.

W języku C# obowiązują określone zasady nazewnictwa zmiennych. Nazwa zmiennej może się składać jedynie z alfanumeryków oraz znaku podkreślenia. Alfanumeryk to cyfra lub litera (mała lub duża), przy czym nazwa zmiennej nie może być słowem kluczowym oraz nazwa zmiennej nie może się zaczynać od cyfry. Wykaz słów kluczowych znajduje się w *Dodatkach* (Tabela 4). W nazwach zmiennych odróżniane są duże i małe litery, przykładowo zmienne *Temperatura* oraz *temperatura* będą widziane w programie jako dwie różne zmienne. Polskie znaki diakrytyczne (np. Ą, ą, ę, ć, itd.) można stosować, ale nie zaleca się¹⁰.

Poniżej przedstawiona jest przykładowa deklaracja dla danej typu *int*, („int” to skrót od słowa „integer”), czyli typu całkowitoliczbowego:

```
int x;
```

Deklarowana jest tu zmienna o nazwie *x* i typie *int*. Od tego miejsca kompilator „wie”, że *x* to zmienna i że jest typu *int*, ale jeszcze nieznana jest jej wartość. Próba wyświetlenia wartości tej zmiennej poprzez instrukcję `Console.WriteLine(x);` zakończy się błędem¹¹. Jeśli wpiszemy wartość do zmiennej *x*, będzie można wyświetlić jej zawartość, co pokazuje przykład 2.1.

¹⁰ W tej sprawie zdania są podzielone. Wolelibyśmy nie roztrząsać tej kwestii. Należy przyjąć zasady ustalone przez zespół programistyczny, w którym się pracuje.

¹¹ Komunikat o błędzie będzie brzmiał „Use of unassigned local variable 'x'”, co oznacza, że użyto zmiennej lokalnej, nie mającej przypisanej wartości. **Zmienna lokalna** – zmienna, która ma zakres lokalny w metodzie, w której jest deklarowana. W C# nie ma zmiennych globalnych. Definiuje się natomiast pola w klasach (podobnie jak zmienne lokalne), ale pola, w przeciwieństwie do zmiennych lokalnych, mają wartości domyślne (zgodnie z typem) – więcej na ten temat w rozdziale 6.

Przykład 2.1.

```
static void Main(string[] args)
{
    int x;
    x = 5;
    Console.WriteLine(x);
    Console.ReadKey();
}
```

Zgodnie z tym, co obrazuje składnia deklaracji zmiennej, można w tej samej linii zadeklarować zmienną i przypisać jej wartość początkową (zainicjalizować). Zatem gdyby zamiast dwóch pierwszych instrukcji umieścić tę jedną `int x = 5;` to program działać będzie tak samo.

W jednej linii można zadeklarować kilka zmiennych (jeśli są tego samego typu), np.:

```
int x, y = 3, z;
```

W powyższej deklaracji zadeklarowane są trzy zmienne typu *int*, o nazwach *x*, *y*, *z*, przy czym jedna z nich (*y*) jest inicjalizowana wartością 3. Poszczególne zmienne na liście rozdziela się przecinkiem.

Jeśli do zmiennej przypiszemy wartość innej zmiennej, to w przypadku typów wartościowych zostanie przekopiuwana wartość. Przykładowo mamy dwie zmienne deklarowane następująco:

```
int x = 0;
int y = x;
```

Ponieważ *int* jest typem wartościowym – po powyższej deklaracji obie zmienne *x* i *y* „żyją swoim własnym życiem”, mimo że mają tę samą wartość. Jeśli później zmienimy tylko *x*, to zmienna *y* nadal będzie równa 0. W przypadku typów referencyjnych będzie inaczej – takie zmienne nie będą miały niezależnego „życia”, ponieważ obie będą wskazywały na to samo miejsce w pamięci. Uzupełnienie tych zagadnień znajduje się w [podrozdziale 6.4](#).

Na koniec tego podrozdziału przedstawimy inny (alternatywny) sposób deklaracji zmiennej. Poniżej są dwie tożsame deklaracje zmiennej *x*.

<code>int x = 0;</code>	<code>int x = new int();</code>
-------------------------	---------------------------------

Po lewej stronie jest deklaracja według omówionej [składni 2.1](#) (zmienna *x* typu *int*, zainicjalizowana wartością 0), natomiast po prawej została użyta inna składnia:

```
Typ Nazwa = new Typ();
```

Składnia 2.2

Operator *new* ma w języku C# kilka ról. Tu służy do tworzenia obiektu i wywoływania specjalnej funkcji, która inicjalizuje zmienną (ta funkcja to tzw. konstruktor i nazywa się tak samo jak typ, szczegóły poznamy w [rozdziale 6](#)). Zmienna typu *int*, podobnie jak i zmienne pozostałych typów wartościowych są w takich przypadkach inicjalizowane wartościami domyślnymi¹². Dla typu *int* wartością domyślną jest 0. Dla typów referencyjnych wartością domyślną jest tzw. **null** (oznacza brak referencji). Składnia 2.2 w najbliższych rozdziałach nie będzie nam potrzebna, ale później przypomnimy (i rozwinimy) ten zapis w kontekście typów referencyjnych – tablic i klas.

2.2 Wyrażenia algebraiczne

W tym podrozdziale w zasadzie powinniśmy zacząć od zapowiedzi, że zapis wyrażeń algebraicznych w programowaniu jest oparty na matematycznych zasadach, co oznacza, że nic nowego nas tu nie zaskoczy. Podobieństw znanych z lekcji matematyki w szkole podstawowej, gimnazjum czy średniej jest więcej niż różnic. Ale mimo tych podobieństw, a może wręcz z ich powodu, wskazana będzie pewna czujność. Podobnie jak w przypadku nauki języków podobnych do już znanego. Np. czeski wyraz „čerstvý” to po polsku „świeży”. A czeski miesiąc „kveten” to nasz „maj”. Będziemy zwracać uwagę na analogiczne pułapki w odniesieniu do kodowania wyrażeń algebraicznych.

2.2.1 Działania bez konwersji typów

Przykład, który będzie omawiany nie zawiera żadnych „pułapek”. Dotyczy danych typu całkowitego, jedyne, co wymaga zwrócenia uwagi to kwestia braku domyślnego operatora mnożenia. W matematyce można opuszczać znak mnożenia między zmiennymi lub liczbą i zmienną (stałą). Np. wzór $O = 2\pi r$ jest całkowicie czytelny bez umieszczania znaków mnożenia. W programowaniu nie można tak zapisywać iloczynu. Zmienne mogą mieć nazwy wieloznakowe i brak operatorów mnożenia rodziłby niejednoznaczność.

Przykład 2.2.

Kolejny przykład, jaki uruchomimy w tym rozdziale będzie obliczał wartość prostego wyrażenia: $2a - 3b$, gdzie *a* i *b* to będą liczby całkowite. Zdefiniujemy treść metody *Main()* następująco:

```
static void Main(string[] args)
{
    int a, b, wynik;
    a = 4;
    b = 1;
    wynik = 2 * a - 3 * b;
```

¹² Wartości domyślne dla wszystkich typów wartościowych przedstawione są na stronie: <http://msdn.microsoft.com/en-gb/library/83fhsxwc.aspx>

```

    Console.WriteLine(wynik);
    Console.ReadKey();
}

```

Omówimy poszczególne linie tego programu. Przy każdej linii wyjaśniamy nie tylko co dana linia wnosi do programu, ale także zagadnienia bezpośrednio związane z danym poleceniem. Jak już wspomniano przy uruchomieniu pierwszego programu – w języku C# poszczególne deklaracje i instrukcje są kończone znakiem średnika.

Linia kodu	Komentarz
<code>int a, b, wynik;</code>	W pierwszej linii deklarujemy zmienne. Ponieważ wyrażenie ma dotyczyć liczb całkowitych, przypisaliśmy dla wszystkich zmiennych typ całkowity – <i>int</i> . Nazwę typu umieszczamy przed listą zmiennych. Zmienne oddzielamy przecinkami. Jest więcej typów całkowitoliczbowych, np. <i>short</i> lub <i>long</i> , które różnią się rozmiarem (zakresem wartości), ale uprościmy początkowy etap nauki przyjmując, że wszystkie całkowite liczby w naszych programach będą typu <i>int</i> . W <i>Dodatkach Tabela 1</i> przedstawia wszystkie typy wbudowane w C#.
<code>a = 4;</code> <code>b = 1;</code>	Przypisanie konkretnych wartości dla obu zmiennych. Ponieważ to pierwsze przypisanie dla tych zmiennych (później można wpisać inne całkowite wartości), nazywane jest inicjalizacją . Wpisane wartości dla omawianego przykładu nie mają większego znaczenia (można wpisać inne). Wkrótce użyjemy instrukcji, dzięki którym będzie możliwe pobranie wartości od użytkownika. Obie linie wraz z poprzednią można by zamienić na taką: <code>int a = 4, b = 1, wynik;</code> W powyższej wersji zarówno deklaracja zmiennych jak i inicjalizacja zmiennych <i>a</i> oraz <i>b</i> jest w tej samej linii.
<code>wynik = 2 * a - 3 * b;</code>	Wartościowanie wyrażenia. W programowaniu nie możemy pominąć operatora mnożenia, jest nim znak „*”. Do zmiennej <i>wynik</i> zostanie przypisana wartość $2*4-3*1$.
<code>Console.WriteLine(wynik);</code>	Metoda <i>WriteLine()</i> wyświetli wynik, czyli wartość 5
<code>Console.ReadKey();</code>	Metoda <i>ReadKey()</i> czeka na wciśnięcie dowolnego klawisza, dzięki jej umieszczeniu okienko z wynikiem zostanie na ekranie do czasu wciśnięcia klawisza.

2.2.2 Modyfikacja typów dla literalów

Każdy literal ma typ domyślny. Liczba całkowita ma domyślny typ *int*. Natomiast liczba z kropką dziesiętną ma domyślny typ *double*. Można jednak zmienić domyślny typ literalu dodając specjalny sufix - literowy modyfikator typu. Poniżej umieszczamy wykaz wybranych modyfikatorów typów dla literalów:

Modyfikator typu	Komentarz
F lub f	Modyfikacja typu literału liczbowego na typ <i>float</i> , np. 3.5f to jest literał 3.5 typu <i>float</i> . Analogicznie 5.7F to literał 5.7 typu <i>float</i> . Deklaracja <i>float x = 34.5</i> ; spowoduje błąd, ponieważ literał 34.5 ma domyślny typ <i>double</i> . Dopiero dopisanie modyfikatora dla typu <i>float</i> pozwoli poprawnie wykonać deklarację z inicjalizacją literału z separatorem dziesiętnym do zmiennej typu <i>float</i> : <i>float x = 34.5f</i> ; lub <i>float x=34.5F</i> ;
D lub d	Modyfikacja typu literału liczbowego na typ <i>double</i> , np. 3d to jest literał 3.0 typu <i>double</i> .
M lub m	Modyfikacja typu literału liczbowego na typ <i>decimal</i> , np. 3m to jest literał 3.0 typu <i>decimal</i> .
L lub l	Modyfikacja typu literału liczbowego na typ <i>long</i> , np. 1323234L to jest literał 1323234 typu <i>long</i> .

Przydatność użycia modyfikatora typu dla literału przedstawimy na przykładzie dotyczącym dzielenia. W matematyce najczęściej spotykane operatory dzielenia to dwukropek, kreska ułamkowa oraz ukośnik. W programowaniu używa się tylko tego ostatniego. Operator dzielenia „/” w języku C# może zostać użyty w dwóch rolach – operatora dzielenia całkowitoliczbowego oraz zwykłego dzielenia. Dzielenie całkowitoliczbowe polega na odrzuceniu części ułamkowej wyniku dzielenia (np. wynik całkowitoliczbowy z wyrażenia $5/2$ równa się 2). Dzielenie całkowitoliczbowe w niektórych językach programowania można uzyskać poprzez specjalny operator (inny niż operator dzielenia)¹³. W języku C# jest tylko jeden operator dzielenia, a to, czy pełni rolę całkowitoliczbowego czy nie – zależy od typu zmiennych, stanowiących dzielną i dzielnik.

Przy okazji omawiania operatora dzielenia, krótko przedstawimy jeszcze jeden operator związany z dzieleniem – „%”, który zwraca resztę z dzielenia¹⁴. Przykładowo, wynikiem działania $10 \% 2$ jest 0. Natomiast wynik działania $10 \% 3$ wynosi 1.

Kolejny program będzie przeliczał temperaturę w stopniach Fahrenheita na temperaturę w stopniach Celsjusza zgodnie z wzorem $C = \frac{5}{9}(F - 32)$. Na obecnym etapie narzuca się zapisanie tego wyrażenia w języku C# jako $C = 5/9*(F-32)$. Wykonamy program z tak zakodowanym wyrażeniem i sprawdzimy jego wynik. Tym razem poprosimy użytkownika o wprowadzenie danej. Definicja metody *Main()* będzie wyglądać tak:

¹³ W języku Pascal operator całkowitoliczbowy to *div*.

¹⁴ W języku Pascal do wyznaczania reszty z dzielenia służy operator *mod* („mod” to skrót od pojęcia „modulo”, które oznacza operację wyznaczania reszty z dzielenia).

Przykład 2.3.

```
static void Main(string[] args)
{
    double F, C;
    Console.WriteLine("Podaj temp. w stopniach Fahrenheita");
    F = double.Parse(Console.ReadLine());
    C = 5 / 9 * (F - 32);
    Console.WriteLine(C);
    Console.ReadKey();
}
```

Pierwszy test możemy przeprowadzić dla $F=41$, wówczas wyrażenie $5/9 * (41-32)$ powinno dać 5. Gdy uruchomimy program widzimy jednak, że tak się nie dzieje. Należy uruchomić program, a następnie przeczytać poniższe uwagi.

Linia kodu	Komentarz
<code>double F, C;</code>	Tu także zaczynamy od deklaracji zmiennych. Ponieważ wyrażenie może dotyczyć liczb rzeczywistych przypisany został dla obu zmiennych typ <i>double</i> . Jest więcej typów dla liczb niecałkowitych (<i>float</i> oraz <i>decimal</i>), które różnią się rozmiarem i precyzją, ale w początkowym etapie nauki najczęściej będziemy używać typu <i>double</i> . W <i>Dodatkach Tabela 1</i> przedstawia wszystkie typy wbudowane w C#.
<code>Console.WriteLine("Podaj temp. w stopniach Fahrenheita");</code> <code>F=double.Parse(Console.ReadLine());</code>	Wyświetlony zostaje komunikat z prośbą o wprowadzenie wartości, a następnie wywołana jest metoda <i>ReadLine()</i> dla konsoli, co oznacza, że program będzie czekał na wprowadzenie danej. Metoda <i>ReadLine()</i> wczytuje dane typu <i>string</i> (łańcuch znaków), a ponieważ potrzebujemy danej typu <i>double</i> korzystamy z metody <i>double.Parse()</i> , która zamieni typ <i>string</i> na typ <i>double</i> . Metody można zagnieżdżać i w tym programie mamy pierwszy tego przykład. Zagnieżdżone metody czytamy „od środka” – program najpierw przeczyta to co wprowadzi użytkownik (i potwierdzi klawiszem <i>Enter</i>), a następnie wartość ta zostanie zmieniona na typ <i>double</i> .
<code>C = 5 / 9 * (F - 32);</code>	Wartościowanie wyrażenia (czyli obliczanie jego wartości). Jeśli wprowadzimy $F = 41$, to oczekujemy wyniku $C = 5$. Po uruchomieniu programu widzimy jednak, że wyświetla się 0, a nie 5. Dlaczego? Ponieważ literały 5 i 9 (o literałach pisaliśmy już w rozdziale 1.1.2) są typu <i>int</i> . Nie było jawnego przypisania typu, a to jest domyślny typ dla literałów całkowitych. Operator dzielenia dla argumentów typu <i>int</i> występuje w roli dzielenia całkowitoliczbowego. Zatem reszta z dzielenia zostanie utracona i wynik $5/9$ zamiast liczby 0,55(5) będzie 0, przez co całe wyrażenie będzie także zerem (zob. rozdział 2.5 o kolejności wykonywania działań). Co zrobić, aby uzyskać oczekiwany wynik? Dokonać tego można na kilka sposobów, pod tabelą zaprezentujemy dwa z nich.
<code>Console.WriteLine(C);</code>	Metoda <i>WriteLine()</i> wyświetli wynik.

Pierwszy sposób poprawienia programu – modyfikator typu

Musimy zmienić typ dla przynajmniej jednego z literałów (5 lub 9). Najpierw posłużymy się modyfikatorem typu, który ustawia inny niż domyślny typ dla literału liczbowego. Dla liczby całkowitej domyślnym typem jest *int*. Jeśli chcemy, aby ta liczba była typu *double* musimy umieścić modyfikator typu jako sufix, czyli tuż za liczbą (bez spacji), dla typu *double* sufix ten to litera „D” lub „d”. Należy zmienić linie programu z wyrażeniem na następującą i uruchomić ponownie program:

```
C = 5d / 9 * (F - 32);
```

Litera „5d” to liczba pięć typu *double*, czyli tak naprawdę 5.0 (dla nas to dalej ta sama liczba, ale dla komputera jest istotny typ literału)¹⁵.

Drugi sposób poprawienia programu – użycie kropki dziesiętnej

Tu także zmienimy typ tylko dla jednego literału, ale tym razem dopisując jedynie kropkę dziesiętną i zero. Zamiast literału 5 będzie 5.0. W celu sprawdzenia należy zmienić linie programu z wyrażeniem na następującą i uruchomić ponownie program:

```
C = 5.0 / 9 * (F - 32);
```

Litera „5.0” jak i każdy literał z separatorem dziesiętnym ma domyślny typ *double*.

2.2.3 Konwersja typów

Także i w przypadku zmiennych nierzadko zdarza się, że zmienne nie mają typu, jaki byłby pożądanym z punktu widzenia działania, jakie chcemy wykonać. Przykładowo mogą być dwie dane typu całkowitego – liczba studentów oraz liczba komputerów w pracowni. Chcemy poznać liczbę studentów przypadających na jedno stanowisko komputerowe. Wynikiem może być np. 1,2. Nie możemy zmieniać w sposób trwały typu dla obu danych, ponieważ ludzi i komputery liczymy w całości. Ale w celu obliczenia stosunku studentów do komputerów, pamiętając o działaniu operatora dzielenia „/” dla danych całkowitych w roli operatora całkowitoliczbowego – musimy zastosować coś analogicznego do modyfikacji typu dla literałów. W przypadku zmiennych (a także stałych) można użyć tzw. **rzutowania**.

Przykład 2.4.

Wykonamy program, który zaprezentuje rzutowanie na przykładzie obliczania stosunku liczby studentów do komputerów. Uruchomimy i przeanalizujemy kod metody *Main()*:

¹⁵ W języku C# jak i wielu innych separatorem dziesiętnym jest kropka (a nie przecinek). Tak jest tylko dla programisty (w kodzie programu). Użytkownik programu może wprowadzać separator dziesiętny zgodnie z ustawieniami regionalnymi dla danego kraju. W omawianym programie użytkownik wprowadza wartość z przecinkiem jako separatorem dziesiętnym, np. dla $F = 45,5$ i otrzymujemy wynik 7,5.

```
static void Main(string[] args)
{
    const int komputery = 24;
    int studenci;
    double wynik;
    Console.WriteLine("Podaj liczbę studentów: ");
    studenci = Convert.ToInt32(Console.ReadLine());
    wynik = (double)studenci / komputery;
    Console.WriteLine(wynik);
    Console.ReadKey();
}
```

Przeanalizujemy program i przy okazji poznamy bliżej kolejny element języka – stałe.

Linia kodu	Komentarz
<code>const int komputery = 24;</code>	Można przyjąć, że liczba komputerów w pracowni (o określonej powierzchni) jest względnie stała. W tym programie liczba komputerów będzie stałą, którą deklarujemy podobnie jak zmienną (także potrzebny jest typ i nazwa), przy czym musi być ten zapis poprzedzony słowem kluczowym const . Stałą należy od razu inicjalizować wartością, której zmienić już nie wolno.
<code>int studenci;</code> <code>double wynik;</code>	Deklaracja zmiennych.
<code>Console.WriteLine("Podaj liczbę studentów: ");</code> <code>studenci = Convert.ToInt32 (Console.ReadLine());</code>	Wyświetlony zostaje komunikat z prośbą o wprowadzenie wartości, a następnie wywołana jest metoda <i>ReadLine()</i> dla konsoli. Metoda <i>ReadLine()</i> wczytuje tylko dane typu <i>string</i> , a ponieważ potrzebujemy danej typu <i>int</i> skorzystamy z metody, która zamieni typ <i>string</i> na <i>int</i> . Tym razem użyjemy metody <i>Convert.ToInt32()</i> (równie dobrze moglibyśmy użyć <i>int.Parse()</i>).
<code>wynik = (double)studenci/komputery;</code>	Kluczowa linia w tym programie, w której pokazujemy rzutowanie zmiennej <i>studenci</i> – z typu <i>int</i> na <i>double</i> . Przed nazwą tej zmiennej, dla której chcemy rzutować typ umieszczamy w okrągłych nawiasach nazwę typu, na który rzutujemy. W tym programie moglibyśmy rzutować także stałą. Podmiana tej linii na taką: <code>wynik = studenci/(double)komputery;</code> da ten sam wynik.

Omówione rzutowanie typów jest przykładem tzw. jawnej konwersji typów (o niejawnej jest mowa w [podrozdziale 2.3](#)). Rzutowanie jawne informuje kompilator, że celowo zamierzamy dokonać konwersji i że jesteśmy świadomi, że może dojść do utraty danych. W przypadku rzutowania danej typu *int* na *double*, jak to miało miejsce w omawianym przykładzie, nie ma takiego niebezpieczeństwa. Typ *double* jest bardziej pojemny (sprawdź rozmiar obu typów w [Tabeli 1](#) w *Dodatkach*). Gdy przekładamy rzeczy z mniejszej szuflady do większej nie istnieje groźba, że coś się nam nie zmieści. Ale musimy się z taką sytuacją

liczyć w działaniu w odwrotną stronę – przy rzutowaniu z typu „większego” na „mniejszy”. Przykładowo rzutowanie zmiennej *double* $x = 2.65$ do typu *int* sprawi, że zmienna x po rzutowaniu będzie miała wartość 2 (część ułamkowa zostanie utracona). W przypadku rzutowania zmiennej typu *double* do *float* skutkiem może być zmniejszenie precyzji¹⁶. Utrata precyzji nie musi powodować błędu, ale wymaga się, aby proces takiej konwersji był kontrolowany przez programistę, stąd konieczne jest jawne wykonanie tej operacji.

Do konwersji typów można także wykorzystać specjalne metody. W dotychczasowych przykładach użyliśmy dwóch takich metod – *Convert.ToInt32()*¹⁷ oraz *double.Parse()*.

Tematyka konwersji typów w niniejszym podręczniku nie została wyczerpana. Gdy Czytelnik dojdzie do bardziej zaawansowanego etapu będzie musiał ją zgłębić.

2.2.4 Funkcje matematyczne

Gdyby w programie zaistniała potrzeba użycia jakiejś funkcji matematycznej, np. sinus albo funkcji obliczającej pierwiastek kwadratowy, czy logarytm, możemy skorzystać ze statycznej klasy *Math*. Rozdział o klasach jest jednym z ostatnich w tym podręczniku i tam Czytelnik dowie się, co to jest klasa statyczna. Teraz wystarczy, że powiemy o niej, że nie tworzymy obiektów dla takiej klasy oraz że używamy jej metod, poprzedzając ich nazwę kropką i nazwą klasy (*Math*). Użyliśmy już wcześniej innej klasy statycznej – *Convert*.

Przykład 2.5.

Obliczymy wyrażenie $y = \sqrt{|\sin x| \cdot \log_2 x}$. Dla uproszczenia pominiemy w tym programie badanie poprawności danej x (nie można liczyć logarytmu dla wartości ujemnych):

```
static void Main(string[] args)
{
    double x, y;
    Console.WriteLine("Podaj x (większe od 0): ");
    x = Convert.ToDouble(Console.ReadLine());
    y = Math.Sqrt(Math.Abs(Math.Sin(x)) * Math.Log(x, 2.0));
    Console.WriteLine(y);
    Console.ReadKey();
}
```

Skupimy się na linii, w której wyliczana jest wartość zmiennej y . Metoda *Math.Sin(x)* zwraca sinus z x , wynik ten stanowi wejście dla metody *Math.Abs()*, która zwraca wartość absolutną. Wynik ten jest mnożony przez logarytm o podstawie 2 z x . Natomiast wynik tego iloczynu stanowi wejście dla metody *Math.Sqrt()*, która zwraca pierwiastek kwadratowy. Można zauważyć, że zapis wyrażenia matematycznego w języku programowania jest „liniowy”. Wszystkie, nawet piętrowe wzory są kodowane w postaci liniowej, a zakres dla

¹⁶ Precyzja wyznacza liczbę miejsc po przecinku.

¹⁷ Pełna lista metod klasy *Convert*: <http://msdn.microsoft.com/en-us/library/bds4fye2.aspx>

dzielenia czy pierwiastkowania oznacza się poprzez okrągłe nawiasy. Przed użyciem metod z klasy *Math* dla funkcji matematycznych należy wcześniej sprawdzić ich specyfikację (jakie przyjmuje argumenty)¹⁸.

2.2.5 Arytmetyka++

W dotychczasowych przykładach użyte zmienne nie zmieniały się w trakcie działania programu. A tymczasem zmiana wartości zmiennych jest zjawiskiem powszechnym w programowaniu. Przytaczaliśmy w rozdziale 1 przykład instrukcji zmieniającej wartość: $x = x + 1$. Otóż w programowaniu tak często występuje potrzeba zwiększenia (lub zmniejszenia) wartości zmiennych o 1, że opracowano skróty dla kodowania tych operacji. Owe „skrótów” to operatory inkrementacji i dekrementacji. **Operator inkrementacji „++”** umieszczony przed lub po nazwie zmiennej dokonuje zwiększenia wartości tej zmiennej o 1. Instrukcja $x++$; jest zatem równoważna z instrukcją $x = x + 1$; Natomiast **operator dekrementacji „--”** umieszczony przed lub po nazwie zmiennej dokonuje zmniejszenia wartości zmiennej o 1. Instrukcja $x--$; jest równoważna z instrukcją $x = x - 1$;

Operatory inkrementacji oraz dekrementacji mogą wystąpić zarówno przed jak i po nazwie zmiennej¹⁹. Czym się różnią oba zapisy? W przypadku, gdy zmienna z takim operatorem występuje jako samodzielna instrukcja, np. $x++$; wówczas położenie operatora nie ma znaczenia. Zarówno instrukcja $++x$; jak i instrukcja $x++$; zwiększają x o 1 i w ich działaniu nie ma żadnej różnicy. Jeżeli jednak inkrementowana (lub dekrementowana) zmienna stanowi fragment jakiegoś wyrażenia, wówczas położenie operatora ma znaczenie.

Przykład 2.6.

```
static void Main(string[] args)
{
    int x = 0, y;
    y = x++ * 2;
    Console.WriteLine(x);
    Console.WriteLine(y);
    Console.ReadKey();
}
```

Prześledźmy powyższy przykład. Po uruchomieniu programu mamy na ekranie wynik: wartość $x=1$, natomiast $y=0$. Zmienna x zwiększyła się o jeden, ale stało się to **po jej użyciu w wyrażeniu** $y = x++ * 2$. Oznacza to, że dotychczasowa wartość (zero) została pomnożona przez dwa, wynik iloczynu został umieszczony w zmiennej y , a dopiero później zmienna x zwiększyła się o 1.

¹⁸ Wykaz metod w klasie *Math* wraz ze specyfikacją jest w dokumentacji MSDN: <http://msdn.microsoft.com/en-us/library/4zfevfw9.aspx>

¹⁹ Używa się specjalnych terminów z przedrostkami „pre” (przed) oraz „post” (po). I tak mamy preinkrementację (np. $++x$), postinkrementację (np. $x++$), predekrementację (np. $--x$) oraz postdekrementację (np. $x--$).

Gdy zmienimy powyższy przykład, zastępując linię, w której obliczana jest wartość zmiennej y na taką: $y = ++x * 2$; otrzymamy inny wynik: $x=1$ oraz $y=2$. Tym razem zmienna x zwiększyła się o 1 **przed jej użyciem w wyrażeniu**. Powinno być łatwo zapamiętać różnicę między tymi dwoma rodzajami inkrementacji (i dekrementacji) – jeśli operator jest **przed** zmienną, to zmienna ta będzie zwiększona (lub zmniejszona) **przed** jej użyciem w wyrażeniu. Jeśli zaś operator jest **po** zmiennej, to zmienna ta będzie zwiększona (lub zmniejszona) **po** jej użyciu w wyrażeniu.

Dla operacji zwiększania i zmniejszania o wartość różną od 1 opracowano specjalne operatory przypisania, takie jak $+=$, $-=$, $*=$. Np. instrukcję $x = x + 2$; można zapisać jako $x+=2$; Obie dokonują tego samego – zwiększają wartość zmiennej x o 2. Lista operatorów przypisania znajduje się w *Dodatkach*, w [Tabeli 2](#).

Jako ciekawostkę można wspomnieć, że nazwa języka C++ ma bezpośredni związek z operatorem inkrementacji. Język C++ to kolejna, unowocześniona wersja języka C.

Poćwiczmy trochę „arytmetykę” operatorów inkrementacji i dekrementacji. Poniższe przykłady wymagają nieco innej analizy niż poprzednie. Tym razem nie uruchamiamy od razu programu, ale staramy się wyliczyć samodzielnie to, co program powinien wykonać. Dopiero później uruchamiamy program i otrzymany wynik porównujemy z tym „naszym”.

Przykład 2.7.

Podaj wartość zmiennych x i y po wykonaniu programu:

```
static void Main(string[] args)
{
    int x, y = 4;
    x = (y += 3);
    x = ++y;
    x = y--;
    Console.WriteLine(x);
    Console.WriteLine(y);
    Console.ReadKey();
}
```

Programu nie uruchamiamy, tylko liczymy samodzielnie. W poniższej tabeli zaprezentujemy, jak najlepiej podejść do rozwiązywania tego typu zadań. Po każdej linii programu notujemy sobie na boku aktualny stan zmiennych. Zadajemy sobie pytanie – jakie wartości będą miały użyte zmienne po wykonaniu danej linii programu?

Linia kodu	x	y	Komentarz
<code>int x, y = 4;</code>	-	4	Po wykonaniu tej linii kodu zmienna x nie ma jeszcze wartości, natomiast zmienna y ma wartość 4.
<code>x = (y += 3);</code>	7	7	Zmienna y zwiększa się o 3, zatem ma wartość 7, a następnie jej wartość zostaje przypisana do zmiennej x .

<code>x = ++y;</code>	8	8	Tu najpierw zwiększa się wartość zmiennej <i>y</i> (wynosi wówczas 8) i wartość ta jest przypisana zmiennej <i>x</i> .
<code>x = y--;</code>	8	7	Tym razem najpierw następuje przypisanie (do zmiennej <i>x</i> przypisuje się wartość <i>y</i> sprzed zmniejszenia, czyli 8), a dopiero później zmienna <i>y</i> zostaje zmniejszona (do wartości 7).

Po uruchomieniu programu otrzymujemy wynik zgodny z przeprowadzonymi obliczeniami: *x* = 8, a *y* = 7.

Przykład 2.8.

Co się wyświetli na ekranie po wykonaniu programu:

```
static void Main(string[] args)
{
    int x, y = 5;
    x = ++y;
    x = y++;
    x = --y;
    x = y--;
    Console.WriteLine(y++);
    Console.ReadKey();
}
```

Podobnie jak poprzednio najpierw policzymy „na piechotę” wartości zmiennych.

Linia kodu	x	y	Konsola	Komentarz
<code>int x, y = 5;</code>	-	5	-	Po wykonaniu tej linii kodu zmienna <i>x</i> nie ma jeszcze wartości, natomiast zmienna <i>y</i> ma wartość 5.
<code>x = ++y;</code>	6	6	-	Zmienna <i>y</i> zwiększa się o 1, zatem ma wartość 6, a następnie jej wartość zostaje przypisana do zmiennej <i>x</i> .
<code>x = y++;</code>	6	7	-	Tu najpierw przypisana jest do zmiennej <i>x</i> dotychczasowa wartość zmiennej <i>y</i> (czyli 6), a dopiero później zwiększa się wartość zmiennej <i>y</i> (wynosi wówczas 7).
<code>x = --y;</code>	6	6	-	Najpierw zmniejsza się zmienna <i>y</i> (ma wówczas wartość 6), a później jest przypisana do zmiennej <i>x</i> .
<code>x = y--;</code>	6	5	-	Do zmiennej <i>x</i> jest przypisana wartość <i>y</i> sprzed zmniejszenia (czyli 6), a następnie zmienna <i>y</i> zostaje zmniejszona (do wartości 5).
<code>Console.WriteLine(y++);</code>	6	6	5	Zmienna <i>x</i> nie zmienia swej wartości w tej instrukcji. Zmienna <i>y</i> zostanie zwiększona o jeden – ale uwaga stanie się to po jej użyciu – tzn. po jej wyświetleniu. Pytanie brzmiało co wyświetli program? Wyświetli wartość 5.

Powyższy przykład jest „nieżyciowy”, tzn. takiego zapisu raczej nie spotkamy w typowym programie (wielokrotne zmiany tej samej zmiennej odbywają się w instrukcjach cyklicznych, które poznamy w kolejnym rozdziale). To tylko trening. Podobnie jak w np. siatkówce – na treningu siatkarze ćwiczą czasem z piłkami lekarskimi, znacznie cięższymi niż normalna piłka siatkowa, albo ćwiczą same pady, choć podczas meczu nigdy nie robią tego z taką intensywnością.

2.3 Wyrażenia logiczne

W świecie rzeczywistym jest wiele obiektów czy zjawisk, które można opisać zerojedynkowo – prawda lub fałsz. W językach programowania przewidziano specjalny typ dla takich danych, który w języku C# nazywa się **bool** (skrót od angielskiego „boolean” – boolowski, logiczny). Wartości dla tego typu to *true* oraz *false* (czyli prawda i fałsz). Wyrażenie logiczne to takie, którego wartość jest typu *bool*. Wartość taką można przypisać do zmiennej typu *bool* albo bezpośrednio wykorzystać w instrukcji sterującej. Ten drugi sposób użycia wartości wyrażenia logicznego poznamy w kolejnym rozdziale.

Wyrażenie algebraiczne składa się z operatorów i argumentów (operandów). Np. wyrażenie $2 + 3$ zawiera operator dodawania oraz dwa argumenty (liczby 2 i 3). Wyrażenie logiczne także składa się z operatorów oraz argumentów, przy czym posiada swój zestaw operatorów, a wszystkie argumenty muszą być typu logicznego. W tym podrozdziale będzie właśnie o tym mowa – o operatorach i argumentach wyrażeń logicznych.

Argumenty wyrażenia logicznego muszą być typu *bool*, ale nie muszą to być wyłącznie zmienne lub stałe typu *bool*. Argumentami mogą być również wyrażenia relacyjne oraz wywołane metody, które zwracają wartość typu *bool*. Metody zostaną omówione w [rozdziale 5](#). Tu skupimy się na wyrażeniach relacyjnych, które składają się z operatora relacji i argumentów, jakimi mogą być dane typu liczbowego lub znakowego²⁰. **Operatory relacji** są znane Czytelnikowi z matematyki – to znak większości ($>$), mniejszości ($<$), większe lub równe ($>=$), mniejsze lub równe ($<=$), równe ($==$) oraz różne ($!=$). Należy zwrócić uwagę na znak dla operatora równości $==$ i uważać, aby nie mylić go z operatorem przypisania $=$. Dwie instrukcje $x = y$ oraz $x == y$ to dwie różne operacje²¹. Pierwsza przypisuje wartość zmiennej y do x , natomiast druga sprawdza, czy wartość x jest równa y . Przykładowe wyrażenie relacyjne $x > 2$ zwróci albo prawdę, albo fałsz (w zależności od wartości zmiennej x).

²⁰ Można porównywać także dane innych typów, ale w tym celu trzeba zdefiniować sposób porównania. Dokonuje się tego poprzez tzw. przeciążenie (inaczej przeładowanie) operatorów.

²¹ Uwaga dla osób znających Pascala: w Pascalu przypisanie wykonuje się przy pomocy dwukropka ze znakiem równości „:=”, natomiast porównanie (czy wartości są równe) realizuje się przy pomocy pojedynczego znaku równości „=”. Może się na początku programowania w języku C# mylić, dlatego należy być czujnym.

W języku C# są cztery **operatory logiczne** dwuargumentowe (tak naprawdę to tylko dwa: koniunkcja i alternatywa – ale każdy z nich w dwóch wariantach) oraz jeden jednoargumentowy (negacja). Tabela przedstawia przykładowe wyrażenia logiczne.

Wyrażenie logiczne	Uwagi
<code>CzyZdrowy</code>	Wyrażenie składa się z jednego argumentu, którym jest zmienna typu <i>bool</i> o nazwie <i>CzyZdrowy</i> .
<code>x >= 2</code>	Wyrażenie składa się z jednego argumentu, którym jest wyrażenie relacyjne (sprawdzające wartość zmiennej <i>x</i>).
<code>CzyZdrowy && x >= 2</code>	Wyrażenie składa się z dwóch argumentów oraz operatora koniunkcji warunkowej (&&). Pierwszy argument jest zmienną typu <i>bool</i> , a drugi to wyrażenie relacyjne.
<code>!CzyZdrowy</code>	Wyrażenie składa się z operatora logicznego negacji (!), który jest jednoargumentowy, oraz zmiennej typu <i>bool</i> .
<code>!CzyZdrowy x >= 2</code>	Wyrażenie składa się z dwóch argumentów oraz operatora alternatywy warunkowej (). Pierwszy argument jest zmienną typu <i>bool</i> (dla której użyto operatora negacji), a drugi to wyrażenie relacyjne.

Przed omówieniem operatorów logicznych przypomnimy podstawowe operacje logiczne zgodnie z algebrą Boole’a:

p	q	Koniunkcja (p && q)	Alternatywa (p q)	Negacja (!p)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Wartość 0 to *false* (fałsz), natomiast wartość 1 to *true* (prawda). I tak koniunkcja dwóch zdań fałszywych jest fałszem. Natomiast alternatywa prawdy i fałszu jest prawdą, itd.

2.3.1 Operatory koniunkcji

W języku C# są dwa rodzaje operatora koniunkcji oraz dwa rodzaje operatora alternatywy²². W przypadku koniunkcji jest to koniunkcja warunkowa „&&” oraz koniunkcja „&” (bezw warunkowa). Na przykładach omówimy różnice między nimi.

²² W Pascalu operatorem koniunkcji jest „AND”, natomiast operatorem alternatywy „OR”.

Przykład 2.9.

Załóżmy, że warunkiem pójścia do kina na dany film jest wiek minimum 18 lat oraz posiadanie pieniędzy na bilet (20 zł). Napišemy program, który wyświetla *true*, jeśli spełniony jest warunek pójścia do kina lub *false* w przeciwnym przypadku.

```
static void Main(string[] args)
{
    double wiek, PLN;
    bool kino;
    Console.WriteLine("Podaj wiek: ");
    wiek = double.Parse(Console.ReadLine());
    Console.WriteLine("Ile masz pieniędzy: ");
    PLN = double.Parse(Console.ReadLine());
    kino = (wiek >= 18 && PLN >= 20);
    Console.WriteLine(kino);
    Console.ReadKey();
}
```

Omówimy linię, w której jest obliczana wartość zmiennej logicznej *kino*. Operator „&&” jest operatorem koniunkcji warunkowej. Warunek pójścia do kina będzie spełniony, gdy wiek jest większy lub równy 18 oraz ilość pieniędzy większa lub równa 20 – wówczas zmienna *kino* przyjmie wartość *true*. W przeciwnym razie będzie miała wartość *false*. Użyte w tej linii programu okrągłe nawiasy nie są konieczne, użyto ich dla czytelności i podkreślenia, że najpierw będzie wyznaczana wartość wyrażenia logicznego, a później wartość ta zostanie przypisana do zmiennej *kino* (o zasadach łączności operatorów będzie mowa wkrótce, w [podrozdziale 2.5](#)).

Z matematyki (logiki) wiemy, że koniunkcja zdań jest prawdziwa, gdy wszystkie zdania składowe są prawdziwe. W naszym przykładzie z kinem oznacza to, że oba warunki (odnośnie wieku i pieniędzy) muszą być spełnione. **Wystarczy, że jeden z nich nie będzie spełniony, a nie ma potrzeby sprawdzać pozostałych.** Operator koniunkcji warunkowej (&&) wykorzystuje tę własność. Jego użycie powoduje, że program nie sprawdza drugiego argumentu wyrażenia logicznego w przypadku, gdy pierwszy argument składowy jest fałszywy. Ponieważ argumenty są sprawdzane (wartościowane) od lewej do prawej najpierw zostanie sprawdzony *wiek*. Jeśli jest mniejszy niż 18 drugiego argumentu koniunkcji warunkowej program już sprawdzać nie będzie – a zmienna *kino* od razu przyjmie wartość *false*.

Użycie drugiego rodzaju operatora koniunkcji „&” spowoduje, że program będzie sprawdzać oba argumenty składowe bezwarunkowo, tzn. bez względu na to, czy pierwszy z nich jest prawdziwy czy nie. W omawianym programie można zamienić linię, w której przypisywana jest wartość logiczna zmiennej *kino* na taką:

```
kino = (wiek >= 18 & PLN >= 20);
```

Po uruchomieniu nie będzie widać różnicy w działaniu obu wersji programu. Aby zobaczyć różnice w działaniu obu wersji operatora koniunkcji (warunkowej i bezwarunkowej) wykonamy inny przykład, w którym w części warunkowej wykonywana będzie (oprócz sprawdzania) jakaś „dodatkowa praca”.

Przykład 2.10.

```
static void Main(string[] args)
{
    int x = 1, y = 2;
    bool wynik;
    wynik = (x >= 2 && y++ >= 2);
    Console.WriteLine(wynik);
    Console.WriteLine(x);
    Console.WriteLine(y);
    Console.ReadKey();
}
```

Operator inkrementacji, jak pamiętamy, jest skrótem instrukcji zwiększającej wartość zmiennej ($y++$ jest równoważne z $y = y + 1$). Po uruchomieniu programu widzimy, że:

- Zmienna *wynik* przyjmuje wartość *false*, co nie budzi wątpliwości, ponieważ pierwszy warunek składowy nie jest spełniony (zmienna *x* nie jest większa lub równa 2),
- Zmienna *x* ma wartość 1. Wartość tej zmiennej nie uległa zmianie względem inicjalizacji,
- Zmienna *y* ma wartość 2. Tu może się pojawić u Czytelnika wątpliwość. Zmienna *y* także nie zmieniła wartości, tzn. ma taką, jaką jej przypisano na początku programu (2), ale w jej przypadku był użyty operator inkrementacji. Do zwiększenia wartości tej zmiennej jednak nie dochodzi, ponieważ operator koniunkcji warunkowej (&&) powoduje, że nie jest sprawdzany drugi argument logiczny, jeśli pierwszy ma wartość *false*.

Przykład 2.10 przeanalizujemy w kilku wariantach (oprócz omówionego). W każdym z wariantów zmienia się tylko jedna linia, ta, w której jest wyrażenie logiczne.

LP	Warianty przykładu 2.10	Uwagi
1	<pre>int x = 1, y = 2; bool wynik; wynik = (x >= 2 && y++ >= 2);</pre>	<p>Operator koniunkcji warunkowej (&&) ma dwa argumenty, którymi są wyrażenia relacyjne. Pierwsze nie jest spełnione, przez co drugi argument nie jest wartościowany (zmienna <i>y</i> nie zmienia swej wartości).</p> <p>Po uruchomieniu: wynik = false; x = 1; y = 2</p>
2	<pre>int x = 1, y = 2; bool wynik; wynik = (x >= 2 & y++ >= 2);</pre>	<p>Operator koniunkcji bezwarunkowej (&) ma dwa argumenty, którymi są wyrażenia relacyjne. Oba wyrażenia relacyjne są sprawdzane, mimo że pierwsze nie jest spełnione.</p> <p>Po uruchomieniu: wynik = false; x = 1; y = 3</p>

3	<pre>int x = 1, y = 2; bool wynik; wynik = (x++ >= 2 && y++ >= 2);</pre>	<p>Operator koniunkcji warunkowej (&&) ma dwa argumenty, którymi są wyrażenia relacyjne. Pierwsze nie jest spełnione (ponieważ operator inkrementacji ++ jest po nazwie zmiennej i porównywana jest „stara” wartość zmiennej x), przez co drugi argument nie jest wartościowany (zmienna y nie zmienia swej wartości).</p> <p>Po uruchomieniu: wynik = false; x = 2; y = 2</p>
4	<pre>int x = 1, y = 2; bool wynik; wynik = (++x >= 2 && y++ >= 2);</pre>	<p>Operator koniunkcji warunkowej (&&) ma dwa argumenty, którymi są wyrażenia relacyjne. Pierwsze jest spełnione (ponieważ operator inkrementacji ++ jest przed nazwą zmiennej i porównywana jest „nowa”, zwiększona wartość zmiennej x), przez co drugi argument jest wartościowany (zmienna y przyjmuje wartość 3).</p> <p>Po uruchomieniu: wynik = true; x = 2; y = 3</p>

Przeanalizowanie przykładu wraz z prezentowanymi wariantami powinno pomóc wyjaśnić działanie obu operatorów koniunkcji, ale może się nasuwać pytanie - który kiedy stosować? Stosujemy najczęściej operator koniunkcji warunkowej (&&), ponieważ dzięki niemu program będzie działał szybciej. Natomiast w przypadku, gdy w danym wyrażeniu logicznym występuje jakieś zadanie do wykonania i zależy nam na tym, aby to zadanie wykonało się bezwarunkowo – wówczas stosujemy bezwarunkowy wariant operatora koniunkcji (&).

2.3.2 Operatory alternatywy

W przypadku alternatywy jest bardzo podobnie. W języku C# są dwa operatory alternatywy – alternatywa warunkowa (||) oraz alternatywa bezwarunkowa (|). Ponieważ w przypadku alternatywy zdań wynik wyrażenia logicznego jest prawdą, gdy przynajmniej jedno ze zdań składowych jest prawdziwe – to w przypadku użycia operatora alternatywy warunkowej zaniechane jest sprawdzanie drugiego argumentu wyrażenia logicznego, jeśli pierwszy jest prawdziwy (ma wartość *true*).

Przykład 2.11.

```
static void Main(string[] args)
{
    int x = 3, y = 10;
    bool wynik;
    wynik = (x >= 2 || y++ >= 2);
    Console.WriteLine(wynik);
    Console.WriteLine(x);
    Console.WriteLine(y);
    Console.ReadKey();
}
```

Ponieważ x jest równy 3 (a zatem $x \geq 2$), całe wyrażenie jest prawdziwe. Zastosowanie operatora alternatywy warunkowej `||` powoduje, że drugi argument wyrażenia nie jest sprawdzany. Zmienna y nie zmienia się (po uruchomieniu programu widzimy, że nadal ma wartość początkową równą 10). Całe wyrażenie logiczne jest prawdziwe (zmienna `wynik` ma wartość `true`).

Zmiana instrukcji z wyrażeniem logicznym w poprzednim przykładzie na taką:

```
wynik = (x >= 2 | y++ >= 2);
```

sprawi, że zmienna y po wykonaniu wyrażenia logicznego zwiększy się (będzie równa 11). Operator alternatywy (bezwarunkowej) wymusza sprawdzenie obu argumentów wyrażenia logicznego (tu warunków składowych). Wynik wyrażenia logicznego będzie `true`, jak i w poprzednim przykładzie.

Zalecenia odnośnie stosowania operatora koniunkcji dotyczą także operatora alternatywy – stosujemy oba operatory w wersji warunkowej (`&&` oraz `||`) z wyjątkiem wyrażen, w których oba argumenty operatora logicznego muszą być wartościowane bez względu na wynik całego wyrażenia.

2.3.3 Złożone wyrażenia logiczne

Wyrażenia logiczne mogą być bardziej złożone niż w poprzednich przykładach. Mogą zawierać także operator negacji (`!`), który jest operatorem jednoargumentowym. Omówimy przykład takiego bardziej złożonego wyrażenia (w kilku wariantach).

Przykład 2.12.

Złożone rzeczy wydają się trudne, gdy nie znamy „klucza” jak je rozłożyć na elementy proste. Przypatrzmy się wyrażeniu w programie:

```
static void Main(string[] args)
{
    int x = 1, y = 2, z = 3;
    bool wynik;
    wynik = (x == 1 || y != 5 && z < 1);
    Console.WriteLine(wynik);
    Console.ReadKey();
}
```

Podobnie jak w matematyce operatory arytmetyczne i logiczne podporządkowane są pewnym regułom. O zasadach tych będzie mowa w [podrozdziale 2.5](#), ale tu warto zwrócić uwagę na ich użycie w przypadku operatorów logicznych. Reguły te związane są m.in. z priorytetami operatorów. **Priorytety operatorów** decydują o kolejności wartościowania argumentów (zob. [Tabela 3](#) w *Dodatkach*). Operator koniunkcji (`&&` oraz `&`) ma wyższy priorytet niż operator alternatywy (`||` oraz `|`). Podobnie jak w arytmetyce operator mnożenia ma wyższy priorytet niż dodawania. Arytmetyczne wyrażenie $2 + 3 * 5$ równa się 17,

ponieważ najpierw wykonane jest mnożenie. W wyrażeniu logicznym, które jest w programie najpierw będzie wartościowana koniunkcja `y != 5 && z < 1` i wartość ta wynosi *false* (ponieważ `z` nie jest mniejsze od 1). Następnie wynik koniunkcji jest uwzględniony w alternatywie z warunkiem `x == 1`, który jest prawdziwy. Alternatywa ta zatem daje wynik końcowy – *true*. Chcąc zmienić kolejność wykonywania działań należy użyć nawiasów. Wykonajmy program podmieniając linię z wyrażeniem logicznym na taką:

```
wynik = ((x == 1 || y != 5) && z < 1);
```

Wówczas wynik końcowy będzie równy *false* – koniunkcja alternatywy (mającej wynik *true*) ze zdaniem fałszywym. Ta operacja jest adekwatna do zmiany kolejności wykonywania działań w wyrażeniu arytmetycznym $(2+3) * 5$.

Jednoargumentowy operator negacji nie wymaga specjalnego komentarza. Jeśli ma dotyczyć zmiennej typu *bool*, poprzedzamy tę zmienną wykrzyknikiem, np. `!CzyZdrowy`. Jeśli ma dotyczyć wyrażenia złożonego – wówczas wyrażenie to należy umieścić w okrągłych nawiasach, np.

```
wynik = !(x == 1 || y != 5 && z < 1);
```

Zmienna `wynik` dla prezentowanego wariantu wyrażenia logicznego z omawianego przykładu będzie miała wartość *false* (negacja wartości wyrażenia w nawiasach).

2.4 Proste operacje na tekstach i znaki specjalne

W jednym z kolejnych rozdziałów zostaną omówione operacje na tekstach (w [podrozdziale 4.2](#)), ale ponieważ już w bieżących programach będą potrzebne podstawowe operacje na danych typu *string*, omówimy już teraz kilka prostych przykładów.

Przykład 2.13.

Napišemy program, który prosi użytkownika o wpisanie tekstu (imienia), a następnie wyświetla komunikat „Cześć <imię>”, gdzie <imię> to tekst, jaki wprowadził użytkownik (np. „Cześć Ewa”). Program ponadto wyświetla poniżej liczbę znaków tekstu.

```
static void Main(string[] args)
{
    string tekst;
    Console.WriteLine("Podaj imię");
    tekst = Console.ReadLine();
    tekst = "Cześć " + tekst;
    Console.WriteLine(tekst);
    Console.WriteLine(tekst.Length); // wyświetli liczbę znaków
    Console.ReadKey();
}
```

Analizę zaczniemy od końca. Program wyświetla liczbę znaków tekstu i używa w tym celu **właściwości** *Length*. Dopiero w szóstym rozdziale wyjaśnimy dokładnie, czym jest właściwość, na razie musi nam wystarczyć informacja, że zmienne (obiekty) mogą mieć właściwości określające pewne cechy (stosownie do typu). I tak, np. łańcuch znakowy (dana typu *string*) ma właściwość *Length*, która podaje długość łańcucha. Jeśli chcemy poznać długość danego łańcucha, piszemy jego nazwę, kropkę oraz *Length* (np. *tekst.Length*).

W dalszej kolejności zwróćmy uwagę na dwie linie programu, w których ustalana jest wartość zmiennej *tekst*. Metoda *ReadLine()*, której już używaliśmy w innych programach wczytuje tekst, jaki wpisze użytkownik. Ale w tym programie nie jest wykorzystana żadna z metod dokonujących konwersji typu, jak to miało miejsce w pracy z danymi typu liczbowego. Tu nie ma takiej potrzeby, metoda *ReadLine()* zwraca daną takiego typu, jaki tu został użyty dla zmiennej *tekst* (czyli *string*). W kolejnej linii, do tej samej zmiennej wpisywane jest wyrażenie *"Cześć " + tekst*. W wyrażeniu tym użyto operatora „+”, który tu nazywa się **operatorem konkatencji**, czyli łączenia. Operator ten łączy oba argumenty – „skleja” je ze sobą. W przykładzie zarówno literał „Cześć” jak i wprowadzony tekst przez użytkownika są danymi typu *string*. Ale konkatencja ma miejsce także wówczas, gdy tylko jeden z argumentów tego operatora jest typu *string*. Popatrzmy na kolejny przykład.

Przykład 2.14.

```
static void Main(string[] args)
{
    string tekst;
    Console.WriteLine("Podaj imię");
    tekst = Console.ReadLine();
    Console.WriteLine(tekst + 10);
    Console.ReadKey();
}
```

Program ten ma wyświetlić wynik wyrażenia *tekst + 10*. Mimo że jeden z argumentów jest liczbą (literał 10 jest typu *int*), to wyrażenie to nie jest arytmetyczne. Obecność argumentu typu *string* powoduje, że drugi argument (liczba 10) zostanie poddany tzw. **niejawnej konwersji** i będzie dołączony do tekstu jako łańcuch znaków „10”. Przykładowy wynik programu to „Ewa10”. Niejawna konwersja dotyczy przejścia z typów „mniejszych” do „większych” (według kryterium rozmiaru)²³ i jest możliwa dla wszystkich typów wartości do typu *string*. W tabeli zostaną omówione różne warianty operacji łączenia tekstu.

²³ Wykaz możliwych konwersji niejawnych dla typów numerycznych w tabeli na stronie <http://msdn.microsoft.com/en-us/library/y5b434w4.aspx>

LP	Przykłady użycia operatora konkatencji (oraz innych)	Uwagi
1	<pre>string tekst = "Apollo "; string wynik = tekst + "13"; Console.WriteLine(wynik);</pre>	Operator konkatencji (+) ma tu dwa argumenty typu <i>string</i> (literał „13” jest ujęty w cudzysłów). Po uruchomieniu: wynik = „Apollo 13”
2	<pre>string tekst = "Apollo "; string wynik = tekst + 13; Console.WriteLine(wynik);</pre>	Operator konkatencji (+) ma tu jeden argument typu <i>string</i> oraz jeden typu <i>int</i> (literał „13” nie jest ujęty w cudzysłów). Dla drugiego argumentu zostanie wykonana automatyczna konwersja typu z <i>int</i> do <i>string</i> (a po tej konwersji wykona się dokładnie to, co w przykładzie poprzednim). Po uruchomieniu: wynik = „Apollo 13”
3	<pre>Console.WriteLine(10 + 11);</pre>	Ponieważ oba argumenty operatora „+” są liczbami – operator ten pełni tu swoją „tradycyjną” rolę operatora arytmetycznego i wyświetli się suma obu liczb. Po uruchomieniu wyświetli się: 21
4	<pre>Console.WriteLine("10"+11);</pre>	Operator konkatencji (+) ma tu jeden argument typu <i>string</i> oraz jeden typu <i>int</i> . Dla drugiego argumentu zostanie wykonana automatyczna konwersja typu z <i>int</i> do <i>string</i> . Po uruchomieniu wyświetli się: 1011
5	<pre>char znak = 'A'; Console.WriteLine("B"+znak); // w apostrofach zapisujemy literały // znakowe (char) // w cudzysłowach literały typu string</pre>	W tym przykładzie operator „+” pełni rolę operatora konkatencji, a zmienna znak (typu <i>char</i>) ulegnie automatycznej konwersji do typu <i>string</i> . Po uruchomieniu wyświetli się: BA
6	<pre>char znak = 'A'; Console.WriteLine('B'+znak);</pre>	Jako wynik ‘B’ + ‘A’ wyświetli się liczba 131 i może to zaskoczyć niektórych. Wyjaśnienie jest następujące – typ <i>char</i> należy do typów całkowitoliczbowych. Wartość liczbową każdego znaku to liczba przypisana w tzw. tablicy kodów ASCII. Skrócona tablica ASCII umieszczona jest w <i>Dodatkach</i> (Tabela 5). Znak ‘A’ ma kod dziesiętny 65, natomiast znak ‘B’ ma kod 66. Suma ich obu to wartość, jaką wyświetla ten program. Zatem operator „+” jest tu operatorem arytmetycznym. Po uruchomieniu wyświetli się: 131
7	<pre>Console.WriteLine("Suma=" + 5+5 + " Iloczyn=" + 5*5);</pre>	Ten przykład może niektórych rozczarować, bo można by oczekiwać, że w wyrażeniu 5+5 operator „+” będzie zwyczajnym plusem (operatorem arytmetycznym). Jest jednak inaczej. Wszystkie plusy mają ten sam priorytet i są wykonywane od lewej do prawej. Najpierw zostanie dołączona 5 do tekstu „Suma=”, a do tekstu „Suma=5” dołączona zostanie druga piątka (w wyniku

		<p>automatycznych konwersji). Natomiast w przypadku operatora mnożenia nie ma niespodzianki, wynik zgodny z oczekiwaniami (operator „*” ma wyższy priorytet niż „+”).</p> <p>Po uruchomieniu wyświetli się: Suma=55 Iloczyn = 25</p>
8	<pre>Console.WriteLine("Suma=" + (5+5) + " Iloczyn=" + 5*5);</pre>	<p>Ten przykład różni się od poprzedniego tylko ujęciem w nawiasy wyrażenia 5+5. Teraz wynik powinien się wszystkim podobać :)</p> <p>Po uruchomieniu wyświetli się: Suma=10 Iloczyn = 25</p>

Na zakończenie opisu podstawowej arytmetyki na tekstach należy wskazać, że nie musi ona ograniczać się tylko do operatora konkatenacji „+”, można by także użyć innych operatorów, ale rzadko się to robi²⁴. Więcej możliwości manipulacji na tekstach dostarczają specjalne metody oraz właściwości z klasy *String*, z których w tym podrozdziale zaprezentowaliśmy tylko właściwość *Length* zwracającą długość łańcucha znaków²⁵. Bardziej zaawansowane operacje na tekstach wykonamy w [podrozdziale 4.2](#).

Podczas omawiania przykładów wspomnieliśmy, że stałe znakowe (literały typu *char*) zapisuje się w apostrofach (np. *char znak* = 'A'). Natomiast stałe tekstowe (literały typu *string*) zapisuje się w cudzysłowach (np. *string tekst* = "Słowo"). Wewnątrz stałych tekstowych mogą być umieszczone tzw. **znaki specjalne**. Znaki specjalne zaczynają się od znaku „\” (backslash), np. '\n' – to znak nowej linii, natomiast '\t' to znak tabulacji. Lista wybranych znaków specjalnych jest umieszczona w *Dodatkach* ([Tabela 7](#)). Znaki specjalne mają typ *char* (mimo że faktycznie są to dwuznaki). Literały tekstowe można poprzedzić znakiem @, wówczas ewentualne znaki specjalne są traktowane dosłownie. Poniższy przykład prezentuje użycie kilku znaków specjalnych.

Przykład 2.15.

```
static void Main(string[] args)
{
    Console.WriteLine("Linia1\nLinia2");           // przejście do nowej linii
    Console.WriteLine("Wzrost 170\tWaga 65");       // \t - tabulacja
    Console.WriteLine(@"Wzrost 170\tWaga 65");      // @ - dosłowna interpretacja
    Console.WriteLine("C:\\Windows\\Temp");        // podwójny "\\" wstawia "\"
    Console.WriteLine("Tytuł filmu: \"Rój\"");     // znak \" wstawia znak "
    Console.ReadKey();
}
```

²⁴ Wymagałoby to tzw. przeciążenia operatorów.

²⁵ Pojęcia „metoda” oraz „właściwość” zostaną wyjaśnione w dalszej części podręcznika (rozdział 5 i 6), na obecnym etapie nie ma potrzeby ich dokładnego omawiania. Każdy rozdział i podrozdział kładzie akcent na określonym zagadnieniu. Najlepiej jest kierować swoją uwagę właśnie na to zagadnienie.

Wynik programu:

```
Linia1
Linia2
Wzrost 170      Waga 65
Wzrost 170\tWaga 65
C:\Windows\Temp
Tytuł filmu: "Rój"
```

2.5 Kolejność wykonywania działań

Kwestia kolejności wykonywania działań miała znaczenie prawie w każdym z dotychczas omówionych przykładów. Tym razem przyjrzymy się jej dokładniej. **Reguły kolejności działań można tak zdefiniować:**

- jeśli operatory mają ten sam priorytet, obowiązują zasady łączności,
- jeśli operatory mają różne priorytety, obowiązuje kolejność według priorytetów operatorów.

Najpierw omówimy **zasady łączności**. Operatory przypisania oraz operatory jednoargumentowe obowiązuje zasada łączności od prawej do lewej. Natomiast operatory dwuargumentowe (za wyjątkiem przypisania) obowiązuje zasada łączności od lewej do prawej. Popatrzmy na przykład:

Przykład 2.16.

```
static void Main(string[] args)
{
    int a, b, c = 10;
    a = b = c;                // od prawej do lewej
    Console.WriteLine(a);
    Console.WriteLine(a + b + c); // od lewej do prawej
    Console.ReadKey();
}
```

W przykładzie jest instrukcja zawierająca dwa operatory przypisania $a = b = c$. Zasada łączności dla operatorów przypisania (od prawej do lewej) sprawi, że najpierw wartość zmiennej c zostanie przypisana do zmiennej b , następnie wartość zmiennej b będzie przypisana do zmiennej a . Jest to zapis równoważny z zapisem $a = (b = c)$.

Pozostałe operatory dwuargumentowe obowiązuje kolejność działań od lewej do prawej. W wyrażeniu $a + b + c$ najpierw zostanie dodana zmienna a do zmiennej b , następnie suma ta będzie dodana do zmiennej c , wyrażenie to jest równoważne z zapisem $(a+b)+c$. Na tym jednak przykładzie, zawierającym identyczne operatory, nie widać różnicy w kolejności wykonywania działań (dodawanie realizowane od drugiej strony dałoby ten sam wynik). Ta różnica uwidacznia się dopiero wówczas, gdy w wyrażeniu występują operatory o różnym priorytecie.

$$6:2(2+1) = ?$$

Jakiś czas temu na różnych stronach internetowych emocjonowano się sporami w sprawie wyniku wyrażenia arytmetycznego $6:2(2+1)$. Jednym wychodziło 9, innym 1. Oba obozy – zwolenników „dziewiątki” oraz „jedyńki” – wytaczały przeciwko sobie argumenty w postaci, mniej lub bardziej poważnie traktowanych, dowodów matematycznych. W jednym z wywodów można było nawet zobaczyć całki. My zaniechamy takiej zabawy, ale wykorzystamy ten przykład jako punkt wyjścia do omówienia kwestii priorytetów operatorów.

Owe spory prowadzono w oderwaniu od problemów programowania, którymi tu się zajmujemy. Bo jest to świat arytmetyki znanej Czytelnikom ze szkoły. W programowaniu zachowano obowiązujące w niej zasady. Wyrażenie $6:2(2+1)$ zapisane w języku programowania wygląda tak $6/2*(2+1)$. Dzielenie i mnożenie ma ten sam priorytet (w matematyce i w programowaniu). Zgodnie z zasadą kolejności działań, jeśli operatory mają ten sam priorytet obowiązują zasady łączności. Czyli dla operatorów dwuargumentowych (innych niż przypisanie) wykonuje się działania od lewej do prawej. Najpierw wykonywane będzie dzielenie $6/2$, wynik z tego dzielenia będzie pomnożony przez sumę $(2+1)$. Zwolennicy „jedyńki” usiłowali przekonać, że znak dzielenia można zamienić na kreskę ułamkową i w ten sposób podzielić 6 przez wyrażenie $2(2+1)$. Nie można jednak tak zrobić, zgodnie z zasadą kolejności działań należy wykonywać je tu od lewej do prawej. Gdy Czytelnik wpisze omawiane działanie jako kod programu `Console.WriteLine(6/2*(2+1));` będzie mógł się przekonać, czy kompilator C# należy do zwolenników „dziewiątki” czy „jedyńki” :).

Umieszczona w *Dodatkach* [Tabela 3](#) zawiera wykaz priorytetów dla wybranych operatorów w języku C#. Podsumowując przedstawione zasady wpływające na kolejność wykonywania działań – jeśli operatory mają ten sam priorytet, obowiązują zasady łączności, czyli dla operatorów przypisania oraz jednoargumentowych – od prawej do lewej. Natomiast dla operatorów dwuargumentowych (innych niż przypisania) – od lewej do prawej. W przypadku, gdy operatory mają różny priorytet wykonywane są one w kolejności wynikającej z priorytetów. Można celowo zmienić tę „naturalną” kolejność używając nawiasów (podobnie jak w arytmetyce)²⁶. Zarówno w matematyce, jak i programowaniu nawiasy stosowane są nie tylko wówczas, gdy muszą (np. w wyrażeniu $2*(2+1)$, aby wymusić dodawanie przed mnożeniem), ale także dla zwiększenia czytelności wyrażenia.

²⁶ W programowaniu nawiasy też są rodzajem operatora, posiadającym wysoki priorytet (w grupie nadrzędnej), zob. [Tabela 3 – wykaz priorytetów operatorów w Dodatkach](#).

Przykład 2.17.

```
static void Main(string[] args)
{
    int x = 1, y = 1;
    Console.WriteLine(x++ + 2 * ++y);
    Console.ReadKey();
}
```

Taki program działa poprawnie (wyświetli się liczba 5), wartość ta wyliczy się na podstawie priorytetów operatorów. Wartość 1 (czyli x sprzed zwiększenia) będzie dodana do iloczynu $2*2$ (zmienna y zostanie zwiększona przed jej użyciem). Niemniej należy przyznać, że wyrażenie takie do czytelnych nie należy i nie zaszkodzi zapisać go przy użyciu nawiasów, np. $(x++) + 2 * (++y)$.

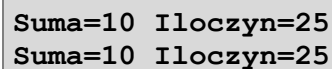
2.6 Prezentacja wyników

W kolejnym rozdziale, w którym analizować będziemy coraz bardziej złożone przykłady przyda się wygodniejszy sposób prezentacji danych. Nadal będą to znane nam metody *Console.Write()* oraz *Console.WriteLine()*, ale w innym wariantcie²⁷.

Przykład 2.18.

```
static void Main(string[] args)
{
    int x = 10, y = 25;
    Console.WriteLine("Suma=" + x + " Iloczyn=" + y);
    Console.WriteLine("Suma={0} Iloczyn={1}", x, y);
    Console.ReadKey();
}
```

W powyższym przykładzie oba wywołania metody *Console.WriteLine()* dają ten sam efekt, widoczny na rysunku:



```
Suma=10 Iloczyn=25
Suma=10 Iloczyn=25
```

Pierwsze wywołanie metody *Console.WriteLine()* powinno być zrozumiałe (bo już robiliśmy takie zadania) – metoda przyjmuje tu tylko jeden argument, jest nim *string*, który jest łączony przy użyciu operatora „+” (w kontekście danych typu *string* nazywanego operatorem konkatencji). A występujące w wyrażeniu zmienne typu *int* są automatycznie konwertowane do typu *string*.

²⁷ O metodzie, która występuje w różnych wariantach mówi się, że to jest metoda przeciążana (lub przeładowana). Przeciążone metody mają tę samą nazwę, ale różnią się listą argumentów. Ich działanie jest podobne, ale nieidentyczne. Więcej na ten temat w [podrozdziale 5.8](#).

Drugie wywołanie metody `Console.WriteLine()` zawiera 3 argumenty (oddzielone przecinkami): jeden łańcuch znakowy oraz dwie zmienne typu `int`. Zatrzymamy się na łańcuchu znakowym `"Suma={0} Iloczyn={1}"`. Zawiera on elementy formatujące `{0}` oraz `{1}`. **Element formatujący** składa się z klamrowych nawiasów, a wewnątrz – z indeksu oraz opcjonalnie z innych parametrów definiujących sposób wyświetlania danych (o czym będzie później). Indeks wskazuje numer argumentu, który ma być wstawiony w miejsce elementu formatującego. Argumenty numerowane są od zera. Ponieważ w omawianym przykładzie mamy dwa argumenty do wyświetlenia (zmienne `x` i `y`) elementy formatujące mają odpowiednio numery 0 i 1. Zatem w miejsce „`{0}`” wyświetli się wartość zmiennej `x`, a w miejsce „`{1}`” wyświetli się wartość zmiennej `y`. Gdyby w programie było więcej zmiennych do wyświetlenia, np. trzy, wówczas wywołanie metody mogłoby wyglądać tak:

```
Console.WriteLine("Suma={0} Iloczyn={1} Różnica={2}", x, y, z);
```

Jak już wspomniano, element formatujący może zawierać opcjonalnie jeszcze inne parametry decydujące o sposobie wyświetlania danych. Ogólnie można by zapisać element formatujący jako: *{indeks, wyrównanie : specyfikator formatu}*. Np. `{0,3:P}` oznacza, że argument o indeksie zero ma być wyrównany do prawej na 3 znakach oraz, że ma być wyświetlony w postaci procentowej. Parametr *wyrównanie* jest liczbą całkowitą określającą minimalną długość tekstu. Jeśli liczba ta jest dodatnia, to tekst zostanie wyrównany do prawej strony, jeśli ujemna, to do lewej. *Specyfikator formatu* zawiera kody formatowania, których jest dużo, w początkowym etapie przydatnych będzie tylko kilka z nich – zostaną przedstawione w kolejnym przykładzie.

Przykład 2.19.

```
static void Main(string[] args)
{
    int suma = 10;
    double x = 12.345678, y = 24.56, wskaznik = 0.45;
    string imie = "Ala", naz = "Nowak";
    Console.WriteLine("1. Imię: {0,15} Nazwisko: {1,20}", imie, naz);
    Console.WriteLine("2. Imię: {0,-15} Nazwisko: {1,-20}", imie, naz);
    Console.WriteLine("3. Wartość {0, 10} to suma", suma);
    Console.WriteLine("4. Wartość {0,-10} to suma", suma);
    Console.WriteLine("5. Wskaźnik: {0,8:P}", wskaznik);
    Console.WriteLine("6. Waga: {0,10} kg", x);
    Console.WriteLine("7. {0,-10:F4} (4 cyfry po przecinku)", x);
    Console.WriteLine("8. Wartość x={0,-10:F4} y={1,-10:F2}", x, y);
    Console.WriteLine("9. A ta liczba będzie w klamrach {{{0}}}", y);
    Console.ReadKey();
}
```

Na początku każdej linii przeznaczonej do wyświetlenia umieszczono kolejny numer w celu ułatwienia analizy programu. Przykład ten najlepiej jest uruchomić i porównywać odpowiednie linie kodu programu i wyświetlane przez dany kod linie na ekranie:

```

1. Imię:           Ala Nazwisko:           Nowak
2. Imię: Ala       Nazwisko: Nowak
3. Wartość        10 to suma
4. Wartość 10     to suma
5. Wskaźnik:      45,00%
6. Waga: 12,345678 kg
7. 12,3457       (4 cyfry po przecinku)
8. Wartość x=12,3457 y=24,56
9. A ta liczba będzie w klamrach {24,56}

```

Po kolei zostaną omówione linie programu:

```

Console.WriteLine("1. Imię: {0,15} Nazwisko: {1,20}", imię, naz);
Console.WriteLine("2. Imię: {0,-15} Nazwisko: {1,-20}", imię, naz);

```

W linii 1 wartość zmiennej *imię* zostanie wyświetlona na 15 znakach, natomiast wartość zmiennej *naz* na 20. Obie te dane będą równane do prawej. Natomiast w linii 2 te same wartości zostaną równane do lewej.

```

Console.WriteLine("3. Wartość {0, 10} to suma", suma);
Console.WriteLine("4. Wartość {0,-10} to suma", suma);

```

W linii 3 i 4 wartość zmiennej *suma* (typu *int*) zostanie wyświetlona na 10 znakach, przy czym w linii 3 liczba będzie równana do prawej, a w linii 4 do lewej.

```

Console.WriteLine("5. Wskaźnik: {0,8:P}", wskaznik);

```

W linii 5 zostanie wyświetlona zmienna *wskaznik*, która ma przypisaną wartość 0,45. Wartość ta zostanie wyświetlona na 8 znakach, równana do prawej i w formacie procentowym (wyświetli się 45,00%).

```

Console.WriteLine("6. Waga: {0,10} kg", x);

```

W linii 6 wyświetlona jest wartość zmiennej *x*, która wynosi 12,345678 na 10 znakach, wyrównanie do prawej. Element formatujący nie zawiera specyfikatora formatu. Wyświetlą się wszystkie cyfry po przecinku.

```

Console.WriteLine("7. {0,-10:F4} (4 cyfry po przecinku)", x);

```

W linii 7 wartość zmiennej *x* wyświetli się na 10 znakach, równana do lewej. Tym razem użyto specyfikatora formatującego „F4”, co oznacza, że wyświetlą się 4 miejsca po przecinku.

```

Console.WriteLine("8. Wartość x={0,-10:F4} y={1,-10:F2}", x, y);

```

W linii 8 wartość zmiennej *x* wyświetli się na 10 znakach, równana do lewej, z 4 miejscami po przecinku. Obok wyświetli się wartość zmiennej *y* na 10 znakach, równana do lewej z dwoma miejscami po przecinku.

```

Console.WriteLine("9. A ta liczba będzie w klamrach {{{0}}}", y);

```

W linii 9 pokazujemy przypadek, gdy wyświetlana dana ma być umieszczona w klamrach. Wówczas element formatujący musi zawierać potrójną klamrę, np. {{{0}}}.

W *Dodatkach* [Tabela 6](#) zawiera wykaz podstawowych specyfikatorów formatowania dla wartości liczbowych.

Wszystkie opisane sposoby użycia metody *Console.WriteLine()* dotyczą także metody *Console.Write()*, która wyświetla dane bez przejścia do nowej linii. Ciąg instrukcji:

```
Console.Write("Suma {0} ", 10);
Console.Write("Iloczyn {0}", 25);
```

wyświetli obie dane w jednej linii:

```
Suma=10 Iloczyn=25
```

2.7 Zadania do samodzielnego rozwiązania

Na końcu każdego z rozdziałów znajdują się zadania do samodzielnego rozwiązania. Dla wygody czytelników, zwłaszcza tych młodszych, wymagane wzory umieszczamy na końcu treści zadania.

Zadanie 2.1.

Napisz program przeliczający temperaturę w stopniach Celsjusza na temperaturę w stopniach Fahrenheita. Program ma prosić użytkownika o podanie temperatury w stopniach Celsjusza. Wzór: $F = 32 + \frac{9}{5}C$.

Zadanie 2.2.

Napisz program, który oblicza deltę dla równania kwadratowego $ax^2 + bx + c = 0$. Program ma prosić użytkownika o podanie współczynników równania a, b oraz c. Wzór: $\Delta = b^2 - 4ac$.

Zadanie 2.3.

Napisz program, który oblicza wskaźnik masy ciała BMI. Program ma prosić użytkownika o podanie wagi w kilogramach oraz wzrostu w metrach. Wzór: $BMI = \frac{masa}{wzrost^2}$.

Zadanie 2.4.

Po wykonaniu poniższych linii programu:

```
int x = 100;
Console.WriteLine(++x * 2);
```

- a) wyświetli się liczba 202 b) wyświetli się liczba 200
- c) będzie błąd d) wyświetli się liczba 201.

Najpierw oblicz wyświetlaną wartość i wybierz jedną z odpowiedzi (a, b, c, d), a dopiero później sprawdź wynik uruchamiając program.

Zadanie 2.5.

Po wykonaniu poniższych linii programu:

```
int x = 2, y = 3;
x *= y * 2;
```

- a) zmienna x=6 b) zmienna x=18
c) zmienna x=24 d) zmienna x=12.

Najpierw oblicz wartość zmiennej i wybierz jedną z odpowiedzi (a, b, c, d), a dopiero później sprawdź wynik uruchamiając program.

Zadanie 2.6.

Jaką wartość będzie miała zmienna x po wykonaniu poniższych instrukcji? Najpierw oblicz wartości zmiennych, a dopiero później sprawdź wynik uruchamiając program.

```
int x, y = 4;
x = (y -= 2);
x = y++;
x = y--;
```

Zadanie 2.7.

Co wyświetli się na ekranie po wykonaniu poniższych instrukcji. Najpierw oblicz wartości zmiennych, a dopiero później sprawdź wynik uruchamiając program.

```
int x, y = 5;
x = ++y * 2;
x = y++;
x = y--;
Console.WriteLine(++y);
```

Zadanie 2.8.

Po wykonaniu poniższych linii programu:

```
bool x;
int y = 1, z = 1;
x = (y == 1 && z++ == 1);
```

zmienne przyjmą wartości:

- a) x=true, y=1, z=2 b) x=1, y=1, z=2
c) x=true, y=1, z=1 d) x=2, y=1, z=2.

Najpierw oblicz wartości zmiennych i wybierz jedną z odpowiedzi (a, b, c, d), a dopiero później sprawdź wynik uruchamiając program.

Zadanie 2.9.

Jaką wartość przyjmą zmienne użyte w programie po wykonaniu poniższych instrukcji? Najpierw ustal wartości zmiennych, a dopiero później sprawdź wynik uruchamiając program.

a)

```
int x = 1, y = 4, z = 2;
bool wynik = (x++ > 1 && y++ == 4 && z-- > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

b)

```
int x = 1, y = 4, z = 2;
bool wynik = (x++ > 1 & y++ == 4 && z-- > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

c)

```
int x = 1, y = 4, z = 2;
bool wynik = (x++ > 1 & y++ == 4 & z-- > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

d)

```
int x = 1, y = 3, z = 4;
bool wynik = (x == 1 || y++ > 2 || ++z > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

e)

```
int x = 1, y = 3, z = 4;
bool wynik = (x == 1 | y++ > 2 || ++z > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

f)

```
int x = 1, y = 3, z = 4;
bool wynik = (x == 1 | y++ > 2 | ++z > 0);
Console.WriteLine("wynik={0} x={1} y={2} z={3}", wynik, x, y, z);
```

Zadanie 2.10.

Po wykonaniu przedstawionego niżej kodu programu zmienna *gestoscZaludnienia* przyjmie wartość 0. Odpowiedz dlaczego i zmień program (w linii, gdzie jest obliczana zmienna *gestoscZaludnienia*) tak, aby wartość tej zmiennej wynosiła 0,1 (czyli 10/100):

```
int powierzchnia = 100, osoby = 10;
double gestoscZaludnienia = osoby/powierzchnia;
Console.WriteLine(gestoscZaludnienia);
```

3 Sterowanie działaniem programu

Posługiwanie się językiem C# w roli zaawansowanego „kalkulatora” już opanowaliśmy, czas zacząć bardziej „prawdziwe” programowanie. Do tej pory wszystkie przykładowe programy wykonywane były przez komputer w sposób liniowy, tzn. polecenia wykonywane były od pierwszego do ostatniego. W praktyce programy komputerowe nie są wykonywane w tak prosty sposób. Prawie w każdym programie przynajmniej raz zachodzi konieczność podjęcia decyzji, wykonania lub niewykonania pewnego fragmentu programu, sprawdzenia pewnych warunków. Dlatego w każdym języku o imperatywnych korzeniach (w tym również w języku C#) występują tzw. instrukcje rozgałęziające, nazywane również **instrukcjami warunkowymi**. Od omówienia instrukcji warunkowych rozpoczniemy ten rozdział.

Ponadto pewne fragmenty programu powinny być wykonywane cyklicznie, czyli powtarzać się określoną liczbę razy lub aż do chwili osiągnięcia jakiegoś stanu. W pierwszym rozdziale porównaliśmy algorytm do przepisu kulinarnego. Czasami można w przepisach napotkać sformułowania, których nie lubią mało wprawni adepci sztuki kulinarnej, w rodzaju „ukręcaj aż do uzyskania konsystencji śmietany”. Zdecydowanie bardziej woleliby dokładne wskazanie typu „zamieszaj 50 razy”. No cóż, nie zawsze jest możliwe określenie liczby cykli dla operacji powtarzalnych. W programowaniu jest podobnie – dlatego udostępniono kilka rodzajów **instrukcji cyklicznych** (inaczej **pętli**), aby programiście było wygodniej zaimplementować dany „przepis” (tzn. algorytm). Po omówieniu instrukcji warunkowych przeanalizujemy trzy rodzaje pętli.

3.1 Instrukcje warunkowe

Instrukcje warunkowe pozwalają na wykonanie określonych fragmentów programu w zależności od spełnionych warunków, czyli umożliwiają rozgałęzienie liniowego programu. Aby zobrazować zachowanie się programu komputerowego możemy sobie wyobrazić spacer ścieżką w parku. Idąc parkową, prostą alejką dochodzimy do rozstaju ścieżek. Możemy kontynuować spacer tylko jedną, wybraną ścieżką. Musimy więc dokonać wyboru, którądy iść. Wybierając dalszy kierunek spaceru możemy sugerować się szerokością ścieżek, ich ukształtowaniem, nawierzchnią itp. Liczba kryteriów jest praktycznie nieograniczona. Podczas wyboru ścieżki możemy zadać pytanie, np. czy ścieżka w prawo jest widokowo bardziej atrakcyjna niż ścieżka w lewo, albo czy dana ścieżka jest stroma. Są to pytania, na które można udzielić odpowiedzi twierdzącej (tak) lub przeczącej (nie). Układając pytania, jakie mają decydować o rozgałęzieniu programu musimy używać wyrażeń logicznych (poznanych w poprzednim rozdziale), które zwracają wynik typu *bool* – prawda lub fałsz.

3.1.1 Instrukcja warunkowa *if*

Instrukcja *if* występuje w praktycznie każdym języku programowania (opartym na paradygmacie imperatywnym), chociaż sposób jej zapisu może się nieco różnić. Instrukcja ta posiada kilka wariantów. W języku C# najprostsza składnia wygląda tak:

```
if (wyrażenie logiczne)
    polecenie;
```

Składnia 3.1

Jeżeli zamiast jednego polecenia ma być wykonana większa ich liczba, wówczas konieczne jest użycie nawiasów klamrowych do oznaczenia **bloku kodu**, czyli grupy instrukcji:

```
if (wyrażenie logiczne)
{
    blok kodu;
}
```

Składnia 3.2

Sposób działania instrukcji *if* wydaje się bardzo prosty, niemniej należy pamiętać o kilku rzeczach, które ułatwią tworzenie poprawnego kodu programu, a przede wszystkim skrócą czas poszukiwania ewentualnych błędów.

Po słowie kluczowym *if* musi wystąpić **wyrażenie logiczne**, które należy umieścić w nawiasach okrągłych. O wyrażeniach logicznych pisaliśmy w [podrozdziale 2.3](#). Tu jedynie przypominamy, że wyrażenie logiczne ma wartość *true* lub *false* (prawda lub fałsz). Prostem wyrażeniem logicznym może być zmienna typu *bool*, ale może też nim być wyrażenie relacyjne (np. $x > 10$) lub metoda, która zwraca wartość typu *bool* (metody poznamy bliżej w [rozdziale 5](#)). Ponadto przypominamy o operatorze używanym w wyrażeniach relacyjnych, służącym do porównywania „==”, ponieważ na etapie tworzenia pierwszych programów z użyciem instrukcji *if* bywa często mylony z operatorem przypisania „=”.

Uwaga!

Instrukcja *if* (podobnie jak wszystkie pozostałe instrukcje sterujące, a także metody, klasy i inne konstrukcje, o których powiemy później) - może obejmować jedno lub więcej poleceń i polecenia te powinniśmy umieszczać w klamrach z tzw. **wcięciem** w prawo (znak tabulacji). Wcięcia są bardzo ważnym elementem programowania, mimo że są ignorowane przez kompilator. Stosujemy je dla zwiększenia czytelności kodu, a tym samym ograniczenia sytuacji błędnych²⁸.

²⁸ W środowisku programistycznym MS Visual Studio jest możliwość automatycznego stosowania wcięć w trakcie pisania kodu. Można także wymusić wyrównanie wcięć dla całego dokumentu (lub wskazanego fragmentu) – opcja *Edit / Advanced / Format Document*.

Przykład 3.1.

```
static void Main(string[] args)
{
    int a, b;
    a = 3;
    b = 5;
    if (a < b)
        Console.WriteLine("{0} jest mniejsze od {1}", a, b);
    Console.ReadKey();
}
```

W powyższym programie deklarowane są zmienne typu całkowitoliczbowego (*int*) o nazwach *a* i *b*, a następnie zmiennym tym przypisywane są wartości, odpowiednio 3 i 5. Instrukcja warunkowa *if* sprawdza, czy *a* jest mniejsze od *b*, czyli w tym konkretnym przypadku czy 3 jest mniejsze od 5. Jeżeli wynikiem wyrażenia jest wartość *true* (prawda) wówczas wykonywane jest kolejne polecenie, czyli wypisywany jest komunikat na ekranie konsoli.

Co jednak się stanie, gdy wyrażenie logicznie nie zwróci prawdy, czyli zwróci fałsz? Na przykład, gdy zmiennej *a* zostanie przypisana wartość 5, a zmiennej *b* wartość 3.

Przykład 3.2.

```
static void Main(string[] args)
{
    int a, b;
    a = 5;
    b = 3;
    if (a < b)
        Console.WriteLine("{0} jest mniejsze od {1}", a, b);
    Console.ReadKey();
}
```

W tym przykładzie wyrażenie logiczne zwraca wartość *false* (fałsz) i tym samym kolejne polecenie nie wykona się (komunikat nie wyświetli się).

Uwaga! W instrukcji *if* nie umieszcza się średnika po wyrażeniu logicznym.

Spójrzmy na fragment programu, w którym jest średnik po wyrażeniu logicznym:

```
int x = 3;
if (x > 5);           // Uwaga! W tej linii nie powinno być średnika!
    Console.WriteLine("{0} jest większe od 5", x);
```

Powyższy kod nie skończy się błędem, ale możemy być zawiedzeni komunikatem głoszącym, że „3 jest większe od 5”. A taki komunikat się pojawi, ponieważ jest średnik tuż

po nawiasie zamykającym wyrażenie logiczne w instrukcji *if*. Wówczas *if* obejmie tylko instrukcję „pustą” (sam średnik), a to co jest w następnej linii (tu komunikat) wykona się bez względu na wartość wyrażenia logicznego.

Jeżeli instrukcja *if* obejmuje pojedyncze i krótkie polecenie, bywa zapisywana w całości w jednej linii, np.: `if (x > 5) y++;`

3.1.2 Instrukcja warunkowa *if..else*

W programach komputerowych występuje także potrzeba przygotowania dwóch oddzielnych poleceń – jednego, które ma być wykonane w przypadku, gdy wyrażenie logiczne zwróci wartość *true* oraz drugiego – wykonywanego w przeciwnym przypadku. Do tego celu można użyć instrukcji warunkowej *if* w wariantcie rozszerzonym *if..else*.

Składnia 3.3

```
if (wyrażenie logiczne)
{
    blok kodu 1;
}
else
{
    blok kodu 2;
}
```

Słowo kluczowe *else* można tutaj tłumaczyć jako „w przeciwnym wypadku”. Jeśli wyrażenie logiczne jest spełnione, to wykonaj (*blok kodu 1*), a w przeciwnym wypadku wykonaj (*blok kodu 2*). W przypadku, gdy dany blok ma zawierać pojedynczą instrukcję można nie używać nawiasów klamrowych. Niemniej dla estetyki, a także zmniejszenia szansy popełnienia błędu zaleca się zawsze stosować klamry, nawet gdy jest w bloku tylko jedna instrukcja. Na łamach tego podręcznika niestety nie stosujemy się do tego zalecenia. Powodem jest głównie chęć zmniejszenia objętości kodu (co jest cenne przy objaśnianiu), Czytelnika jednak zachęamy do tego, aby je stosował w swoich programach.

Popatrzmy na przykład:

Przykład 3.3.

```
static void Main(string[] args)
{
    int a = 3, b = 3;
    if (a < b)
        Console.WriteLine("{0} jest mniejsze od {1}", a, b);
    else
        Console.WriteLine("{0} nie jest mniejsze od {1}", a, b);
    Console.ReadKey();
}
```

Można zadać pytanie: co w sytuacji, gdy wartości zmiennych a i b będą sobie równe? Fragment programu przedstawiony w przykładzie 3.3 zadziała prawidłowo: wyrażenie logiczne zwróci wartość *false* (bo nie jest prawdą, że np. 3 jest mniejsze od 3), tym samym na ekranie pojawi się informacja: „3 nie jest mniejsze od 3”. Tu nie powinno być żadnych wątpliwości, niemniej zdarza się, że podczas pisania pierwszych programów niektórym umyka oczywisty fakt, że jeżeli a nie jest mniejsze od b , to nie oznacza, że a jest większe od b .

Korzystając z zasad logiki klasycznej można zapisywać skomplikowane wyrażenia logiczne. Operatory logiczne zostały omówione w [podrozdziale 2.3](#). Przedstawiliśmy tam operator koniunkcji warunkowej `&&` oraz operator alternatywy warunkowej `||`. Rozbudowując wyrażenie logiczne należy uwzględniać kolejność wykonywania poszczególnych operacji (priorytety operatorów). Bezpiecznie jest umieszczać poszczególne składowe złożonego wyrażenia logicznego w nawiasach. Konieczne jest także uwzględnianie [wyników operacji logicznych](#), które przypomnieliśmy w podrozdziale 2.3.

Przykład 3.4.

```
static void Main(string[] args)
{
    int a = 1;
    if ((a == 1) || (a == -1))
        Console.WriteLine("Wartość bezwzględna równa 1");
    else
        Console.WriteLine("Wartość bezwzględna różna od 1");
    Console.ReadKey();
}
```

Wyrażenie logiczne w przykładzie 3.4 jest czytelne (alternatywa dwóch wyrażeń relacyjnych). Popatrzmy na kolejny program, który jest zmodyfikowaną wersją [przykładu 2.12](#) omówionego w poprzednim rozdziale. Nie ma tu żadnych nawiasów (oprócz pary nawiasów obowiązkowych dla instrukcji *if*) i bez znajomości priorytetów operatorów trudno byłoby to wyrażenie poprawnie zinterpretować.

Przykład 3.5.

```
static void Main(string[] args)
{
    int x = 1, y = 2, z = 3;
    if (x == 1 || y != 5 && z < 1)
        Console.WriteLine("Warunek spełniony");
    else
        Console.WriteLine("Warunek niespełniony");
    Console.ReadKey();
}
```


Ponieważ operator koniunkcji ma wyższy priorytet niż operator alternatywy, to wyrażenie to będzie wartościowane zgodnie z zapisem:

```
x == 1 || (y != 5 && z < 1)
```

czyli najpierw będzie wartościowana koniunkcja (w omawianym przykładzie jest fałszem, ponieważ z nie jest mniejsze od 1), a następnie wartościowana będzie alternatywa zdania prawdziwego (bo x jest równe 1) z fałszywym (wynikiem koniunkcji). Ponieważ alternatywa zdania prawdziwego i fałszywego jest prawdą wyświetli się napis „Warunek spełniony”.

W wyrażeniu logicznym można użyć wyrażeń arytmetycznych lub operacji tekstowych, co pokazuje kolejny przykład, w którym zastosowano proste operacje na tekstach omówione w [podrozdziale 2.4](#).

Przykład 3.6.

```
static void Main(string[] args)
{
    string imie, nazwisko, tekst;
    Console.WriteLine("Podaj imie");
    imie = Console.ReadLine();
    Console.WriteLine("Podaj nazwisko");
    nazwisko = Console.ReadLine();
    if (imie.Length + nazwisko.Length < 30)
    {
        tekst = imie + " " + nazwisko;
        Console.WriteLine(tekst);
    }
    else
    {
        Console.WriteLine("Imię i nazwisko jest za długie!");
    }
    Console.ReadKey();
}
```

W wyrażeniu logicznym sprawdzane jest, czy suma długości łańcuchów znakowych jest mniejsza od 30. Jeśli jest mniejsza, to wykona się blok kodu zawierający dwie instrukcje, jedna łącząca imię z nazwiskiem, a druga wyświetlająca połączony tekst. Jeśli jednak warunek nie jest spełniony to wyświetli się komunikat o tym, że imię i nazwisko jest za długie.

3.1.3 Zagnieżdżanie instrukcji warunkowych

Instrukcja warunkowa *if..else* pozwala wykonać jeden z dwóch bloków poleceń. Nierzadko jednak zachodzi potrzeba rozpatrzenia większej liczby wariantów. W tym celu można instrukcję *if* zagnieżdżać, czyli umieszczać jedną instrukcję *if* w drugiej. Przy takim rozwiązaniu konieczne jest sprawdzenie warunków w liczbie o jeden mniejszej od liczby

dopuszczalnych opcji. W kolejnym przykładzie mamy do rozpatrzenia trzy warianty: zysk, strata, dochód zerowy. Dla ich rozpoznania zostaną użyte dwie instrukcje *if..else*.

Przykład 3.7.

```
public static void Main()
{
    double przychod = 1000.00, koszty = 1020.50;
    double dochod = przychod - koszty;
    if (dochod == 0)
    {
        Console.WriteLine("Mamy dochód zerowy");
    }
    else
    {
        if (dochod > 0)
            Console.WriteLine("Mamy zysk");
        else
            Console.WriteLine("Mamy stratę");
    }
    Console.ReadKey();
}
```

Korzystając z możliwości zagnieżdżania szczególnie zalecane jest korzystanie z nawiasów klamrowych, a dla czytelności kodu źródłowego należy pamiętać o stosowaniu wcięć. W [przykładzie 3.7](#) zagnieżdżanie następowało w sekcji *else* (dla przeciwnego wypadku), ale może także następować po sekcji *if*, jak w poniższym fragmencie programu:

```
if (dochod != 0 )
{
    if (dochod > 0)
        Console.WriteLine("Mamy zysk");
    else
        Console.WriteLine("Mamy stratę");
}
else
{
    Console.WriteLine("Mamy dochód zerowy");
}
```

Wracając do [przykładu 3.7](#) – gdy liczba możliwych opcji jest duża (występuje wiele zagnieżdżonych instrukcji *if* w sekcji *else*), stosowanie wcięć kodu źródłowego może zmniejszyć jego czytelność. Dlatego w takiej sytuacji proponuje się stosowanie wielowarunkowego wariantu instrukcji *if..else if*, w którym zapisuje się instrukcje *else* oraz *if* w jednej linii, z jednolitymi wcięciami kodu, jak w poniższym przykładzie.

Przykład 3.8.

```
static void Main(string[] args)
{
    Console.WriteLine("Wpisz nr dnia tygodnia");
    string number = Console.ReadLine();
    if (number == "1")
        Console.WriteLine("Poniedziałek");
    else if (number == "2")
        Console.WriteLine("Wtorek");
    else if (number == "3")
        Console.WriteLine("Środa");
    else if (number == "4")
        Console.WriteLine("Czwartek");
    else if (number == "5")
        Console.WriteLine("Piątek");
    else if (number == "6")
        Console.WriteLine("Sobota");
    else if (number == "7")
        Console.WriteLine("Niedziela");
    else
        Console.WriteLine("Nie ma takiego dnia tygodnia");
    Console.ReadKey();
}
```

3.1.4 Operator warunkowy

W języku C# istnieje operator, którego zasada działania jest podobna do działania instrukcji *if..else*. Jest nim operator warunkowy.

Składnia 3.4

(wyrażenie logiczne)? wyrażenie1 : wyrażenie2

Operator warunkowy pozwala na sprawdzenie wyrażenia logicznego, które zapisane jest z lewej strony operatora w nawiasach okrągłych. Jeśli spełnione jest wyrażenie logiczne, to operator warunkowy zwraca *wyrażenie1*, w przeciwnym wypadku zwraca *wyrażenie2*. Przykład użycia operatora warunkowego:

Przykład 3.9.

```
static void Main(string[] args)
{
    int y, x = 1;
    y = (x > 0) ? ++x : --x;
    Console.WriteLine(y);
    Console.ReadKey();
}
```

W przypadku, gdy wyrażenie logiczne jest spełnione (a tak jest w tym przykładzie), operator warunkowy zwróci wyrażenie będące bezpośrednio po znaku zapytania, czyli zwiększoną wartość zmiennej *x*. W przeciwnym wypadku zostanie zwrócone wyrażenie będące po znaku dwukropka, czyli zmniejszona wartość zmiennej *x*.

Umieszczamy obok siebie dwa równoważne rozwiązania, jedno z przykładu 3.9 z użyciem operatora warunkowego oraz drugie wykonane za pomocą instrukcji *if..else*.

<pre>// Operator warunkowy (jak w przykładzie 3.9) int y, x = 1; y = (x > 0) ? ++x : --x; Console.WriteLine(y);</pre>	<pre>// Instrukcja if..else int y, x = 1; if (x > 0) y = ++x; else y = --x; Console.WriteLine(y);</pre>
--	--

Rozwiązanie z operatorem warunkowym jest krótsze i to jest jego główna zaleta. Jeśli zawarte w nim wyrażenia nie są zbyt złożone – można go stosować nie tracąc na czytelności kodu.

Zasada działania operatora warunkowego przypomina instrukcję warunkową *if..else*, niemniej korzystanie z tego operatora związane jest z pewnymi ograniczeniami. Przede wszystkim, w przeciwieństwie do instrukcji *if..else*, po prawej stronie operatora warunkowego mogą znaleźć się jedynie wyrażenia, które zwracają wartość typu wbudowanego. Nie można tu użyć wywołania metod, które nic nie zwracają, takich jak *Console.WriteLine()*. Nie można też użyć bloków instrukcji. Zatem każde użycie operatora warunkowego można zmienić i zapisać w postaci składni instrukcji *if..else*, ale nie można każdej instrukcji *if..else* „przetłumaczyć” na operator warunkowy (np. nie można by zmienić instrukcji *if..else* z [przykładu 3.6](#)). Operator warunkowy jest wykorzystywany nie tylko w języku C#²⁹.

3.1.5 Instrukcja warunkowa *switch..case*

[Przykład 3.8](#) prezentuje wykorzystanie wielowarunkowego wariantu instrukcji *if..else*. To rozwiązanie pozwala sprawdzać bardziej skomplikowane wyrażenia logiczne. Jednak w sytuacji, gdy program komputerowy powinien w odpowiedni sposób reagować na proste warunki w postaci pojedynczych wartości zmiennej istnieje możliwość użycia instrukcji wielowarunkowej *switch..case*.

Początkowo składnia polecenia *switch..case* może się wydać skomplikowana. Jednak po jej przeanalizowaniu, okazuje się, że jej budowa jest przejrzysta i uporządkowana, a sposób jej użycia jest intuicyjny.

²⁹ Operator warunkowy jest wykorzystany w innych językach programowaniach (np. C++, Java), ale także w środowisku SQL Server Integration Services 2008.

```
switch (zmienna)
{
    case pierwsza możliwość:
        polecenie lub polecenia;
        break;
    case druga możliwość:
        polecenie lub polecenia;
        break;
    [default:
        polecenie lub polecenia;
        break;]
}
```

Po słowie kluczowym *switch* w nawiasach okrągłych powinna pojawić się nazwa zmiennej. Wartość tej zmiennej będzie sprawdzana w kolejnych fragmentach programu rozpoczynających się słowem kluczowym *case* (które po polsku oznacza „przypadek”). W przypadku znalezienia wartości odpowiadającej zmiennej zostanie wykonane odpowiednie polecenie lub grupa poleceń. Jeżeli sprawdzone zostaną wszystkie możliwe wartości umieszczone po słowach kluczowych *case*, a mimo tego żadna z nich nie będzie odpowiadała wartości zmiennej (zawartej po słowie *switch*), to zostanie wykonane polecenie lub blok poleceń umieszczony na końcu, po słowie *default* (blok opcjonalny).

Poniższy przykład wyświetla w języku polskim nazwę liczby (słownie), która przechowywana jest w zmiennej *liczba*.

Przykład 3.10.

```
static void Main(string[] args)
{
    Console.WriteLine("Podaj liczbę z zakresu 0-2");
    int liczba = int.Parse(Console.ReadLine());
    switch (liczba)
    {
        case 0:
            Console.WriteLine("zero");
            break;
        case 1:
            Console.WriteLine("jeden");
            break;
        case 2:
            Console.WriteLine("dwa");
            break;
        default:
            Console.WriteLine("Nieznana wartość");
            break;
    }
    Console.ReadKey();
}
```

W przykładzie zadeklarowano zmienną *liczba*, której wartość ma wpisać użytkownik. W instrukcji *switch* program sprawdza, czy wartość tej zmiennej odpowiada jednej z kolejnych wartości umieszczonych po słowach kluczowych *case*. Jeśli użytkownik wpisze przykładowo liczbę 2, to zostaną wykonane instrukcje dla tego przypadku (zawarte po poleceniu „case 2:”) – wyświetlenie tekstu „dwa” oraz polecenie *break*. W instrukcji *switch* (w języku C#) w każdym bloku *case* musi wystąpić instrukcja przekazująca sterowanie w inne miejsce programu. W tym przykładzie jest to instrukcja *break*. Po jej napotkaniu sterowanie programu przenoszone jest poza blok instrukcji *switch*.

Czasami zachodzi konieczność, aby po wykonaniu bloku kodu dla danego przypadku został wykonany dodatkowo kod dla innego przypadku (zapisany w innej instrukcji *case*). Możliwość taką daje zamiana polecenia *break* na polecenie **goto case** *wartość*, gdzie *wartość* odpowiada blokowi *case*, który powinien zostać wykonany.

Przykład 3.11.

```
static void Main(string[] args)
{
    double cena = 0.0;
    Console.WriteLine("Podaj porcję (S / M / L)");
    string porcja = Console.ReadLine();
    switch (porcja)
    {
        case "S":
            cena += 4.5;
            break;
        case "M":
            cena += 2.0;
            goto case "S";
        case "L":
            cena += 3.0;
            goto case "S";
        default:
            Console.WriteLine("Podano zły symbol");
            break;
    }
    Console.WriteLine(cena);
    Console.ReadKey();
}
```

W przykładzie 3.11 obliczana jest cena w zależności od podanej porcji (np. kawy, frytek itp.). Porcja *S* (mała) ma cenę 4,5 zł, porcja *M* (średnia) jest o 2 zł droższa od porcji *S*, natomiast porcja *L* (duża) jest o 3 zł droższa od porcji *S*.

Korzystając z możliwości, jaką daje użycie polecenia *goto case* należy zachować ostrożność. Łatwo można doprowadzić do zapętlenia instrukcji *case*. W powyższym przykładzie, gdyby instrukcję *break* (występuje tylko jeden raz) zamienić na instrukcję *goto*

case "M" lub *goto case "L"*, program sam nie zakończyłby działania, ponieważ nieustannie wykonywane byłyby polecenia zawarte w odpowiednich instrukcjach *case*.

Wykorzystując instrukcję *switch..case* należy pamiętać o kilku obowiązujących zasadach:

- Zmienna, której wartość będzie testowana (której nazwa pojawia się po słowie *switch*) nie może być dowolnego typu. Wśród dopuszczalnych typów tej zmiennej są: *bool*, *char*, *string*, *int*, *enum*. Próba sprawdzania wartości zmiennej typu *double* lub *float* zakończy się komunikatem o błędzie.
- Linia programu zawierająca słowo kluczowe *switch* i umieszczoną w nawiasie okrągłym nazwę testowanej zmiennej nie powinna kończyć się znakiem średnika.
- Nie mogą pojawić się dwie takie same wartości po słowie kluczowym *case*. W takim przypadku kompilator nie będzie wiedział, który blok kodu wykonać, dlatego poinformuje o tym fałszywym komunikatem o błędzie.
- Wartości, które pojawiają się po słowie kluczowym *case* muszą być w postaci literału lub wyrażenia używającego literałów (np. 7-2 będzie interpretowane jako 5).
- Po każdym słowie kluczowym *case* może pojawić się tylko jedna wartość (a nie lista wartości).
- Linia zawierająca słowo kluczowe *case* powinna kończyć się znakiem dwukropka.
- Polecenia, które powinny zostać wykonane w ramach jednej wartości *case* muszą zostać zakończone instrukcją *break* lub *goto case* (blok *case* może kończyć także instrukcja *return*, którą poznamy w rozdziale 5).
- Fragment programu pomiędzy słowami *case* i *break* lub *case* i *goto case* stanowi blok kodu, dlatego nie ma potrzeby stosowania w tym przypadku nawiasów klamrowych.
- Należy szczególnie uważać stosując instrukcję *goto case*.
- Blok *default* jest opcjonalny.
- Blok kodu do wykonania może być pusty (bez poleceń i bez *break*), wówczas oznacza to, że kolejna opcja (a dokładniej pierwszy niepusty blok *case*) uwzględni także powyższe „puste” przypadki, tak jak w fragmencie programu:

```
case -1:
case 0:
case 1:
    Console.WriteLine("Przypadek -1 lub 0 lub 1");
    break;
case 2:
    Console.WriteLine("Przypadek 2");
    break;
```

Mimo kilku ograniczeń, instrukcja *switch..case* bardzo ułatwia pisanie kodu w sytuacjach, gdy program powinien reagować na wiele różnych wartości zmiennej.

Streszczenie informacji na temat instrukcji warunkowych wraz z krótkimi przykładami znajduje się w [Tabeli 8](#) w *Dodatkach*.

3.2 Instrukcje cykliczne – pętle

W programie komputerowym zachodzi nierzadko konieczność wykonania wielu bardzo podobnych do siebie działań. Przykładem takich operacji może być wyświetlenie kilku kolejnych liczb naturalnych, obliczenie wartości silnia dowolnej liczby lub oczekiwanie na naciśnięcie przez użytkownika konkretnego znaku na klawiaturze.

Aby wypisać kolejne liczby naturalne (np. z przedziału od 1 do 9) można skorzystać z 9 poleceń `Console.WriteLine()`. Należy zauważyć jednak, że te polecenia będą się od siebie różnić tylko jednym elementem – wartością wyświetlanej liczby. Gdyby zaś rozszerzyć zakres wyświetlanych wartości np. do 1000, nie trzeba nikogo przekonywać, że pisanie w tym celu 1000 niemal identycznych linii jest po prostu stratą czasu i energii (zarówno programisty jak i komputera). Dlatego też w każdym języku programowania występują pętle – konstrukcje, które korzystając ze stosunkowo prostego zapisu kodu programu umożliwiają wielokrotne wykonanie określonego zbioru poleceń.

W językach programowania, w tym oczywiście również w C#, występuje kilka rodzajów pętli. Wspólną cechą większości z nich jest fakt wykorzystania wyrażeń logicznych (omówionych w [podrozdziale 2.3](#)). Przedstawione zostaną trzy rodzaje pętli. Pierwsza z nich to pętla *for*.

3.2.1 Pętla *for*

W pętli *for* występuje zmienna, która nazywana jest **licznikiem pętli** (lub także **zmienną sterującą**). Dzięki tej zmiennej istnieje możliwość określania, ile razy ma się wykonać pętla (czyli liczbę przebiegów pętli). Oprócz licznika pętli, duże znaczenie ma wyrażenie logiczne, które pozwala decydować, czy pętla się wykona czy też nie oraz polecenia wykonywane w samej pętli.

Składnia pętli *for* przedstawia się następująco:

```
for ([inicjalizacja]; [wyrażenie logiczne]; [iteracja])
{
    // ciało pętli (instrukcje)
}
```

Składnia 3.6

Algorytm pętli *for* można przedstawić w poniższy sposób:

- zainicjalizuj wartość początkową licznika pętli,
- jeżeli wyrażenie logiczne, które powinno wykorzystywać wartość licznika pętli ma wartość *true*, to wykonaj polecenia zawarte w pętli, po czym zmień wartość licznika pętli (sekcja *iteracja*). Powtarzaj ten krok, dopóki jest spełnione wyrażenie logiczne,
- jeżeli wyrażenie logiczne zwraca wartość *false* – zakończ działanie pętli.

Dalsze wyjaśnienia dotyczące składni będą oparte na przykładzie zawierającym pętlę, która wyświetla kolejne liczby naturalne od 1 do 9:

Przykład 3.12.

```
static void Main(string[] args)
{
    for (int i = 1; i < 10; i++)
    {
        Console.WriteLine(i);
    }
    Console.ReadKey();
}
```

Przyjrzymy się bliżej definicji pętli *for* z powyższego przykładu:

Składnik instrukcji <i>for</i>	Uwagi
for	Słowo kluczowe języka C#, które rozpoczyna pętlę, po tym słowie w nawiasach okrągłych mogą pojawić się kolejne trzy składowe definicji pętli oddzielone średnikami.
i = 1	Zmienna <i>i</i> jest licznikiem pętli, w omawianym przykładzie zainicjalizowano ją wartością równą 1.
i < 10	Wyrażenie logiczne, które jest sprawdzane za każdym razem przed wykonaniem poleceń pętli. Jeżeli wynik wyrażenia logicznego będzie równy wartości <i>true</i> – pętla wykona się, jeżeli <i>false</i> - nastąpi zakończenie działania pętli.
i++	Operacja inkrementacji, czyli zwiększania wartości zmiennej o 1. To samo działanie można zapisać jako <i>i = i+1</i> .

A teraz prześledzimy dokładnie sposób działania programu z przykładu 3.12:

- W pierwszym kroku inicjalizowana jest wartość licznika pętli wartością 1.
- Następnie sprawdzane jest wyrażenie logiczne. Podczas pierwszego przejścia pętli (pierwszej iteracji), gdy zmienna *i* ma wartość równą 1 wynikiem tego wyrażania jest wartość *true* dlatego: wykonywane jest polecenie pętli wyświetlające na ekranie bieżącą wartość zmiennej *i* (czyli wyświetli się liczba 1). Później zwiększana jest o jeden wartość licznika pętli. Krok ten jest powtarzany, dopóki jest spełnione wyrażenie logiczne (będzie tu tak dla *i* równego 2, 3... aż do 9).
- Gdy wartość wyrażenia logicznego przyjmuje wartość *false*, co stanie się w tym programie z chwilą, gdy zmienna *i* będzie równa 10 – wówczas pętla kończy pracę, nie wykonuje już poleceń. Zatem liczby 10 program ten już nie wypisze.

Aby zamiast kolejnych 9 liczb wyświetlić np. liczby od 1 do 1000 wystarczy drobna modyfikacja przedstawionej pętli, która ogranicza się do zmiany wyrażenia logicznego:

```
i <= 1000
```

Jak łatwo zauważyć licznik pętli można zainicjalizować dowolną wartością. Jednak dla czytelności kodu oraz w celu eliminowania prostych błędów licznik inicjalizowany jest zazwyczaj wartością 1 (lub w przypadku tablic wartością równą 0).

Pętle zadeklarowane poniżej wykonają się taką samą liczbę razy:

```
for (int i = 1 ; i <= 100 ; i++)
for (int i = 0 ; i < 100 ; i++)
for (int i = 20 ; i < 120 ; i++)
```

Licznik pętli jest zmienną, a ta podlega zasadom dotyczącym zmiennych (między innymi zmienna musi zostać zadeklarowana i może przyjmować wartości z zakresu wartości swojego typu), które zostały opisane w [podrozdziale 2.1](#). Powyższe przykłady pętli *for* deklarowały zmienną jednocześnie inicjalizując ją wartością początkową. Nierzadko jednak programiści wykorzystują w pętli zmienną już zadeklarowaną w programie. Możliwy jest zatem taki zapis:

```
int i;
for (i = 1; i <= 100; i++)
```

Omawiając działanie pętli *for* warto zwrócić uwagę na tę część pętli, która odpowiada za zmianę licznika pętli. W przedstawionych do tej pory przykładach wykorzystywany był operator inkrementacji, czyli zwiększania wartości licznika o 1. Nic nie stoi jednak na przeszkodzie, aby wykorzystywać operator dekrementacji, czyli zmniejszania wartości o 1, np.:

```
for (int i = 100 ; i > 0 ; i--)
```

W powyższej deklaracji pętli licznik pętli jest inicjalizowany wartością 100 i zmniejsza się w każdej iteracji o 1. Wyrażenie logiczne sprawdza, czy licznik pętli jest większy od zera (gdy będzie równy zero pętla skończy działanie).

Fragment pętli *for* odpowiedzialny za zmianę wartości licznika pętli można zapisać bez stosowania operatorów inkrementacji lub dekrementacji, tak jak to pokazują dwie kolejne deklaracje pętli:

```
for (int i = 1 ; i <= 100 ; i = i + 1)
for (int i = 1 ; i <= 100 ; i = i + 20)
```

Jak widać, taka możliwość pozwala na zmianę wartości licznika pętli o dowolną wartość, nie tylko o +1 lub -1. Można też używać operatorów przypisania, które stanowią skrócony zapis instrukcji arytmetycznych, np. += lub -=, jak w poniższej deklaracji pętli (zob. wykaz operatorów w [Tabeli 2](#) w *Dodatkach*):

```
for (int i = 1 ; i <= 100 ; i += 2)
```

Podobnie jak w przypadku instrukcji warunkowej *if*, również pętla *for* pozwala na zagnieżdżanie, czyli umieszczanie jednej pętli w drugiej.

Przykład 3.13.

```
static void Main(string[] args)
{
    for (int i = 1; i <= 5; i++)
        for (int j = 1; j <= 5; j++)
            Console.WriteLine("i = {0}, j = {1}", i, j);
    Console.ReadKey();
}
```

W przedstawionym przykładzie pętla wewnętrzna (wykorzystująca jako swój licznik zmienną *j*) wykonuje się 5 razy dla każdej wartości zmiennej *i* (licznika pętli zewnętrznej, który też wynosi 5). Zatem liczba przebiegów pętli wewnętrznej będzie równa $i * j$ (tu 25). Program ten wypisuje informacje o bieżących wartościach obu liczników pętli, jakie przyjmują w kolejnych przebiegach pętli wewnętrznej. Należy zwrócić uwagę na to, że pętla zewnętrzna obejmuje tu tylko jedną instrukcję (brak tu klamer) – a tą instrukcją jest kolejna pętla *for*. Ta druga (wewnętrzna) pętla *for* także obejmuje tylko jedną instrukcję – polecenie *Console.WriteLine()*.

Poniżej fragment programu, w którym pętla zewnętrzna obejmuje dwie instrukcje:

```
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
        Console.Write(j);
    Console.WriteLine();
}
```

Tu pętla zewnętrzna *for* obejmuje pętlę wewnętrzną oraz polecenie przejścia do nowej linii (taki efekt przynosi wywołanie metody *Console.WriteLine()* bez argumentów). Po uruchomieniu programu zawierającego powyższe pętle, w pięciu kolejnych liniach zostaną wypisane ciągi cyfr 12345. Wyświetleniem każdego ciągu cyfr zajmuje się pętla wewnętrzna. Cyfry tworzą ciąg (są w jednym wierszu), ponieważ w wewnętrznej pętli *for* użyta została metoda *Console.Write()*, która nie powoduje przejścia do nowej linii. Dopiero po wyświetleniu każdego wiersza, już w zasięgu pętli zewnętrznej, wykonywane jest przejście do nowej linii.

Zagnieżdżając pętle należy szczególną uwagę zwrócić na ich liczniki (zmienne sterujące), tak aby każda z pętli operowała swoim własnym licznikiem. Zmienna będąca licznikiem pętli nie powinna pełnić tej roli w obu pętlach – zewnętrznej i wewnętrznej (kompilator zezwoliłby na to w przypadku deklaracji zmiennej sterującej przed deklaracją obu pętli). Zaleca się również, w celu ograniczenia możliwości pomyłek, aby nazwy obu liczników nie były zbyt podobne.

Nietypowe przykłady użycia pętli *for*

Prezentowane wyżej przykłady, wraz z omówionymi w komentarzach wariantami, dotyczą typowego użycia pętli *for*. Chcielibyśmy po krótkce pokazać także te mniej typowe przykłady, które są rzadziej stosowane, ale w określonych sytuacjach mogą się okazać przydatne. Te specyficzne sytuacje nie wystąpią jednak w najbliższym okresie nauki, dlatego podczas pierwszego czytania podręcznika zalecamy, aby przykłady 3.14 – 3.17 przejrzeć bez wnikliwej analizy. Można także ominąć opis tych przykładów i przejść od razu do [opisu błędów związanych z wykorzystaniem pętli *for*](#).

Elementy pętli *for*, które występują w nawiasach okrągłych (inicjalizacja licznika, warunek logiczny, zmiana wartości licznika) nie muszą konieczne być zapisywane dokładnie w tym miejscu. Możliwe jest np. wcześniejsze inicjalizowanie licznika pętli, należy jednak pamiętać o zachowaniu odpowiedniej liczby średników występujących w nawiasach, tak aby kompilator nie miał wątpliwości jak interpretować zapis pętli *for*.

Przykład przedstawiający możliwość wcześniejszego inicjalizowania licznika pętli:

Przykład 3.14.

```
static void Main(string[] args)
{
    int i = 0;
    for (; i < 10; i++)
        Console.WriteLine(i);
    Console.ReadKey();
}
```

Przykład, w którym operacja zmiany wartości licznika pętli została przeniesiona do ciała pętli:

Przykład 3.15.

```
static void Main(string[] args)
{
    for (int i = 0; i < 10; )
    {
        Console.WriteLine(i);
        i++;
    }
    Console.ReadKey();
}
```

Kolejny przykład łączy oba te rozwiązania:

Przykład 3.16.

```
static void Main(string[] args)
{
    int i = 0;
```

```

for (; i < 10; )
{
    Console.WriteLine(i);
    i++;
}
Console.ReadKey();
}

```

Wszystkie przedstawione do tej pory przykłady pętli *for* wykorzystywały jako licznik pętli zmienną typu całkowitoliczbowego (*int*). Nic nie stoi jednak na przeszkodzie, aby licznik pętli był innego typu liczbowego, np.:

```

for (decimal i = 0.1m; i <= 1; i = i + 0.1m)
    Console.WriteLine(i);

```

Przypominamy, że „0.1m” to literal z modyfikatorem „m”, czyli typu *decimal* (modyfikatory typu dla literalów opisano w [podrozdziale 2.2.2](#)).

Pętla *for* może wykorzystywać więcej niż jedną zmienną. Takie rozwiązanie zawiera kolejny program:

Przykład 3.17.

```

static void Main(string[] args)
{
    int i, j, k;
    for (i = 0, j = -10, k = 1; i > (j + k); i--, j++, k++)
    {
        Console.WriteLine(" i = {0} j = {1} k = {2}", i, j, k);
    }
    Console.ReadKey();
}

```

Wynik działania programu:

```

i = 0   j = -10 k = 1
i = -1  j = -9  k = 2
i = -2  j = -8  k = 3

```

Z powyższego przykładu wynika, że możliwe jest inicjalizowanie oraz zmiana wartości wielu zmiennych (w tym przypadku trudno nazywać te zmienne licznikami pętli). Jednak wyrażenie logiczne będące warunkiem działania pętli musi być tylko jedno. W praktyce programowania rzadko wykorzystuje się tę możliwość pętli *for*, niemniej jednak warto pamiętać, że istnieje.

Częste błędy związane z wykorzystaniem pętli *for*

Należy zwrócić uwagę, że linia programu rozpoczynająca się słowem kluczowym *for* (podobnie jak w przypadku instrukcji *if*) nie kończy się znakiem średnika. Pojawienie się średnika na końcu tej linii nie jest błędem składniowym (kompilator nie poinformuje o

błędzie podczas uruchamiania programu). Średnik potraktowany zostanie jako instrukcja pusta, w konsekwencji wielokrotnie wykonywać się będzie właśnie ta pusta instrukcja, zaś polecenie, które w zamiarze programisty powinno pojawić się w pętli, zostanie wykonane jednokrotnie po jej zakończeniu.

Przykład 3.18.

```
static void Main(string[] args)
{
    int i;
    for (i = 0; i < 10; i++);
        Console.WriteLine(i);
    Console.ReadKey();
}
```

Jak pokazuje powyższy przykład, pojawienie się łatwego do przeoczenia znaku średnika tuż za nawiasem zamykającym definicję pętli *for* istotnie zmienia sposób działania programu. W powyższym przykładzie zamiast wyświetlenia na ekranie kolejnych liczb od 0 do 9, zostanie wyświetlona jedynie wartość 10. Tego rodzaju błędy są bardzo uciążliwe, ponieważ nie są komunikowane przez kompilator (składnia jest poprawna), a objawiają się błędnym działaniem programu. Gdyby jednak zmienna *i* będąca licznikiem pętli została zadeklarowana bezpośrednio w pętli (a nie jak w przedstawionym przykładzie przed pętlą *for*), kompilator zasygnalizowałby błąd, ponieważ zmienna *i* dostępna byłaby jedynie w ramach pętli. Linia programu wyświetlająca bieżącą wartość zmiennej *i* jest poza pętlą³⁰, dlatego kompilator nie wiedziałby, czym tak naprawdę jest *i* umieszczone jako argument polecenia *Console.WriteLine()*.

W początkowej nauce programowania z wykorzystaniem pętli *for* należy szczególną uwagę zwrócić na warunki brzegowe działania pętli, a więc na to, kiedy pętla się rozpoczyna i kiedy się zakończy, a w konsekwencji ile razy zostanie wykonana. Bardzo często błędy popełniane w tym miejscu powodują, że pętla wykonuje się o jeden raz za mało lub za dużo. Nierzadko przyczyną może być użycie operatora „<=” w sytuacji, gdy powinien być użyty „<” lub odwrotnie (bądź pomylenie operatorów „>=” i „>”).

Ważną zasadą, o której należy pamiętać wykorzystując pętlę *for*, jest unikanie modyfikowania wartości licznika pętli w środku pętli (oczywiście poza sytuacją przedstawioną w [przykładach 3.15 i 3.16](#)). Konstrukcja pętli *for* umożliwia sterowanie wartością licznika pętli w sposób automatyczny i ingerencja programisty w czasie wykonywania pętli w wartość licznika jest niewskazana. Zmiana wartości licznika może skutkować błędną liczbą iteracji pętli lub co gorsza zjawiskiem „zapętlenia się” programu czyli działania pętli nieskończoną liczbę razy.

³⁰ Przypominamy, jeśli tuż po deklaracji pętli (lub instrukcji *if*) nie ma bloku w klamrach, to pętla obejmuje tylko jedną instrukcję (tę najbliższą). Skoro w omawianym przykładzie objęła instrukcję pustą (średnik), to kolejna instrukcja *Console.WriteLine()* jest już poza pętlą. I tak jest, mimo że dla czytelności kodu zrobiliśmy odpowiednie wcięcia (wcięcia nie są interpretowane przez kompilator).

Przykład 3.19.

```
static void Main(string[] args)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i--);
    }
}
```

W powyższym przykładzie po wyświetleniu wartości zmiennej *i* następuje zmniejszenie jej wartości. W efekcie licznik pętli podczas sprawdzania wartości wyrażenia logicznego nigdy nie będzie przyjmować wartości większej od 0, co doprowadzi do „nieskończonego” działania pętli.

3.2.2 Pętle *while* i *do..while*

Dokładne zrozumienie działania pętli *for* ułatwi zapoznanie się z zasadami działania kolejnych pętli. Jedną z nich jest pętla *while*.

Zapis pętli *while* jest prostszy od pętli *for*, chociaż występują w niej podobne elementy.

```
while (wyrażenie logiczne)
{
    // ciało pętli (instrukcje)
}
```

Składnia 3.7

Algorytm pętli *while* można przedstawić w następujący sposób:

- jeżeli wyrażenie logiczne ma wartość *true*, to wykonaj instrukcje zawarte w pętli. Powtarzaj ten krok, dopóki jest spełnione wyrażenie logiczne,
- jeżeli wyrażenie logiczne zwraca wartość *false* – zakończ działanie pętli.

Przykład wyświetlający kolejne liczby z przedziału od 0 do 9, tym razem w wersji z użyciem pętli *while*:

Przykład 3.20.

```
static void Main(string[] args)
{
    int i = 0;
    while (i < 10)
    {
        Console.WriteLine(i);
        i++;
    }
    Console.ReadKey();
}
```

Analizując działanie pętli *while* można wskazać wspólne elementy z pętlą *for* jednak umieszczone są one w innych miejscach. W pętli *while* na ogół także występuje zmienna³¹, która pojawiając się w wyrażeniu logicznym decyduje o tym, czy pętla zostanie wykonana. Zmiennej tej jednak nie będziemy nazywali licznikiem pętli.

Pierwszą różnicą pomiędzy pętlami *while* i *for* jest to, że nawias występujący po słowie kluczowym *while* zawierać może jedynie wyrażenie logiczne, nie umieszcza się w nim deklaracji zmiennych, jak to ma miejsce w przypadku pętli *for*. Zadeklarowanie zmiennej i przypisanie jej wartości powinno się odbyć przed rozpoczęciem pętli *while*³². Dodatkowo należy pamiętać, że polecenia, które wykonywane są w pętli powinny modyfikować wartość zmiennej wykorzystywanej w wyrażeniu logicznym. W przeciwnym wypadku (gdy w ciele pętli nie nastąpi zmiana wartości tej zmiennej) pętla wykonywać się będzie nieskończoną liczbę razy. W przykładzie 3.20 efekt taki zostanie osiągnięty po usunięciu instrukcji *i++*; (która inkrementuje zmienną *i*).

Osoby rozpoczynające naukę programowania zadają pytania dotyczące różnic pomiędzy obiema pętlami nie tylko pod względem składni, ale także ich stosowania – której w jakich sytuacjach najlepiej użyć. Najogólniej można powiedzieć, że pętlę *for* wykorzystywać powinno się wówczas, gdy w momencie rozpoczęcia pętli znana jest liczba jej wykonań (np. wypisanie dziesięciu liczb). Pętlę *while* wykorzystuje się wtedy, gdy nie wiadomo, ile razy pętla ma się wykonać, a znany jest tylko warunek, jaki musi być spełniony, aby polecenia w pętli zostały wykonane. Przykładem może być kod programu:

Przykład 3.21.

```
static void Main(string[] args)
{
    Console.WriteLine("Czas: {0}", DateTime.Now);
    Console.WriteLine("Ponownie pokazać aktualny czas? (t/n)");
    string odpowiedz = Console.ReadLine();
    while (odpowiedz != "n")
    {
        Console.WriteLine("Czas: {0}", DateTime.Now);
        Console.WriteLine("Ponownie pokazać aktualny czas? (t/n)");
        odpowiedz = Console.ReadLine();
    }
}
```

³¹ Zamiast zmiennej może być wywołanie metody, która zwraca wartość typu *bool*. W tym rozdziale nie będziemy wykonywać takich przykładów. Metody omówiono w [rozdziale 5](#).

³² Jest możliwe przypisanie zmiennej wewnątrz wyrażenia logicznego, ale wówczas cała instrukcja przypisania musi być objęta nawiasami okrągłymi, np. `while((x=y+1)>5) {...}`. Ten zapis spowoduje, że najpierw będzie obliczona suma *y+1*, następnie jej wynik będzie umieszczony w zmiennej *x* i dopiero później zmienna *x* będzie porównana z wartością 5. W tym rozdziale, w prezentowanych przykładach, nie będziemy wykonywać inicjalizacji zmiennych wewnątrz wyrażenia logicznego. Jest to bardziej przydatne, gdy do zmiennej przypisuje się wynik jakiejś metody (np. czytającej znaki z pliku). Ale będzie taka okazja w rozdziale kolejnym.

Przykładowy wynik programu:

```
Czas: 2014-01-04 19:18:52
Ponownie pokazać aktualny czas? (t/n)
t
Czas: 2014-01-04 19:19:33
Ponownie pokazać aktualny czas? (t/n)
t
Czas: 2014-01-04 19:19:46
Ponownie pokazać aktualny czas? (t/n)
t
Czas: 2014-01-04 19:20:37
Ponownie pokazać aktualny czas? (t/n)
n
```

Program wyświetla aktualny czas (z użyciem struktury *DateTime*, o której nieco więcej powiemy w rozdziale 6), po czym pyta, czy ponownie pokazać aktualny czas. I dopóki odpowiedź jest różna od „n”, wyświetlany jest aktualny czas systemowy komputera. Przed pętlą jest deklarowana zmienna *odpowiedz* typu *string*, następnie pobierany jest z klawiatury od użytkownika ciąg znaków, który zostaje przypisany do zmiennej *odpowiedz*. Jeśli wprowadzony tekst nie będzie równy „n”, wyrażenie logiczne pętli *while* zwróci wartość *true* i zostaną wykonane polecenia w środku pętli. Pierwszym poleceniem w pętli jest wyświetlenie bieżącego czasu. W dalszej części (po wyświetleniu zapytania) następuje przypisanie do zmiennej *odpowiedz* tekstu wprowadzonego z klawiatury. Jeżeli użytkownik wpisze literę „n”, wówczas wyrażenie logiczne zwróci wartość *false* i tym samym pętla zostanie zakończona, a program przestanie wyświetlać aktualny czas.

Uruchamiając powyższy program nie wiemy, ile razy pętla będzie się wykonywać. Użytkownik może już za pierwszym razem wprowadzić literę „n” (tym samym pętla tak naprawdę nie wykona się ani razu, ponieważ już podczas pierwszego sprawdzenia wyrażenie logiczne zwróci wartość *false*). Użytkownik może jednak dziesiątki, a nawet setki razy wpisywać inne znaki, powodując wykonanie poleceń w pętli.

Zauważmy, że w powyższym przykładzie dwukrotnie pojawia się sekwencja trzech instrukcji: wyświetlenie aktualnego czasu, pytanie dla użytkownika, czy ponownie wyświetlić czas oraz wczytanie odpowiedzi użytkownika. Pierwszy raz przed pętlą, po to, aby wyświetlić czas (zakładamy, że program przynajmniej raz wyświetli czas, a to, czy będzie wyświetlał jego aktualizacje oraz ile razy, zależy od użytkownika), a także w celu zainicjalizowania wartości zmiennej *odpowiedz*. Drugi raz ta sekwencja poleceń pojawia się w środku pętli, umożliwiając użytkownikowi wielokrotne dokonanie decyzji, czy ma być ponownie wyświetlony aktualny czas.

Umieszczanie identycznych poleceń w programie nie jest dobrym rozwiązaniem. Dlatego też oprócz pętli *while* jest dostępna jej „siostrzana” odmiana: pętla *do..while*.

Składnia 3.8

```
do
{
    // ciało pętli (instrukcje)
} while (wyrażenie logiczne)
```

Sposób działania pętli *do..while* jest bardzo podobny do działania pętli *while*. Tu także występuje wyrażenie logiczne, od którego zależy, czy pętla będzie wykonana (dla wartości *true*), czy też zostanie zakończona (dla wartości *false*)³³. Różnica pomiędzy tymi pętlami polega na tym, że w pętli *while* warunek logiczny sprawdzany jest **przed pierwszym** wykonaniem pętli, a w *do..while* warunek logiczny testowany jest **po pierwszym** wykonaniu pętli. W konsekwencji pętla *do..while* wykonana zostanie przynajmniej jeden raz, podczas gdy pętla *while* może nie zostać wykonana w ogóle, gdyby od razu wynikiem wyrażenia logicznego była wartość *false*.

Program z [przykładu 3.21](#) wykonamy teraz z użyciem pętli *do..while*.

Przykład 3.22.

```
static void Main(string[] args)
{
    string odpowiedz;
    do
    {
        Console.WriteLine("Czas: {0}", DateTime.Now);
        Console.WriteLine("Ponownie pokazać aktualny czas? (t/n)");
        odpowiedz = Console.ReadLine();

    } while (odpowiedz != "n");
}
```

Zamieniając pętlę *while* z [przykładu 3.21](#) na pętlę *do..while* wyeliminowano konieczność dwukrotnego użycia instrukcji (wyświetlenie czasu, zapytania i wczytanie odpowiedzi), w tej wersji występują one tylko raz – wewnątrz pętli *do..while*³⁴.

Podczas nauki zasad działania pętli *do..while* oraz *while* należy zwrócić uwagę na pewien szczegół, który jednak, jak większość szczegółów w programowaniu jest bardzo istotny. W przypadku pętli *while* nie umieszcza się średnika w linii, która zawiera wyrażenie logiczne, ponieważ bezpośrednio po sprawdzeniu wyrażenia powinny zostać wykonane

³³ Uwaga dla osób znających Pascala. W języku Pascal istnieje pętla REPEAT .. UNTIL(warunek), w której spełnienie warunku (wartość *true*) powoduje wyjście z pętli (jej zakończenie), czyli odwrotnie niż w przypadku pętli *do..while* w języku C#.

³⁴ W tym konkretnym przykładzie można by poradzić sobie ze zbędnym powieleniem instrukcji także w ramach pętli *while*, inicjalizując wstępnie zmienną *odpowiedz* wartością różną od „n” (np. „t”). Rozwiązanie takie jednak nie zawsze jest dobre, czasami wstępna inicjalizacja nie jest wskazana. Inaczej mówiąc, są sytuacje, gdy ten problem jest trudniej „obejść” i znacznie wygodniej jest użyć pętli *do..while*.

polecenia zawarte w pętli, a nie instrukcja pusta. W przypadku pętli *do..while* wyrażenie logiczne jest ostatnim elementem pętli, dlatego linię, która je zawiera należy zakończyć znakiem średnika.

Przedstawione pętle mimo różnego zapisu służą temu samemu – wielokrotnemu wykonaniu fragmentu kodu programu. Dlatego istnieje możliwość używania tych pętli zamiennie. Oczywiście ze względu na istotne różnice pomiędzy pętlą *for* i dwoma pozostałymi, zamiana tej pętli będzie wiązała się z większą modyfikacją kodu, niż w przypadku zamiany pętli *while* na *do..while* czy też odwrotnie.

Często osoby rozpoczynające naukę programowania zastanawiają się, dlaczego do dyspozycji mają aż trzy rodzaje pętli, skoro można je stosować zamiennie i przynajmniej teoretycznie wystarczyłoby znać zasadę działania jednej z nich. Okazuje się jednak, że różnorodność pętli to duże ułatwienie dla programisty. W zależności od aktualnych potrzeb i algorytmu tworzonego programu wybrać należy tę pętlę, która w danej sytuacji jest prostsza do zastosowania i w konsekwencji jej zapis jest bardziej intuicyjny.

Gdy przed rozpoczęciem działania pętli wiadomo, ile razy ma się ona wykonać, najczęściej optymalnym wyborem będzie wybór pętli *for*. Gdy jednak nie jest znana dokładna liczba iteracji, należy skorzystać z możliwości pętli *do..while* lub *while*. Wybór pomiędzy tymi pętlami powinien być dokonywany na podstawie informacji, czy polecenia zawarte w pętli **muszą** zostać wykonane przynajmniej raz (wtedy należy skorzystać z pętli *do..while*, jak w przykładzie 3.22), czy też dopuszczalna jest sytuacja, w której polecenia pętli mogą nie zostać wykonane w ogóle (wtedy należy użyć pętli *while*).

Na początkowym etapie nauki pętli dobrym treningiem jest implementowanie takiej samej funkcjonalności programu z wykorzystaniem każdej z pętli. Takie zadanie wykonamy najpierw wspólnie. Spójrzmy na przykład:

Przykład 3.23.

```
static void Main(string[] args)
{
    Console.WriteLine("Podaj dodatni wykładnik");
    int wykladnik = Convert.ToInt16(Console.ReadLine());
    if (wykladnik > 0)
    {
        int potega = 1;
        for (int i = 1; i <= wykladnik; i++)
        {
            potega = potega * 2;
            Console.WriteLine("2 do {0,2} = {1,2}", i, potega);
        }
    }
    Console.ReadKey();
}
```

Przykładowy wynik programu:

```
Podaj dodatni wykładnik
5
2 do 1 = 2
2 do 2 = 4
2 do 3 = 8
2 do 4 = 16
2 do 5 = 32
```

Program prosi użytkownika o podanie dodatniego wykładnika potęgi, a następnie w kolejnych liniach wyświetla potęgę liczby 2 dla wykładników od 1 do wartości podanej przez użytkownika. Obliczanie potęgi odbywa się w pętli *for* poprzez wielokrotne mnożenie zmiennej *potega* przez liczbę 2. Należy zwrócić uwagę, że zmienna *potega* jest wstępnie inicjalizowana wartością 1. Na marginesie warto wspomnieć, że w przypadku wielokrotnie wykonywanego dodawania – zmienną, w której ma być wypracowana suma, inicjalizuje się najczęściej zerem. Wracając do przykładu, jeśli użytkownik wpisze jako wykładnik 5, program wykona 5 przebiegów pętli. Kolejne przebiegi przedstawia tabela:

Przebieg pętli (<i>i</i>)	Wartość zmiennej <i>potega</i>
1	potega = potega * 2, czyli potega = 1 * 2
2	potega = potega * 2, czyli potega = 2 * 2
3	potega = potega * 2, czyli potega = 4 * 2
4	potega = potega * 2, czyli potega = 8 * 2
5	potega = potega * 2, czyli potega = 16 * 2

Ostatecznie zmienna *potega* przyjmuje wartość 32 (czyli 16 * 2).

Poniżej (w tabeli) umieszczamy trzy równoważne rozwiązania dla analizowanego programu, jedno przy użyciu pętli *for* (jak w przykładzie 3.23), drugie przy użyciu pętli *while* oraz trzecie z pętlą *do..while*.

Pętla <i>for</i>	<pre>for (int i = 1; i <= wykladnik; i++) { potega = potega * 2; Console.WriteLine("2 do {0,2} = {1,2}", i, potega); }</pre>
Pętla <i>while</i>	<pre>int i = 1; while (i <= wykladnik) { potega = potega * 2; Console.WriteLine("2 do {0,2} = {1,2}", i, potega); i++; }</pre>

Pętla <i>do..while</i>	<pre> int i = 1; do { potega = potega * 2; Console.WriteLine("2 do {0,2} = {1,2}", i, potega); i++; } while (i <= wykladnik); </pre>
---------------------------	---

W tym konkretnym programie większość programistów wybrałoby zapewne rozwiązanie z pętlą *for*, ponieważ po podaniu przez użytkownika wykładnika potęgi program będzie „wiedział”, ile razy pętla ma się wykonać. Ale nic nie stoi na przeszkodzie, aby użyć jednego z pozostałych rozwiązań. Należy jednak uważać na rozwiązanie z pętlą *do..while* i pamiętać, że w przypadku tej pętli jeden przebieg wykona się bezwarunkowo. Gdyby w przykładzie 3.23 usunąć instrukcję *if*, która sprawdza, czy wprowadzono dodatni wykładnik, wpisanie przez użytkownika wykładnika zerowego lub ujemnego (dla rozwiązania z pętlą *do..while*) spowodowałoby wyświetlenie potęgi 2^1 (a tak nie powinno być w tym programie). Zachęcamy do samodzielnego sprawdzenia takiego wariantu programu.

Na koniec jeszcze jedna uwaga odnośnie pętli *while* i *do..while*. W obu tych rozwiązaniach konieczne jest umieszczenie w pętli polecenia, powodującego zwiększanie wartości zmiennej *i* (użytej w wyrażeniu logicznym). Nie musi to jednak być osobna instrukcja, jak w powyższej tabeli. Równie dobrze można stosować poniższy zapis:

```

while (i <= wykladnik)
{
    potega = potega * 2;
    Console.WriteLine("2 do {0,2} = {1,2}", i++, potega);
}

```

Podobnie byłoby dla pętli *do..while*. W powyższym wariantcie zmienna *i* zmienia się w poleceniu *Console.WriteLine()*. Operator inkrementacji *++* jest umieszczony po nazwie zmiennej, co oznacza, że zostanie użyta wartość tej zmiennej sprzed zmiany, a dopiero po wykonaniu polecenia ulegnie zwiększeniu o jeden. Ten zapis jest krótszy (o jedną linię) i z tego powodu chętnie stosowany. Osobom, które dopiero rozpoczynają naukę zalecamy, aby w pierwszych programach z użyciem pętli *while* (lub *do..while*) używały dłuższego zapisu, z inkrementacją zmiennej w osobnej linii.

3.2.3 Polecenia *break* i *continue*

W wielu dobrze napisanych algorytmach nie pojawia się konieczność przerywania pętli przed jej naturalnym zakończeniem. Zdarzają się jednak sytuacje, w których powinno nastąpić natychmiastowe przerywanie działania pętli. Należy wtedy skorzystać ze słowa kluczowego *break* (dokładnie tego samego, które pojawia się w instrukcji warunkowej *switch..case*).

Przykład 3.24.

```
static void Main(string[] args)
{
    int a = 0;
    do
    {
        a++;
        if (a == 5)
            break;           // przerwij pętlę
        Console.WriteLine(a);
    } while (true);
    Console.ReadKey();
}
```

Powyższy przykład spowoduje wyświetlenie na ekranie konsoli liczb od 1 do 4. Gdy zmienna *a* będzie równa 5, wykonana zostanie instrukcja *break*; która przerywa działanie pętli. Gdyby tej instrukcji nie było, pętla działałaby nieskończoną liczbę razy, ponieważ w miejscu wyrażenia logicznego po słowie *while* umieszczono wartość *true*.

W przypadku pętli zagnieżdżonych polecenie *break* przerywa działanie tylko tej pętli, w której wewnątrz zostało użyte. Popatrzmy na przykład:

Przykład 3.25.

```
static void Main(string[] args)
{
    for (int i = 1; i <= 3; i++)
    {
        Console.WriteLine("Liczby w {0} wierszu:", i);
        for (int j = 1; j <= 5; j++)
        {
            if (j == 3) break;
            Console.Write(j + ",");
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}
```

Wynik programu:

```
Liczby w 1 wierszu:
1,2,
Liczby w 2 wierszu:
1,2,
Liczby w 3 wierszu:
1,2,
```

W programie są dwie pętle *for*, jedna zagnieżdżona w drugiej. Polecenie *break* jest tylko w wewnętrznej pętli. Pętla wewnętrzna powinna mieć 5 przebiegów, ale ponieważ w

środku tej pętli jest instrukcja *if*, która dla licznika pętli *j* równego 3 powoduje przerwanie pętli, to faktycznie wykonywane są tylko 3 przebiegi. W ostatnim (trzecim) przebiegu jest wykonana tylko instrukcja *if*, a w dwóch poprzednich (oprócz sprawdzania warunku), wyświetlają się kolejne wartości licznika pętli (dla *j* równego 1 i 2). Działanie tego programu potwierdza, że pętla wewnętrzna była przerywana, ale pętla zewnętrzna nie. Pętla zewnętrzna (dla licznika *i*) miała zdefiniowane trzy przebiegi i tyle się wykonało.

Obok możliwości przerywania pętli istnieje również możliwość jej wznowienia z zaniechaniem wykonania poleceń, jakie zostały do wykonania w danym przebiegu. Służy do tego celu instrukcja *continue*. Można powiedzieć, że polecenie *continue* powoduje przerwanie jedynie bieżącego przebiegu pętli (a nie działania całej pętli jak w przypadku *break*).

Przykład 3.26.

```
static void Main(string[] args)
{
    for (int i = 1; i <= 6; i++)
    {
        if (i == 4)
            continue;           // pominięcie dalsze instrukcje i wznów pętle
        Console.WriteLine(i);
    }
    Console.ReadKey();
}
```

W wyniku działania powyższej pętli na ekranie zostaną wyświetlone liczby od 1 do 6, jednak bez liczby 4. Gdy wartość licznika pętli będzie równa 4, zostanie wykonane polecenie *continue*, tym samym w tej iteracji pominięte zostanie polecenie *Console.WriteLine()*.

Informacja odnośnie zagnieżdżonych pętli i polecenia *break* dotyczy także polecenia *continue*. Polecenie *continue* ma wpływ tylko na tę pętlę, w której wnętrzu zostało umieszczone. Można łatwo to sprawdzić, zmieniając w [przykładzie 3.25](#) tylko jedno słowo – zamiast *break* należy wpisać *continue*. Wówczas program pokaże wynik:

```
Liczby w 1 wierszu:
1,2,4,5,
Liczby w 2 wierszu:
1,2,4,5,
Liczby w 3 wierszu:
1,2,4,5,
```

W pętli wewnętrznej nastąpiło pominięcie instrukcji wyświetlania licznika pętli równego 3 (zgodnie z warunkiem w instrukcji *if*). Natomiast pętla zewnętrzna wykonała się zgodnie z planem (definicją) – trzykrotnie.

Język C# oferuje cztery rodzaje pętli. Do tej pory przedstawione zostały trzy z nich. Ostatnia pętla jest dosyć specyficzna, ponieważ pozwala na przetwarzanie danych złożonych

(zawierających wiele wartości). Pętla ta (*foreach*) zostanie przedstawiona w kolejnym rozdziale poświęconym [tablicom](#). Skrócony opis pętli zawiera [Tabela 9](#) w *Dodatkach*.

3.3 Zadania

Rozdział trzeci omawia wszystkie instrukcje sterujące. Wyjaśniliśmy tu dość dużo konstrukcji językowych, posiłkując się możliwie prostymi przykładami. Z punktu widzenia zakresu, jaki obejmuje ten podręcznik rozdział trzeci moglibyśmy określić jako przełomowy. To znaczy pojawiły się już pewne „schody” i aby ułatwić Czytelnikowi wykonanie zadań w punkcie 3.3.2 (zadania do samodzielnego rozwiązania), wyjątkowo w tym rozdziale umieszczamy także kilka zadań rozwiązanych.

3.3.1 Zadania z rozwiązaniami

Zadanie 3.a.

Napisz program, który dla podanej przez użytkownika liczby całkowitej sprawdza, czy jest parzysta oraz czy jest ujemna.

Ponieważ liczba może być zarówno parzysta jak i ujemna, musimy wykorzystać dwie niezależne instrukcje *if..else*. Jedna z nich sprawdza parzystość (w tym celu użyjemy operatora %, który zwraca resztę z dzielenia), a druga sprawdza, czy liczba jest ujemna.

```
static void Main(string[] args)
{
    Console.WriteLine("Wprowadź liczbę całkowitą");
    int liczba = int.Parse(Console.ReadLine());
    if (liczba % 2 == 0)
        Console.WriteLine("{0} jest liczbą parzystą", liczba);
    else
        Console.WriteLine("{0} jest liczbą nieparzystą", liczba);
    if (liczba < 0)
        Console.WriteLine("{0} jest liczbą ujemną", liczba);
    else
        Console.WriteLine("{0} nie jest liczbą ujemną", liczba);
    Console.ReadKey();
}
```

Przykładowy wynik programu:

```
Wprowadź liczbę całkowitą
-14
-14 jest liczbą parzystą
-14 jest liczbą ujemną
```

Zadanie 3.b.

Napisz program pobierający od użytkownika dwie liczby całkowite. Program powinien wypisać parzyste liczby znajdujące się pomiędzy podanymi wartościami.

Można wykonać ten program tak jak poniżej, sprawdzając w pętli każdą liczbę z zadanego przedziału:

```
static void Main(string[] args)
{
    Console.WriteLine("Wprowadź liczbę 1");
    int liczba1 = int.Parse(Console.ReadLine());
    Console.WriteLine("Wprowadź liczbę 2");
    int liczba2 = int.Parse(Console.ReadLine());
    Console.Write("Liczby parzyste: ");
    for (int i = liczba1; i <= liczba2; i++)
    {
        if (i % 2 == 0)
        {
            Console.Write(i + ",");
        }
    }
    Console.ReadKey();
}
```

Można jednak zrobić inaczej. Sprawdzić jedynie liczby podane przez użytkownika, czy są parzyste. W przypadku liczby otwierającej przedział, jeśli nie jest parzysta, należy zwiększyć jej wartość o jeden. Natomiast liczbę kończącą przedział, jeśli nie jest parzysta, należy zmniejszyć o jeden. A następnie można wyświetlać liczby zwiększane w kolejnych krokach iteracji o wartość 2, jak w poniższym fragmencie programu:

```
liczba1 = (liczba1 % 2 == 0) ? liczba1 : liczba1 + 1;
liczba2 = (liczba2 % 2 == 0) ? liczba2 : liczba2 - 1;
for (int i = liczba1; i <= liczba2; i+=2)
{
    Console.Write(i + ",");
}
```

W powyższym rozwiązaniu użyto operatorów warunkowych do ustalenia przedziałów – jeśli dana liczba jest parzysta, to operator warunkowy zwróci liczbę bez zmian, w przeciwnym razie doda lub odejme 1 (w zależności od tego, czy to początek przedziału, czy koniec). W pętli *for* licznik pętli jest zwiększany o 2.

Zamiast operatorów warunkowych można użyć instrukcji *if*:

```
if (liczba1 % 2 != 0)
    liczba1 = liczba1 + 1;
if (liczba2 % 2 != 0)
    liczba2 = liczba2 - 1;
```

```
for (int i = liczba1; i <= liczba2; i += 2)
{
    Console.Write(i + ",");
}
```

Zadanie 3.c.

Dwaj koledzy Janek i Karol zadłużyli się na 80 zł. Umówili się między sobą, że każdy z nich codziennie będzie odkładał na spłatę zadłużenia 20% swojego dziennego zarobku, Janek zarabia dziennie 50 zł, a Karol 40 zł. Napisz program, który przy pomocy pętli *do..while* sprawdzi po ilu dniach obaj koledzy uzbierają kwotę potrzebną do spłaty swojego długu oraz wypisze wartość uzbieranej kwoty w każdym dniu.

Wykorzystamy pętlę *do..while*, która będzie wykonywana tak długo, dopóki dług jest większy niż uzbierana w danym dniu kwota przeznaczona na spłatę.

```
static void Main(string[] args)
{
    int i = 1;
    double dlug = 80, zarobekJanka = 50, zarobekKarola = 40;
    double splata = 0;
    do
    {
        splata += 0.2 * zarobekJanka + 0.2 * zarobekKarola;
        Console.WriteLine("Dzień = {0}   Spłata = {1}", i++, splata);
    } while (dlug > splata);
    Console.ReadKey();
}
```

Wynik programu:

Dzień = 1	Spłata = 18
Dzień = 2	Spłata = 36
Dzień = 3	Spłata = 54
Dzień = 4	Spłata = 72
Dzień = 5	Spłata = 90

W piątym dniu zostanie zgromadzona kwota, która wystarczy do spłaty długu.

Zadanie 3.d.

Napisz program, który wyświetla na ekranie konsoli „kwadrat” zbudowany ze znaku „*”. Liczbę wierszy (a tym samym kolumn) ma podać użytkownik.

W programie wykorzystamy dwie pętle *for* (jedna zagnieżdżona w drugiej). Pętla zewnętrzna wykona dwa zadania: uruchomienie pętli wewnętrznej (służącej do wyświetlenia jednego wiersza gwiazdek) oraz umieszczenie znaku końca linii (czyli wywołanie instrukcji *Console.WriteLine()* bez argumentu). Pętla wewnętrzna wyświetli w tej samej linii tyle gwiazdek, ile „kwadrat” ma mieć wierszy (i kolumn).

```
static void Main(string[] args)
{
    Console.WriteLine("Wprowadź liczbę wierszy");
    int n = int.Parse(Console.ReadLine());
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            Console.Write("*");
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}
```

Przykładowy wynik programu:

```
Wprowadź liczbę wierszy
5
*****
*****
*****
*****
*****
```

Program ten można wykorzystać podczas pracy nad zadaniem 3.9 – wszystkie wymienione tam „figury” są oparte na „kwadracie”. Przy czym w niektórych miejscach ma być znak „*”, a w niektórych znak spacji. To, jaki znak ma być umieszczony zależy od lokalizacji tego znaku, co możemy rozpoznać po zmiennych sterujących pętlą (*i* oraz *j*).

3.3.2 Zadania do samodzielnego rozwiązania

Zadanie 3.1.

Napisz program, który sprawdza, czy podany rok jest rokiem przestępnym. Rok przestępny dzieli się bez reszty przez 4, nie dzieli się przez 100 (za wyjątkiem lat podzielnych przez 400).

Zadanie 3.2.

Napisz program pobierający od użytkownika dwie liczby całkowite. Program powinien wyświetlać informację, czy druga liczba jest dzielnikiem pierwszej.

Zadanie 3.3.

Napisz program pobierający od użytkownika 3 liczby. Program ma wyświetlić wartość największej z nich.

Zadanie 3.4.

Napisz program – prosty kalkulator, który wczytuje od użytkownika wartości dwóch zmiennych typu *double* oraz znak operacji (+ lub – lub * lub /), a następnie wyświetla wynik

operacji dla podanych wartości. Przykładowo użytkownik wprowadził znak „+” i liczby 1,5 oraz 2,5, program powinien wyświetlić sumę obu liczb, czyli 4,0.

Zadanie 3.5.

Napisz program obliczający liczbę pierwiastków równania kwadratowego. Program ma prosić użytkownika o podanie współczynników równania, a następnie ma wyświetlić stosowny komunikat.

Zadanie 3.6.

Napisz program, który oblicza wskaźnik masy ciała BMI. Program ma prosić użytkownika o podanie wagi w kg oraz wzrostu w metrach. Wzór: $BMI = \frac{masa}{wzrost^2}$ (treść [zadania 2.3](#)).

- a) Po obliczeniu wskaźnika BMI program powinien wyświetlać stosowną informację w zależności od wartości wskaźnika:
- $< 18,5$ – niedowaga,
 - $18,5$ – $24,99$ – wartość prawidłowa,
 - $\geq 25,0$ – nadwaga.
- b) Korzystając z Wikipedii rozszerz program, tak aby wyświetlał komentarz według poszerzonej klasyfikacji zakresów wskaźnika BMI.

Zadanie 3.7.

Wykonaj program z [przykładu 3.8](#) (str. 64) z użyciem instrukcji *switch..case* (zamiast *if..else*).

Zadanie 3.8.

Pobierz od użytkownika wartość średniej ocen. Program ma wyświetlać informacje o wysokości przysługującego stypendium zgodnie z poniższą tabelą:

Średnia ocen		Kwota stypendium
Od	Do	
2,00	3,99	0,00 zł
4,00	4,79	350,00 zł
4,80	5,00	550,00 zł

Zadanie 3.9.

Napisz program, którego efektem działania będą następujące „figury” utworzone ze znaku gwiazdki (*). Liczbę wyświetlanych wierszy podaje użytkownik.

*	*****	*	*****
**	****	**	* *
***	**	***	* *
****	*	****	****
a	b	c	d

Zadanie 3.10.

Napisz program obliczający $n!$ (n silnia), gdzie n jest podane przez użytkownika.

Zadanie 3.11.

Napisz program obliczający ile kolejnych liczb całkowitych (rozpoczynając od wartości 1) należy dodać do siebie, aby suma przekroczyła wartość 100.

Zadanie 3.12.

Napisz program pobierający od użytkownika liczby całkowite. Program ma pobierać te liczby do czasu, gdy użytkownik wprowadzi wartość 0 (zero). Wynikiem działania programu ma być informacja o sumie wprowadzonych przez użytkownika liczb.

Zadanie 3.13.

Napisz program obliczający sumę szeregu $W(n)=1 - 2 + 3 - 4 + \dots \pm n$, gdzie n jest dowolną liczbą naturalną, którą program ma wczytać.

Zadanie 3.14.

Liczba N jest doskonała, gdy jest równa sumie swych dzielników mniejszych od niej samej np. $6=1+2+3=6$ – jest liczbą doskonałą. Napisz program znajdujący liczby doskonałe w przedziale $\langle 1, n \rangle$, gdzie n podaje użytkownik.

Zadanie 3.15.

Dysponując monetami 1 zł, 2 zł, 5 zł sprawdź, na ile różnych sposobów można wypłacić 10 zł. Napisz program, który wyświetli w oknie konsoli wszystkie możliwe kombinacje.

4 Operacje na typach referencyjnych – tablice i typ *string*

W [podrozdziale 2.1](#) przedstawione zostały podstawy dotyczące zmiennych. Jak pokazują przykłady zamieszczone w poprzednich rozdziałach, zmienne są powszechnym elementem programu komputerowego. Poprzez zmienne możemy kontrolować przebieg programu, odpowiednio definiując liczniki, badając ich stan – generalnie zapisywać w nich wszystkie te informacje, które wspomagają przetwarzanie. Możemy też w zmiennych zapisywać dane – te, które mają ulegać przetwarzaniu. Ten podział nie jest sztywny, dążymy tu jedynie do przedstawienia danych jako pewnego fragmentu modelu rzeczywistości. Informację o tym, że rok ma 12 miesięcy możemy zapisać w postaci literału 12 (nie spodziewamy się zmiany kalendarza). Jakąś stałą matematyczną lub fizyczną (ze względu na precyzję) lepiej będzie zapisać w postaci stałej (deklarowanej przy użyciu modyfikatora *const*). Natomiast informację o cenie danej akcji giełdowej lepiej już zapisać w zmiennej. W świecie rzeczywistym dane występują jednak w pewnych grupach (zbiorach). Np. cena akcji danego przedsiębiorstwa zmienia się w czasie, a nas mogą interesować wszystkie ceny z danego okresu. Inaczej mówiąc w programowaniu potrzebujemy złożonych struktur, które pozwolą odzwierciedlić, na ile to możliwe, „naturalny charakter” zbioru danych. W programowaniu takich specjalnych konstrukcji pozwalających zapisywać złożone struktury jest dość dużo. W dodatku możemy je zagnieżdżać (jak i w życiu – ceny akcji jednej firmy to zbiór danych, a jeśli mamy takie zbiory dla wielu firm, to jest to już „zbiór zbiorów”). W języku C# można takie grupy danych zapisywać m.in. w tzw. **kolekcjach**. Jest kilka rodzajów kolekcji, a najprostszym z nich jest tablica jednowymiarowa. Ponieważ tablice służą do przechowywania grupy danych, a te mogą być dość liczne i zajmować sporo miejsca w pamięci – tablice umieszcza się na stercie jako obiekty. Tablice mają zatem typ referencyjny. O typach referencyjnych pisaliśmy w [punkcie 2.1.1](#).

Oprócz tablic omówimy tu także dane typu *string*. Typ tekstowy to typ wbudowany, ale jak można sprawdzić na wykazie typów wbudowanych (w [Tabeli 1](#) w *Dodatkach*) – typ ten jest typem referencyjnym. Z podobnego powodu, jak tablice. Teksty mogą mieć duży rozmiar. Można zapisać w zmiennej łańcuchowej cały tekst „Pana Tadeusza” autorstwa naszego wieszczka narodowego. A te wszystkie znaki w książce – to przecież też zbiór danych (zbiór znaków). Niemniej typ *string* jest dość wyjątkowy, bo choć jest referencyjny, to programista ma możliwość używania go jakby był typem prostym, o czym już Czytelnik miał okazję się przekonać. W tym jednak rozdziale, w ostatniej jego części, będzie możliwość lepiej poznać typ *string* – od tej jego „referencyjnej” strony.

4.1 Tablice

Jest kilka rodzajów tablic w języku C#, zaczniemy od omawiania najprostszych tablic jednowymiarowych, następnie omówimy tablice regularne dwuwymiarowe oraz tablice nieregularne (tzw. postrzępione). Na koniec podrozdziału przedstawimy kilka metod klasy *Array*, których można użyć do operacji na tablicach.

4.1.1 Tablice jednowymiarowe

Na pewnym etapie nauki programowania okazuje się jednak, że „zwykłe” zmienne, takie jakie wykorzystywane były do tej pory, posiadają pewne ograniczenia, które z czasem stają się bardzo uciążliwe. Dla przykładu założmy, że chcemy napisać program komputerowy, który pozwoli na przechowywanie informacji o wieku uczestników pewnej wycieczki. Informacje te najpierw trzeba pobrać (np. od użytkownika), następnie należy je zapamiętać, aby w późniejszym etapie móc wykonywać na nich podstawowe działania (np. obliczenie średniej wieku uczestników wycieczki, czy uzyskanie informacji o wieku najstarszej lub najmłodszej osoby).

Teoretycznie nic nie stoi na przeszkodzie, aby w tym celu wykorzystywać zmienne, w sposób, w jaki wykorzystywane były do tej pory. Jeżeli w wycieczce udział będzie brało 5 osób, można korzystając z pięciu zmiennych typu *int* zapamiętać odpowiednie informacje³⁵.

Przykład 4.1.

```
static void Main(string[] args)
{
    int uczestnik_1 = 19;
    int uczestnik_2 = 34;
    int uczestnik_3 = 23;
    int uczestnik_4 = 54;
    int uczestnik_5 = 31;
    double srednia = (uczestnik_1 + uczestnik_2 + uczestnik_3 +
                     uczestnik_4 + uczestnik_5) / 5.0;
    Console.WriteLine(srednia);
    Console.ReadKey();
}
```

W przykładzie użyto pięciu zmiennych i obliczono średnią. Na marginesie zwracamy uwagę, że suma zmiennych podzielona tu została przez literał 5.0, który ma domyślny typ *double* i dzięki temu zastosowany operator dzielenia „/” pozwoli uzyskać dokładny wynik (32,2). Przypominamy, że w przypadku, gdy oba argumenty operatora dzielenia „/” są typu całkowitoliczbowego, to operator ten zwraca tylko część całkowitą wyniku (więcej na ten temat zob. w [punkcie 2.2.2](#)).

³⁵ Do przechowywania informacji o wieku uczestników wystarczający byłby typ *byte*, jednak kolejne przykłady będą nadal operować na naszym „długim” typie całkowitym – *int*.

Obliczenie średniej arytmetycznej wieku uczestników wycieczki nie było zbyt kłopotliwe (dla 5 osób). Trochę bardziej skomplikowane byłoby sprawdzenie, ile lat ma najmłodszy lub najstarszy uczestnik wycieczki. W tym celu należałoby skorzystać z kilku instrukcji warunkowych *if*. Co się jednak stanie, gdy uczestników wycieczki będzie nie 5, a np. 50 lub 100? Ile dodatkowych zmiennych trzeba będzie wykorzystać? Ile dodatkowych linii programu trzeba napisać? Ile czasu zmarnuje na to programista? Rozwiązaniem tego problemu jest wykorzystanie struktury złożonej, zawierającej zbiór elementów. Takie struktury danych nazywane są **kolekcjami**, spośród których najprostszą jest tablica jednowymiarowa.

Tablica to struktura danych pozwalająca na przechowywanie zbioru elementów określonego typu. Najprostszą tablicę można zilustrować w poniższy sposób:

uczestnicy:

19	34	23	54	31
----	----	----	----	----

W powyższym przykładzie tablica ma nazwę *uczestnicy* i przechowuje 5 wartości określających wiek uczestników wycieczki. Tablica pozwala na przechowywanie bardzo wielu wartości i co wynika z przedstawionej definicji pojęcia *tablica* – wszystkie te wartości muszą być tego samego typu.

Tablice w języku C# są obiektami (typu referencyjnego). Aby móc używać w programie tablicy należy zadeklarować zmienną tablicową oraz utworzyć samą tablicę (obiekt tablicy). W przypadku tablic istnieje kilka możliwości ich deklarowania. W rozdziale drugim prezentowaliśmy [składnię 2.2](#) mówiąc, że taka składnia deklaracji zmiennej (z użyciem operatora *new*) dla danych typów wbudowanych nie jest konieczna i że wrócimy do tej postaci mając do czynienia z danymi typów referencyjnych – w tym właśnie tablicami. Spójrzmy na jeden z możliwych sposobów deklaracji tablicy:

Składnia 4.1

```
Typ[] nazwa = new Typ[rozmiar];
```

W porównaniu ze składnią 2.2 tu muszą wystąpić nawiasy kwadratowe³⁶. To, co jest w składni 4.1 ujęte w jednej linii może być zapisane także w dwóch osobnych liniach:

```
Typ[] nazwa;           // deklaracja zmiennej tablicowej
nazwa = new Typ[rozmiar]; // utworzenie tablicy (obektu) i przypisanie referencji
                           // (wskazującej na utworzoną tablicę) do zmiennej
                           // tablicowej
```

³⁶ W tej składni (jak i wszystkich dla tablic) nawiasy kwadratowe nie oznaczają opcjonalności, należy je tu czytać „dosłownie”.

Pisząc w rozdziale drugim ([punkt 2.1.1](#)) o typie referencyjnym wyjaśniliśmy, że dane typu referencyjnego zapisane są na stercie, natomiast na stosie zapisywane są referencje do tych danych. Po wykonaniu instrukcji deklarującej zmienną tablicową, np. `int[] tab;` - istnieje już w programie zmienna tablicowa, ale ma na razie wartość `null` (czyli jest to „pusta” referencja). Natomiast przypisanie do tej zmiennej (poprzez operator `new`) obiektu tablicy z podanym jej rozmiarem powoduje, że referencja (zmienna tablicowa) pokazuje już na utworzoną tablicę (wypełnioną wartościami domyślnymi w zależności od typu).

Możemy deklarować tablicę także od razu inicjalizując jej elementy, według składni:

Składnia 4.2

```
Typ[] nazwa = new Typ[] {Lista inicjalizacyjna};
Lub skrócona wersja:
Typ[] nazwa = {Lista inicjalizacyjna};
```

W deklaracji tablicy wraz z inicjalizacją także ma miejsce deklaracja zmiennej tablicowej oraz obiektu tablicy (mimo że w skróconej wersji nie ma operatora `new`). Lista inicjalizacyjna zawiera elementy tablicy i rozmiar tablicy zostanie ustalony automatycznie na podstawie ich liczby.

Wracając do przykładu z uczestnikami wycieczki, najpierw zadeklarujemy tablicę używając składni 4.2, a następnie omówimy po kolei elementy deklaracji:

```
int[] uczestnicy = new int[] { 19, 34, 23, 54, 31 };
```

Po nazwie typu liczbowego określającego typ wartości, jakie będą przechowywane w tablicy (np. `byte`, `int`, `float`) pojawiają się puste nawiasy kwadratowe. Nawiasy te informują kompilator, że deklarowana zmienna będzie zmienną tablicową. Następnie pojawia się nazwa zmiennej. Tablica ta jest od razu inicjalizowana wartościami, dlatego też (analogicznie jak to było w przypadku „zwykłych” zmiennych) po nazwie zmiennej umieszczony jest operator przypisania, czyli znak „=”.

W języku C# każda zmienna jest obiektem. Podstawowe informacje dotyczące obiektów znajdują się w dalszej części podręcznika, w [rozdziale 6](#). W tej chwili wystarczy informacja, że obiekty tworzone są z wykorzystaniem słowa kluczowego `new`. Po nim zostaje umieszczony typ elementów tablicy oraz para nawiasów kwadratowych informująca, że tworzony obiekt będzie tablicą. Wreszcie konkretne wartości, którymi tablica zostanie zainicjalizowana wypisane są w nawiasach klamrowych i oddzielane przecinkami. Nawiasy kwadratowe występujące po słowach `new int` mogą zawierać informację o liczbie elementów (rozmiar tablicy), jakie będą przechowywane w tablicy. Jednak liczba ta musi być zgodna z liczbą elementów, które zostały wypisane w nawiasach klamrowych. Dlatego też, eliminując potencjalne miejsce popełnienia błędu, najlepiej jest nie wpisywać rozmiaru w tej deklaracji.

Zgodnie ze [składnią 4.2](#) można tę sama tablicę zadeklarować krócej:

```
int[] uczestnicy = { 19, 34, 23, 54, 31 };
```

W skróconej wersji brak operatora *new* oraz nazwy typu po prawej stronie operatora przypisania, ale obiekt jest tworzony tak samo, jak w przypadku pełnej wersji zapisu.

Zadeklarowana tablica *uczestnicy* posiada 5 elementów, każdy z nich jest typu *int*. Liczba elementów, które mogą być w tablicy przechowywane w tym samym czasie określa jej rozmiar. Aby możliwy był dostęp do poszczególnych elementów tablicy, są one numerowane. W języku C# elementy tablicy numerowane są (indeksowane) począwszy od wartości 0. Na początku nauki programowania może to być trochę mylące (od dziecka jesteśmy nauczani zliczać wszystko począwszy od 1). Dostęp do konkretnego elementu tablicy możliwy jest poprzez podanie nazwy tablicy i indeksu elementu umieszczonego w nawiasie kwadratowym. Wyświetlenie na ekranie konsoli wszystkich elementów tablicy *uczestnicy* umożliwia poniższy kod programu.

Przykład 4.2.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    Console.WriteLine(uczestnicy[0]);
    Console.WriteLine(uczestnicy[1]);
    Console.WriteLine(uczestnicy[2]);
    Console.WriteLine(uczestnicy[3]);
    Console.WriteLine(uczestnicy[4]);
    Console.ReadKey();
}
```

Pierwszy element tablicy ma indeks równy 0. Indeks elementu ostatniego, w tablicy 5-cio elementowej równy jest 4.

Jak widać w przykładzie 4.2 poszczególne linie kodu różnią się jedynie wartością indeksu tablicy. Wiadomo, od jakiej wartości indeksu należy rozpocząć wyświetlanie elementów tablicy (od indeksu równego 0), wiadomo też, że indeks za każdym razem zwiększany jest o 1, aż do momentu, gdy osiągnie wartość równą 4. Można zatem w programie z przykładu 4.2 użyć pętli. Pętla *for* wyświetlająca wszystkie elementy tablicy może wyglądać następująco:

Przykład 4.3.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    for (int i = 0; i < 5; i++)
        Console.WriteLine(uczestnicy[i]);
    Console.ReadKey();
}
```

To samo zadanie za pomocą pętli *do..while* przedstawia poniższy przykład:

Przykład 4.4.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    int i = 0;
    do
    {
        Console.WriteLine(uczestnicy[i]);
        i++;
    } while (i < 5);
    Console.ReadKey();
}
```

Należy podkreślić, że indeksy tablicy są wartościami całkowitymi, dlatego w przykładach 4.3 oraz 4.4 zmienna wykorzystywana jako indeks tablicy (*i*) jest typu *int*.

Przedstawione przykłady pokazują przewagę tablic nad „prostymi” zmiennymi w sytuacji, gdy mamy pracować z grupą danych określonego typu. Gdyby uczestników wycieczki było 50, należałoby odpowiednio zmienić deklarację tablicy oraz wyrażenie logiczne dla pętli użytej do przetwarzania tej tablicy.

Poniżej zadeklarujemy i omówimy tablicę według [składni 4.1](#):

```
int[] tablica = new int[10];
```

lub w wersji z zapisem w dwóch osobnych liniach:

```
int[] tablica;
tablica = new int[10];
```

Liczba umieszczona w nawiasach kwadratowych wskazuje na liczbę elementów tablicy. Została tu zadeklarowana 10-cio elementowa tablica liczb typu *int*. W tablicy tej element pierwszy posiada indeks równy 0, a element ostatni indeks równy 9.

Uwaga!	Ponieważ indeksy tablic numerowane są od wartości 0 to ostatni element deklarowanej tablicy ma indeks o 1 mniejszy od rozmiaru (czyli w tablicy 10-elementowej ostatni element ma indeks równy 9).
---------------	--

Zadeklarowana tablica, której nie zainicjalizowano, zostaje wypełniona wartościami domyślnymi dla danego typu. Np. wszystkie elementy zadeklarowanej tablicy typu *int* zostaną wypełnione wartościami równymi 0.

Aby zapisać dowolną wartość (zgodną z typem elementów tablicy) należy (podobnie jak w przypadku wyświetlania) określić nazwę zmiennej i numer indeksu, do którego nowa wartość ma zostać wpisana.

Przykład 4.5.

```
static void Main(string[] args)
{
    int[] tablica = new int[3];
    tablica[0] = 19;
    tablica[1] = 34;
    tablica[2] = 23;
}
```

Deklarację tablicy oraz przypisanie wartości jej elementom w przykładzie 4.5 można zrobić w jednej linii z użyciem [składni 4.2](#). Poniżej zestawiono obok siebie oba równoważne rozwiązania:

// Deklaracja bez inicjalizacji, przypisanie w osobnych liniach (przykład 4.5)

```
int[] tablica = new int[3];
tablica[0] = 19;
tablica[1] = 34;
tablica[2] = 23;
```

// Deklaracja z inicjalizacją

```
int[] tablica = {19, 34, 23};
```

Osoby rozpoczynające naukę programowania popełniają nierzadko błędy związane z indeksowaniem tablic. Wyświetlanie całej tablicy umożliwiają pętle. Zgodnie z zasadami tworzenia pętli (zob. [podrozdział 3.2](#)), konieczne jest określenie warunków ich działania. W przypadku tablic, warunek ten określać powinien, które elementy tablicy mają zostać uwzględnione. W [przykładach 4.3 i 4.4](#) wyświetlano wszystkie elementy tablicy 5-cio elementowej, dlatego wyrażenia logiczne obu pętli zdefiniowano jako: $i < 5$. Gdyby w tym wyrażeniu logicznym wykorzystano nierówność nieostrą ($i \leq 5$) program zakończyłby się komunikatem o błędzie (pojawiającym się w trakcie działania programu, a nie na etapie jego kompilacji). Błąd wynikałby z próby dostępu do elementu tablicy o indeksie 5, a tablica 5-cio elementowa nie ma elementu o indeksie równym 5. Najwyższa wartość indeksu w takiej tablicy wynosi 4.

Druga trudność wynikająca z wykorzystania pętli w obsłudze tablic pojawia się, gdy programista z jakichś powodów w trakcie pracy nad programem zmienia długość tablicy. Na przykład zmienna tablicowa, która do tej pory posiadała 10 elementów, okazuje się za mała i jest rozszerzana do 100 elementów. Ta zmiana wymusza modyfikację warunków działania pętli wykorzystujących tę tablicę, tak aby w dalszym ciągu uwzględniane były wszystkie elementy tablicy. Jeżeli w programie występuje wiele pętli obsługujących taką tablicę, o popełnienie błędu nie trudno. Dlatego też dobrym rozwiązaniem jest wykorzystywanie w kodzie programu właściwości *Length*. W [podrozdziale 2.4](#) przedstawiono właściwość *Length*

dla zmiennej typu *string* (która zwraca długość tekstu). Zmienne tablicowe również udostępniają kilka właściwości³⁷.

Przykład 4.6.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    for (int i = 0; i < uczestnicy.Length; i++)
        Console.WriteLine(uczestnicy[i]);
    Console.ReadKey();
}
```

Powyższa pętla od pętli z [przykładu 4.3](#) różni się drobnym szczegółem. W wyrażeniu logicznym wykorzystano informację o długości (rozmiarze) tablicy *uczestnicy.Length* - czyli właściwość zmiennej tablicowej określającą liczbę elementów tablicy. Dzięki temu programista ma pewność, że niezależnie od zmian, jakie zostaną dokonane w deklaracji zmiennej tablicowej *uczestnicy* pętla ta uwzględni wszystkie jej elementy.

Zwracamy uwagę na fakt, iż *uczestnicy.Length* wyznacza długość (rozmiar) tablicy *uczestnicy*, a więc liczbę jej elementów. Dlatego też warunek logiczny musi uwzględniać nierówność ostrą (<). Umieszczenie w przykładzie 4.6. warunku nieostrego (<=) wygeneruje błąd, ponieważ tablica nie posiada elementu o indeksie równym jej długości.

Programista nie musi znać rozmiaru tablicy (ani jej elementów), możliwe jest, że rozmiar zostanie odczytany dopiero w trakcie działania programu, co pokazuje kolejny przykład:

Przykład 4.7.

```
static void Main(string[] args)
{
    Console.WriteLine("Ile chcesz wpisać imion?");
    int rozmiar = Convert.ToInt32(Console.ReadLine());
    string[] imiona = new string[rozmiar];
    for (int i = 0; i < imiona.Length; i++)
    {
        Console.WriteLine("Podaj {0} imię", i+1);
        imiona[i] = Console.ReadLine();
    }
    for (int i = 0; i < imiona.Length; i++)
    {
        Console.Write(imiona[i] + ", ");
    }
    Console.ReadKey();
}
```

³⁷ Więcej o właściwościach w języku C# powiemy w [rozdziale 6](#).

Przykładowy wynik programu:

```
Ile chcesz wpisać imion?
3
Podaj 1 imię
Anna
Podaj 2 imię
Olga
Podaj 3 imię
Jan
Anna, Olga, Jan,
```

Program pyta użytkownika o liczbę imion, jakie zamierza wprowadzić. Na podstawie tej wartości (zapamiętanej w zmiennej *rozmiar*) deklarowana jest tablica o nazwie *imiona* (typu *string*). Następnie użytkownik wprowadza imiona. Na końcu, przy pomocy osobnej pętli (dla potwierdzenia), wyświetlone zostają w jednej linii wszystkie wprowadzone imiona.

Wykorzystanie pętli *for* w tablicach pozwala w bardzo dowolny sposób odwoływać się do elementów tablicy. Na przykład możliwe jest wyświetlenie wartości elementów tablicy od tyłu (rozpoczynając od elementu ostatniego, a kończąc na elemencie pierwszym). Prezentuje to poniższy przykład:

Przykład 4.8.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    for (int i = uczestnicy.Length - 1; i >= 0; i--)
        Console.WriteLine(uczestnicy[i]);
    Console.ReadKey();
}
```

Warto zauważyć, że w tym programie licznik pętli (zmienna *i*) inicjalizowany jest wartością o jeden mniejszą od długości tablicy. Zapobiega to próbie odwołania się do nieistniejącego elementu tablicy.

Indeksy tablicy są liczbami, dlatego nic nie stoi na przeszkodzie, aby określając wartość indeksu jednocześnie wykonywać na nim pewne operacje, co pokazuje kolejny przykład:

Przykład 4.9.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    for (int i = 1; i <= uczestnicy.Length; i++)
        Console.WriteLine(uczestnicy[i - 1]);
    Console.ReadKey();
}
```

W powyższym przykładzie zmienna *i* zmienia się od wartości równej 1 (wartość inicjalizowana) do wartości 5 (długość tablicy), dlatego też wyświetlenie wszystkich

elementów tablicy powoduje konieczność odjęcia od bieżącej wartości zmiennej *i* liczby 1. Gdyby w powyższym przykładzie licznik pętli *i* zainicjalizowano wartością równą 0, wygenerowany zostałby błąd, ponieważ numery indeksów tablicy nie mogą mieć wartości ujemnych (`uczestnicy[0-1]`). Natomiast gdyby w powyższym przykładzie nierówność nieostrą (`<=`) zamienić na ostrą (`<`), podczas wyświetlania pominięty zostałby ostatni element tablicy *uczestnicy*.

Ciekawy przykład może stanowić zadanie polegające na przypisaniu elementom jednej tablicy elementów tablicy drugiej, lecz w kolejności odwrotnej.

Przykład 4.10.

```
static void Main(string[] args)
{
    int[] uczestnicy = new int[] { 19, 34, 23, 54, 31 };
    int[] odwrotnie = new int[uczestnicy.Length];

    // Wpisywanie elementów do tablicy odwrotnie
    for (int i = uczestnicy.Length - 1; i >= 0; i--)
        odwrotnie[uczestnicy.Length - i - 1] = uczestnicy[i];

    // Wyświetlenie elementów tablicy odwrotnie
    for (int i = 0; i < odwrotnie.Length; i++)
        Console.WriteLine(odwrotnie[i]);
    Console.ReadKey();
}
```

W powyższym przykładzie tablica *uczestnicy* jest deklarowana oraz inicjalizowana wartościami. W kolejnym wierszu deklarowana jest druga tablica *odwrotnie*, która nie posiada zainicjalizowanych wartości, a jej długość jest równa długości tablicy *uczestnicy*. Dzięki takiemu rozwiązaniu ewentualna zmiana liczby elementów tablicy *uczestnicy* w sposób automatyczny przełoży się na odpowiednią zmianę długości tablicy *odwrotnie* zapewniając, że obie tablice będą miały taką samą liczbę elementów. Pętla *for* odczytuje elementy tablicy *uczestnicy* „od tyłu” (licznik zmniejsza się od wartości równej długości tablicy pomniejszonej o jeden aż do wartości równej 0), przepisując odczytane wartości do drugiej tablicy (*odwrotnie*) „od przodu”. Gdy zmniejsza się licznik pętli (czyli maleje wartość indeksu tablicy *uczestnicy*), rośnie wartość indeksu tablicy *odwrotnie*. W efekcie działania programu tablica *odwrotnie* będzie „lustrzanym odbiciem” tablicy *uczestnicy*. Będzie zawierać te same wartości elementów, jednak ułożone w odwrotnej kolejności.

Uwaga!

Korzystając z pętli do obsługi tablic należy szczególną uwagę zwrócić na warunki brzegowe działania pętli. Bardzo często popełniane błędy polegają na nieuwzględnieniu skrajnych elementów tablicy (szczególnie elementu ostatniego) lub niedozwolonym przekroczeniu dopuszczalnej wartości indeksu tablicy.

Aby zapobiec tego typu problemom, w języku C# dostępny jest czwarty rodzaj pętli – pętla *foreach*.

Pętla *foreach* („dla każdego”), jest przeznaczona do obsługi tzw. kolekcji, czyli zbiorów elementów. Jak już pisaliśmy, tablica jest jedną z kolekcji. Pętla *foreach* pozwala uzyskać kolejno dostęp do wszystkich elementów tablicy, odciążając programistę od konieczności numeracji indeksów i wynikających z niej problemów.

Poniższy przykład wyświetla zawartość wszystkich elementów tablicy *uczestnicy*.

Przykład 4.11.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    foreach (int x in uczestnicy)
        Console.WriteLine(x);
    Console.ReadKey();
}
```

W pętli *foreach* nie występuje znany z pętli *for* licznik pętli. W jego miejsce deklarowana jest pojedyncza zmienna (w tym przypadku nazwana *x*), która powinna być takiego samego typu, jak elementy tablicy wykorzystywanej w pętli, której nazwa pojawia się po słowie kluczowym *in*. Kompilator w sposób automatyczny na początku każdej iteracji pobiera wartość bieżącego elementu tablicy (rozpoczynając od pierwszego i przesuwając się w kolejnych iteracjach aż do elementu ostatniego) i przypisuje jego wartość zadeklarowanej w pętli zmiennej (*x*). Dlatego w środku pętli wystarczy wyświetlić jedynie wartość zmiennej (*x*), bez potrzeby odwoływania się do indeksów tablicy, czy też pobierania informacji o jej długości.

Pętla *foreach* jest bardzo wygodnym rozwiązaniem wykorzystywanym do obsługi tablic. Pętla ta posiada jednak pewne ograniczenie. Korzystając z tej pętli **nie ma możliwości wpisywania** wartości do tablicy. Pętla *foreach* umożliwia jedynie odczyt kolejno wszystkich elementów tablicy. W sytuacji, gdy konieczna jest zmiana wartości elementów tablicy, należy korzystać z pozostałych pętli (*for*, *do..while*, *while*).

Korzystając z pętli *foreach* warto pamiętać o następujących jej cechach:

- pętla *foreach* zawsze wykonuje tyle iteracji, ile jest elementów w tablicy (kolekcji),
- pętla *foreach* zawsze rozpoczyna przeglądanie tablicy od jej pierwszego elementu i kończy działanie na elemencie ostatnim. Jeżeli zachodzi konieczność zmiany tego kierunku, konieczne jest wykorzystanie innej pętli.
- Pętla *foreach* nie pozwala programiście na dostęp do numeru indeksu. Jeżeli wartość indeksu tablicy jest istotna, konieczne jest wykorzystanie innej pętli.
- Pętla *foreach* pozwala jedynie na odczyt wartości elementów tablicy. Jeżeli zachodzi konieczność ich modyfikowania, konieczne jest wykorzystanie innej pętli.

Do tego miejsca krok po kroku omówiliśmy, czym jest tablica oraz jak ją w języku C# można deklarować i odczytywać jej elementy przy pomocy różnych pętli. Wyszliśmy od przykładu z wieloma zmiennymi (przechowującymi wiek uczestników wycieczki). Pokazaliśmy „nieekonomiczne” rozwiązanie w [przykładzie 4.1](#), gdzie liczona została średnia z pięciu zmiennych. Teraz obliczymy średnią przy użyciu tablicy. Przykład ten (jak i kolejny) wykonamy przy użyciu pętli *for*, zachęcając Czytelnika, aby po przeanalizowaniu programu samodzielnie przerobił na wersję z pętlą *foreach*.

Przykład 4.12.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    int suma = 0;
    double srednia;
    Console.Write("Wiek uczestników: ");
    for (int i = 0; i < uczestnicy.Length; i++)
    {
        Console.Write("{0}, ", uczestnicy[i]);
        suma += uczestnicy[i];
    }
    srednia = (double)suma / uczestnicy.Length;
    Console.WriteLine();
    Console.WriteLine("Średnia: {0}", srednia );
    Console.ReadKey();
}
```

Wynik programu:

```
Wiek uczestników: 19, 34, 23, 54, 31,
Średnia: 32,2
```

W pętli *for* wyświetlane są elementy tablicy (w tej samej linii) oraz obliczana jest wartość zmiennej *suma*. Zmienna ta została przed pętlą zainicjalizowana wartością 0, a wewnątrz pętli – w każdym jej przebiegu dodawana jest do niej wartość elementu tablicy wyznaczonego przez indeks *i*.

Poszczególne przebiegi pętli przedstawia tabela:

Przebieg pętli (<i>i</i>)	Wartość zmiennej <i>suma</i>
0	<code>suma = suma + uczestnicy[0],</code> czyli <code>suma = 0 + 19</code>
1	<code>suma = suma + uczestnicy[1],</code> czyli <code>suma = 19 + 34</code>
2	<code>suma = suma + uczestnicy[2],</code> czyli <code>suma = 53 + 23</code>
3	<code>suma = suma + uczestnicy[3],</code> czyli <code>suma = 76 + 54</code>
4	<code>suma = suma + uczestnicy[4],</code> czyli <code>suma = 130 + 31</code>

Po wykonaniu pętli zmienna *suma* wynosi 161. Wartość ta jest dzielona przez liczbę uczestników (czyli rozmiar tablicy). Ponieważ zmienna *suma* jest typu *int* oraz liczba elementów tablicy również – zapis `suma/uczestnicy.Length` oznaczałby dzielenie całkowitoliczbowe, a wynikiem byłaby liczba 32. Aby uzyskać dokładną średnią (z miejscami po przecinku) musimy dokonać rzutowania typu. I tak jest zrobione w tym przykładzie. Przypominamy o tym jedynie, ponieważ te zagadnienia zostały już omówione w [punkcie 2.2.3.](#)

I na koniec tego podrozdziału pokażemy przykład, w którym znajduje się największa wartość w tablicy:

Przykład 4.13.

```
static void Main(string[] args)
{
    int[] uczestnicy = { 19, 34, 23, 54, 31 };
    int max = uczestnicy[0];    // tymczasowe maksimum
    for (int i = 1; i < uczestnicy.Length; i++)
    {
        if (uczestnicy[i] > max)
        {
            max = uczestnicy[i];
        }
    }
    Console.WriteLine("Najstarszy uczestnik ma {0} lat(a)", max);
    Console.ReadKey();
}
```

Po deklaracji i inicjalizacji tablicy deklarowana jest zmienna *max*, w której będzie maksymalny element tablicy (wiek najstarszego uczestnika). Inicjalizujemy tę zmienną tymczasowo wartością początkowego elementu tablicy (o indeksie 0). To tak, jakbyśmy przyjęli założenie, że największą wartością jest początkowy element tej tablicy, a następnie w kolejnych krokach będziemy to założenie weryfikować. Proszę zwrócić uwagę, że licznik pętli *for* jest tu inicjalizowany wartością 1 (a nie 0) – bo nie ma potrzeby porównywać

początkowego elementu tablicy ze samym sobą. Zatem przed wykonaniem pętli zmienna *max* jest równa 19. W tabeli przedstawimy kolejne przebiegi pętli:

Przebieg pętli (<i>i</i>)	Porównanie	Wartość zmiennej <i>max</i>
1	Czy uczestnicy[1] > 19, czyli czy 34 > 19 → Tak	max = 34 (nowa wartość max)
2	Czy uczestnicy[2] > 34, czyli czy 23 > 34 → Nie	max = 34 (bez zmian)
3	Czy uczestnicy[3] > 34, czyli czy 54 > 34 → Tak	max = 54 (nowa wartość max)
4	Czy uczestnicy[4] > 54, czyli czy 31 > 54 → Nie	max = 54 (bez zmian)

Ostatecznie po wyjściu z pętli w zmiennej *max* jest wartość 54 i jest to maksymalny element tej tablicy.

Przedstawione w tym podrozdziale przykłady prezentowały możliwości obsługi tablic jednowymiarowych, będących odpowiednikiem pojęcia wektora w matematyce. W dalszej części omówimy tablice wielowymiarowe.

4.1.2 Tablice dwuwymiarowe

Zmienne tablicowe wykorzystywane w języku C# (oraz w większości innych języków programowania) mogą posiadać więcej niż jeden wymiar. To znaczy, że każdy element tablicy może być reprezentowany przez więcej niż jeden indeks. Na przykład każdy element tablicy dwuwymiarowej będzie dostępny poprzez wartość dwóch indeksów. W sposób graficzny taką tablicę można zaprezentować jako macierz prostokątną:

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

Dostęp do każdego elementu powyższej tablicy możliwy jest po podaniu numeru wiersza oraz numeru kolumny. Na przykład wartość 8 znajduje się w drugim wierszu i trzeciej kolumnie. Przy czym należy pamiętać o tym, że numery indeksów rozpoczynają się od wartości 0, zatem indeksy dla tego elementu to 1 i 2.

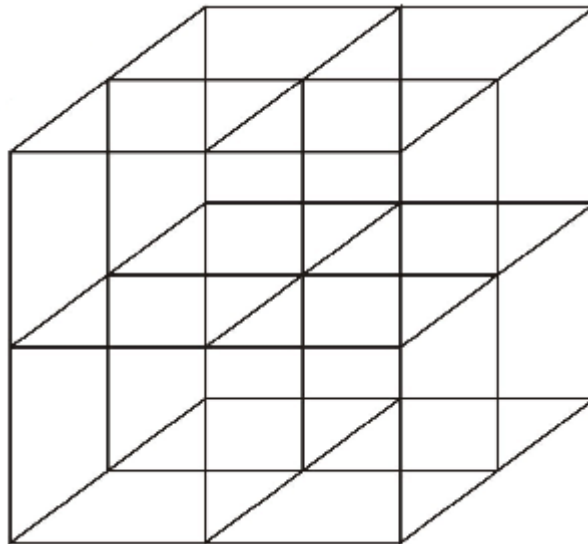
Działania związane z tablicami wielowymiarowymi (np. deklarowanie, inicjalizowanie, dostęp do poszczególnych elementów tablicy) wykonywane są w sposób analogiczny do obsługi tablic jednowymiarowych.

Poniżej jest przykładowa deklaracja tablicy dwuwymiarowej (nazywanej tablicą *prostokątną* lub *regularną*):

```
int[ , ] tablica_2d = new int [3,5];
```

Jedyną różnicą pomiędzy deklaracją tablicy jednowymiarowej i dwuwymiarowej jest przecinek pojawiający się w nawiasach kwadratowych i związana z tym konieczność podania dwóch rozmiarów – dla obu wymiarów tablicy (liczba wierszy, liczba kolumn).

Analogicznie możliwe jest zadeklarowanie tablicy trójwymiarowej, którą w sposób graficzny przedstawić można jako figurę przestrzenną:



Przykładowa deklaracja takiej tablicy może wyglądać następująco:

```
int[ , , ] tablica_3d = new int[2, 2, 2];
```

Nic nie stoi na przeszkodzie, aby w programie komputerowym wykorzystywać tablice o większej liczbie wymiarów. Tablicę taką zdecydowanie łatwiej zadeklarować niż przedstawić w sposób graficzny. Poniższa linia kodu deklaruje tablicę 5-cio wymiarową.

```
int[ , , , , ] tablica_5d = new int[4,6,7,3,5];
```

Deklarując zmienną tablicową należy mieć na uwadze wielkość pamięci, jaka zajmowana będzie przez zmienną. Na przykład 10-cio elementowa tablica jednowymiarowa liczb typu *int* zajmie w pamięci komputera 40 bajtów (10 x 32 bity / 8). Tablica dwuwymiarowa [10,10] pozwalająca na przechowanie 100 wartości typu *int* zajmie 400 bajtów. Tablica trójwymiarowa [2,2,2] (np. deklarowana wyżej tablica o nazwie *tablica_3d*)

wykorzystywać będzie 32 bajty, a zmienna tablicowa dla tablicy 5-cio wymiarowej (deklarowana wyżej jako *tablica_5d*) – 10 080 bajtów.

Kolejne przykłady związane z tablicami wielowymiarowymi ograniczać będą się jedynie do tablic dwuwymiarowych. Jak jednak pokazują dotychczasowe przykłady deklarowania tablic, w przypadku większej liczby wymiarów postępować należy analogicznie.

Inicjalizowanie wartości elementów tablicy dwuwymiarowej umożliwia poniższy zapis:

```
int[,] tablica_2d = new int[,] { { 1, 2 },
                                { 3, 4 },
                                { 5, 6 },
                                { 7, 8 } };
```

lub krócej:

```
int[,] tablica_2d = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Zadeklarowana w ten sposób tablica ma rozmiar [4,2] (czyli 4 wiersze i 2 kolumny). Dostęp do poszczególnych elementów takiej tablicy możliwy jest po podaniu wartości obu indeksów, np.:

```
Console.WriteLine(tablica_2d [2,0]);
Console.WriteLine(tablica_2d [2,1]);
Console.WriteLine(tablica_2d [1,1]);
```

Wyświetlenie wszystkich elementów tablicy możliwe jest poprzez wykorzystanie pętli *foreach*:

Przykład 4.14.

```
static void Main(string[] args)
{
    int[,] tablica_2d = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
    foreach (int x in tablica_2d)
        Console.WriteLine(x);
    Console.ReadKey();
}
```

Jednak, aby wyświetlić wartości elementów tablicy w sposób uwzględniający odpowiednie wiersze i kolumny konieczne jest wykorzystanie innych pętli, np. pętli *for*.

Przykład 4.15.

```
static void Main(string[] args)
{
    int[,] tablica_2d = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
    for (int a = 0; a < 4; a++)
    {
        for (int b = 0; b < 2; b++)
        {
            Console.Write("{0,3}", tablica_2d[a, b]);
        }
        Console.WriteLine();
    }
    Console.WriteLine("Rozmiar: " + tablica_2d.Length);
    Console.ReadKey();
}
```

Powyższy przykład wykorzystuje w wyrażeniach logicznych literały 4 i 2 (ponieważ zadeklarowana tablica ma rozmiar [4,2], indeks *a* zmienia wartość od 0 do 3, a indeks *b* od 0 do 1). Jak już jednak wspomniano, należy wykorzystywać możliwości języka C#, tak aby zmiany wykonywane w jednym miejscu kodu źródłowego nie wymuszały modyfikacji wielu innych miejsc. Dlatego w kilku innych przykładach proponowaliśmy wykorzystywanie właściwości zmiennej tablicowej, która podaje jej rozmiar (*tablica.Length*), zob. [przykład 4.6](#). W przypadku tablic wielowymiarowych informacja o długości (rozmiarze) tablicy jest jedną liczbą, która wskazuje całkowitą liczbę elementów tablicy (np. dla tablicy zainicjalizowanej w powyższym przykładzie właściwość *tablica_2d.Length* wynosi 8). Czy istnieje więc możliwość wyświetlenia za pomocą pętli *for* wszystkich elementów tablicy, ale w taki sposób, by nie podawać górnych granic obu indeksów jako literałów?

Można w tym celu wykorzystać metodę *GetLength()*. Metoda ta zwraca liczbę elementów w danym wymiarze. Trzeba jedynie w nawiasach okrągłych wpisać numer wymiaru (numeracja od 0), jak w prezentowanym przykładzie:

Przykład 4.16.

```
static void Main(string[] args)
{
    int[,] tablica_2d = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
    for (int a = 0; a < tablica_2d.GetLength(0); a++)
    {
        for (int b = 0; b < tablica_2d.GetLength(1); b++)
        {
            Console.Write("{0,3}", tablica_2d[a, b]);
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}
```

Dla tablicy dwuwymiarowej *GetLength(0)* zwraca długość (rozmiar) pierwszego wymiaru, czyli liczbę wierszy, natomiast *GetLength(1)* zwraca długość drugiego wymiaru, czyli liczbę kolumn

4.1.3 Tablice postrzępione

W języku C# obok tablic jednowymiarowych i wielowymiarowych dostępne są także tzw. **tablice postrzępione** (ang. jagged array), nazywane również tablicami nieregularnymi. Tablica postrzępiona jest tablicą, której elementy są tablicami (czyli jest „tablicą tablic”). Elementy postrzępionej tablicy mogą być różnych wymiarów i rozmiarów.

W tablicy dwuwymiarowej (prostokątnej) każdy wiersz ma taką samą liczbę elementów. W tablicy postrzępionej liczba elementów w poszczególnych wierszach może być różna, np.:

1	2		
3	4	5	6
7	8	9	

Można więc powiedzieć, że powyższa tablica postrzępiona jest tablicą tablic jednowymiarowych o różnych długościach. Sformułowanie to ma swoje odzwierciedlenie w zapisie deklaracji takiej tablicy:

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

Taką samą tablicę postrzępioną można zadeklarować w inny sposób:

```
int[][] tab = new int[3][];
tab[0] = new int[2];
tab[1] = new int[4];
tab[2] = new int[3];
```

Deklaracja tablicy postrzępionej wraz z inicjalizowaniem jej wartości możliwa jest w następujący sposób:

```
int[][] tab =
{
    new int[] {1,2},
    new int[] {3,4,5,6},
    new int[] {7,8,9}
};
```

Podobnie jak w przypadku każdej tablicy w języku C# - wszystkie elementy tablicy postrzępionej muszą być tego samego typu.

Sposób dostępu do poszczególnych elementów tablicy postrzępionej również wskazuje na to, że jest to tablica tablic:

Przykład 4.17.

```
static void Main(string[] args)
{
    int[][] tab =
    {
        new int[] {1,2},
        new int[] {3,4,5,6},
        new int[] {7,8,9}
    };
    Console.WriteLine(tab[0][0]); // wypisze 1
    Console.WriteLine(tab[1][2]); // wypisze 5
    Console.WriteLine(tab[2][2]); // wypisze 9
    Console.ReadKey();
}
```

Uzyskanie informacji o liczbie elementów poszczególnych wierszy tablicy postrzępionej możliwe jest w następujący sposób:

Przykład 4.18.

```
static void Main(string[] args)
{
    int[][] tab =
    {
        new int[] {1,2},
        new int[] {3,4,5,6},
        new int[] {7,8,9}
    };
    Console.WriteLine(tab[0].Length); // wypisze 2
    Console.WriteLine(tab[1].Length); // wypisze 4
    Console.WriteLine(tab[2].Length); // wypisze 3
    Console.ReadKey();
}
```

Podobnie jak w przypadku tablic wielowymiarowych, również w tablicach postrzępionych istnieje możliwość dodania „kolejnego wymiaru”. Ponieważ jednak tablica postrzępiona jest tablicą tablic, bardziej poprawne będzie określenie mówiące o kolejnym poziomie zagnieżdżenia tablicy. Popatrzmy na przykład:

Przykład 4.19.

```
static void Main(string[] args)
{
    int[][][] tab =
    {
        new int[][]
        {
            new int[] {1,2},
            new int[] {3,4,5}
        }
    };
    Console.Write(tab[0][1][2]); // wypisze 5
    Console.ReadKey();
}
```

Dostęp do poszczególnych elementów takiej tablicy jak powyższym przykładzie wymaga wskazania trzech indeksów (np. `tab[0][1][2]`).

Deklarację tablicy postrzępionej można jeszcze bardziej komplikować, np. poprzez wykorzystanie na różnych poziomach zagnieżdżenia tablic o różnych wymiarach, co pokazuje kolejny przykład:

Przykład 4.20.

```
static void Main(string[] args)
{
    int[,][,] tab =
    {
        new int[,] { {1,2}, {3,4} },
        new int[,] { {5,6,7}, {8,9,10} }
    };
    Console.WriteLine(tab[1][0, 2]); // wypisze 7
    Console.ReadKey();
}
```

Wyświetlenie wartości elementu tej tablicy wymaga odwołania się w indeksie pierwszym do tablicy jednowymiarowej, a w indeksie drugim do tablicy dwuwymiarowej.

Wyświetlanie zawartości całej tablicy postrzępionej umożliwia pętla *foreach* przedstawiona poniżej.

Przykład 4.21.

```

static void Main(string[] args)
{
    int[][] tab =
    {
        new int[] {1,2},
        new int[] {3,4,5,6},
        new int[] {7,8,9}
    };
    foreach (int[] podtablica in tab)
    {
        foreach (int x in podtablica)
        {
            Console.Write("{0,2}", x);
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}

```

Przykład ten uwidacznia to, o czym już pisaliśmy, że tablica postrzępiona jest tablicą tablic. Wyświetlenie zawartości całej tablicy realizowane jest dzięki zagnieżdżonym pętlom. Deklarowana w pętli zewnętrznej zmienna *podtablica* sama jest tablicą (świadczą o tym nawiasy kwadratowe umieszczone za typem *int*). Pętla wewnętrzna *foreach* (deklarująca zmienną *x*) funkcjonuje tak, jak w dotychczas prezentowanych przykładach.

Postrzeganie tablicy postrzępionej jako tablicy tablic uwidacznia także kolejny przykład, w którym wyświetlenie elementów tablicy postrzępionej odbywa się przy pomocy pętli *for* oraz właściwości *Length* (jakiej używaliśmy w tablicach jednowymiarowych):

Przykład 4.22.

```

static void Main(string[] args)
{
    string[][] zespoly = {
        new string[] { "Adam", "Karol" },
        new string[] { "Ola", "Ela", "Jan" } };
    for (int i = 0; i < zespoly.Length; i++)
    {
        for (int j = 0; j < zespoly[i].Length; j++)
        {
            Console.Write("{0,-10}", zespoly[i][j]);
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}

```

Użycie właściwości *Length* dla „tablicy tablic”, czyli *zespoly.Length*, zwraca liczbę podtablic. Użycie tej właściwości dla każdej podtablicy zwraca liczbę jej elementów (np. *zespoly[0].Length* wynosi 2 – bo są dwa elementy w pierwszej podtablicy).

Tablice postrzępione przydatne są dla zbiorów danych o nieregularnej strukturze. W powyższym przykładzie w tablicy postrzępionej umieszczone są imiona członków zespołów, a liczba osób w poszczególnych zespołach nie jest taka sama.

4.1.4 Wybrane metody klasy *Array*

Każda tablica w języku C# dziedziczy po klasie *Array*. Wstępne informacje o dziedziczeniu znajdują się w rozdziale 6, tu wystarczy przyjąć, że dzięki temu mamy do dyspozycji szereg użytecznych mechanizmów do manipulacji na tablicach. To właśnie dzięki klasie *Array* mogliśmy korzystać z właściwości *Length* oraz metody *GetLength()* do określenia rozmiaru tablicy. Metody tej klasy pozwalają także przekopiować fragmenty tablicy, posortować ją czy przeszukać.

Obok właściwości *Length* dostępna jest w tej klasie także właściwość *Rank*, która zwraca liczbę wymiarów danej tablicy. Np. dla tablicy o deklaracji *int[,] tab = new int[4,5]*; właściwość *Rank* zwróci liczbę 2 (tablica dwuwymiarowa). Właściwość ta jest przydatna jedynie w przypadku tablic regularnych, ponieważ w przypadku tablic postrzępionych zwraca wartość 1 (tablica postrzępiona widziana jest jako jednowymiarowa tablica tablic).

W dalszej części przedstawimy kilka przydatnych metod klasy *Array*. Zaczniemy od metody *Copy()*, która umożliwia skopiowanie fragmentu jednej tablicy do drugiej:

Przykład 4.23.

```
static void Main(string[] args)
{
    int[] a = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
    int[] b = new int[10];
    Array.Copy(a, 2, b, 3, 5);
    foreach (int x in b)
    {
        Console.Write("{0}, ", x);
    }
    Console.ReadKey();
}
```

Wynik programu:

```
0, 0, 0, 33, 44, 55, 66, 77, 0, 0,
```

Efektem działania powyższego programu jest skopiowanie pięciu (na co wskazuje ostatni argument metody *Copy()*) elementów tablicy *a* do tablicy *b*. Kopiowanie elementów z tablicy źródłowej rozpocznie się od elementu o indeksie 2, elementy te zostaną wstawione do

tablicy docelowej, rozpoczynając od indeksu 3 (pozostałe elementy tablicy *b* są wypełniane zerami, bo 0 jest wartością domyślną dla elementów tablicy typu *int*). Warto zapoznać się z innymi trzema wariantami wykorzystania tej metody³⁸.

[Przykład 4.10](#) umożliwiał odwrócenie elementów tablicy za pomocą pętli. Klasa *Array* oferuje wygodniejsze rozwiązanie w postaci metody *Reverse()*:

Przykład 4.24.

```
static void Main(string[] args)
{
    int[] tab = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Array.Reverse(tab);
    foreach (int x in tab)
        Console.Write("{0,2}", x);
    Console.ReadKey();
}
```

Bardzo często zachodzi potrzeba posortowania elementów tablicy. W klasie *Array* można użyć do tego celu metody *Sort()*, która w jednym z wariantów sortuje elementy tablicy domyślnie w porządku rosnącym, jak w przykładzie:

Przykład 4.25.

```
static void Main(string[] args)
{
    int[] tab = { 4, 2, 6, 23, 1, 3, 7, 0 };
    Array.Sort(tab); // sortowanie tablicy
    for (int i = 0; i < tab.Length; i++)
        Console.WriteLine(tab[i]);
    Console.ReadKey();
}
```

Przydatna może być także metoda *IndexOf()*, która zwraca indeks pierwszego wystąpienia szukanego elementu. Popatrzmy na przykład:

Przykład 4.26.

```
static void Main(string[] args)
{
    // elementy tablicy   0       1       2       3       4
    string[] imiona = { "Ala", "Ola", "Ela", "Tola", "Ela" };
    Console.WriteLine(Array.IndexOf(imiona, "Ela")); // wypisze 2
    Console.WriteLine(Array.IndexOf(imiona, "Iza")); // wypisze -1
    Console.ReadKey();
}
```

³⁸ O metodzie, która występuje w różnych wariantach mówi się, że to jest metoda przeciążana (lub przeładowana). Przeciążone metody mają tę samą nazwę, ale różnią się listą argumentów. Ich działanie jest podobne, ale nieidentyczne. Więcej na ten temat w [podrozdziale 5.8](#).

Metoda *IndexOf()* dla wyszukanego w tablicy elementu „Ela” zwraca indeks 2 (czyli pozycję pierwszego wystąpienia). Natomiast dla tekstu „Iza”, którego nie ma w tej tablicy, zwraca wartość -1. Podobne działanie ma metoda *LastIndexOf()*, która zwraca ostatnie wystąpienie szukanego elementu. W obu tych metodach należy podać szukany element zgodnie z typem tablicy.

Metody *Reverse()*, *Sort()*, *IndexOf()* oraz *LastIndexOf()* stosować można jedynie dla tablic jednowymiarowych.

Umiejętność korzystania z tablic (dowolnego typu) w powiązaniu z możliwościami, jakie zawierają zaimplementowane w języku C# metody pozwala programistom szybko tworzyć kod programu i skupiać się na istotnych elementach ich działania, a nie na poszczególnych czynnościach, takich jak sortowanie tablic, wyszukiwanie tekstu itp. W języku C# są dostępne także inne kolekcje (takie jak listy, kolejki czy słowniki), które oferują więcej możliwości, a obsługiwane są w podobny sposób, jak tablice. Streszczenie informacji na temat tablic znajduje się w [Tabeli 10](#) w *Dodatkach*.

4.2 Operacje na tekstach

Pogłębimy w tym podrozdziale umiejętności w zakresie operacji na tekstach, czyli danych typu *string*. Nazwa typu *string* (z małej litery „s”) to alias dla klasy *String* (pisanej z dużej litery „S”). Skoro alias (alternatywna nazwa), to można powiedzieć, że to jest to samo. Na ogół jednak wersji *string* używa się do deklaracji zmiennej, natomiast do wywołania statycznych metod³⁹ dla tej klasy używa się oryginalnej nazwy klasy, czyli *String*. Jak już pisaliśmy, typ *string* jest typem referencyjnym, takim dość szczególnym, bo pozwala pracować z danymi w taki sposób, że można chwilami zapomnieć o tym, że jest typem referencyjnym. Podczas deklaracji danej tego typu nie używa się operatora *new*⁴⁰, nie trzeba nigdzie deklarować, jaki będzie maksymalny rozmiar deklarowanego łańcucha znakowego. Ot tak sobie można wykorzystywać ten typ, nie martwiąc się o takie szczegóły. W tym podrozdziale wyraźniej odczujemy fakt, iż *string* jest typem referencyjnym. Po pierwsze zobaczymy, że możemy do poszczególnych znaków zadeklarowanego łańcucha znakowego odnosić się, tak, jakby łańcuch ten był tablicą znaków. Bo tak właśnie wygląda typ *string* „od kuchni” – tekst jest przechowywany w tablicy znaków. W klasie *String* zadbano o wiele różnych szczegółów, tak aby programowanie w zakresie pracy z tekstami było wygodniejsze. Klasa ta oferuje szereg użytecznych metod ułatwiających operowanie na tekstach, poznamy wybrane z nich.

³⁹ Tworzenie metod i ich rodzaje poznamy wkrótce, [w rozdziale 5](#).

⁴⁰ Jakbyśmy bardzo chcieli, to możemy użyć operatora *new*, wyglądałoby to np. tak: `char[] tab = { 'p', 'r', 'o', 'g', 'r', 'a', 'm' }; String x = new String(tab);` - po tych instrukcjach zmienna *x* będzie zawierać łańcuch znakowy „program”.

4.2.1 Tekst jako tablica znaków

Większość przedstawionych w poprzednim podrozdziale tablic zawierała liczby. Oczywiście istnieje możliwość tworzenia tablic posiadających elementy innego typu np. logicznego lub znakowego. Ciekawe możliwości niesie ze sobą fakt, że w języku C# ciągi znaków (dane typu *string*) mogą być traktowane jako tablice znaków, co pokazuje przykład:

Przykład 4.27.

```
static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    Console.WriteLine(tekst[0]);    // wypisze "A"
    Console.WriteLine(tekst[4]);    // wypisze "m"
    Console.WriteLine(tekst[7]);    // wypisze "k"
    Console.ReadKey();
}
```

Wszystkie zasady obsługi tablic jednowymiarowych odnoszą się również do ciągów znaków. Poniższy przykład wyświetla tekst po jednym znaku w każdej linii:

Przykład 4.28.

```
static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    foreach (char litera in tekst)
        Console.WriteLine(litera);
    Console.ReadKey();
}
```

Korzystając z faktu, że ciągi znakowe mogą być traktowane jak tablice, w łatwy sposób można wyświetlić tekst od tyłu, co obrazuje kolejny przykład:

Przykład 4.29.

```
static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    for (int i = tekst.Length - 1; i >= 0; i--)
        Console.Write(tekst[i]);
    Console.ReadKey();
}
```

Czasami w programie komputerowym zachodzi konieczność policzenia, ile razy w danym tekście występuje konkretna litera (lub inny znak). Wykonamy takie zadanie przy użyciu pętli *foreach*:

Przykład 4.30.

```
static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    int liczba_znakow = 0;
    foreach (char litera in tekst)
        if (litera == 'a') liczba_znakow++;
    Console.WriteLine("Litera a wystąpiła {0} razy", liczba_znakow);
    Console.ReadKey();
}
```

W powyższym przykładzie należy zwrócić uwagę na warunek instrukcji *if*. Zmienna *tekst* typu *string* traktowana jest jak tablica znaków. W języku C# znaki (czyli wartości typu *char*) powinny być umieszczane pomiędzy apostrofami (w odróżnieniu od wartości typu *string*, które powinny być ograniczane znakami cudzysłowu, np. "a" to jest łańcuch znakowy złożony z jednego znaku). Podkreślić należy również, że mała litera 'a' jest zupełnie innym znakiem niż wielka litera 'A' (zob. wykaz kodów ASCII w [Tabeli 5](#) w *Dodatkach*).

4.2.2 Wybrane metody klasy *String*

String to nazwa typu zmiennej tekstowej (częściej używa się aliasu *string* pisanego z małej litery „s”), lecz przede wszystkim to nazwa klasy, która zgodnie z zasadami obiektowości posiada pola i metody, czyli udostępnia informacje o tekście, a także umożliwia wykonywanie na nim operacji. Tworzenie metod i klas to temat dwóch kolejnych rozdziałów. Tu jednak będziemy doskonalić użytkowanie gotowych klas i metod. Klasa *String* jest dobrym poligonem doświadczalnym do tego celu.

Często w programie komputerowym zachodzi potrzeba pobrania („wycięcia”) z dłuższego tekstu jego fragmentu (np. z tekstu o długości 11 znaków trzeba pobrać 6 kolejnych począwszy od znaku piątego). Efekt ten można uzyskać korzystając z dowolnej pętli. Wygodniejsze jednak będzie wykorzystanie metody *Substring()*.

Przykład 4.31.

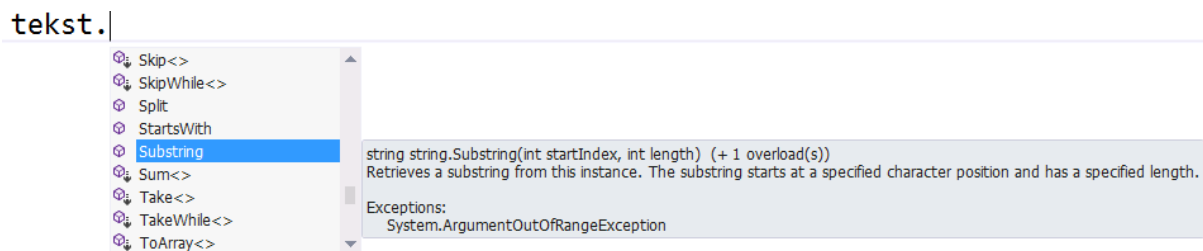
```
static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    string fragment;
    fragment = tekst.Substring(4, 6);
    Console.WriteLine(fragment);    // wypisze "ma kot"
    Console.ReadKey();
}
```

W powyższym przykładzie deklarowane są zmienne tekstowe, a następnie do zmiennej o nazwie *fragment* wpisywany jest efekt działania metody *Substring()*. Pierwszy argument tej metody wskazuje indeks (pozycję) znaku, od którego rozpocznie się operacja pobrania tekstu. Drugi argument określa liczbę kolejnych znaków, które zostaną pobrane. W efekcie działania

omawianego przykładu w oknie konsoli zostanie wyświetlony tekst: „ma kot” czyli znaki o numerach 4,5,6,7,8,9 (numeracja znaków od 0, jak w przypadku elementów tablic).

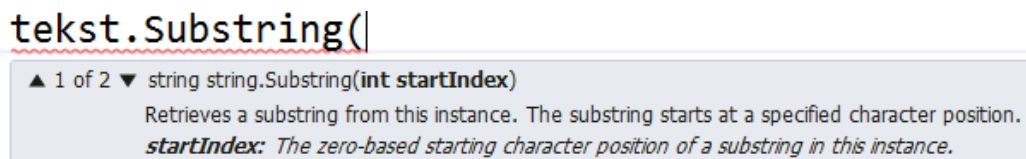
Metoda *Substring()* może być użyta także w nieco inny sposób. Jeżeli zostanie podany tylko jeden jej argument, wtedy zostanie pobrany fragment tekstu od znaku podanego jako argument aż do końca tekstu źródłowego. Opisywaną metodę można wykorzystać na dwa sposoby, istnieją jednak metody, które posiadają nawet kilkanaście różnych wariantów działania. Uczenie się na pamięć tych możliwości nie ma żadnego sensu. Należy skorzystać z informacji, które w czytelny sposób prezentuje edytor kodu źródłowego środowiska Visual Studio.

Gdy w edytorze zostanie wpisana nazwa zmiennej typu *string*, a następnie bezpośrednio po niej pojawi się znak kropki, edytor wyświetli listę wszystkich pól i metod dostępnych do wykorzystania. Po wybraniu dowolnej pozycji zostanie wyświetlony jej opis. Informację prezentowaną przez edytor kodu źródłowego dla metody *Substring()* przedstawia poniższy rysunek.

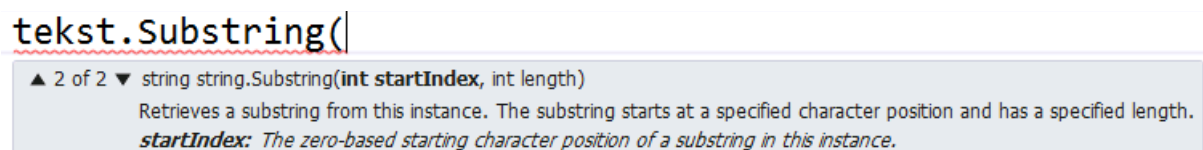


Prezentowana informacja umożliwia zapoznanie się z przeznaczeniem wybranej metody oraz co istotne, określa typ zwracanej wartości. Metoda *Substring()* zwraca fragment tekstu, (czyli daną typu *string*).

Dodatkowo po wybraniu metody z listy i wprowadzeniu z klawiatury znaku otwierającego nawias półokrągły edytor prezentuje dalsze szczegółowe informacje:



Z prezentowanej informacji można odczytać liczbę różnych sposobów wykorzystania danej metody, a także szczegółowy opis wybranego wariantu. I tak dla metody *Substring()* pierwszy z dwóch wariantów wymaga podania jednego argumentu (typu *int*) – numeru znaku, od którego rozpocznie się pobieranie tekstu.



Drugi wariant wykorzystania metody *Substring()* wymaga podania dwóch argumentów – indeksu, od którego rozpocznie się pobranie oraz długości „wycinanego” tekstu (oba argumenty powinny być typu *int*).

Umiejętność odczytywania prezentowanych przez edytor informacji jest kluczowa dla sprawnego posługiwania się dużą liczbą dostępnych pól i metod, z jeszcze większą liczbą możliwych wariantów ich zastosowania. Informacje te są nieocenioną pomocą w czasie nauki programowania, dlatego warto zwracać na nie uwagę (no i oczywiście doskonalić swój język angielski).

Kolejną użyteczną metodą jest *Compare()*. Pozwala ona na porównanie dwóch tekstów.

Przykład 4.32.

```
static void Main(string[] args)
{
    int wynik;
    string tekst1 = "Kowalski";
    string tekst2 = "Nowak";
    wynik = String.Compare(tekst1, tekst2); // wypisze -1
    Console.WriteLine(wynik);
    Console.ReadKey();
}
```

Warto zauważyć, że metoda *Compare()* nie jest wywoływana poprzez nazwę zmiennej (jak to miało miejsce w przypadku metody *Substring()*), lecz nazwę klasy (*String*). Wynikiem metody *Compare()* jest liczba typu *int* równa 0, gdy dwa porównywane teksty są takie same lub inna wartość liczbową (różna od 0), gdy teksty różnią się od siebie⁴¹. Jak można odczytać z prezentowanej przez edytor informacji, istnieje 10 różnych wariantów użycia metody *Compare()*, między innymi metoda ta pozwala ignorować wielkość znaków porównywanych tekstów. Gdy interesuje nas jedynie proste sprawdzenie, czy teksty są równe, zamiast metody *Compare()* można użyć operatora równości `==`.

Operacja konkatencji (czyli łączenia ciągów znakowych) realizowana może być za pomocą metody *Concat()*:

⁴¹ Gdy tekst pierwszy jest „mniejszy” od tekstu drugiego metoda *Compare()* zwraca wartość ujemną, gdy tekst pierwszy jest „większy” od drugiego metoda zwraca wartość dodatnią. Porządek alfabetyczny pozwala stwierdzić kompilatorowi, który tekst jest „większy” bądź „mniejszy”. W omawianym przykładzie tekst pierwszy jest „mniejszy” wg kolejności alfabetycznej od tekstu drugiego, dlatego metoda *Compare()* zwraca wartość ujemną.

Przykład 4.33.

```
static void Main(string[] args)
{
    string tekst_zlaczony;
    string tekst1 = "Ala ma kota";
    string tekst2 = " i psa";
    tekst_zlaczony = string.Concat(tekst1, tekst2);
    Console.WriteLine(tekst_zlaczony);
    Console.ReadKey();
}
```

Operację konkatencji możemy także wykonać, tak jak już to nieraz robiliśmy, przy użyciu operatora „+”, np. tekst1 + tekst2.

Aby przekonać się, czy pewien fragment tekstu zawiera się w innym tekście można skorzystać z metody *IndexOf()*. Popatrzmy na przykład:

Przykład 4.34.

```
static void Main(string[] args)
{
    string tekst = "być albo nie być";
    Console.WriteLine(tekst.IndexOf("być")); // wypisze 0
    Console.ReadKey();
}
```

Zwracana przez metodę *IndexOf()* wartość nieujemna wskazuje, od którego znaku w zmiennej tekstowej rozpoczyna się pierwsze wystąpienie poszukiwanego fragmentu. Gdy fragment tekstu nie zostanie znaleziony, metoda zwraca wartość równą -1. Metodą podobną w działaniu jest *LastIndexOf()*, która szuka zadanego fragmentu od końca. I tak dla tekstu z omawianego przykładu wywołanie *tekst.IndexOf("być")* zwróci 0 (pierwsze wystąpienie „być” w tym tekście zaczyna się od pozycji 0), natomiast wywołanie *tekst.LastIndexOf("być")* zwróciłoby 13 (ostatnie wystąpienie „być” zaczyna się od pozycji 13). Przypominamy o numeracji łańcucha od 0.

A co, jeśli potrzebowalibyśmy przeanalizować wszystkie wystąpienia szukanego fragmentu? Wówczas metodę *IndexOf()* lub *LastIndexOf()* musielibyśmy umieścić w pętli, popatrzmy na takie rozwiązanie:

Przykład 4.35.

```
static void Main(string[] args)
{
    string tekst = "być albo nie być";
    int pozycja, start = 0;
    Console.WriteLine("0123456789012345"); // pomocniczo, aby numerować znaki
    Console.WriteLine(tekst);
    Console.WriteLine("szukany tekst \"być\" jest na pozycjach:");
```

```

while ((pozycja = tekst.IndexOf("być", start)) >= 0)
{
    Console.Write("{0}, ", pozycja);
    start = pozycja + 3;    // dalsze szukanie 3 znaki dalej (bo "być" ma 3 znaki)
}
Console.ReadKey();
}

```

Wynik programu:

```

0123456789012345
być albo nie być
szukany tekst "być" jest na pozycjach:
0, 13,

```

Takie rozwiązanie pozwoli znaleźć wszystkie wystąpienia szukanego tekstu. W przykładzie są tylko dwa wystąpienia słowa „być”, ale gdyby ich było więcej, ten program też działałby prawidłowo. W programie wyjaśnienia wymaga jedynie pętla *while*. Ponieważ możemy testować różne teksty, nie możemy założyć, że ma miejsce choć jedno wystąpienie szukanego tekstu. Dlatego lepiej jest użyć pętli *while*, która przed pierwszym przebiegiem sprawdzi, czy jest jakieś wystąpienie zadanego fragmentu. Tu używamy takiej wersji metody *IndexOf()*, która ma dwa argumenty, pierwszy to szukany tekst, a drugi to pozycja, od której należy zacząć przeszukiwanie. Ustalamy początkowo wartość tej pozycji (poprzez zmienną *start*) na 0. Ale w każdym nowym przebiegu pętli zmienna *start* przyjmuje nową wartość (powiększoną o długość szukanego tekstu). Jeszcze krótkie wyjaśnienie odnośnie zapisu:

```
while( (pozycja = tekst.IndexOf("być", start)) >= 0)
```

Pamiętamy, że w pętli *while* musi być wyrażenie logiczne. I tak tu jest. Wspominaliśmy omawiając pętlę *while*, że można dokonać przypisania w wyrażeniu logicznym, ale operacja ta musi być ujęta w nawiasach okrągłych. Najpierw wołana jest tu metoda *IndexOf()*, jej rezultat jest przypisany do zmiennej *pozycja*, a dopiero później zmienna *pozycja* jest porównywana z zerem (czy jest większa lub równa 0).

Wstawienie fragmentu tekstu do istniejącego łańcucha znaków umożliwia metoda *Insert()*, co pokazuje przykład:

Przykład 4.36.

```

static void Main(string[] args)
{
    string tekst = "Ala ma kota";
    string nowy_tekst;
    nowy_tekst = tekst.Insert(7, "kanarka i ");
    Console.WriteLine(nowy_tekst);
    Console.ReadKey();
}

```

Pierwszy argument metody *Insert()* wskazuje miejsce (numer znaku), w które zostanie wstawiony tekst, będący drugim argumentem tej metody. Powyższy przykład powoduje zapisanie w zmiennej *nowy_tekst* ciągu znaków „Ala ma kanarka i kota”.

Pracując ze zmiennymi typu *string* warto poznać możliwości innych metod klasy *String* takich jak: *Remove()*, *ToLower()*, *ToUpper()*, *Trim()*, *Join()*. Skrócony opis wybranych metod klasy *String* zawiera [Tabela 11](#) (Dodatki).

Do obsługi łańcuchów znakowych można także zastosować inną klasę – *StringBuilder*. Klasa *String* ma pewną „ukrytą wadę”. Każda wartość obiektu tej klasy jest „niezmienna” (ang. *immutable*). To znaczy, że raz wpisana do określonego miejsca w pamięci pozostaje tam bez zmian lub jest usuwana. A programista, gdy zmienia w programie wartość zmiennej łańcuchowej – może nie wiedzieć, że w niewidocznym dla jego oczu miejscu – pamięci komputera zmieniony łańcuch tworzony jest tak naprawdę w nowym miejscu. Dopisanie jednej litery do tekstu mającego 1000 znaków nie oznacza, że się dopisuje w pamięci tylko ta nowa litera. Cały nowy łańcuch wpisuje się do nowego miejsca w pamięci. Ponieważ to typ referencyjny, po takiej podmianie na stosie zmienia się zawartość referencji (czyli naszej zmiennej typu *string*) – będzie wskazywać już na inne miejsce w pamięci. Jeśli przykładowo w pętli należy wykonać wiele zmian jednego dużego tekstu to praca z klasą *String* będzie mało ekonomiczna. Klasa *StringBuilder* jest pod tym względem bardziej wydajna. Nie będziemy jej tu omawiać, bo klasa *String* na obecnym etapie nauki jest wystarczająca i mimo swej „ukrytej wady” bardzo powszechna oraz użyteczna w typowych operacjach na łańcuchach (nie wymagających częstych zmian dużego tekstu). Niemniej zachęcamy Czytelnika, aby po przeczytaniu tego podręcznika samodzielnie zapoznał się z klasą *StringBuilder*.

4.3 Zadania do samodzielnego rozwiązania

Zadanie 4.1.

Napisz program, który pozwoli zapełnić n -elementową tablicę jednowymiarową liczb całkowitych wartościami podanymi przez użytkownika. Na początku działania programu użytkownik podaje liczbę elementów tablicy, a następnie poszczególne wartości jej elementów. Po wypełnieniu całej tablicy program powinien wypisać je w oknie konsoli.

Zadanie 4.2.

Napisz program kopiujący z danej tablicy liczb całkowitych *tab1* do nowej tablicy *tab2* wyłącznie wartości dodatnie. Obie tablice mają być jednowymiarowe o rozmiarze równym 10 (czyli 10-elementowe). Elementy pierwszej tablicy (*tab1*) należy wpisać w trakcie deklaracji tej tablicy.

Zadanie 4.3.

Napisz program wyświetlający informacje o wypełnionej przez użytkownika tablicy n -elementowej:

- wartość i pozycja największego elementu,
- wartość i pozycja najmniejszego elementu,
- średnia wartości wszystkich elementów tablicy,
- liczba wartości dodatnich w tablicy.

Zadanie 4.4.

Napisz program, który podaje, ile jest liczb pierwszych w tablicy 100 elementowej typu *int*. Tablicę należy wypełnić losowymi wartościami. Wskazówka: Poniższy fragment programu pokazuje działanie klasy *Random* (która zawiera generator liczb pseudolosowych) – w pętli zostanie wyświetlonych 100 liczb wybranych losowo z zakresu 1 – 999 (o zakresie decydują argumenty podane w wywołaniu metody *Next()*⁴²).

```
Random rand = new Random();
for (int i = 0; i < 100; i++)
    Console.Write("{0,8}", rand.Next(1, 1000));
```

Zadanie 4.5.

Dana jest n -elementowa tablica liczb całkowitych *tab1*. Napisz program kopiujący wartości elementów tablicy *tab1* do tablicy *tab2* (o tym samym rozmiarze) z przesunięciem o jedną pozycję. To znaczy, że element w tablicy źródłowej o indeksie 0 powinien znaleźć się w tablicy docelowej pod indeksem 1, element o indeksie 1 ma być w tablicy docelowej pod indeksem 2 itd. Element ostatni tablicy źródłowej ma być elementem o indeksie 0 w tablicy docelowej.

Zadanie 4.6.

Napisz program, który deklaruje i inicjalizuje dwuwymiarową tablicę liczb rzeczywistych o rozmiarze 5 x 5. Program ma wyświetlić elementy tablicy (wiersz po wierszu), a następnie wyświetlić sumę elementów znajdujących się na głównej przekątnej tablicy (główna przekątna – od elementu o indeksach 0,0 do elementu o indeksach n,n).

Zadanie 4.7.

Napisz program, który dodaje dwie macierze o rozmiarze 2 x 3. Elementy macierzy należy umieścić w tablicach dwuwymiarowych w trakcie deklaracji. Program ma wyświetlić

⁴² Metoda *Next()* z klasy *Random* występuje w kilku przeciążonych wariantach. Omawiamy tu tylko wariant przyjmujący dwa argumenty (typu *int*). Dla przykładowego wywołania *rand.Next(x1,x2)* – metoda ta zwróci wartość losową z przedziału lewostronnie domkniętego $[x1,x2)$. Fakt, że jest to przedział lewostronnie domknięty oznacza, że losowa wartość będzie większa lub równa wartości pierwszego argumentu (tu *x1*) oraz będzie mniejsza od drugiego argumentu (tu *x2*). Zatem nie może przyjąć wartości górnego zakresu. Dlatego, jeśli chcemy mieć zakres 1-999, należy wywołać metodę *Next()*, jak w prezentowanym fragmencie programu, dla argumentów 1 i 1000.

macierz wynikową. Wskazówka: Dodawanie macierzy – macierz wynikowa C zawiera elementy, które stanowią sumę elementów macierzy A i B o odpowiednich indeksach, tzn. element w zerowym wierszu i zerowej kolumnie macierzy A jest dodawany do elementu o tych samych indeksach macierzy B, element A [0,1] do B [0,1]... itd.

Zadanie 4.8.

Uzupełnij poniższy kod programu o wszystkie dni tygodnia i przy użyciu pętli wyświetl zawartość tablicy: w każdym wierszu dany dzień tygodnia w trzech językach (polskim, angielskim, niemieckim).

```
string[,] dniTygodnia;
dniTygodnia = new string[2, 3]; // pamiętaj o zmianie rozmiaru tablicy
dniTygodnia[0, 0] = "poniedziałek";
dniTygodnia[1, 0] = "wtorek";
dniTygodnia[0, 1] = "monday";
dniTygodnia[1, 1] = "tuesday";
dniTygodnia[0, 2] = "montag";
dniTygodnia[1, 2] = "dienstag";
```

Zadanie 4.9.

Napisz program obliczający liczbę wyrazów w łańcuchu znaków wprowadzonym przez użytkownika. Należy przyjąć, że wyrazy to ciągi znaków rozdzielone spacją.

Zadanie 4.10.

Napisać program, który pobierze datę w formacie DD-MM-RRRR, z której pobierze miesiąc i wyświetli jego nazwę słownie.

Zadanie 4.11.

Napisz program analizujący częstość występowania poszczególnych znaków w łańcuchu znaków wprowadzonym przez użytkownika. Np. dla wprowadzonego tekstu „abrakadabra” program powinien wyświetlić informacje: a – 5, b – 2, r – 2, k – 1, d – 1.

Zadanie 4.12.

Napisz program, który dla zadeklarowanej niżej zmiennej łańcuchowej wyświetli jej zawartość, poda liczbę wierszy oraz poda liczbę znaków w każdym wierszu.

// fragment powieści A. A. Milne, "Kubuś Puchatek"

```
string tekst = "W parę godzin później, gdy noc zbierała się do odejścia,\n" +
    "Puchatek obudził się nagle z uczuciem dziwnego przygnębienia.\n" +
    "To uczucie dziwnego przygnębienia miewał już nieraz i wiedział,\n" +
    "co ono oznacza. Był głodny. Więc poszedł do spiżarni,\n" +
    "wgramolił się na krzeselko, sięgnął na górną półkę, ale nic nie znalazł.";
```

Zadanie 4.13.

Napisz program, który przeanalizuje dany łańcuch pod kątem wielokrotnego występowania słów w tekście. Przykładowo dla zmiennej łańcuchowej o zawartości: „Kiedy idzie się po miód z balonikiem, to trzeba się starać, żeby pszczoły nie wiedziały, po co się idzie – odpowiedział Puchatek” – program powinien wypisać raport o słowach powielonych w tym tekście: idzie – 2 razy, po – 2 razy, się – 3 razy.

Zadanie 4.14.

W danej firmie środki trwałe mają identyfikatory złożone z kilku liter, myślnika oraz czterech cyfr. Te cztery cyfry to rok zakupu danego środka trwałego. Przykładowe identyfikatory to: KOMG-2002, BH-2010. Napisz program, który deklaruje 5-cio elementową tablicę typu *string* dla środków trwałych, którą należy zainicjalizować przykładowymi identyfikatorami w czasie deklaracji. Program ma dla każdego środka trwałego podać liczbę lat, jakie upłynęły od jego zakupu.

5 Metody

W pierwszym rozdziale ([punkt 1.1.6](#)) pisaliśmy, że obiektowe języki programowania opisują model danego fragmentu rzeczywistości uwzględniając powiązanie danych obiektu z jego zachowaniem. Dla obiektów danej klasy można przypisać określone grupy zadań, jakie mogą być na nich wykonywane. Podstawowym narzędziem służącym do opisu zachowania obiektów w klasach – są **metody** (funkcje). Istnieje kilka różnych rodzajów metod, oznaczonych odpowiednimi modyfikatorami. W tym rozdziale skupimy się na metodach statycznych (z modyfikatorem *static*). Powodem jest możliwość użycia takich metod bez konieczności budowania własnych klas, czym zajmiemy się w następnym rozdziale. Metody statyczne będą dla nas poligonem, pozwalającym objaśnić, a następnie przetestować szereg elementów programistycznych związanych z używaniem wszystkich rodzajów metod (statycznych i niestatycznych). Do elementów tych należą: definicja metody, przekazywanie parametrów, zwracanie wartości, wywołanie metod, przeciążanie metod oraz rekurencja.

5.1 Metody statyczne

Metoda statyczna jest to metoda klasy, która nie jest wywoływana dla konkretnego obiektu tej klasy. Metody statyczne z reguły służą do obsługi składowych statycznych klas (o składowych powiemy więcej w [rozdziale 6](#)). Wykorzystywaliśmy już metody statyczne, m.in. metody matematyczne z klasy *Math* (np. do obliczania wartości funkcji sinus – *Math.Sin(x)*), zob. [przykład 2.5](#)). Teraz będziemy budować własne metody.

Popatrzmy na metodę (funkcję) jak na „maszynę”, która ma coś do wykonania. Każda taka maszyna ma coś na wejściu, coś wykonuje (przetwarza) i w efekcie daje coś na wyjściu. Sokowirówka przykładowo na wejściu ma owoce, po uruchomieniu przeciera miąższ, a na wyjściu jest sok i resztki stałe owoców. To, co owa „maszyna” może przyjąć na wejściu nazywamy argumentami. Dla sokowirówki może to być np. jabłko. Wówczas wynikiem będzie sok jabłkowy. Użycie innego owocu da w efekcie inny sok. Ale sposób przetwarzania, przypisany dla danej „maszyny” jest taki sam. Nie musimy używać osobnych sokowirówek dla poszczególnych rodzajów owoców i warzyw. Swoje metody tworzymy dla takich wyodrębnionych funkcjonalności, które wykonywane dla różnych wejść (w ramach określonego typu⁴³) – generują odpowiednie wyniki.

Po utworzeniu nowego projektu dla konsoli pojawia się wstępnie kod, zwróćmy uwagę na jego wewnętrzną część:

⁴³ Najczęściej dla obiektów, wówczas same obiekty określają jedno z wejść, o tym powiemy więcej w [punkcie 6.3.2](#). W bieżącym rozdziale skupiamy się na metodach statycznych, które są wywoływane dla klasy, a nie dla konkretnych obiektów.

```
class Program
{
    static void Main(string[] args)
    {
    }
}
```

Automatycznie tworzona jest klasa *Program* zawierająca statyczną metodę *Main()*. W jej ciele umieszczaliśmy dotychczas kod programu. W programach, jakie będziemy analizować w tym podrozdziale, klasa *Program* będzie poszerzona o jeszcze jedną metodę statyczną, naszą własną. Przykładowy szkielet programów będzie wyglądał następująco:

```
class Program
{
    static void NaszaMetoda(int x)
    {
    }

    static void Main(string[] args)
    {
    }
}
```

Metod (statycznych i niestycznych) może być w danej klasie dużo. Ale na tych dwóch pokażemy podstawowe zasady dotyczące **komunikacji** między metodami. Każda metoda ma coś do wykonania i musi być zapewniona komunikacja między pozostałymi częściami programu, aby program działał poprawnie jako całość.

5.2 Definicja metody

Definicja metody składa się z kilku elementów (elementy ujęte w nawiasy kwadratowe są opcjonalne):

Składnia 5.1

```
[Modyfikatory] Typ Nazwa ([Lista argumentów])
{
    [Ciało metody]
}
```

Elementy definicji metody:

- *Deklaracja metody*:
 - *Modyfikatory* - określają zachowanie i dostępność metody,
 - *Typ* – typ danej zwracanej przez metodę,
 - *Nazwa* – nazwa metody, różna od nazwy klasy, w której została zdefiniowana,

- *Lista argumentów* – lista argumentów przekazywanych do metody,
- *Ciało metody* – inaczej treść metody, kod (zbiór instrukcji) realizujący działanie metody.

Pierwsza linia definicji, która obejmuje modyfikatory, typ, nazwę i listę argumentów stanowi **deklarację metody**. Deklaracja metody wraz z ciałem metody – stanowią **definicję metody**. Przybliżymy poszczególne elementy definicji metody.

Modyfikatory

Można wyróżnić dwa rodzaje modyfikatorów: dostępu i zachowania. Pełną listę modyfikatorów dostępu przedstawimy i omówimy w [rozdziale 6](#), tu krótko powiemy o dwóch: *public* i *private*. Modyfikator *public* udostępnia daną metodę na zewnątrz dla innych klas. Modyfikator *private* umożliwia użycie metody tylko wewnątrz klasy. W naszych programach (w tym rozdziale) wykorzystana będzie tylko jedna klasa, przez co modyfikator dostępu dla metod może być *private* – i taki jest domyślnie. Deklaracja *static void Metoda(int x)* jest równoznaczna z deklaracją: *private static void Metoda(int x)*. W skład modyfikatorów określających zachowanie (rodzaj) metody wchodzi: *static* (dla metod statycznych), *new*, *virtual*, *override*, *sealed* oraz *abstract*.

Typ

Każda metoda może zwracać jakąś wartość i w deklaracji należy opisać typ tej wartości. Jeśli nic nie zwraca, to zamiast nazwy typu umieszcza się słowo kluczowe *void*. Przykładowa metoda statyczna o deklaracji *static void Metoda(int x)* nie zwraca nic. Natomiast metoda *static int Metoda(int x)* zwraca wartość typu *int*.

Nazwa

Nazwa metody stanowi wraz z listą argumentów identyfikator metody. Nazwa metody musi być różna od nazwy klasy (metoda, która ma tę samą nazwę, co klasa jest konstruktorem klasy i pełni specjalne funkcje w klasie, szerzej na ten temat w [podrozdziale 6.2.2](#)). Mogą być w danej klasie metody o tej samej nazwie, ale muszą różnić się listą argumentów (są to tzw. metody przeciążone). Zasady odnośnie nazewnictwa metod są takie same, jak w przypadku nazewnictwa zmiennych: może się składać jedynie z alfanumeryków oraz znaku podkreślenia. Alfanumeryk to cyfra lub litera (mała lub duża). Przy czym nazwa metody nie może być słowem kluczowym oraz nie może się zaczynać od cyfry.

Lista argumentów

Lista argumentów zawiera argumenty metody oddzielane przecinkami. Dla każdego argumentu musi być podany typ oraz jego nazwa. Metoda może nie mieć żadnych argumentów, wówczas umieszcza się pustą parę nawiasów. Przykładowa metoda o deklaracji *static void Metoda(int x, float y)* ma dwa argumenty, jeden typu *int*, drugi typu *float*. Opcjonalnie na liście argumentów mogą znajdować się słowa kluczowe *ref*, *out* lub *params*

(umieszcza się je przed nazwą typu), decydujące o sposobie przekazania parametru – o tym szerzej będzie w dalszej części rozdziału.

Ciało metody

Ciało metody, inaczej treść metody – to kod ujęty w klamrowe nawiasy, który realizuje zadanie przypisane dla metody. W ciele metody można deklarować zmienne lokalne⁴⁴, umieszczać instrukcje i wywołania innych metod. Deklarowane zmienne nie mogą mieć tej samej nazwy, co nazwy argumentów. Jeżeli metoda ma określony typ zwracanej wartości (inny niż *void*), w ciele metody musi być instrukcja (instrukcje) *return*, która zwraca daną określonego typu.

Popatrzmy na przykład:

Przykład 5.1.

```
class Program
{
    static void Odejmij(int x, int y)
    {
        Console.WriteLine(x - y);
    }

    static void Main(string[] args)
    {
        Odejmij(4, 3);           // wywołanie metody
        Console.ReadKey();
    }
}
```

Najpierw przeczytamy deklarację metody. Metoda *Odejmij()* jest metodą prywatną (modyfikator *private* jest domyślny dla składowych klas). Jest metodą statyczną (modyfikator *static*). Nie zwraca żadnej wartości (*void*). Przyjmuje dwa argumenty typu *int*. W ciele metody jest tylko jedna instrukcja – wywołanie metody *Console.WriteLine()*, która wyświetla wynik różnicy obu argumentów *x* i *y*.

Tak zdefiniowana metoda, aby mogła zostać wykonana przez program musi być **wywołana**. Wywołanie znajduje się w ciele metody *Main()*. Ponieważ jest to metoda statyczna w tej samej klasie (w klasie *Program*) wystarczy, że podamy nazwę metody wraz z argumentami. Gdybyśmy chcieli tę samą metodę wywołać z innej klasy, to po pierwsze metoda ta musiałaby być zadeklarowana jako *public*, a po drugie w wywołaniu musielibyśmy podać także nazwę klasy, np. *Program.Odejmij(4,3)*. Wróćmy jednak do naszych (tymczasowo) uproszczonych warunków w tym przykładzie (jedna klasa) i wywołania *Odejmij(4,3)*. W wywołaniu podajemy argumenty, które odpowiadać będą liście argumentów

⁴⁴ Zmienna lokalna ma zakres lokalny w metodzie, w której jest deklarowana. W C# nie ma zmiennych globalnych. Definiuje się natomiast pola w klasach (podobnie jak zmienne lokalne), ale pola, w przeciwieństwie do zmiennych lokalnych, mają wartości domyślne (zgodnie z typem) – więcej na ten temat w rozdziale 6

metody. Skoro w deklaracji metody *Odejmij()* określono, że przyjmuje dwa argumenty typu *int*, to w wywołaniu powinny być dwa argumenty typu *int*⁴⁵. Co robi program po wywołaniu metody? Oczywiście wraca do miejsca wywołania, do kolejnej instrukcji. W omawianym przykładzie program wróci do metody *Main()* i przejdzie do instrukcji *Console.ReadKey()*.

Ponieważ metoda *Odejmij()* zgodnie z deklaracją ma nic nie zwracać (typ zwracanej wartości jest *void*), brak w jej ciele instrukcji *return*. Taka instrukcja pojawi się w kolejnym przykładzie.

Przykład 5.2.

```
class Program
{
    static double Dziel(double x, int y)
    {
        return (x / y);           // zakładamy, że y jest różne od zera
    }

    static void Main(string[] args)
    {
        Console.WriteLine(Dziel(1.5, 3)); // wywołanie metody
        Console.ReadKey();
    }
}
```

Metoda *Dziel()* jest statyczna, zwraca wartość typu *double*, przyjmuje jeden argument typu *double* oraz jeden typu *int*. W jej ciele jest instrukcja *return*, która zwraca wynik działania *x/y*. Przyjrzymy się wywołaniu metody, które jest w ciele metody *Main()*:

```
Console.WriteLine(Dziel(1.5, 3));
```

Ponieważ metoda *Dziel()* zwraca wartość, możemy posłużyć się jej wywołaniem w wyrażeniu lub w wywołaniu innej metody. Wywołanie metod może być zagnieżdżone. Rezultat metody „wewnętrznej” jest argumentem wejściowym dla metody „zewnętrznej”. Kod metody *Main()* z powyższego przykładu przeanalizujemy w kilku wariantach.

⁴⁵ W kontekście metod można spotkać w podręcznikach do programowania dwa terminy: „argument” oraz rzadziej „parametr”. W definicji metody, która obejmuje deklarację i ciało metody mówi się o argumentach formalnych (parametrach formalnych), natomiast w wywołaniu metody używane jest pojęcie „argumenty aktualne” (parametry aktualne). Więcej na ten temat można znaleźć w dokumentacji MSDN: <http://msdn.microsoft.com/en-us/library/f81cdka5.aspx>. W podręczniku tym używać będziemy głównie terminu „argument” zarówno dla argumentów formalnych jak i aktualnych, rozróżniane będą wg kontekstów.

Wariantwołania metody <i>Dziel()</i>	Komentarz
<pre>static void Main(string[] args) { double wynik; wynik = Dziel(1.5, 3); Console.WriteLine(wynik); Console.ReadKey(); }</pre>	<p>W tej wersji rezultat metody <i>Dziel()</i> jest przypisywany do zmiennej <i>wynik</i>. Rozwiązanie takie byłoby polecane, gdyby wynik tej metody miał być jeszcze wykorzystany w innym miejscu programu.</p> <p>Wyświetli się liczba 0,5 (tak samo jak w wersji bazowej przykładu 5.2).</p>
<pre>static void Main(string[] args) { Console.WriteLine(Dziel(1.5,3)*2); Console.ReadKey(); }</pre>	<p>Tu wynik metody <i>Dziel()</i> jest mnożony przez 2 (a zatem użyty w wyrażeniu). Iloczyn ten jest argumentem dla metody <i>Console.WriteLine()</i>.</p> <p>Wyświetli się liczba 1.</p>
<pre>static void Main(string[] args) { double x = 0.5; int y = 2; Console.WriteLine(Dziel(x,y)); Console.ReadKey(); }</pre>	<p>W tym przypadku w wywołaniu metody <i>Dziel()</i> użyto jako argumentów zmiennych (a nie literałów jak poprzednio).</p> <p>Nazwy zmiennych użyte w wywołaniu jako argumenty nie mają związku z nazwami argumentów w definicji metody. W tym programie obie zmienne można równie dobrze nazwać <i>a</i> i <i>b</i>. Albo zmienić <i>x</i> i <i>y</i> miejscami, tzn.: <code>double y = 0.5; int x = 2;</code> <code>Console.WriteLine(Dziel(y,x));</code></p> <p>Wyświetli się liczba 0,25.</p>

Na koniec omawiania przykładu 5.2 rozważymy jeszcze kwestię zwracania wartości i instrukcji warunkowej. Gdyby zmienić definicję metody *Dziel()* na następującą:

```
static double Dziel(double x, int y)
{
    if (y != 0)
    {
        return (x / y);
    }
}
```

to pojawi się komunikat błędu o treści: „not all code paths return a value”. Oznacza to, że nie wszystkie ścieżki w kodzie tej metody umożliwiają zwracanie wartości (a musi być zwrócona wartość typu *double*). I błąd ten wystąpi nawet jeśli we wszystkich wywołaniach tej metody drugi argument będzie różny od zera. Poprawny będzie kod:

```
static double Dziel(double x, int y)
{
    if (y != 0)
    {
        return (x / y);
    }
}
```

```

        else
        {
            return (0);
        }
    }

```

Ale bardziej zalecane jest rozwiązanie, w którym unikamy powielania instrukcji *return*, np. takie:

```

static double Dziel(double x, int y)
{
    double wynik = 0;
    if (y != 0)
    {
        wynik = x / y;
    }
    return (wynik);
}

```

W powyższym rozwiązaniu przypisana jest domyślnie wartość, jaką metoda ma zwrócić. Bez względu na to, czy program przejdzie ścieżkę zgodnie z warunkiem dla instrukcji *if*, czy też nie – zwracana zmienna *wynik* ma już jakąś wartość.

Należy wspomnieć w tym miejscu, że do obsługi błędów w języku C# służą tzw. **wyjątki**. Obsługa wyjątków to jedno z bardzo istotnych zagadnień, z jakimi należy się zapoznać po przeczytaniu tej książki.

5.3 Przekazywanie argumentów przez wartość

Dla typów wbudowanych⁴⁶ domyślnym sposobem przekazywania argumentów jest przekazywanie przez wartość. Oznacza to, że wywoływana metoda otrzymuje kopię wartości podanych w argumentach. Ewentualne zmiany wartości argumentów wewnątrz metody nie będą widziane na zewnątrz (czyli argumenty te mają charakter wejściowy). Typ zmiennej przekazywanej do metody jako argument musi być zgodny z zadeklarowanym typem argumentu lub typem, który może zostać skonwertowany w sposób niejawny.

⁴⁶ Wykaz typów wbudowanych w C# znajduje się w [Tabeli 1 w Dodatkach](#).

Przykład 5.3.

```

class Program
{
    static void Dodaj(int a)
    {
        a++;
        Console.WriteLine("Argument z wnętrza metody: " + a);
    }

    static void Main(string[] args)
    {
        int x = 5;
        Console.WriteLine("Przed wywołaniem metody: " + x);
        Dodaj(x);
        Console.WriteLine("Po wywołaniu metody: " + x);
        Console.ReadKey();
    }
}

```

Wewnątrz metody *Dodaj()* została zmieniona wartość argumentu *a* (zwiększona o jeden), ale nie wpływa to na wartość argumentu aktualnego, jaki jest w wywołaniu. Po wykonaniu metody wartość zmiennej *x* nadal wynosi 5 (jak przed zmianą). Program po uruchomieniu wyświetla informacje:

```

Przed wywołaniem metody: 5
Argument z wnętrza metody: 6
Po wywołaniu metody: 5

```

W wywołaniu *Dodaj(x)* zmienna *x* została przekazana przez wartość. W metodzie *Dodaj()* została zrobiona lokalna kopia tej wartości i to dlatego jej zmiana nie jest widoczna po wyjściu z metody.

Wspomniano o niejawnej konwersji podczas wywołania metody. W prosty sposób można to sprawdzić zmieniając w metodzie *Main()* instrukcje *int x = 5;* na *short x = 5;*. Po takiej zmianie program będzie działał tak samo, a argument metody *Dodaj()* zostanie niejawnie konwertowany do typu *int*. Nie jest jednak możliwa niejawna konwersja z typu *double* do *int*, zmiana deklaracji na *double x = 5;* będzie skutkować błędem „cannot convert from double to int”⁴⁷.

5.4 Przekazywanie argumentów przez referencję

Przekazywanie argumentów przez referencję oznacza, że do metody przekazywany jest adres argumentów. Wówczas wewnątrz metody nie jest robiona kopia lokalna, metoda

⁴⁷ Wykaz możliwych konwersji niejawnych dla typów numerycznych jest w tabeli na stronie <http://msdn.microsoft.com/en-us/library/y5b434w4.aspx>

pracuje na oryginalnych danych (posługując się ich adresem), a po wywołaniu metody ewentualne zmiany argumentu są widziane na zewnątrz (tam, gdzie była wołana metoda). W przypadku typów wbudowanych, aby przekazać argumenty przez referencję należy oznaczyć je przy pomocy jednego z modyfikatorów: *ref* lub *out*. Wybór odpowiedniego modyfikatora zależy od tego, gdzie będzie inicjalizowany argument, jeśli w miejscu wywołania metody, to *ref*, jeśli wewnątrz metody, to *out*. Przeanalizujemy to na przykładach.

Przykład 5.4.

```
class Program
{
    static void Dodaj(ref int a)
    {
        a++;
        Console.WriteLine("Argument z wnętrza metody: " + a);
    }

    static void Main(string[] args)
    {
        int x = 5;
        Console.WriteLine("Przed wywołaniem metody: " + x);
        Dodaj(ref x);
        Console.WriteLine("Po wywołaniu metody: " + x);
        Console.ReadKey();
    }
}
```

Kod przykładu 5.4 jest prawie identyczny jak poprzedniego. Różnica dotyczy jedynie wywołania argumentu, który wołany jest przez referencje. Użyto modyfikatora *ref* (w deklaracji metody na liście argumentów przed typem, a w wywołaniu tuż przed nazwą). Program po uruchomieniu wyświetla:

```
Przed wywołaniem metody: 5
Argument z wnętrza metody: 6
Po wywołaniu metody: 6
```

Użycie modyfikatora *ref* sprawiło, że zmienna *x* w metodzie *Main()*, która została użyta jako argument metody *Dodaj()*, po wywołaniu metody zachowuje zmienioną wartość (równą 6). Ponieważ argumenty z modyfikatorem *ref* przekazywane są w obu kierunkach (do metody oraz z metody) nazywane są także argumentami wejścia/wyjścia.

Przykład 5.5.

```

class Program
{
    static void Dodaj(out int x, out int y)
    {
        x = 2;
        y = 5;
        Console.WriteLine("Dodaj(): x={0} y={1}", x, y);
    }

    static void Main(string[] args)
    {
        int a, b;           // deklaracja, brak inicjalizacji
        Dodaj(out a, out b);
        Console.WriteLine("Main(): a={0} b={1}", a, b);
        Console.ReadKey();
    }
}

```

Kod tego przykładu także jest podobny do poprzednich, ale tym razem są dwa argumenty przekazywane przez referencje z użyciem modyfikatora *out*. Inicjalizacja (i ewentualne przetwarzanie) takich argumentów następuje wewnątrz metody. Na zewnątrz muszą być jedynie zadeklarowane. Argumenty z modyfikatorem *out* to tzw. argumenty wyjściowe, nie można ich przekazać do metody. Przekazywanie argumentów przez referencję (*ref* lub *out*) jest jednym ze sposobów uzyskania efektu „zwracania” przez metodę więcej niż jednej wartości wynikowych. Wynik omawianego programu:

Dodaj() : x=2 y=5 Main() : a=2 b=5

5.5 Lista argumentów o zmiennej długości

Do metody można przekazać listę argumentów wejściowych o zmiennej długości (zmiennej liczbie argumentów). Prześledźmy przykład:

Przykład 5.6.

```

class Program
{
    static void Elementy(params int[] tab)
    {
        for (int i = 0; i < tab.Length; i++)
        {
            Console.WriteLine(tab[i]);
        }
        Console.WriteLine();
    }
}

```

```

static void Main(string[] args)
{
    Elementy(1, 2, 3); // 1) wywołanie dla listy argumentów

    int[] tab1 = new int[3] { 18, 26, 67 };
    Elementy(tab1);    // 2) wywołanie z użyciem tablicy
    Console.ReadKey();
}
}

```

Metoda *Elementy()* przyjmuje jako argument tablicę typu *int*, argument ten oznaczony jest modyfikatorem *params*. Wewnątrz metody wyświetlane są elementy tablicy. Ponieważ do odczytania rozmiaru tablicy użyto właściwości *Length*, metoda ta może wyświetlić dowolną liczbę elementów. Metodę z argumentem posiadającym modyfikator *params* można wywołać na dwa sposoby. W pierwszym z nich podajemy listę argumentów. Może być tak jak w przykładzie *Elementy(1,2,3)*, ale równie dobrze może być np. *Elementy(1,2,3,6,9)*. Drugi sposób wywołania metody polega na użyciu tablicy jako argumentu. W kolejnym punkcie będzie mowa o przekazywaniu tablic jako argumentów, ale już teraz nadmienimy, że tablica przekazywana jest przez referencję, co oznacza, że gdyby w metodzie *Elementy()* były instrukcje zmieniające zawartość tablicy, to po wyjściu z metody zmiany te będą widoczne w tablicy *tab1*.

Jest także możliwość przekazania listy argumentów o zmiennej długości mających różne typy, np. *NazwaMetody(1, 'a', "test")*⁴⁸.

Metoda zawierająca argument z modyfikatorem *params* może mieć więcej (zwykłych) argumentów, ale argument *params* musi być na końcu.

5.6 Przekazywanie i zwracanie tablic

Można do metody przekazać daną typu złożonego (np. tablicę, obiekt). Podobnie metoda może także zwracać daną typu złożonego. Tu zajmiemy się przekazywaniem i zwracaniem tablic. **Tablice przekazywane są przez referencję** (a referencja zawiera adres). Oznacza to, że po powrocie z metody ewentualne zmiany w tablicy są widoczne po wywołaniu metody.

⁴⁸ Przykład dla argumentów różnego typu jest w dokumentacji MSDN: [http://msdn.microsoft.com/en-us/library/w5zay9db\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/w5zay9db(v=vs.71).aspx)

Przykład 5.7.

```

class Program
{
    static void Wielkie(string[] tab)
    {
        for (int i = 0; i < tab.Length; i++)
        {
            tab[i] = tab[i].ToUpper();
        }
    }

    static void Main(string[] args)
    {
        string[] tab1 = { "jeden", "dwa", "trzy" };
        Wielkie(tab1);      // wywołanie metody (tablica argumentem)

        for (int i = 0; i < tab1.Length; i++)
        {
            Console.Write(tab1[i] + " ");
        }
        Console.ReadKey();
    }
}

```

Metoda *Wielkie()* przyjmuje jako argument tablicę jednowymiarową typu *string*. Teksty w tablicy zamieniane są na duże litery. Po wywołaniu wyświetlona zostaje tablica (z zachowanymi zmianami):

JEDEN DWA TRZY

Program przypomina jeden z wariantów wywołania metody *Elementy()* z poprzedniego przykładu (z modyfikatorem *params*), przy czym w przeciwieństwie do rozwiązania z modyfikatorem *params* nie jest możliwe wywołanie metody *Wielkie()* z listą argumentów, np. *Wielkie("ala", "kot")*. W wywołaniu musi być tablica (podajemy tylko nazwę tablicy bez rozmiaru).

Kolejny przykład zawierać będzie metodę, która zwraca tablicę.

Przykład 5.8.

```

class Program
{
    static int[] Liczby(int rozmiar)
    {
        int[] tab = new int[rozmiar];
    }
}

```

```

        for (int i = 0; i < rozmiar; i++)
        {
            tab[i] = i;
        }
        return (tab);           // zwraca tablicę
    }

    static void Main(string[] args)
    {
        int[] tab1 = Liczby(5); // wywołanie metody
        for (int i = 0; i < tab1.Length; i++)
        {
            Console.Write(tab1[i] + " ");
        }
        Console.ReadKey();
    }
}

```

W deklaracji metody *Liczby()* przy typie zwracanej wartości podano także kwadratowe nawiasy, co oznacza, że metoda będzie zwracać tablicę. Metoda *Liczby()* przypisuje elementom tablicy kolejne liczby (od zera). W instrukcji *return* należy podać nazwę zwracanej tablicy. Wywołanie metody zostało użyte bezpośrednio w przypisaniu zwracanej tablicy do utworzonej tablicy *tab1*.

Przedstawione przykłady dotyczyły tablic jednowymiarowych, ale oczywiście przekazywanie i zwracanie tablic dotyczy wszystkich tablic – także wielowymiarowych. Poniżej jest kod bazujący na [przykładzie 5.7](#), który przekazuje do metody tablicę dwuwymiarową (prostokątną).

Przykład 5.9.

```

class Program
{
    static void Wielkie(string[,] tab)
    {
        for (int i = 0; i < tab.GetLength(0); i++)
        {
            for (int j = 0; j < tab.GetLength(1); j++)
            {
                tab[i, j] = tab[i, j].ToUpper();
            }
        }
    }

    static void Main(string[] args)
    {
        string[,] tab1 = {{ "jeden", "dwa", "trzy" },
                           { "one", "two", "three" }};
        Wielkie(tab1); // wywołanie metody (tablica argumentem)
    }
}

```

```

    for (int i = 0; i < tab1.GetLength(0); i++)
    {
        for (int j = 0; j < tab1.GetLength(1); j++)
        {
            Console.Write("{0,-8}", tab1[i, j]);
        }
        Console.WriteLine();
    }
    Console.ReadKey();
}

```

W przykładzie metoda *Wielkie()* przyjmuje tablicę dwuwymiarową prostokątną, co zostało określone w deklaracji metody. W jej ciele do odczytu dokładnego rozmiaru dla obu wymiarów użyto metody *GetLength()* z numerem wymiaru jako argumentem (wymiar numeruje się od zera). Wynik programu:

JEDEN	DWA	TRZY
ONE	TWO	THREE

Na zakończenie omawiania tablic w kontekście metod należy zwrócić uwagę na to, że efekt udostępnienia elementów tablicy poza metodą (do miejsca wywołania) można uzyskać zarówno poprzez przekazanie jak i zwracanie. To kiedy zwracać tablicę, a kiedy przekazać zależy m.in. od miejsca inicjalizacji tablicy.

5.7 Argumenty domyślne

Jest możliwe zadeklarowanie w metodzie tzw. argumentów domyślnych, czyli takich, których nie trzeba podawać w wywołaniu (bo przyjmą wartości domyślne).

Przykład 5.10.

```

class Program
{
    static int Dodaj(int x, int y = 0)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(Dodaj(1));           // wyświetli się 1
        Console.WriteLine(Dodaj(1, 5));       // wyświetli się 6
        Console.ReadKey();
    }
}

```

Metoda *Dodaj()* ma na liście argumentów jeden argument zwykły, a drugi domyślny. Domyślny ma przypisaną wartość domyślną 0. Wywołanie *Dodaj(1)* sprawi, że w ciele metody do liczby 1 zostanie dodana wartość domyślna drugiego argumentu, czyli 0. Dla wywołania *Dodaj(1,5)* zwrócona zostanie suma obu przekazanych argumentów.

Argumenty domyślne (może ich być więcej niż jeden) muszą być umieszczone na końcu listy argumentów⁴⁹.

5.8 Metody przeciążone

Nazwa metody wraz z listą argumentów tworzy tzw. sygnaturę. W uproszczony sposób możemy przedstawić sygnaturę dla przykładowej metody o deklaracji *int MojaMetoda(int x, double y)* w postaci *MojaMetoda_int_double*. Jak widać, zwracany typ nie ma wpływu na postać sygnatury. W danej klasie zdefiniowane metody muszą się różnić sygnaturą, ale mogą mieć taką samą nazwę. Metody, które mają tę samą nazwę i różnią się listą argumentów nazywane są metodami **przeciążonymi** lub przeładowanymi (ang. overloaded). Przeciążanie metod, podobnie jak pozostałe omówione w tym podrozdziale tematy, dotyczy wszystkich rodzajów metod, nie tylko statycznych, na których (obecnie) testujemy działanie metod.

Przykład 5.11.

```
class Program
{
    static string OpiszTyp()
    {
        return "Metoda bez argumentów";
    }

    static string OpiszTyp(int x)
    {
        return "Liczba całkowita";
    }

    static string OpiszTyp(string x)
    {
        return "łańcuch znaków";
    }

    static string OpiszTyp(double x, int y)
    {
        return "Liczba double i liczba całkowita";
    }
}
```

⁴⁹ Można zadać pytanie – jak postąpić w przypadku, gdy na końcu listy argumentów jest kilka argumentów domyślnych, a chcemy skorzystać z wartości domyślnej argumentu, który jest nieostatni. Można w tym celu posłużyć tzw. *argumentem nazwanym*, więcej na ten temat można znaleźć na stronie dokumentacji MSDN <http://msdn.microsoft.com/pl-pl/library/dd264739.aspx>.

```

static void Main(string[] args)
{
    Console.WriteLine(OpiszTyp());
    Console.WriteLine(OpiszTyp(10));
    Console.WriteLine(OpiszTyp("Apollo 10"));
    Console.WriteLine(OpiszTyp(100.45, 10));
    Console.ReadKey();
}
}

```

W przykładzie są cztery metody przeciążone. Inaczej moglibyśmy powiedzieć, że metoda *OpiszTyp()* występuje tu w czterech wariantach (wersjach). Metody zwracają jedynie tekst opisu typu argumentów, ale dzięki prostocie definicji tych metod możemy skupić się na podstawowych aspektach. W metodzie *Main()* są cztery wywołania, to, która przeciążona metoda zostanie wykonana zależy od listy argumentów (czyli sygnatury). Wynik programu:

```

Metoda bez argumentów
Liczba całkowita
Łańcuch znaków
Liczba double i liczba całkowita

```

Wywołanie bez argumentów wywoła wersję bezargumentową, wywołanie z liczbą całkowitą wywoła wersję z jednym argumentem typu *int*, itd. Wydaje się to wszystko oczywiste, niemniej jest jeszcze kilka elementów komplikujących ten prosty mechanizm. Do elementów tych należą: niejawne konwersje, argumenty domyślne oraz uwzględnienie kontekstu dziedziczenia. Ten ostatni problem nie będzie w tym podręczniku omawiany.

Przeanalizujemy przykład z argumentem domyślnym:

Przykład 5.12.

```

class Program
{
    static int Daj(int x = 0)    // argument domyślny
    {
        return x;
    }

    static int Daj()
    {
        return -1;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(Daj());
        Console.ReadKey();
    }
}

```


Przed uruchomieniem programu można mieć wątpliwość, która z wersji metody zostanie wywołana, ponieważ wywołanie metody *Daj()* pasuje do obu definicji. Wykonana zostanie wersja druga (bezargumentowa), jeśli jednak usuniemy tę wersję, to wywołana zostanie pierwsza (z argumentem domyślnym). Oznacza to, że w przypadku, gdy do danego wywołania pasuje zarówno metoda dokładnie według sygnatury oraz jednocześnie pasuje metoda z zastosowaniem argumentu domyślnego – to kompilator wybiera to dokładne dopasowanie. I dla potwierdzenia jeszcze inny przykład - gdyby były dwie metody o deklaracjach: *int Daj(int x, int y = 0)* oraz *int Daj(int x)*, to wywołanie *Daj(1)* pasuje w sposób dokładny według sygnatury tylko do drugiej deklaracji, zatem ta druga wersja przeciążonej metody zostałaby wywołana.

W kolejnym przykładzie przeanalizujemy problem niejawnej konwersji:

Przykład 5.13.

```
class Program
{
    static int Dodaj(int x, int y)    // 1) oba argumenty int
    {
        return x + y;
    }
    static int Dodaj(int x, short y) // 2) jeden int, drugi short
    {
        return -1 * (x + y);        // ta wersja zwróci liczbę ujemną
    }

    static void Main(string[] args)
    {
        int a = 1, b = 5;
        short c = 1;
        Console.WriteLine(Dodaj(a, b)); // wyświetli się 6 (metoda 1)
        Console.WriteLine(Dodaj(a, c)); // wyświetli się -2 (metoda 2)
        Console.ReadKey();
    }
}
```

Drugie wywołanie *Dodaj(a,c)* pasuje potencjalnie do obu metod, przy czym do drugiej metody (przyjmującej argumenty *int* i *short*) pasuje dokładnie (nie jest potrzebna niejawna konwersja). I właśnie ta druga wersja metody *Dodaj()*, z dokładnym dopasowaniem listy argumentów, zostanie wykonana. Wystarczy, że usuniemy drugi wariant metody, a wówczas wykonany zostanie pierwszy wariant (z niejawną konwersją dla drugiego argumentu z *short* do *int*).

A teraz skomplikujemy sytuację uwzględniając w jednym programie zarówno argument domyślny jak i niejawną konwersję:

Przykład 5.14.

```

class Program
{
    static int Dodaj(short x, int y = 0) // 1) short oraz int domyślny
    {
        return x + y;
    }
    static int Dodaj(int x) // 2) jeden argument int
    {
        return -1 * x; // ta wersja zwróci liczbę ujemną
    }

    static void Main(string[] args)
    {
        short b = 3;
        Console.WriteLine(Dodaj(b)); // wyświetli się 3 (metoda 1)
        Console.ReadKey();
    }
}

```

Metoda *Dodaj()*wołana jest z jednym argumentem typu *short*. Żadna z metod nie jest dokładnie dopasowana do sygnatury. Pierwsza wersja przyjmuje dwa argumenty, jeden typu *short*, a drugi domyślny typu *int*, natomiast druga wersja metody przyjmuje jeden argument typu *int*. Zatem co kompilator wykona – przyjmie domyślny argument czy niejawną konwersję? Po uruchomieniu widzimy, że została wykonana pierwsza wersja metody (z argumentem domyślnym), dla której stopień dopasowania jest wyższy.

Prezentowane przykłady były trywialne, dobre do pokazania „jak to działa”, ale niezbyt przydatne do pokazania „po co to”. Przeciążamy metody wówczas, gdy działanie we wszystkich wersjach ma być na tyle podobne, aby zasadne było używanie tej samej nazwy metody, ale jednocześnie, gdy występują pewne (nieduże) różnice mające swoje źródło w argumentach wejściowych.

Dla wbudowanych klas, takich jak np. *String*, można sprawdzić w dokumentacji dostępne metody i ich przeciążania. W ramach podstawowego kursu programowania chcielibyśmy Czytelnika zachęcać do usamodzielnienia się i sięgania do dokumentacji języka oraz podręczników dla zaawansowanych odbiorców. Na stronie dokumentacji MSDN jest lista metod w klasie *String*⁵⁰, niektóre z nich były omawiane w [podrozdziale 4.2](#). Przykładowo metoda *Substring()* jest opisana tam jako *overloaded* (czyli przeciążona). Napisano także, że zwraca podłańcuch dla obiektu (łańcucha znaków). Metoda ta może być użyta na dwa sposoby. Po kliknięciu w nazwę metody na wykazie pojawiają się do wyboru dwa przeładowania tej metody:

- Substring Method (Int32),

⁵⁰ [http://msdn.microsoft.com/en-us/library/System.String_methods\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/System.String_methods(v=vs.71).aspx)

- Substring Method (Int32, Int32).

Z opisu zawartego w dokumentacji dowiadujemy się, że pierwsza wersja ma tylko jeden argument typu *int* i zwraca podłańcuch począwszy od podanej pozycji do końca. Natomiast druga ma dwa argumenty typu *int* i zwraca podłańcuch od podanej pozycji, mający tyle znaków, ile wskazuje drugi argument. Druga wersja metody jest bardziej uniwersalna, tzn. przy jej pomocy można uzyskać podłańcuch zarówno licząc od podanej pozycji do końca, jak i określoną liczbę znaków. Ale w przypadku, gdy chcemy pobrać podłańcuch od podanego znaku do końca wygodniej jest użyć tej pierwszej wersji (nie trzeba obliczać liczby znaków potrzebnych do pobrania). Ponieważ metody te nie są statyczne, ich wołanie wymaga podania obiektu.

5.9 Rekurencja

Gdy mówimy studentom o rekurencji, a w tle na ekranie wyświetla się slajd z nagłówkiem „Definicja rekurencji” oraz poniżej zdanie „**Żeby zrozumieć rekurencję trzeba najpierw zrozumieć rekurencję**”, można dostrzec uśmiechy, a nierzadko zdarza się, że ktoś zgłasza błąd. Tą groteskową „definicją” (znaną wśród informatyków) budzimy intuicję słuchacza. I oczywiście w uśmiechach tych i skonsternowanych spojrzeniach ujawnia się ona prawidłowo. To jest dobry moment, aby przejść do pokazania kilku przykładowych przejawów rekurencji w geometrii i naturze. Jednym z nich jest trójkąt Sierpińskiego. Otrzymuje się go poprzez narysowanie trójkąta równobocznego, a następnie połączenie środków boków. Wówczas powstają 4 mniejsze trójkąty, środkowy się „wycina”, a z każdym pozostałym wykonuje się to samo. Na poniższych rysunkach widać kilka kolejnych etapów:



Na każdym z etapów tworzenia trójkąta Sierpińskiego wykonuje się te same operacje, w wyniku czego dany fragment trójkąta jest podobny do całości – takie podobieństwo nazywa się **samopodobieństwem**⁵¹. W naturze samopodobieństwo jest bardzo powszechne, przejawia się m.in. w strukturze liścia paproci, kalafiora, rzeki wraz z jej dorzeczem, błyskawicy, naczyń krwionośnych.

Prawdziwa definicja rekurencji brzmi oczywiście inaczej, ale gdy się już zrozumie rekurencję, wówczas tamta groteskowa definicja nie wydaje się już taka błędna: **Rekurencja (ang. recursion) to w logice, programowaniu i w matematyce odwoływanie się np. funkcji lub definicji do samej siebie.**

⁵¹ Obiekty samopodobne nazywane są fraktalami.

Ponieważ samopodobieństwo jest dość powszechnym zjawiskiem, także i w programowaniu przydatny jest mechanizm budujący samopodobne struktury – jest nim właśnie rekurencja.

Zacniemy od klasycznego przykładu metody rekurencyjnej – obliczającej silnię. Rekurencyjna definicja silni wygląda następująco:

$$n! = \begin{cases} 1 & \text{dla } n = 1 \\ n \cdot (n-1)! & \text{dla } n \geq 1 \end{cases}$$

Definicja nie obejmuje przypadku dla 0!, który trzeba zdefiniować osobno: 0! = 1. Silnia 4! według tej definicji to 4*(4-1)!, czyli 4*3!. Natomiast 3! możemy wyrazić przy pomocy tego samego wzoru jako 3*(3-1)!. Obliczamy podobnie 2! jako 2*(2-1)!. W końcu zostaje 1!, dla którego już nie ma wołania rekurencyjnego i które wynosi 1. Ostateczny wynik 4! jest obliczany jako iloczyn 1*2*3*4.

Przykład 5.15.

```
class Program
{
    static int Silnia(int n)
    {
        if (n <= 1) return 1;
        else return n * Silnia(n - 1);
    }

    static void Main(string[] args)
    {
        Console.WriteLine(Silnia(4));
        Console.ReadKey();
    }
}
```

Metoda *Silnia()* ma bardzo prostą definicję, tzn. zwięzłą, co jest jedną z cech rekurencji. Zawiera dwie linie, w pierwszej realizowany jest warunek opuszczenia rekurencji. Rekurencyjne wywołanie metody musi się zakończyć – albo decyduje o tym warunek opuszczenia rekurencji (jak w tym programie) albo skończony charakter danego zasobu, który jest rekurencyjnie przetwarzany (np. drzewo katalogów). Gdyby usunąć z analizowanego programu warunek opuszczenia rekurencji, pojawiłby się błąd *StackOverflowException* – czyli przepełnienie stosu. W drugiej linii metody *Silnia()* następuje wywołanie samej siebie dla *n-1* a wynik tego wywołania jest mnożony przez *n*.

W dotychczasowych programach operacje o charakterze wielokrotnym wykonywane były przy pomocy iteracji (np. pętli *for* lub *while*). W tym programie (rekurencyjnej metodzie) wielokrotnie wykonywane mnożenie wykonywane jest bez udziału iteracji. Na początkowym etapie nauki niektórzy popełniają błąd i próbują łączyć te obie konstrukcje programistyczne, np. umieszczając w pętli *for* rekurencyjne wywołania. W podejściu rekurencyjnym

wielokrotność (powtarzalność) operacji zapewnia wołanie metody samej siebie. W kodzie mamy *return* i kolejne wołanie tej samej metody. Wołania te kładzione są na stosie i to tam są pamiętane wartości argumentów z poszczególnych etapów. Dla omawianego programu stos ten można zobrazować następująco (należy czytać stos od dołu):

Silnia(1)	n= 1	→ zwraca wartość 1
Silnia(2)	n= 2	→ zwraca 2*Silnia(1)
Silnia(3)	n= 3	→ zwraca 3*Silnia(2)
Silnia(4)	n= 4	→ zwraca 4*Silnia(3)

Zaraz po tym, gdy na wierzchu stosu pojawi się wywołanie dla *Silnia(1)*, pozycja ta ze stosu jest usuwana. Metoda w tym wywołaniu zwraca konkretną wartość i jest przekazana do miejsca wywołania, czyli o jeden „schodek” stosu niżej. Teraz stos będzie wyglądał tak:

Silnia(2)	n= 2	→ zwraca wartość 2*1
Silnia(3)	n= 3	→ zwraca 3*Silnia(2)
Silnia(4)	n= 4	→ zwraca 4*Silnia(3)

I dalej podobnie, wywołanie metody *Silnia()* dla $n=2$ zwraca wartość i jest usuwane ze stosu. Kolejna postać stosu:

Silnia(3)	n= 3	→ zwraca wartość 3*2
Silnia(4)	n= 4	→ zwraca 4*Silnia(3)

W kolejnym kroku:

Silnia(4)	n= 4	→ zwraca wartość 4*6
-----------	------	----------------------

Proszę zwrócić uwagę, że w chwili, gdy na stosie są wszystkie cztery wywołania metody *Silnia()*, to one się wzajemnie „nie widzą”, każda ma na stosie lokalną kopię swoich argumentów i zmiennych (tu jest tylko jeden argument n). My już nie musimy się martwić o wartość argumentu n w poszczególnych wywołaniach, nie sterujemy nią jak wartością zmiennej sterującą w pętli *for*. Dopiero, gdy wywołanie będące na wierzchu stosu jest realizowane, czyli zwracana jest wartość, wartość ta staje się „widoczna” w kolejnym wywołaniu. Jak widać pierwsze wywołanie metody *Silnia()* dla $n=4$ musi „czekać” na stosie najdłużej na wykonanie.

Rekurencyjny kod metody obliczającej silnię porównamy z wersją iteracyjną.

```
static int Silnia(int n)
{
    if (n<=1) return 1;
    else return n*Silnia(n-1);
}
```

```
static int Silnia(int n)
{
    int s = 1;
    for (int i = 1; i <=n; i++)
        s = s * i;
    return (s);
}
```

W kolorze czerwonym⁵² zaznaczono w obu wersjach warunek zakończenia rekurencji i iteracji (w przypadku iteracji koniec następuje, gdy warunek nie jest spełniony). Natomiast w kolorze zielonym zaznaczono odpowiadające sobie fragmenty metod realizujące mnożenie, wykonywane wielokrotnie. Mnożenie 1*2*3*4 (czy w odwrotnej kolejności) jest wykonywane w etapach, w każdym mnożone są tylko dwie liczby, wynik jest zapamiętywany (w wersji iteracyjnej) lub przekazywany (w wersji rekurencyjnej) i wykorzystany w kolejnym etapie.

W kolejnym przykładzie policzymy największy wspólny dzielnik (NWD) dla dwóch podanych liczb przy użyciu algorytmu Euklidesa. Krótko przypomnimy sam algorytm. W wersji podstawowej porównywane są dwie liczby, z tych dwóch liczb tworzymy nową parę: pierwszą z liczb jest liczba mniejsza, natomiast drugą jest różnica liczby większej i mniejszej. Krok ten jest powtarzany tak długo, aż obie liczby staną się sobie równe – wartość tych liczb to szukany największy wspólny dzielnik. Ta wersja algorytmu ma wadę, która ujawnia się, gdy jest znaczna różnica pomiędzy liczbami. Druga wersja algorytmu zamiast odejmowania wykonuje dzielenie modulo, reszta działania algorytmu jest podobna, przy czym algorytm jest wykonywany tak długo, aż jedna z liczb przyjmie wartość zero, a wówczas ostatnia niezerowa reszta jest szukanym dzielnikiem.

Przykład 5.16.

```
class Program
{
    static int NWD(int a, int b)
    {
        if (b == 0) return a;
        else return NWD(b, a % b);
    }
    static void Main(string[] args)
    {
        int a, b;
        Console.WriteLine("Podaj a i b");
        a = int.Parse(Console.ReadLine());
        b = int.Parse(Console.ReadLine());
        Console.WriteLine("NWD = {0}", NWD(a, b));
        Console.ReadKey();
    }
}
```

⁵² Objasnienia dotyczące kolorowej czcionki prosimy sprawdzić w wersji elektronicznej podręcznika.

Metoda rekurencyjna *NWD()* ma dwa argumenty typu *int*, a w ciele jest warunek opuszczenia rekurencji oraz wywołanie rekurencyjne. Zobrazujemy działanie algorytmu dla liczb 27 i 15. W pierwszym kroku $a = 27$, natomiast $b = 15$. Reszta z dzielenia $27 \% 15$ wynosi 12, wówczas nowe wartości liczb wynoszą $a = 15$, $b = 12$. W kolejnym kroku obliczana jest reszta $15 \% 12$, która wynosi 3. Po tym kroku $a = 12$, $b = 3$, reszta z dzielenia równa się 0 (koniec algorytmu), a wynikiem jest ostatnia niezerowa reszta, czyli 3. W przykładzie nie ma jawnego podstawiania aktualnej wartości b , wykorzystuje się do tego celu listę argumentów metody (b to drugi argument metody).

Porównamy wersję rekurencyjną metody *NWD()* z iteracyjną.

```
static int NWD(int a, int b)
{
    if (b == 0) return a;
    else return NWD(b, a % b);
}
```

```
static int NWD(int a, int b)
{
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

Kolorem czerwonym⁵³ oznaczono warunek zakończenia rekurencji i odpowiednio iteracji (iteracja kończy się, gdy warunek nie jest spełniony). Kolorem zielonym oznaczono treść obu metod odpowiadającą za wielokrotne obliczanie reszt z dzielenia i podstawianie nowych wartości badanych par liczb. Przy czym w wersji rekurencyjnej nie ma jawnego podstawiania (przypisywania), a jedynie metoda w kolejnym wywołaniu przyjmuje aktualne wartości argumentów.

Podręcznikowe przykłady – takie jak obliczanie silni czy wyznaczanie elementów ciągu Fibonacciego (zob. zadania do samodzielnego wykonania) – spełniają bardzo dobrze swoją rolę objaśniającą działanie rekurencji, ale nie nadają się jako przykłady praktycznych zastosowań rekurencji w programowaniu. Do praktycznych zastosowań należą m.in.: przeszukiwanie drzewiastych struktur (np. przeszukiwanie katalogów, przeszukiwanie pliku XML), grafika fraktalna. Ponadto rekurencja jest powszechna w języku Prolog oraz w językach funkcyjnych.

Zaletą rekurencji jest przejrzystość kodu, należy jednak pamiętać także o wadach, takich jak wzrost zapotrzebowania pamięciowego programu. Każde kolejne wywołanie metody wymaga zapamiętania danych na stosie. Wadą rekurencji jest także niezależne rozwiązywanie problemów, które mogłyby być rozwiązane w jednym miejscu (raz), co

⁵³ Objaśnienia dotyczące kolorowej czcionki prosimy sprawdzić w wersji elektronicznej podręcznika.

zwiększa złożoność obliczeniową. Klasycznym przykładem takiego „marnotrawstwa” jest algorytm wyznaczania elementu ciągu Fibonacciego – w wersji rekurencyjnej program działa dłużej niż w iteracyjnej, o czym Czytelnik będzie mógł się sam przekonać po wykonaniu jednego z zadań w sekcji z zadaniami do samodzielnego wykonania. Każda konstrukcja programistyczna i jej alternatywy mają swoje wady i zalety, nie można podać uniwersalnych porad. Na początkowym etapie nauki najlepiej jest zapoznawać się z możliwościami języka programowania. Wraz z doświadczeniem programista nabywa coraz więcej umiejętności w zakresie odpowiedniego doboru konstrukcji do zadań, jakie rozwiązuje.

5.10 Zadania do samodzielnego rozwiązania

Zadanie 5.1.

Napisz program zawierający metodę statyczną obliczającą temperaturę w stopniach Fahrenheita na temperaturę w stopniach Celsjusza. Metoda ma przyjmować jeden argument (temperaturę w stopniach Fahrenheita) i zwracać temperaturę w stopniach Celsjusza.

Zadanie 5.2.

Napisz program wczytujący 3 liczby rzeczywiste a , b , x , a następnie wywołujący metodę, która sprawdza, czy liczba x należy do przedziału obustronnie otwartego (a, b) . Metoda sprawdzająca ma zwrócić wartość logiczną, którą należy zinterpretować w metodzie *Main()* z podaniem stosownego komunikatu.

Zadanie 5.3.

Napisz program, który ma znaleźć współrzędne punktu po przesunięciu o dany wektor. W metodzie *Main()* wczytaj od użytkownika współrzędne punktu A oraz zadeklaruj współrzędne wektora *wek* [3, 2], a następnie wywołaj metodę o nazwie *Przesun()*, która ma przesunąć punkt A o wektor *wek* (dodać odpowiednie współrzędne). Współrzędne punktu (jako dwie zmienne typu *double*) mają zostać przesłane do tej metody przez referencję, a współrzędne wektora (także jako dwie zmienne typu *double*) przez wartość. Metoda *Przesun()* ma nic nie zwracać (*void*), aktualne współrzędne punktu mają być pamiętane dzięki użyciu argumentów przesyłanych przez referencje. Program ma wyświetlić współrzędne punktu po przesunięciu o wektor *wek*. Przykładowo, gdyby użytkownik podał początkowe współrzędne punktu A (2, 1), to wówczas program znajdzie położenie punktu A po przesunięciu w miejscu o współrzędnych (5, 3) (czyli 2+3, 1+2).

Zadanie 5.4.

Napisz program, który mnoży elementy tablicy jednowymiarowej przez zadaną liczbę. Mnożenie ma być wykonane w metodzie statycznej przyjmującej jako argumenty tablicę typu *int* oraz liczbę całkowitą (mnożnik).

Wykonaj zadanie w dwóch wariantach:

- Wewnątrz metody tworzona jest nowa tablica wynikowa i ta tablica jest zwracana przez metodę (tablica wejściowa nie zmienia się),
- Wyniki mnożenia elementów tablicy są umieszczane w tablicy wejściowej (wówczas metoda ta ma niczego nie zwracać).

Przykładowo dla tablicy o elementach {1,4,6,8,2} oraz mnożniku 2 program powinien wyświetlić tablicę {2,8,12,16,4}.

Zadanie 5.5.

Napisz program, który wypisze na ekranie znaki w kształcie prostokąta. Program ma prosić użytkownika o podanie rozmiaru prostokąta: długość i szerokość, a następnie znak, którym ma być wypełniony prostokąt. Napisz metodę *Rysuj()*, która wypisze na konsoli prostokąt, przesyłając jako argument długość, szerokość oraz znak wypełnienia. Wywołaj metodę dwa razy, za drugim razem podaj na odwrót argumenty dla długości i szerokości. W wyniku działania programu powinny zostać wyświetlone dwa prostokąty, jeden „leżący” oraz drugi „stojący”. Przykładowy przebieg działania programu na rysunku:

```
Podaj długość: 6
Podaj szerokość: 3
Podaj znak: *

*****
*****
*****

***
***
***
***
***
***
```

Zadanie 5.6.

Uzupełnij program z zadania 5.4 (dowolny wariant) o metodę przeciążoną przyjmującą tablicę typu *string* oraz mnożnik typu *int*. W tym przypadku metoda ma powielać łańcuch znaków (konkatenować tyle razy, ile wynika z mnożnika). Przykładowo dla tablicy o elementach {"ala", "kot", "dom"} oraz mnożniku 2 program powinien wyświetlić tablicę {"alaala", "kotkot", "domdom"}.

Zadanie 5.7.

Napisz statyczną metodę, która oblicza wyrażenie:

$W = (x+1) + (x+2) + (x+3) + \dots + (x+n)$. W metodzie *Main()* wywołaj funkcję dla x i n (liczb naturalnych) wczytanych z klawiatury.

Zadanie 5.8.

Napisz metodę, która oblicza sumę cyfr liczby naturalnej x . W programie głównym wywołaj funkcję dla x wczytanego z klawiatury. Przykładowo jeśli użytkownik wpisze 125, to metoda powinna zwrócić wartość 8 ($1+2+5=8$).

Zadanie 5.9.

Wykonaj program znajdujący n -ty wyraz ciągu Fibonacciego według wzoru

$$F_n = \begin{cases} n & \text{dla } n = 0 \vee n = 1 \\ F_{n-1} + F_{n-2} & \text{dla } n > 1 \end{cases}$$

Wykonaj program w dwóch wariantach: w jednym metoda znajdująca wyraz ciągu ma być rekurencyjna, a w drugim ma wykorzystać iteracyjne podejście (z użyciem pętli).

Zadanie 5.10.

Jaki będzie rezultat metody *Oblicz()* wywołanej z parametrem $n = 5$? Zadanie wykonaj najpierw bez udziału kompilatora, a dopiero później uruchom program i sprawdź otrzymany wynik.

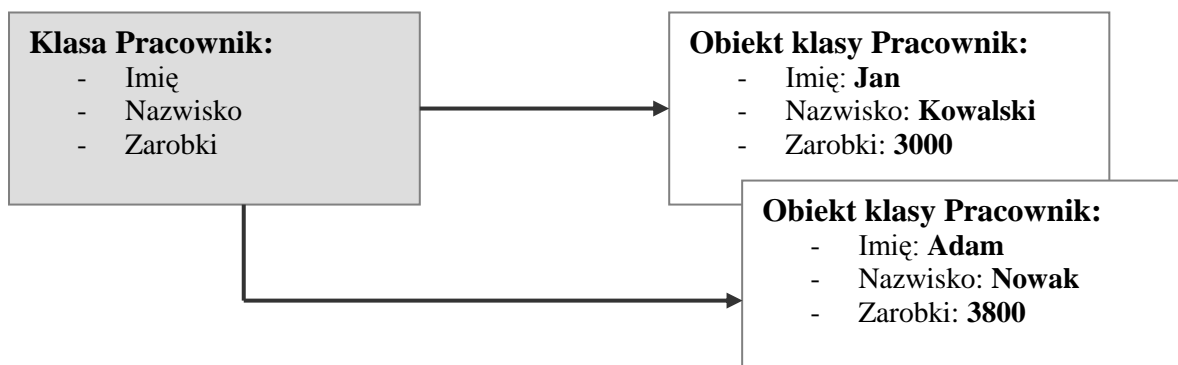
```
static int Oblicz(int n)
{
    if (n <= 1) return (1);
    else return (n + Oblicz(n - 1));
}
```

6 Wprowadzenie do tworzenia klas

Koncepcja programowania obiektowego została wstępnie przedstawiona w pierwszym rozdziale podręcznika ([punkt 1.1.6](#)). Pisaliśmy tam, że poznawany przez nas świat to zbiór obiektów. A obiekty te mogą podlegać określonym dla nich zachowaniom. Obiektowe języki programowania opisują model danego fragmentu rzeczywistości uwzględniając **powiązanie danych obiektu z jego zachowaniem** oraz powiązania między obiektami. Powiązanie danych z zachowaniem nie jest jedyną cechą języków obiektowych, ale na tej skupimy się najpierw. Pozostałe i bardzo ważne cechy zostaną omówione na końcu tego rozdziału – co stanowić będzie jednocześnie „mapę” podstawowych zagadnień paradygmatu obiektowego ze wskazaniem elementów omówionych w tym podręczniku oraz tych, jakie zostaną Czytelnikowi do poznania.

6.1 Klasa a obiekt

Klasa to pewnego rodzaju szablon, który zawiera charakterystykę obiektu tworzonego na podstawie tego szablonu – zarówno jego dane składowe jak i operacje, które można na nim wykonać. Klasą może być np. kot jako gatunek ssaka. Natomiast obiektem tej klasy jest konkretny kot (mający np. swoje imię, wagę, kolor sierści itp.). Obiekt jest zatem egzemplarzem danej klasy. Konkretnego kota możemy pogłaskać czy nakarmić, ale nie możemy tego zrobić z gatunkiem „kot”. Gatunek możemy jedynie opisać, zdefiniować. Wyobraźmy sobie inną klasę – *Pracownik*. Pracowników mogą charakteryzować takie dane jak imię, nazwisko, data urodzenia, adres zamieszkania czy zarobki. Klasa *Pracownik* jedynie wyszczególnia (definiuje) dane opisujące pracowników, a obiektami tej klasy są konkretni pracownicy.

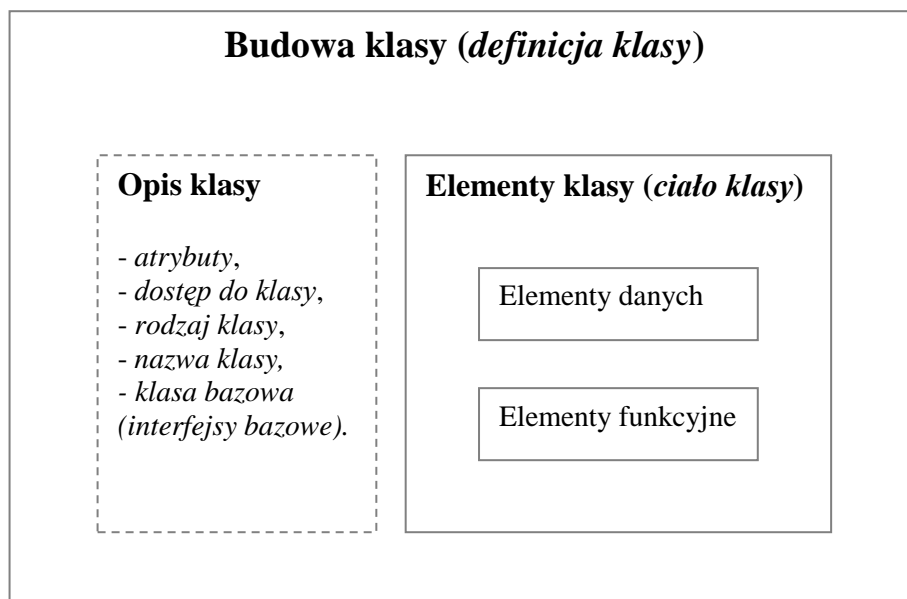


Nazwa klasy powinna wskazywać na obiekty, jakie będzie można na jej podstawie tworzyć. Jeśli obiektami mają być pracownicy, to dobrą nazwą dla takiej klasy jest wyraz *Pracownik*. Obok obowiązkowych zasad dotyczących nazewnictwa w C# (takich samych jak w przypadku nazw zmiennych i metod), istnieją dodatkowe zalecane standardy odnośnie

nazywania klas. Zgodnie z tymi zaleceniami, nazwy klas powinno się tworzyć przy pomocy rzeczowników lub rzeczowników z przymiotnikami (zazwyczaj w liczbie pojedynczej). Ponadto zaleca się, aby nazwy klas zaczynać się z dużej litery, a jeśli nazwa ma kilka słów, kolejne słowa także powinny zaczynać się z dużej litery, np. *PracownikZwolniony*.

6.2 Budowa klasy

Skoro klasa ma być szablonem, na podstawie którego tworzone będą obiekty, należy ten szablon odpowiednio zdefiniować. Budowę klasy opisuje szereg elementów, zostały one zgrupowane i przedstawione na rysunku oraz w składni.



Na rysunku widoczne są dwie grupy elementów składających się na budowę klasy: opis klasy oraz elementy klasy (zawarte w ciele klasy). Składnia wygląda następująco:

Składnia 6.1

```
[Atrybuty][Modyfikatory] class Nazwa [:lista elementów bazowych]
{
    Ciało klasy
}
```

Opis klasy (deklaracja klasy) zawiera dwa elementy obowiązkowe – słowo kluczowe *class* oraz nazwę klasy (identyfikator), a ponadto może zawierać kilka informacji opcjonalnych, opisujących charakter klasy. W prezentowanej składni opis klasy zawiera się w pierwszej linii. Ciało klasy (w składni ujęte w klamrach) definiuje zawartość szablonu (klasy), na podstawie którego będą tworzone obiekty. Poszczególne składowe opisu klasy oraz budowa ciała klasy zostaną przedstawione w dalszej części podrozdziału.

6.2.1 Opis klasy

W opisie klasy, obok słowa kluczowego *class* oraz nazwy klasy (które są obowiązkowe) występować mogą dodatkowe dane opisujące klasę: atrybuty, dostęp do klasy, rodzaj klasy oraz elementy bazowe (po których dziedziczy klasa). Zasady odnośnie tworzenia **nazwy klasy** są takie same jak w przypadku zmiennych i metod, a o dodatkowych zaleceniach pisaliśmy pod koniec podrozdziału 6.1. **Atrybuty** są opisowymi znacznikami, których można używać do tworzenia uzupełniających informacji na temat typów (klas), elementów i właściwości. W tym podręczniku nie będziemy omawiać tego zagadnienia. **Dostęp do klasy** określa dostępność do klasy z zewnątrz (z innych klas). Dostęp do klasy definiuje się poprzez tzw. modyfikatory dostępu (które zostaną wkrótce omówione). Można wyróżnić kilka **rodzajów klasy** – klasa abstrakcyjna, klasa nie podlegająca dziedziczeniu, klasa statyczna oraz „zwykła” klasa. W przypadku tworzenia klasy innej niż ta ostatnia należy umieścić modyfikator rodzaju klasy. Każda klasa może dziedziczyć po innej klasie oraz implementować jeden lub więcej interfejsów, wówczas w opisie klasy należy wskazać po dwukropku nazwę **klasy bazowej** (czy interfejsów). Interfejsy oraz zagadnienie dziedziczenia zostaną w tym podręczniku jedynie wstępnie przedstawione pod koniec rozdziału.

Spośród wymienionych opcjonalnych elementów opisu klasy opiszemy szerzej jedynie modyfikatory (modyfikatory dostępu oraz modyfikatory określające rodzaj klasy). Pozostałe nie będą w początkowym etapie tworzenia własnych klas potrzebne.

Modyfikatory dostępu do klasy

Można wyróżnić pięć rodzajów modyfikatorów dostępu: *public*, *private*, *protected*, *internal* oraz *protected internal*. Przy opisie modyfikatorów używa się terminu *zestaw* (ang. *assembly*) – wykonywalny kod programów w środowisku .NET jest umieszczany w tzw. zestawach (inne polskie nazwy to kompilat, podzespół, pakiet). Modyfikator **public** oznacza, że klasa jest dostępna z poziomu każdego zestawu. Modyfikatory **private** oraz **protected** dotyczą tylko klas zagnieżdżonych – *private* oznacza, że dostęp ogranicza się do klasy zawierającej (czyli tej, w której dana klasa jest zagnieżdżona), a *protected* ponadto daje dostęp klasom potomnym klasy zawierającej. W przypadku modyfikatora **internal** dostęp dotyczy klas z tego samego zestawu – jest to domyślny modyfikator klasy. Modyfikator **protected internal** oznacza, że dostęp ogranicza się do danego zestawu oraz klas potomnych względem klasy zawierającej. W naszych pierwszych klasach będziemy używać modyfikatora *public* i na razie nie jest konieczne, aby Czytelnik dobrze rozumiał wszystkie wymienione modyfikatory.

Modyfikatory określające rodzaj klasy

Dla „zwykłej” klasy nie umieszcza się żadnego modyfikatora dla określenia rodzaju. Istnieją jednak pewne klasy specjalne, dla których jest to konieczne. Wyróżnia się trzy modyfikatory określające rodzaj klasy:

- *abstract* – modyfikator dla klasy abstrakcyjnej. Klasa abstrakcyjna wykorzystywana jest jedynie w postaci klasy bazowej (inne klasy po niej dziedziczą) i nie można na jej podstawie tworzyć obiektów. Klasy abstrakcyjne nie będą omawiane w tym podręczniku,
- *sealed* – modyfikator dla klasy, która nie może pełnić roli klasy bazowej,
- *static* – modyfikator dla klasy statycznej, zawierającej wyłącznie składowe statyczne.

Poniżej dwie przykładowe deklaracje klas:

```
public class Figura
{
    // ciało klasy
}
```

Na podstawie powyższego opisu klasy można odczytać, że klasa *Figura* jest klasą o dostępie publicznym. Ponieważ nie ma modyfikatora rodzaju – oznacza to, że jest to „zwykła” klasa.

```
public static class Osoba
{
    // ciało klasy
}
```

Kolejny przykład przedstawia statyczną klasę *Osoba* o dostępie publicznym. Proszę zwrócić uwagę, że w przypadku umieszczania w deklaracji obu typów modyfikatorów (dostępu i rodzaju), najpierw umieszcza się modyfikator dostępu.

6.2.2 Elementy klasy (ciało klasy)

Wśród elementów klasy możemy wyróżnić dwie grupy: elementy danych oraz elementy funkcyjne, co odzwierciedla charakterystyczne w programowaniu obiektowym powiązanie danych i zachowania (tego co się z danymi dzieje). Elementy danych przechowują niezbędne do działania obiektu dane (np. w klasie *Osoba* – nazwisko osoby, jej wiek, adres itp.). Elementy funkcyjne to zdefiniowane w klasie bloki kodu umożliwiające klasie wykonywanie określonych prac na obiektach. W stosunku do wszystkich elementów klasy (danych i funkcyjnych) można zdefiniować dostęp poprzez modyfikatory dostępu.

Modyfikatory dostępu

Modyfikatory dostępu, zarówno w przypadku dostępu do elementów klasy jak i do całej klasy – są takie same. Interpretacja w kontekście elementów klasy wydaje się być nieco inna, co wynika z faktu, że modyfikatory *private* i *protected* dotyczą tylko składowych klasy (zatem w kontekście klas dotyczą jedynie klas zagnieżdżonych). Ale faktycznie to są te same modyfikatory, opiszemy je ponownie, uwzględniając tym razem kontekst elementów klasy. Modyfikator **public** oznacza, że dany element klasy jest dostępny zarówno z wnętrza jak i spoza klasy. Modyfikator **private** oznacza, że element jest dostępny tylko z wnętrza klasy – ten modyfikator jest domyślny dla elementów klasy. Modyfikator **protected** daje dostęp do danego elementu z wnętrza klasy oraz klasom pochodnym. W przypadku modyfikatora

internal dostęp dotyczy klas z tego samego zestawu. Modyfikator **protected** **internal** oznacza, że składowa zadeklarowana jest widoczna z wnętrza klasy, w której została zadeklarowana (lub klasy pochodnej od niej) oraz z wnętrza zestawu, w którym się znajduje. W analogii do portali społecznościowych takich jak facebook, możemy powiedzieć, że te dane, które użytkownik portalu udostępnia wszystkim są „public”, dane dostępne tylko dla użytkownika są „private”, natomiast dostęp dla znajomych odpowiada modyfikatorowi „protected”. W naszych przykładowych programach będziemy używać tylko modyfikatorów *public* i *private*.

Elementy danych

Elementy danych to tzw. *pola* (lub po prostu *dane*). Składnia definicji pojedynczej danej wygląda następująco:

[Modyfikatory] typ *Nazwa* [= wartość domyślna];

Składnia 6.2

Możliwe modyfikatory to modyfikator dostępu i (lub) modyfikator określający charakter pola (np. statyczne, stałe). Typ oraz nazwa pola to elementy obowiązkowe w definicji pola. Opcjonalnie można przypisać wartość domyślną (dla stałych jest to obowiązkowe).

Przedstawimy przykład definicji klasy z elementami danych i na nim omówimy szczegóły.

Przykład 6.1.

```
public class Pracownik
{
    public string nazwisko;
    private double zarobki;
    public static int liczbaPracownikow;
    public const double etat = 1.0;
}
```

Klasa *Pracownik* zawiera cztery pola:

- *nazwisko* – typu *string*, do którego jest dostęp publiczny,
- *zarobki* – typu *double*, do którego jest dostęp prywatny,
- *liczbaPracownikow* – statyczne typu *int*, do którego jest dostęp publiczny,
- *etat* – stałe typu *double*, do którego jest dostęp publiczny.

W klasie *Pracownik* wszystkie pola mają typ wbudowany⁵⁴, ale w klasach mogą być dane dowolnego typu (np. tablice, typy definiowane przez inne klasy). Spośród czterech wymienionych pól tej klasy dwa mają dodatkowe modyfikatory. Przed nazwą typu pola

⁵⁴ Zob. listę typów wbudowanych w *Dodatkach* (Tabela 1).

liczbaPracownikow jest słowo kluczowe *static*, co oznacza, że jest to składowa statyczna, która dotyczy całej klasy, a nie jednego konkretnego obiektu. W deklaracji danej *etat* użyte zostało słowo kluczowe *const* – oznacza, że jest to dana stała. Stałe należy od razu inicjalizować wartością, której zmienić już nie wolno.

Elementy funkcyjne

Elementy funkcyjne odpowiadają za zachowanie obiektów – są to metody, konstruktory, destruktory, delegaty i zdarzenia, indeksery oraz operatory zdefiniowane przez użytkownika (przeciążone). Do tej grupy przypiszemy także właściwości, mimo że ten element jednocześnie dotyczy elementów danych. Z wymienionych elementów funkcyjnych omówimy w tym podręczniku tylko trzy – *metody*, *konstruktory* oraz *właściwości*. Od tego miejsca wyjaśniać będziemy definicję klasy w oparciu o kompletne przykłady.

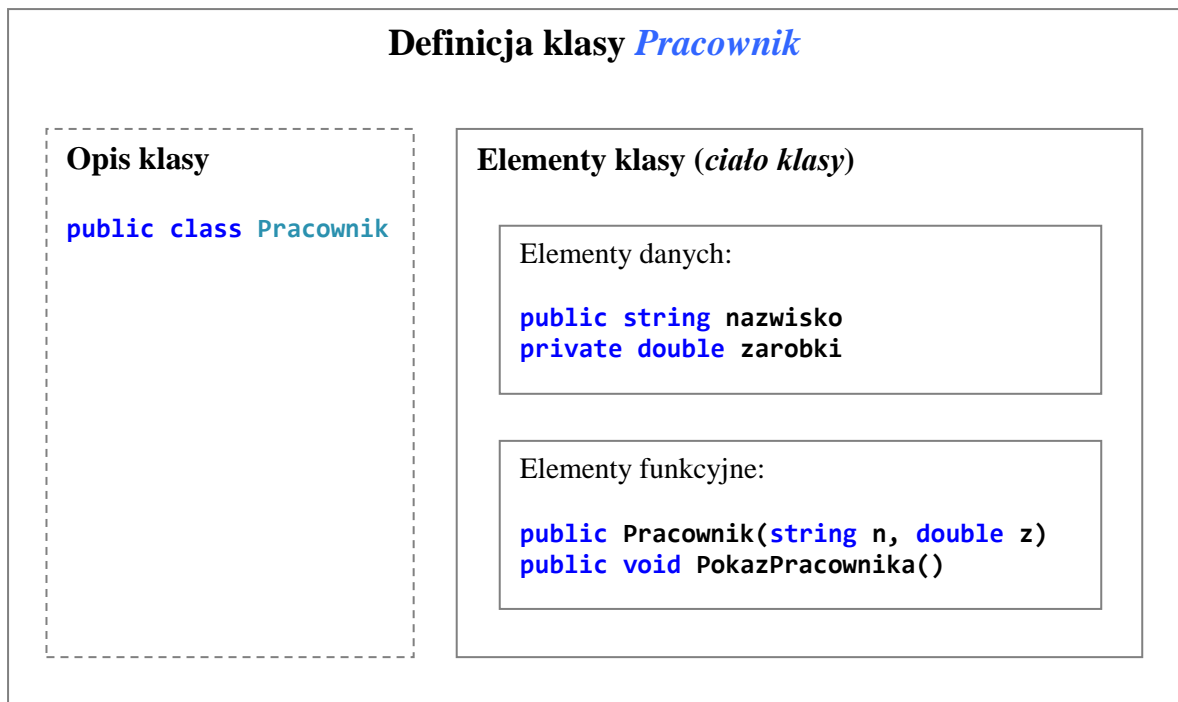
Przykład 6.2.

```
public class Pracownik
{
    public string nazwisko;
    private double zarobki;

    public Pracownik(string n, double z)           // konstruktor
    {
        nazwisko = n;
        zarobki = z;
    }
    public void PokazPracownika()                 // metoda
    {
        Console.WriteLine("{0,-15} {1,10}", nazwisko, zarobki);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Pracownik p1 = new Pracownik("Kowalski", 1000);
        p1.PokazPracownika();
        Console.ReadKey();
    }
}
```

Po uruchomieniu⁵⁵ tego programu wyświetli się nazwisko pracownika i kwota jego zarobków. Przeanalizujemy kod klasy *Pracownik*. Klasa ta posiada dwa pola (*nazwisko* i *zarobki*) oraz dwa elementy funkcyjne: konstruktor oraz metodę, co uwidocznione zostało na poniższym rysunku.

⁵⁵ Testując przykłady z rozdziału 6 należy wkleić cały kod przykładu w miejsce pustej sekcji *class Program {}*, wpisanej automatycznie przez środowisko Visual Studio.



Konstruktor to specjalna metoda, która jest wykonywana w chwili tworzenia obiektu, najczęściej służy do inicjalizacji pól obiektu. Konstruktor musi się nazywać tak samo jak klasa, nie może nic zwracać (nie umieszcza się w jego definicji nawet słowa kluczowego *void*) i musi być publiczny⁵⁶. Konstruktory można przeciążać jak zwykłe metody. W tym przykładzie konstruktor jest tylko jeden, przyjmuje dwa argumenty o typach takich samych jak pola klasy. W ciele konstruktora jest inicjalizacja pól klasy wartościami argumentów. Definiowanie konstruktora nie jest obowiązkowe, do deklaracji obiektu można użyć tzw. konstruktora domyślnego, o tym powiemy w dalszej części rozdziału. W przypadku definiowanego konstruktora nie ma potrzeby, aby były w nim inicjalizowane wszystkie pola – ponieważ te pola, które nie będą jawnie inicjalizowane będą miały przypisane wartości domyślne (w zależności od typu).

Metodę klasy definiuje się według [składni 5.1](#) prezentowanej w poprzednim rozdziale. W tamtym rozdziale opisane zostały zagadnienia dotyczące wszystkich metod, ale przykłady były wykonywane jedynie dla metod statycznych. Metody statyczne nie dotyczą konkretnego obiektu, dlatego w ich przypadku najlepszym sposobem przekazania danych do wnętrza metody było użycie argumentów. Metody „zwykłe” mają dostęp do danych obiektu, dla którego są wykonywane – tych danych nie trzeba przekazywać do metody w postaci argumentów. Publiczna metoda *PokazPracownika()* nic nie zwraca (*void*), ani nie przyjmuje argumentów. W jej ciele jest instrukcja *Console.WriteLine()* wyświetlająca oba pola klasy – nazwisko i zarobki. Metoda ta ma dostęp do obu tych danych (jedna jest z nich *private*, a

⁵⁶ Jeśli konstruktor ma być wykorzystywany standardowo, tzn. do tworzenia obiektów to musi być publiczny. Ale pamiętajmy, że dla klas statycznych (zawierających składowe statyczne) nie ma potrzeby tworzenia obiektów (np. w celu wykorzystania metod z klasy *Math* nie tworzyliśmy obiektów), wówczas konstruktor mógłby być prywatny.

druga *public*), ponieważ jest zdefiniowana wewnątrz klasy i ma dostęp do wszystkich składowych klasy.

6.3 Użycie zdefiniowanej klasy

Omówiona klasa została zdefiniowana w celu jej użycia, tzn. tworzenia obiektów na jej podstawie oraz wykonywania określonych operacji na tych obiektach. Program z [przykładu 6.2](#) wykorzystamy do omówienia deklaracji obiektu oraz wywoływania metody dla obiektów. Przykład ten po drobnych modyfikacjach będzie analizowany w tym podrozdziale jeszcze kilkakrotnie, a w każdym z wariantów będzie akcentowane konkretne zagadnienie.

6.3.1 Deklaracja obiektu

Spójrzmy na zawartość metody *Main()* zdefiniowanej w klasie *Program* z [przykładu 6.2](#). Najpierw omówimy linię:

```
Pracownik p1 = new Pracownik("Kowalski", 1000);
```

Deklarowany jest tu obiekt o nazwie *p1* i zanim przejdziemy do szczegółów wrócimy na chwilę do „zwykłych” zmiennych, które mają typ wartościowy (np. *int*, *double*, *char*). W języku C# każda zmienna jest obiektem. W rozdziale drugim pisaliśmy, że zmienną, definiowaną jako *int x = 0*; można zdefiniować także w taki sposób:

```
int x = new int();
```

Operator *new* tworzy obiekt i wywołuje konstruktor⁵⁷. W dalszej części podręcznika będzie mowa o strukturach, które są podobne do klas, ale różnią m.in. tym, że są typu wartościowego. Ujawniając wiedzę „od kuchni” – wszystkie dane typu prostego (w tym *int*) są zapisane tak naprawdę poprzez struktury. Konstruktor inicjalizuje obiekt wartością domyślną dla danego typu (dla *int* jest to 0). Prezentowany zapis (realizowany według [składni 2.2](#)) w odniesieniu do danych typu wartościowego nie jest konieczny, można użyć samego operatora przypisania: *int x = 0*. Natomiast w przypadku deklaracji obiektów definiowanych klas musimy korzystać z pełnego zapisu. Jest drobna różnica składniowa, mianowicie przy inicjalizowaniu klas można (opcjonalnie) podać listę argumentów dla konstruktora (przypominamy, że konstruktor nazywa się tak samo jak klasa).

Składnia 6.3

```
Typ Nazwa = new Typ([lista argumentów]);
```

⁵⁷ Tworzenie obiektów i wywoływanie konstruktora – to nie jest jedyna rola operatora *new* w języku C#. Na stronie dokumentacji MSDN opisane są wszystkie role <http://msdn.microsoft.com/en-gb/library/51y09td4.aspx>, jednak na obecnym etapie nauki wystarczy znajomość operatora *new* w roli jaką tu omawiamy.

Obiekt można deklarować albo w jednej linii (jak w omawianym przykładzie i jak prezentuje składnia 6.3), albo w dwóch osobnych:

```
Pracownik p1;
p1 = new Pracownik("Kowalski", 1000);
```

Literały "Kowalski" oraz 1000 to argumenty dla konstruktora. Przypomnimy definicję konstruktora z analizowanego przykładu:

```
public Pracownik (string n, double z)      // konstruktor
{
    nazwisko = n;
    zarobki = z;
}
```

Zgodnie z tym co pisaliśmy o konstruktorach, konstruktor ten nazywa się jak klasa (*Pracownik*), jest publiczny i nic nie zwraca (nie ma nawet słowa kluczowego *void*). W ciele tego konstruktora są tylko dwie instrukcje przypisania. Konstruktor przypisuje tu argumenty do obu pól klasy. Konstruktor jak i każda metoda ma dostęp do wszystkich pól własnej klasy.

Deklarowany obiekt *p1* prezentuje pracownika o nazwisku „Kowalski” i zarobkach równych 1000 zł. Obiekty deklaruje się po to, aby wykonać na nich jakąś „pracę”, do czegoś użyć. W naszym prostym przykładzie jest tylko jedna taka „praca” – mianowicie wyświetlenie pól obiektu na ekranie.

6.3.2 Wywołanie metody dla obiektu

Przechodzimy w analizowanym [przykładzie 6.2](#) do ciała metody *Main()*, do instrukcji:

```
p1.PokazPracownika();
```

W instrukcji tejwołana jest metoda *PokazPracownika()* dla obiektu *p1*. Niemal od początku tego podręcznika w wykorzystanych przykładach prezentowaliśmy użycie gotowych klas, dlatego sam zapis wywołania metod nie jest czymś nowym. Teraz jednak jest dobry moment, aby uporządkować informacje na ten temat.

Rodzaj metody	Wywołanie metody
Metoda statyczna	<p>NazwaKlasy.NazwaMetody([lista argumentów])</p> <p>Przykładowe wywołania metod statycznych z klasy <i>Math</i>:</p> <pre>Math.Sin(x) // zwraca sinus Math.Log(x, 2.0) // oblicza logarytm</pre> <p><i>Math</i> to nazwa klasy. Jeśli metoda statyczna jest wołana spoza własnej klasy, musimy poprzedzić nazwę metody kropką i nazwą klasy (zob. przykład użycia metod klasy <i>Math</i> w przykładzie 2.5).</p> <p>Jeżeli metoda statyczna jest wołana wewnątrz klasy, można wywołać ją podając tylko nazwę (zob. przykład 5.1), np. <i>Odejmij(4,3)</i>.</p>

Metoda obiektu („zwykła”)	Obiekt.NazwaMetody([lista argumentów]) Przykładowe wywołanie: <code>p1.PokazPracownika()</code> <i>p1</i> to obiekt, natomiast <i>PokazPracownika()</i> to metoda. Jeżeli metoda jest wołana przez inną metodę tej samej klasy, także można w wywołaniu pominąć nazwę obiektu, ale zaleca się wówczas użyć słowa kluczowego <i>this</i> , np. <i>this.NazwaMetody()</i> .
------------------------------	--

W [podrozdziale 4.2](#) omówiono przykłady metod dla łańcuchów znakowych z klasy *String*, niektóre z tych metod były wywoływane dla klasy, a niektóre dla obiektu. Mamy nadzieję, że po zapoznaniu się z umieszczonymi tu wyjaśnieniami, tamte przykłady nie będą już budzić żadnych wątpliwości.

Czytelnicy, którzy ominęli [rozdział 5](#) powinni (teraz lub wkrótce) do niego zajrzeć. Opisane tam zagadnienia, w szczególności dotyczące definicji metod oraz przekazywania argumentów dotyczą także metod pracujących na obiektach i tu już nie będziemy tych zagadnień ponownie objaśniać.

6.3.3 Zmiana wartości pól obiektu

Wykorzystamy kod [przykładu 6.2](#) po drobnej modyfikacji. Zmiana dotyczy tylko ciała metody *Main()*, dodamy nową linię, w której jest zmiana pola obiektu (*nazwisko*). Taka zmiana pola tuż po deklaracji obiektu nie byłaby specjalnie uzasadniona, ale wykonujemy ją w celu przetestowania wpisywania wartości pola obiektu.

```
class Program
{
    static void Main(string[] args)
    {
        Pracownik p1 = new Pracownik("Kowalski", 1000);
        p1.nazwisko = "Nowak";           // zmiana wartości pola obiektu
        p1.PokazPracownika();
        Console.ReadKey();
    }
}
```

Ponieważ pole *nazwisko* zostało zadeklarowane jako *public*, to jest możliwa zmiana wartości pola obiektu poza macierzystą klasą *Pracownik*. Natomiast próba kompilacji programu z instrukcją *p1.zarobki = 2000;* zakończyłaby się błędem, ponieważ pole *zarobki* ma modyfikator dostępu *private*, czyli nie można tego pola używać poza klasą.

Do pola obiektu odwołujemy się podobnie jak do metody – najpierw piszemy nazwę obiektu, kropkę, a następnie nazwę pola. Poza daną klasą mamy dostęp jedynie do tych elementów klasy (pól i metod), które mają odpowiedni modyfikator dostępu. Przy omawianiu modyfikatorów dostępu dla elementów klasy pisaliśmy, że domyślnym modyfikatorem dla

danych jest *private*. Wydaje się to dość zrozumiałe, że lepiej założyć prywatność takich zasobów jak dane klasy⁵⁸. Takie też jest zalecenie, aby pola klasy miały dostęp *private*. W związku z tym jak komunikować się z obiektem, jeśli w danej klasie wszystkie pola byłyby o dostępie *private*? Wprawdzie można by stworzyć publiczne metody, które zajmowałyby się wpisywaniem danych jak i ich odczytem. Ale dla wygody programisty takie „niby-metody” pełniące rolę komunikacyjną z polami klasy już są. Nazywają się właściwościami. W kolejnym podrozdziale opiszemy jak z nich korzystać.

6.3.4 Właściwości

Właściwości służą do kontrolowania dostępu (odczytu i zapisu) do danego pola klasy. Wśród elementów funkcyjnych wymieniliśmy właściwości z zastrzeżeniem, że dotyczą one także elementów danych. W kolejnym przykładzie jedno z pól (*nazwisko*) ma dostęp *public*, natomiast drugie (*zarobki*) ma dostęp *private*.

Przykład 6.3.

```
public class Pracownik
{
    public string nazwisko;
    private double zarobki;
    public double Zarobki // właściwość dla pola zarobki
    {
        get { return zarobki; }
        set { zarobki = value; }
    }
    public void PokazPracownika() // metoda
    {
        Console.WriteLine("{0,-15} {1,10}", nazwisko, zarobki);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Pracownik p1 = new Pracownik();
        p1.nazwisko = "Kowalski";
        p1.Zarobki = 1250.0; // użycie akcesora set
        p1.PokazPracownika();
        Console.WriteLine(p1.Zarobki); // użycie akcesora get
        Console.ReadKey();
    }
}
```

⁵⁸ Więcej o korzyściach z takiego ukrywania danych będzie w [podrozdziale 6.6](#).

Jak to już sprawdziliśmy, nie można użyć pól prywatnych poza klasą. Tu jednak jest to możliwe, dzięki zdefiniowaniu właściwości o nazwie *Zarobki*. W pewnym uproszczeniu moglibyśmy powiedzieć, że element klasy o nazwie *Zarobki* (tzn. właściwość) to specjalna metoda, która składa się z dwóch „metod wykonawczych” – tzw. **akcesorów** o nazwach *get* i *set*. Akcesor *get* jest wykonywany, gdy w programie użyto danej właściwości „do odczytu”, natomiast akcesor *set* jest wykonywany, gdy program napotka instrukcję przypisania dla danej właściwości (czyli gdy użyto właściwości „do zapisu”). Właściwość ma dostęp publiczny i typ taki sam jak pole, którego dotyczy, a w jej ciele znajdują się definicje obu akcesorów. W omawianym przykładzie oba akcesory zawierają tylko jedną instrukcję (może być ich więcej). Akcesor *get* zwraca wartość pola (*return zarobki;*). Akcesor *set* przypisuje do pola wartość *value* – to nie jest zmienna, ale niejawny parametr, który symbolizuje wartość przypisywaną do właściwości (ma taki sam typ jak dana właściwość). W metodzie *Main()* są dwie instrukcje, które „wywołują” odpowiednie akcesory (oznaczone w komentarzach kodu).

Na marginesie omawianego tematu, proszę zwrócić uwagę na to, że w tym przykładzie brak jest definicji konstruktora. W czasie deklaracji obiektu wykorzystany został tzw. **konstruktor domyślny**, który nie zawiera żadnych argumentów. Konstruktor domyślny inicjalizuje wszystkie pola wartościami domyślnymi (w zależności od typu). Użycie w omawianym przykładzie konstruktora domyślnego nie wynika z faktu, że jest tu właściwość, tzn. bez względu na to czy właściwości są zdefiniowane czy nie – konstruktor może być definiowany jawnie lub domyślnie.

Właściwość powinna mieć dostęp publiczny, ale niekoniecznie oba akcesory muszą być publiczne. Poniżej umieszczono definicję właściwości z modyfikatorem *private* dla akcesora *set*. Dla tak zdefiniowanej właściwości nie można przypisać wartości na zewnątrz klasy, możliwy będzie jedynie jej odczyt.

```
public double Zarobki
{
    get { return zarobki; }
    private set { zarobki = value; }
}
```

W przypadku, gdy właściwości nie mają żadnej dodatkowej pracy do wykonania⁵⁹, co ma miejsce w omawianych przykładach (tzn. akcesor *get* udostępnia daną do odczytu, a *set* do zapisu – i nie robią nic więcej) począwszy od wersji C# 3.0 jest możliwe użycie tzw. **automatycznych właściwości**⁶⁰. Mają one uproszczoną (zwięzłą) składnię i z tego powodu są chętnie używane.

⁵⁹ Właściwości można wykorzystać także do dodatkowych zadań. Np. akcesor *get* może zwrócić wartość po przeliczeniu na odpowiednią jednostkę, a *set* może sprawdzać, czy parametr niejawny *value* ma dozwoloną wartość dla danego pola (zob. przykład <http://msdn.microsoft.com/en-us/library/w86s7x04.aspx>).

⁶⁰ Nazwa „automatyczna właściwość” nie oddaje w pełni „automatycznego” charakteru tej właściwości, gdyby tę nazwę uzupełnić musiałaby brzmieć mniej więcej tak „automatycznie implementowana właściwość z

Przykład 6.4.

W tym przykładzie względem poprzedniego zmieniona została jedynie definicja właściwości. Po tych zmianach program działa tak samo. Omówimy zmiany w składni.

```
public class Pracownik
{
    public string nazwisko;
    public double Zarobki { get; set; }           // automatyczna właściwość

    public void PokazPracownika()                // metoda
    {
        Console.WriteLine("{0,-15} {1,10}", nazwisko, Zarobki);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Pracownik p1 = new Pracownik();
        p1.nazwisko = "Kowalski";
        p1.Zarobki = 1250.0;                      // użycie akcesora set
        p1.PokazPracownika();
        Console.WriteLine(p1.Zarobki); // użycie akcesora get
        Console.ReadKey();
    }
}
```

Po wprowadzonych zmianach z dwóch elementów klasy będących w poprzednim przykładzie, tzn. jednego elementu danych (pola) oraz jednego elementu funkcyjnego (właściwości), zrobił się jeden „wspólny”, który może budzić wątpliwość, czym on właściwie jest – elementem danych czy elementem funkcyjnym. Przypatrzmy się obu rozwiązaniom jednocześnie:

```
// Przykład 6.3
private double zarobki;
public double Zarobki
{
    get { return zarobki; }
    set { zarobki = value; }
}
```

```
// Przykład 6.4 (automatyczna właściwość)
public double Zarobki { get; set; }
```

Po lewej stronie mamy deklaracje prywatnego pola *zarobki* i poniżej publiczną właściwość dla tego pola o nazwie *Zarobki*⁶¹, która definiuje bloki instrukcji dla akcesorów

automatycznym definiowaniem prywatnego pola anonimowego”, ale zostaniemy przy tej krótszej nazwie (w języku angielskim używa się nazwy „auto-implemented properties”).

⁶¹ Zaleca się, aby nazwy pól zaczynać małą literą (ewentualne kolejne słowa od dużej, np. *zarobkiPracownika*), natomiast nazwy właściwości zaleca się pisać od dużej litery (ewentualne kolejne słowa także od dużej, np. *ZarobkiPracownika*).

get i *set*. A teraz przyjrzyjmy się prawej stronie, gdzie jest definicja automatycznej właściwości. Mamy tu tylko jedną składową, która jest publiczna. W tym przypadku kompilator automatycznie utworzy prywatną składową (jako anonimowe pole, do którego dostęp możliwy będzie jedynie przez właściwość). Nie definiujemy tu kodu dla akcesorów, umieszczamy jedynie ich nazwy (które mogą być poprzedzone modyfikatorem dostępu).

Uwaga! Modyfikator *public* dotyczy dostępu do właściwości, a nie składowej (która wygeneruje się automatycznie).

Podobnie jak w przypadku pełnej definicji właściwości i tutaj można wybiórczo ustalić dostęp do akcesorów. Przykładowo jest możliwa deklaracja, która udostępnia akcesor *set* jako prywatny:

```
public double Zarobki {get; private set; }
```

Jak można zauważyć, postać deklaracji automatycznej właściwości upodobniła się „wizualnie” do deklaracji pól. Niemniej nadal jest to definicja właściwości (uproszczonej) i będziemy ją traktować jako element funkcyjny klasy.

6.3.5 Składniki statyczne

Klasa może zawierać składowe statyczne, zarówno pola jak i metody. Składowe statyczne nie wymagają tworzenia obiektów i dotyczą całej klasy, a nie konkretnego obiektu.

Przykład 6.5.

```
public class Pracownik
{
    public static int liczbaPrac;           // pole statyczne
    public string Nazwisko { get; set; }   // właściwość
    public double Zarobki { get; set; }    // właściwość

    public Pracownik(string naz, double zar) // konstruktor
    {
        liczbaPrac++;
        Nazwisko = naz;
        Zarobki = zar;
    }
    static Pracownik()                     //konstruktor statyczny
    {
        liczbaPrac = 0;
    }
    public void PokazPracownika()           // metoda
    {
        Console.WriteLine("{0,-15} {1,10}", Nazwisko, Zarobki);
    }
}
```



```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Liczba {0}", Pracownik.liczbaPrac);

        Pracownik p1 = new Pracownik("Kowalski", 1250.0);
        p1.PokazPracownika();

        Pracownik p2 = new Pracownik("Nowak", 1340.0);
        p2.PokazPracownika();

        Console.WriteLine("Liczba {0}", Pracownik.liczbaPrac);
        Console.ReadKey();
    }
}

```

Wynik programu:

Liczba 0	
Kowalski	1250
Nowak	1340
Liczba 2	

W klasie *Pracownik* dodane zostało jedno pole statyczne *liczbaPrac*, które będzie zawierać liczbę utworzonych obiektów w klasie. Liczba pracowników (obiektów tej klasy) nie jest cechą konkretnego pracownika, ale całej grupy. Liczba ta będzie się zwiększać w trakcie tworzenia obiektów, czyli w konstruktorze (jest tam instrukcja *liczbaPrac++*).

W deklaracji składników statycznych umieszcza się słowo kluczowe *static*. Domyślnie każdy statyczny składnik klasy jest inicjalizowany zgodnie z typem (typ *int* wartością 0). Można też dokonać jawnej inicjalizacji w trakcie deklaracji:

```
public static int liczbaPrac = 0;
```

Powyższa inicjalizacja byłaby wykonana w programie tylko raz, jeszcze przed utworzeniem obiektów (a nie przy tworzeniu każdego obiektu, bo wówczas nie miałoby to sensu). Ale co w sytuacji, gdy do inicjalizacji potrzebne byłoby wykonanie kilku różnych czynności, np. połączenie z bazą danych i pobranie jakiejś wartości? Rozwiązaniem, które jest uniwersalne i pozwala inicjalizować pole statyczne w każdej sytuacji jest **konstruktor statyczny**. W omawianym przykładzie wykorzystaliśmy właśnie to rozwiązanie, mimo że tu akurat konstruktor statyczny nie wykonuje specjalnych zadań (inicjalizuje pole statyczne *liczbaPrac* wartością 0).

Uwaga! Konstruktor statyczny nie może mieć modyfikatorów dostępu ani argumentów. Jest wywoływany automatycznie jeszcze przed utworzeniem obiektów.

Odwołanie do składników statycznych wymaga podania nazwy klasy, znaku kropki oraz nazwy składnika. W przykładzie są dwa odwołania, przed utworzeniem i po utworzeniu dwóch obiektów:

```
Console.WriteLine("Liczba {0}", Pracownik.liczbaPrac);
```

Metody statyczne tworzyliśmy w poprzednim rozdziale, ale wówczas nasza uwaga była kierowana na same metody, a nie na ich role w klasie. Można wymienić co najmniej dwie grupy takich zastosowań – pierwsza stosowana na ogół w klasach statycznych, które w całości zostały pomyślane jako „narzędziowe” (np. klasa *Math*). Druga grupa zastosowań dotyczy zwykłej klasy, w której metoda statyczna wykonuje jakąś pracę dla całej klasy, niezależnie od obiektów lub dla wybranych obiektów (otrzymanych jako argumenty metody). O klasach statycznych krótka wzmianka jest w kolejnym akapicie, natomiast przykład metody statycznej pracującej dla klasy będzie w kolejnym podpunkcie, gdzie wykorzystamy tablicę obiektów jako argument metody statycznej, sumującej zarobki wszystkich pracowników.

Jak już wspomnieliśmy w bieżącym rozdziale można tworzyć klasy statyczne, czyli takie, które posiadają wszystkie składowe statyczne. Poznanym przykładem klasy statycznej jest klasa *Math*, która zawiera metody obliczające różne funkcje matematyczne oraz definicję stałych matematycznych (liczba π , podstawa logarytmu naturalnego e). Do wykorzystania tych funkcji nie potrzebujemy tworzyć żadnych obiektów. Wyżej użyliśmy wobec tego typu klas określenia „narzędziowe”. I na ogół chodzi o uniwersalne „narzędzia”, mogące się przydać w wielu klasach bez względu na to jaki fragment rzeczywistości modelują⁶².

6.3.6 Tablice obiektów

W [przykładzie 6.5](#), aby zademonstrować zliczanie obiektów utworzyliśmy dwa obiekty typu *Pracownik* o nazwach *p1* i *p2*. W przypadku większej liczby obiektów wygodniej będzie posłużyć się kolekcją. W tym podręczniku omówiona została tylko jedna z kolekcji, jaką jest tablica ([podrozdział 4.1](#)). W kolejnym przykładzie obiekty będą umieszczone w **tablicy obiektów**.

Przykład 6.6.

```
public class Pracownik
{
    public string Nazwisko { get; set; }    // właściwość
    public double Zarobki { get; set; }    // właściwość
    public Pracownik(string naz, double zar)    // konstruktor
    {
        Nazwisko = naz;
        Zarobki = zar;
    }
}
```

⁶² Przykład definicji prostej klasy statycznej, której metody zajmują się przeliczaniem jednostek dla temperatury jest na stronie [http://msdn.microsoft.com/en-us/library/79b3xss3\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/79b3xss3(v=vs.90).aspx).

```

public void PokazPracownika() // metoda
{
    Console.WriteLine("{0,-15} {1,10}", Nazwisko, Zarobki);
}
public static double Sumuj(Pracownik[] tab) // metoda statyczna
{
    double suma = 0;
    for (int i = 0; i < tab.Length; i++)
    {
        suma += tab[i].Zarobki;
    }
    return suma;
}
}
class Program
{
    static void Main(string[] args)
    {
        Pracownik[] tab = new Pracownik[3];
        tab[0] = new Pracownik("Kowalski", 1250.0);
        tab[1] = new Pracownik("Nowak", 1340.0);
        tab[2] = new Pracownik("Abacki", 2210.0);

        foreach (Pracownik p in tab)
        {
            p.PokazPracownika();
        }
        Console.WriteLine("Suma {0}", Pracownik.Sumuj(tab));
        Console.ReadKey();
    }
}

```

Wynik programu:

Kowalski	1250
Nowak	1340
Abacki	2210
Suma	4800

Podczas omawiania tego przykładu skupimy uwagę na dwóch nowych elementach – tablicy obiektów oraz metodzie statycznej. Najpierw omówimy tablicę obiektów.

Tablica obiektów

Deklaracja tablicy obiektów nie różni się składniowo od deklaracji „zwykłej” tablicy (zob. [składnia 4.1](#)). Ponieważ tablica obiektów ma zawierać obiekty klasy, to podczas inicjalizacji elementów tablicy musimy użyć składni dla tworzenia obiektów. Można zadeklarować tablicę obiektów jednocześnie inicjalizując jej elementy. Poniżej umieszczono oba alternatywne warianty.

```
// Deklaracja tablicy obiektów i osobno inicjalizacja elementów (jak w przykładzie 6.6)
Pracownik[] tab = new Pracownik[3];
tab[0] = new Pracownik("Kowalski", 1250.0);
tab[1] = new Pracownik("Nowak", 1340.0);
tab[2] = new Pracownik("Abacki", 2210.0);
```

```
// Deklaracja tablicy obiektów wraz z inicjalizacją elementów
Pracownik[] tab = {new Pracownik("Kowalski", 1250.0),
                   new Pracownik("Nowak", 1340.0),
                   new Pracownik("Abacki", 2210.0)};
```

Utworzenie tablicy obiektów ułatwia pracę na obiektach. W programie wykorzystano pętlę *foreach* do uruchomienia metody wyświetlającej pola obiektu. Równie dobrze można by użyć innej pętli. Zaprezentujemy poniżej obok siebie dwa warianty uruchomienia metody dla obiektów w tablicy – z zastosowaniem pętli *foreach* oraz pętli *for*. Zachęcamy Czytelnika, aby przetestował prezentowane alternatywne warianty.

```
// Pętla foreach (jak w przykładzie 6.6)
foreach (Pracownik p in tab)
{
    p.PokazPracownika();
}
```

```
// Pętla for
for (int i = 0; i < tab.Length; i++)
{
    tab[i].PokazPracownika();
}
```

Pętla *foreach* będzie pracować na obiektach typu *Pracownik* umieszczonych w tablicy *tab*. Dla każdego obiektu, który jest w tej tablicy (dla każdego pracownika) zostaną wyświetlone jego dane. W pętli *for* wykorzystano właściwość *Length* zwracającą rozmiar tablicy, a w ciele pętli wywołano metodę *PokazPracownika()* dla każdego obiektu umieszczonego w tablicy *tab*.

Metoda statyczna

W programie zdefiniowano metodę statyczną obliczającą sumę zarobków dla wszystkich pracowników. Suma zarobków nie jest cechą jednego konkretnego pracownika, ale dotyczy klasy *Pracownik*. Metoda statyczna *Sumuj()* przyjmuje jako argument tablicę obiektów typu *Pracownik*, a zwraca daną typu *double* (sumę zarobków). Wywołanie tej metody w klasie *Program*, w metodzie *Main()* wymaga podania nazwy klasy, kropki oraz nazwy metody wraz z argumentem (nazwą tablicy).

6.4 Typ referencyjny w kolejnej odsłonie

Na początku rozdziału drugiego, w [punkcie 2.1.1](#) pisaliśmy o typach referencyjnych i wartościowych. W wielu podręcznikach opisuje się te zagadnienia z użyciem fachowych terminów związanych z przydzielaniem pamięci w środowisku uruchomieniowym, ale opisy

te nie są dość przystępne dla osób zaczynających przygodę z programowaniem. W naszym podręczniku wyjaśniamy ten problem „w odcinkach”, stopniowo odsłaniając jego meritum.

W rozdziale drugim posłużyliśmy się metaforą archiwum, w którym znajdują się segmenty pełne szuflad (analogia sterty) oraz biurko z bieżącymi dokumentami (analogia stosu). Mówiliśmy, że na stercie umieszczane są dane typów referencyjnych, do których m.in. należą typy definiowane w klasach. Wówczas na stosie umieszczane są referencje do obiektów, a obiekty na stercie. Naszego Czytelnika, po kursie objętym w tym podręczniku, zachęcamy, aby pogłębił zagadnienia dotyczące przydziału pamięci w środowisku .NET w odpowiednich materiałach kierowanych dla zaawansowanych programistów. Tu jedynie czynimy starania, aby po przeczytaniu naszej książki fachowe źródła stały się bardziej zrozumiałe. Omówimy przykłady obrazujące różnice w przypisywaniu danych typu referencyjnego i typu wartościowego.

Zacniemy od przykładu, który nam przypomni zachowanie typu wartościowego.

Przykład 6.7.

```
static void Main(string[] args)
{
    int p1 = 5, p2 = p1;
    Console.WriteLine("p1 = {0}", p1);
    Console.WriteLine("p2 = {0}", p2);
    p1 = 8;
    Console.WriteLine();
    Console.WriteLine("Wartości po zmianie p1");
    Console.WriteLine("p1 = {0}", p1);
    Console.WriteLine("p2 = {0}", p2);
    Console.ReadKey();
}
```

Po uruchomieniu tego przykładu widzimy na ekranie:

```
p1 = 5
p2 = 5

Wartości po zmianie p1
p1 = 8
p2 = 5
```

Początkowo obie zmienne *p1* i *p2* (typu *int*) miały tę samą wartość. Ponieważ obie są typu wartościowego, to w chwili przypisania *p2=p1* do zmiennej *p2* została przekopiowana wartość (a nie adres). Gdy później zmienna *p1* zmieniła wartość na 8, to zmienna *p2* ma nadal wartość 5.

A teraz zobaczmy jak to wygląda w przypadku przypisania danej typu referencyjnego.

Przykład 6.8.

```

public class MojaKlasa
{
    public int Dana { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        MojaKlasa p1 = new MojaKlasa();
        p1.Dana = 5;
        MojaKlasa p2 = p1;

        Console.WriteLine("p1.Dana = {0}", p1.Dana);
        Console.WriteLine("p2.Dana = {0}", p2.Dana);

        p1.Dana = 8;

        Console.WriteLine();
        Console.WriteLine("Wartości po zmianie obiektu p1");

        Console.WriteLine("p1.Dana = {0}", p1.Dana);
        Console.WriteLine("p2.Dana = {0}", p2.Dana);

        Console.ReadKey();
    }
}

```

Wynik tego programu:

```

p1.Dana = 5
p2.Dana = 5

Wartości po zmianie obiektu p1
p1.Dana = 8
p2.Dana = 8

```

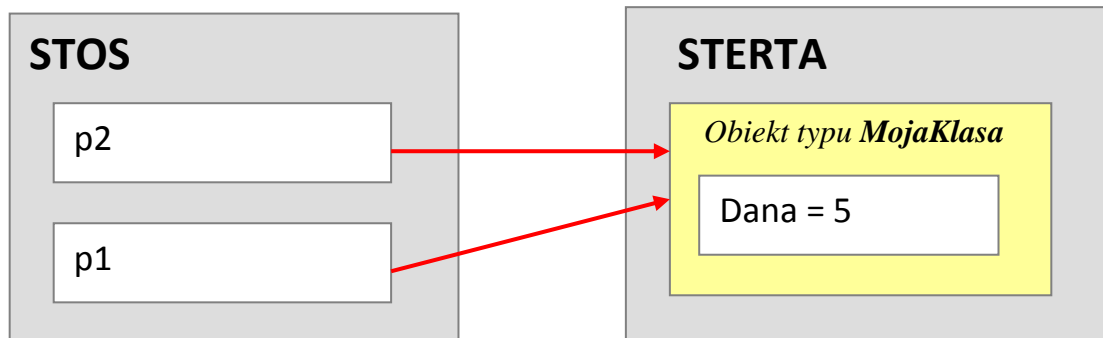
W tym programie jest zdefiniowana klasa o nazwie *MojaKlasa*, posiada ona tylko jedną składową – właściwość *Dana*. Nie ma zdefiniowanego konstruktora, czyli zostanie wygenerowany automatycznie konstruktor domyślny. W metodzie *Main()* deklarowany jest obiekt *p1* typu *MojaKlasa*. Deklarowany jest poprzez operator *new*, który fizycznie zarezerwuje pamięć dla obiektu. Poniżej jest deklarowany dla tej samej klasy obiekt *p2*, ale jest inicjalizowany poprzez przypisanie obiektu *p1*. Adekwatna operacja, która była w poprzednim przykładzie dla zmiennych typu *int*:

```
int p1 = 5, p2 = p1;
```

spowodowała, że w zmiennej *p2* była kopia wartości zmiennej *p1*. W innym miejscu podręcznika użyliśmy przenośnego sformułowania – że obie zmienne „żyją własnym życiem”. Zmiana jednej zmiennej nie wpływa na wartość drugiej. Tak było w poprzednim przykładzie, który dotyczył typów wartościowych. W bieżącym przykładzie *p1* i *p2* to dane typu referencyjnego. Instrukcja:

```
MojaKlasa p2 = p1;
```

nie powoduje kopiowania wartości, ale kopiuje jedynie referencję. Przydział pamięci dla tego przypadku można zaprezentować w sposób uproszczony na rysunku:



Wracając do analogii z archiwum - w chwili, gdy wykonana jest powyższa instrukcja przypisania obiektu, dodawana jest na stos (biurko) jeszcze jedna referencja do tego samego obiektu, będącego wciąż w tym samym miejscu na stercie (w segmencie). Nie ma w deklaracji obiektu *p2* operatora *new*, nie ma zatem nowego „bytu”. Jest tylko jeszcze jedno odwołanie do tego samego miejsca. Konsekwencje tego widać w dalszym działaniu programu. Po zmianie właściwości *Dana* dla obiektu *p1* – zmianę widać zarówno poprzez odwołanie *p1.Dana* jak i *p2.Dana*.

Gdybyśmy chcieli mieć w programie oddzielne dwa obiekty, a nie tylko dwie referencje do jednego obiektu (z których jeden jest kopią drugiego), oba musiałyby być stworzone przez operator *new*, a efekt kopiowania danych składowych można by zapewnić poprzez zdefiniowanie **konstruktora kopiującego**. Takie rozwiązanie przedstawia kolejny przykład⁶³.

Przykład 6.9.

```
public class MojaKlasa
{
    public int Dana { get; set; }
    public MojaKlasa(int d)           // konstruktor
    {
        this.Dana = d;
    }
}
```

⁶³ W omawianym przykładzie jest tylko jedna składowa (właściwość *Dana*). Korzyść z zastosowania konstruktora kopiującego (w celu kopiowania składowych obiektu) jest szczególnie widoczna w przypadku klas posiadających więcej składowych.

```

    public MojaKlasa(MojaKlasa kopia)    // konstruktor kopiujący
    {
        this.Dana = kopia.Dana;
    }
}
class Program
{
    static void Main(string[] args)
    {
        MojaKlasa p1 = new MojaKlasa(5);
        MojaKlasa p2 = new MojaKlasa(p1); // wywołanie konstr. kopiującego

        Console.WriteLine("p1.Dana = {0}", p1.Dana);
        Console.WriteLine("p2.Dana = {0}", p2.Dana);

        p1.Dana = 8;

        Console.WriteLine();
        Console.WriteLine("Wartości po zmianie obiektu p1");

        Console.WriteLine("p1.Dana = {0}", p1.Dana);
        Console.WriteLine("p2.Dana = {0}", p2.Dana);

        Console.ReadKey();
    }
}

```

Wynik programu:

```

p1.Dana = 5
p2.Dana = 5

Wartości po zmianie obiektu p1
p1.Dana = 8
p2.Dana = 5

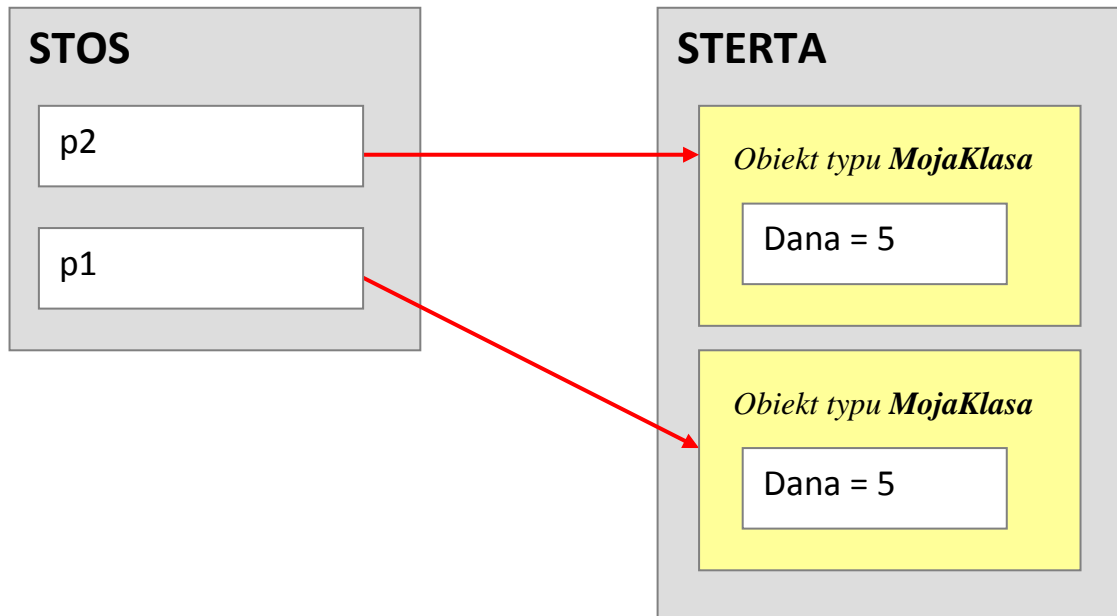
```

W przykładzie tym są dwa przeciążone konstruktory, jeden „zwykły” (inicjalizujący), a drugi kopiujący. Konstruktor kopiujący przyjmuje jako argument obiekt tego samego typu, co definiowana klasa. W tym programie konstruktor kopiujący nie ma zbyt wiele pracy, ponieważ jest tylko jedno pole. W obu konstruktorach użyto słowa kluczowego *this*, które pozwala na odwołanie do bieżącego obiektu (dla którego np. wołany jest konstruktor albo metoda). W tym programie *this* nie było konieczne (program i tak by działał dobrze), ale w ciele konstruktora kopiującego było to szczególnie zalecane ze względów „estetycznych”, aby podkreślić, że pole *Dana* dla bieżącego obiektu ma być wypełnione wartością pola o tej samej nazwie innego obiektu. Użycie słowa kluczowego *this* byłoby niezbędne, gdyby w ciele danej metody lub konstruktora była zmienna lub argument o tej nazwie co składowa.

W [przykładzie 6.9](#) oba obiekty zostały stworzone przy pomocy operatora *new*. Dzięki temu na sterce są fizyczne dwa obiekty, a po wykonaniu instrukcji:

```
MojaKlasa p2 = new MojaKlasa(p1);
```

uproszczony schemat przydziału pamięci wygląda następująco:



Po deklaracji obiektu *p2* z użyciem konstruktora kopiującego na sterce powstaje drugi obiekt o tych samych składowych (tu tylko jedna składowa *Dana*). Zmiana jednego obiektu nie będzie wpływać na stan drugiego obiektu.

6.5 Struktury a klasy

Struktura jest typem tworzonym przez programistę, przypominającym klasę. Ponieważ już poznaliśmy podstawy budowy klas, zaczniemy omawiać struktury od wskazania różnic między klasami a strukturami:

- Struktura jest typem wartościowym (klasa jest typem referencyjnym),
- Struktura nie może dziedziczyć po klasie, ani być przedmiotem dziedziczenia (ale może implementować interfejsy),
- Struktury nie obsługują destruktorów⁶⁴,
- W strukturze nie można zadeklarować konstruktora bez argumentów,
- Składowe struktury nie mogą być inicjalizowane w momencie deklarowania,
- Różnica w składni – dla struktur używa się słowa kluczowego *struct* (dla klas *class*)
- Domyślny poziom dostępu do składowych w strukturach jest publiczny (w klasach prywatny).

⁶⁴ Nie omawialiśmy destruktorów dla klas – są wykorzystywane w procesie odzyskiwania pamięci.

Struktura jest typem wartościowym, co stanowi jedną z ważniejszych różnic względem klas. W nawiązaniu do poprzedniego podrozdziału, gdyby w [przykładzie 6.8](#) zmienić tylko jedną rzecz – mianowicie słowo kluczowe *class* na *struct* wówczas wynik programu byłby taki sam jak dla [przykładu 6.7](#) (dla danych typu wartościowego). Zachęcamy Czytelnika, aby to sprawdził. W przypadku struktury instrukcja podobna do poniższej:

```
DefiniowanyTyp p2 = p1;
```

powoduje, że na stosie kopiowane są wszystkie składowe danego obiektu (a nie referencja). To tłumaczy, dlaczego zmiana dla jednego z dwóch obiektów (p1 lub p2) nie powoduje jednocześnie zmiany dla drugiego (mamy wówczas dwa obiekty, a nie dwie referencje do jednego obiektu).

Typy, których obiekty zawierają same składowe typu wartościowego bardzo często implementowane są właśnie jako struktury. I tak przy użyciu struktur definiowane są typy proste. Inne przykłady struktur w środowisku .NET to struktura *Color* mająca właściwości określające kolor (np. Red, White, Blue i itd.), a także struktura *DateTime*, która jest wykorzystywana w działaniach na datach i godzinach. Poniżej umieszczamy przykład wykorzystujący strukturę *DateTime*.

Przykład 6.10.

```
static void Main(string[] args)
{
    DateTime t1 = DateTime.Now;
    Console.WriteLine("Czas początkowy t1: {0}", t1);
    int licznik = 0;
    for (int i = 0; i < int.MaxValue; i++)
        licznik++;
    Console.WriteLine("Wartość zmiennej licznik {0}", licznik);
    DateTime t2 = DateTime.Now;
    Console.WriteLine("Czas końcowy t2: {0}", t2);
    Console.WriteLine("Różnica t2-t1: {0}", t2 - t1);
    Console.WriteLine("Dziś jest {0} dzień roku", t1.DayOfYear);
    Console.ReadKey();
}
```

Wynik programu:

```
Czas początkowy t1: 2014-01-05 20:45:12
Wartość zmiennej licznik 2147483647
Czas końcowy t2: 2014-01-05 20:45:18
Różnica t2-t1: 00:00:05.8906250
Dziś jest 5 dzień roku
```

Program rozpoczyna się od deklaracji obiektu struktury *DateTime* o nazwie *t1*. Przypisana jest temu obiektowi wartość właściwości *Now* (bieżący czas)⁶⁵. W pętli *for*

⁶⁵ Pozostałe właściwości struktury *DateTime* opisane są na stronie MSDN [http://msdn.microsoft.com/en-us/library/system.datetime_properties\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.datetime_properties(v=vs.110).aspx)

zwiększana jest wartość zmiennej *licznik*, a my sprawdzimy jak długo to zadanie jest wykonywane. Proszę zwrócić uwagę na warunek zakończenia działania pętli – użyliśmy tam właściwości dla struktury *Int32* (mającej alias „*int*”) o nazwie *MaxValue*. Właściwość ta zwraca największą możliwą wartość dla danej określonego typu. Użycie tej właściwości pozwoli nam wydłużyć czas działania pętli bez ryzyka, że nie zmieścimy się w limicie dla typu *int*. Po uruchomieniu programu zobaczymy na konsoli najpierw czas początkowy *t1*. Po kilku sekundach pojawiają się kolejne linie, które będą zawierać: wartość zmiennej *licznik* (równą właściwości *MaxValue*), czas końcowy (*t2*) oraz różnicę obu czasów (*t2 – t1*). Zgodnie z wynikiem prezentowanym wyżej czas wykonania pętli wyniósł blisko 6 sekund. W ostatniej linii wyświetla się dzień roku, w tym celu wykorzystana została właściwość *DayOfYear*.

Definiowanie struktury niewiele różni się od definiowania klasy. Kolejny przykład zawiera definicję struktury *Kwadrat*.

Przykład 6.11.

```
struct Kwadrat
{
    int bok;
    ConsoleColor kolor;
    public Kwadrat(int bok1, ConsoleColor kolor1)
    {
        bok = bok1;
        kolor = kolor1;
    }
    public void RysujKwadrat()
    {
        Console.ForegroundColor = kolor;
        for(int i = 1; i <= bok; i++)
        {
            for(int j = 1; j <= bok; j++)
                Console.Write("*");
            Console.WriteLine();
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Kwadrat k1 = new Kwadrat(5, ConsoleColor.Blue);
        k1.RysujKwadrat();
        Console.ReadKey();
    }
}
```

Wynikiem tego programu jest „narysowany” ze znaku gwiazdki kwadrat w kolorze niebieskim. Struktura *Kwadrat* ma dwa pola: długość boku kwadratu oraz kolor. Kolor dla konsoli zapisany jest przy użyciu danej typu wyliczeniowego *ConsoleColor*. O typie wyliczeniowym nie mówiliśmy w tym podręczniku, zachęamy do zapoznania się z tym typem. Tu wystarczy wspomnieć, że typ wyliczeniowy jest typem wartościowym i że za jego pomocą można zdefiniować listę wartości, np. dni tygodnia, kolory w palecie, itp.^{66,67}.

Zdefiniowany konstruktor inicjalizuje oba pola struktury. Natomiast publiczna metoda *RysujKwadrat()* „rysuje” tekstowo kwadrat ze znaku gwiazdki. Kolor oraz długość boku kwadratu (czyli liczba znaków) są polami składowymi tej struktury i metoda ta z nich korzysta. Zawartość pola *kolor* jest przypisywana do właściwości *Console.ForegroundColor*, która określa kolor wyświetlanych znaków.

Obie konstrukcje programistyczne, klasa oraz struktura, są podobne, ale nieidentyczne i może zrodzić się pytanie, od czego uzależniać decyzję, którą z nich zastosować w danym miejscu programu. Najpierw spójrzmy na planowany kod programu pod kątem dziedziczenia (w przypadku struktur nie ma obsługi dziedziczenia). Przy założeniu, że nie potrzebujemy dziedziczenia kierujemy się w wyborze (użyć klasy czy struktury) kwestią pamięci. Rezerwacja i zwalnianie pamięci na stosie jest zdecydowanie szybsze od tych samych operacji wykonywanych na stercie. Dlatego dla przechowywania danych o niewielkich rozmiarach lepiej jest wykorzystać typy wartościowe, a dla danych o dużych rozmiarach lepiej użyć typów referencyjnych. Ponieważ struktura jest typem wartościowym a klasa jest typem referencyjnym stosujemy klasę, gdy ma ona obsługiwać duże zbiory danych, a strukturę, gdy ma pracować na niewielkich zbiorach. Ważna jest też częstość wywołań metod mających jako argument obiekt (obiekty) – jeśli jest duża to lepiej zastosować klasę.

6.6 Cechy programowania obiektowego

W programowaniu obiektowym programy definiuje się za pomocą obiektów, łączących stan (elementy danych) i zachowanie (elementy funkcyjne). Program komputerowy w paradygmacie obiektowym możemy sobie wyobrazić jako „ożywienie” takich obiektów i podjęcie przez nie współpracy w celu wykonywania określonego zadania. Aby taka „współpraca” była możliwa konieczne są mechanizmy pozwalające powiązać klasy obiektów oraz określenie zasad współpracy.

Język C# wspomaga trzy główne cechy programowania obiektowego: hermetyzację, dziedziczenie, polimorfizm. Wymienione cechy programowania obiektowego omówimy krótko wraz z podaniem konstrukcji językowych, które je realizują. W tym

⁶⁶ Szerzej na temat typu wyliczeniowego na stronie MSDN: <http://msdn.microsoft.com/pl-pl/library/sbdt4032.aspx>

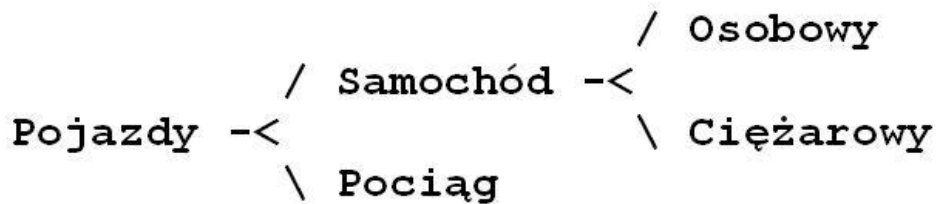
⁶⁷ Dostępne kolory dla danej typu wyliczeniowego *ConsoleColor* wymienione są na stronie MSDN: [http://msdn.microsoft.com/pl-pl/library/system.consolecolor\(v=vs.110\).aspx](http://msdn.microsoft.com/pl-pl/library/system.consolecolor(v=vs.110).aspx)

podręczniku przedstawiliśmy tylko niektóre z tych konstrukcji, przez co fragmenty nawiązujące do pozostałych mogą być nie dość jasne. Umieszczamy je jednak, bo jednocześnie są wskazaniem – co zostało Czytelnikowi do poznania w zakresie nauki programowania.

Hermetyzacja (enkapsulacja) – to kontrolowany dostęp do wnętrza klasy. Hermetyzacja polega na ukrywaniu pewnych danych składowych lub metod danej klasy, tak aby były one dostępne tylko metodom wewnętrznym danej klasy lub innym „uprawnionym”. Zasadę hermetyzacji stosuje się powszechnie w życiu, użytkownicy sprzętu RTV czy AGD mają do dyspozycji określone przyciski na obudowie i nie jest wskazane, aby zaglądali do środka i usiłovali „naprawiać”. Natomiast producent oraz jego serwisy mają już takie uprawnienia i kompetencje. Motywacją ukrywania pewnych elementów przed użytkownikiem (rzeczy albo klasy w programie) jest przede wszystkim ograniczenie sytuacji błędnych. Hermetyzację umożliwiają *modyfikatory dostępu* (np. *private*, *protected*) oraz tzw. *interfejsy*, które określają możliwe metody współpracy. *Właściwości*, które służą do kontroli dostępu do pól klasy także wspierają hermetyzację. Modyfikatory dostępu oraz właściwości omówiliśmy w tym podręczniku, interfejsów nie.

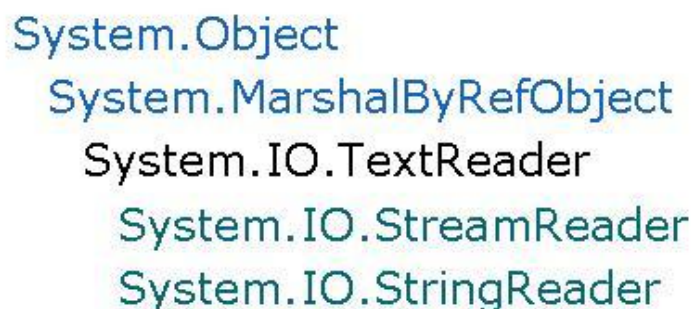
Polimorfizm to inaczej wielopostaciowość (wielość form). W świecie rzeczywistym otaczają nas obiekty pod pewnymi względami podobne, ale nieidentyczne. Podczas pisania programu wygodnie jest traktować takie różne (ale spokrewnione) dane w jednolity sposób, np. wywołać zarówno dla koła jak i kwadratu metodę obliczającą pole figury. Jeśli poszczególne figury, kwadraty czy koła są obiektami różnych klas, a ponadto oczekujemy aby „jednolity sposób” polegał na wywołaniu metody o tej samej nazwie i liście argumentów (ale różnym działaniu) – to przeciążanie metod nie jest rozwiązaniem. Potrzebny jest mechanizm, który obsłużyłby wielopostaciowość nie tylko w ramach jednej klasy, ale w obrębie grupy powiązanych między sobą klas. I to zapewnia właśnie polimorfizm (wraz z dziedziczeniem). Mówimy w programowaniu o dwóch rodzajach polimorfizmu: statycznym i dynamicznym. W wersji statycznej wielopostaciowość udostępniają takie konstrukcje jak *typy generyczne* (wykorzystywane w kolekcjach z typem $\langle T \rangle$) oraz przeciążone operatory (statyczny polimorfizm jest ustalany na etapie kompilacji). W wersji dynamicznej wielopostaciowość udostępniana jest poprzez *metody wirtualne* (*virtual*) – konkretna wersja metody może być ustalona dopiero w czasie wykonywania programu (na podstawie typu obiektu). Żadnej z konstrukcji językowych wspierających polimorfizm nie omawialiśmy w tym podręczniku.

Dziedziczenie w programowaniu to mechanizm wyrażania podobieństw między klasami, dzięki któremu można uprościć definicje klas do już zdefiniowanych.



Na rysunku prezentujemy przykładowy diagram dziedziczenia. Klasy *Osobowy* i *Ciężarowy* dziedziczą po klasie *Samochód*, czyli inaczej powiemy, że to są klasy pochodne klasy *Samochód*. *Samochód* jest klasą bazową dla obu tych klas. Natomiast *Samochód* i *Pociąg* to klasy pochodne klasy *Pojazdy*. Cechy wspólne dla wszystkich pojazdów można zdefiniować w klasie bazowej *Pojazdy* (np. maksymalna prędkość pojazdu). Dzięki temu definicja klas *Samochód* oraz *Pociąg* będzie uproszczona (bo będzie zawierać tylko te cechy, które są charakterystyczne dla danego rodzaju pojazdu, np. liczba wagonów w klasie *Pociąg*). Dziedziczenie wspomaga polimorfizm oraz hermetyzację. Konstrukcje językowe, które umożliwiają dziedziczenie są wyspecyfikowane w składni: klas (*class*), klas abstrakcyjnych (*abstract class*) oraz interfejsów (*interface*), w miejscu, gdzie po dwukropku podaje się nazwę klasy bazowej lub listę interfejsów, jakie klasa ma implementować. Ponadto, używa się słowa kluczowego *base* do wywołania konstruktora lub metody klasy bazowej. W języku C# klasa może dziedziczyć tylko po jednej klasie, ale może dziedziczyć po wielu interfejsach. W odniesieniu do interfejsów częściej mówi się, że klasa implementuje interfejs niż dziedziczy po nim, co wynika z charakteru interfejsu, który jest czymś w rodzaju „szablonu” obowiązkowych składowych klasy. Tych zagadnień nie omawialiśmy w tym podręczniku.

Temat dziedziczenia trochę rozwinie, czyniąc niejako wstęp do tego zagadnienia. W języku C# wszystkie typy (klasy) dziedziczą po klasie *System.Object* i tworząc własne klasy nie definiujemy jawnie tego powiązania. Używaliśmy wielokrotnie bibliotecznej klasy *Console* (posiadającej m.in. metodę *WriteLine()*). W dokumentacji MSDN możemy sprawdzić diagram dziedziczenia (ang. inheritance hierarchy) dla tej klasy. Klasa *Console* dziedziczy bezpośrednio po klasie *System.Object*. Natomiast diagram dziedziczenia dla bibliotecznych klas *StreamReader* oraz *StringReader* (do obsługi tzw. strumieni) wygląda następująco:



W nazwie klasy *System.Object* – słowo *System* to nazwa tzw. **przestrzeni nazw** (ang. namespace). Stosowanie przestrzeni nazw pomaga porządkować kod projektów. Nie musimy się martwić, że w jednym programie (pisanym przez zespoły programistów) pojawią się klasy o tej samej nazwie i spowoduje to konflikt. Bo nawet jeśli się pojawią, to będą w oddzielnej przestrzeni nazw i będą przez program widziane jako różne. Przestrzeń nazw moglibyśmy porównać do kontekstu. Przestrzenią nazw może być przykładowo mieszkanie Jana, jego rodzina nie musi wołać go po imieniu i nazwisku, wystarczy samo imię. W bloku obok, w innym mieszkaniu może mieszkać inny Jan. Ale to drugie mieszkanie to inna przestrzeń nazw. Dopiero gdy obaj panowie o imieniu „Jan” spotkają się gdzieś razem, albo będzie mowa o tej dwójce – wówczas używanie przestrzeni nazw związanej z rodziną czy mieszkaniem (np. nazwisko lub adres) będzie przydatne do jednoznacznej identyfikacji. Na początku programu można określić używane w programie przestrzenie nazw przy pomocy instrukcji *using*. Przykładowo jeśli mamy w programie instrukcję *using System*; to możemy użyć metody klasy *Console* bez podawania pełnej nazwy (z przestrzenią nazw), wystarczy tylko nazwa klasy. I tak robiliśmy w naszych programach – pisaliśmy *Console.WriteLine()* zamiast *System.Console.WriteLine()*.

Wracając do dziedziczenia, co to znaczy, że wszystkie klasy dziedziczą po klasie *System.Object*? Oznacza to m.in., że wszystkie klasy mają dostęp do metod zdefiniowanych w klasie *System.Object*, np.:

- *Equals()* – sprawdza równość referencji (ale nie wartości obiektów),
- *GetType()* – zwraca typ obiektu,
- *ToString()* – zwraca ciąg reprezentujący aktualny obiekt.

Czytelnik może się łatwo o tym przekonać kopiując poniższe dwie linie do programów z [przykładów 6.8](#) i [6.9](#) (na końcu metody *Main()*):

```
Console.WriteLine(p1.ToString());
Console.WriteLine(p1.Equals(p2));
```

Zanim krótko omówimy, co zrobią te metody dla naszych obiektów – prosimy zwrócić uwagę na to, że tych metod nie definiowaliśmy w klasie *Pracownik*, a wołane są dla obiektów tej klasy. Możemy ich używać właśnie dzięki dziedziczeniu. Tak jak mówiliśmy klasa *System.Object* jest wyjątkowa, ponieważ dziedziczy się po niej niejawnie.

Pierwsza z tych metod – *ToString()* zwróci jedynie nazwę klasy wraz z przestrzenią nazw. Możemy jednak napisać w klasie *Pracownik* swoją wersję tej metody i „przykryć” tę z klasy *System.Object*. Nasza wersja mogłaby np. zwrócić łańcuch znaków zawierający wartości pól obiektu. Metoda *Equals()* porównuje referencje, w przykładach 6.8 i 6.9 da różny wynik. W naszej klasie byłaby bardziej przydatna metoda porównująca wartości obiektów a nie referencje. Także możemy wykonać swoją wersję tej metody przykrywając (a ściślej – ukonkretniając) jej zawartość.

Twórcy klasy *System.Object* nie mogli znać konkretnych potrzeb w zakresie działania tych metod w klasach pochodnych. Dlatego zdefiniowano je jako „wirtualne” (są to metody oznaczone modyfikatorem *virtual*). To co one wykonują może się przydać programiście, ale z góry przewidziano, że autor danej klasy może potrzebować nieco innego działania tych metod i wówczas zdefiniuje te metody „po swojemu” w ciele tworzonej klasy, dając im tę samą nazwę i oznaczając modyfikatorem *override*. Te zagadnienia wychodzą poza ramy tego podręcznika i jedynie je tu sygnalizujemy.

I na zakończenie krótkie wyjaśnienie odnośnie terminu *programowanie obiektowe*. Angielski odpowiednik brzmi *object-oriented programming*, w polskich podręcznikach można napotkać dwa tłumaczenia: *programowanie obiektowe* oraz *programowanie zorientowane obiektowo*. To drugie, bardziej dosłowne tłumaczenie lepiej oddaje podejście do „obiektości” w takich językach jak C# czy C++, ale bardziej upowszechniło się to pierwsze – programowanie obiektowe. Także i w tym podręczniku zdecydowaliśmy się na „programowanie obiektowe”. Poniżej umieszczamy rysunek, który schematycznie definiuje programowanie zorientowane obiektowo (programowanie obiektowe).



W tym podręczniku zrealizowaliśmy paradygmat obiektowy tylko w zakresie konstrukcji umożliwiających łączenie danych i zachowania (czyli tworzyliśmy proste klasy). Dalszym krokiem w nauce programowania dla naszego Czytelnika jest poznanie konstrukcji językowych odpowiedzialnych za dziedziczenie i polimorfizm. Ponadto nieodzowne będzie poznanie gotowych klas bibliotecznych wspomagających takie operacje jak – obsługa wyjątków, obsługa strumieni plikowych, czy praca z bazą danych, które występują w typowych programach.

6.7 Zadania do samodzielnego rozwiązania

Zadanie 6.1.

Napisz program, który tworzy klasę *Prostokat*, zawierającą dwie prywatne dane składowe: *dlugosc*, *szerokosc*, dwie prywatne metody: *powierzchnia()*, *obwod()* oraz metodę publiczną – *Prezentuj()* (która wyświetla powierzchnię i obwód prostokąta) i konstruktor inicjalizujący. W metodzie *Main()* zdefiniuj obiekt i uruchom dla niego metodę *Prezentuj()*.

Zadanie 6.2.

Uzupełnij program z poprzedniego zadania o definicję tablicy obiektów dla prostokątów. W metodzie *Main()* wyświetl powierzchnie oraz obwód wszystkich prostokątów w tablicy używając (wewnątrz pętli *foreach*) metody publicznej *Prezentuj()*.

Zadanie 6.3.

Uzupełnij program z poprzedniego zadania o definicję metody statycznej, która podaje powierzchnię największego prostokąta.

Zadanie 6.4.

Zdefiniuj klasę, która pozwoli na rejestrację zużycia energii elektrycznej. Klasa powinna pozwalać na:

- rejestrację początkowego i bieżącego stanu licznika energii,
- uzyskanie danych o początkowym oraz bieżącym stanie licznika,
- obliczanie zużytej energii.

Zadanie 6.5.

Napisz program tworzący klasę *Punkt* do obsługi punktów na płaszczyźnie. Klasa ta ma zawierać: konstruktor, którego argumentami będą współrzędne punktu, metodę składową *Przesun()*, realizującą przesunięcie o zadane wielkości oraz metodę składową *Wyswietl()* wypisującą aktualne współrzędne punktu. Współrzędne punktu mają być zdefiniowane poprzez właściwości.

Zadanie 6.6.

Napisz program (używając klasy *Punkt* zdefiniowanej w poprzednim zadaniu), który przechowuje dane o trzech punktach w tablicy obiektów i sprawdza przy pomocy statycznej metody, czy leżą one na jednej prostej.

Zadanie 6.7.

Zdefiniuj klasę *Odcinek* składającą się z dwóch punktów klasy *Punkt*. W klasie *Odcinek* zdefiniuj metodę, która obliczy długość odcinka.

Zadanie 6.8.

Zdefiniuj klasę *Prostopadloscian*, która pozwoli na reprezentację danych opisujących długość, szerokość i wysokość prostopadłościanu. W klasie zaimplementuj metody pozwalające na obliczenie objętości prostopadłościanu, oraz porównanie objętości dwóch prostopadłościanów.

Zadanie 6.9.

Wykonaj zadania 6.1 oraz 6.2 z użyciem struktury (zamiast klasy).

Zadanie 6.10.

Napisz program z użyciem struktury *KandydatNaStudia*, która ma posiadać następujące pola: *nazwisko*, *punktyMatematyka*, *punktyInformatyka*, *punktyJęzykObcy*. W trzech ostatnich polach mają być zapisane punkty za przedmioty zdawane na maturze (dla uproszczenia uwzględniamy tylko jeden poziom zdawanej matury, np. podstawowy). Jeden punkt to jeden procent (tj. student, który ma 55% z matematyki ma mieć 55 punktów z tego przedmiotu). Struktura ma posiadać metodę obliczającą łączną liczbę punktów kandydata wg przelicznika: 0,6 punktów z matematyki + 0,5 punktów z informatyki + 0,2 punktów z języka obcego. W metodzie *Main()* utwórz obiekty dla struktury (jako elementy tablicy) dla kilku kandydatów i pokaż listę kandydatów, zawierającą nazwisko i obok, w tej samej linii, obliczoną łączną liczbę punktów.

7 Poprawianie błędów w programie

Pojawianie się błędów w pisanych programach jest czymś naturalnym i najlepiej jest to od razu zaakceptować i polubić. Nie tyle same błędy, co proces ich śledzenia i usuwania – bo to jest obowiązkowy i zajmujący немало czasu element pracy programisty. Bywa, że napisanie krótkiego i prostego programu zajmuje początkującemu programiście kilka godzin, z czego zdecydowana większość to czas przeznaczony na szukanie i poprawę błędów. Rodzi to u niektórych pewną frustrację, ponieważ w chwili, gdy już program działa dana osoba wie jak blisko była rozwiązania i daremny wydaje się jej czas błędzenia poprzedzający poprawę programu. Proszę nam wierzyć, taki wysiłek nigdy nie jest daremny. Etap „błędzenia” – to bardzo ważny etap nauki, angażujący emocjonalnie, budzący aktywność umysłu do wielokrotnego analizowania danego fragmentu programu i mobilizujący do osiągnięcia celu. Wszystko to potrzebne jest podczas pracy w tym zawodzie.

Postaramy się w tym rozdziale pomóc Czytelnikowi nabyć sprawności w wyszukiwaniu i poprawianiu błędów. W pierwszej kolejności chcielibyśmy pomóc wypracować odpowiednie podejście do tego zagadnienia. Początkowo, podczas nauki programowania, można obserwować u niektórych osób dwa niedojrzałe podejścia do poprawiania błędów: „przypadkowe” oraz „magiczne”. Podejście „przypadkowe” polega na tym, że dana osoba nie próbuje analizować kodu np. szukać linii, gdzie ma wstawić brakujący nawias, ale wstawia gdziekolwiek i sprawdza, czy już można poprawnie skompilować program. Podejście „magiczne” polega na wykonywaniu w danym miejscu poprawek, ponieważ w innym podobnym miejscu programu (albo w innym programie) pomogły. Odbyna się to niestety bez krytycznego wglądu w charakterystykę bieżącego problemu, a powielane „poprawki” traktowane są jak lekarstwo, którego działania nie rozumiemy, ale wierzymy w jego „moce uzdrawiające”. W prawidłowym podejściu element empiryczny (prób i błędów) także pełni ważną rolę, ale w przeciwieństwie do omówionych podejść jest poprzedzony analizą kodu i przemyśleniem scenariusza eksperymentów.

Popelniane błędy podzielić można na dwie główne kategorie: błędy zgłaszane na etapie kompilacji programu oraz błędy ujawniające się podczas działania programu. Środowisko programistyczne udostępnia przydatne narzędzia do analizy błędów. W dwóch kolejnych podrozdziałach omówimy narzędzia pomocne do poprawiania błędów w ramach wymienionych kategorii.

7.1 Poprawianie błędów zgłaszanych na etapie kompilacji

Środowisko programistyczne sygnalizuje błędy już na etapie pisania programu (w edytorze), ale funkcjonalność tę można wyłączyć. Nie można jednak zrezygnować z etapu kompilacji programu, czyli tłumaczenia kodu na taki, który mógłby być wykonywany na

maszynie⁶⁸. Dlatego w nazwie tej grupy błędów akcentujemy „etap kompilacji”, a nie „etap edycji”.

Wykonamy kilka typowych błędów. Najpierw jednak napiszemy poprawny program, który w dalszej części będziemy „psuć” i opisywać proces szukania i poprawiania błędu.

Przykład 7.1.

```
static void Main(string[] args)
{
    Console.WriteLine("Wprowadź liczbę");
    int liczba = int.Parse(Console.ReadLine());
    if (liczba >= 0)
    {
        Console.WriteLine("Pierwiastek {0}", Math.Sqrt(liczba));
    } // Klamra do usunięcia
    Console.WriteLine("Potęga {0}", Math.Pow(liczba, 2));
    Console.ReadKey();
}
```

Program ten prosi użytkownika o wprowadzenie liczby. Dla liczby nieujemnej wypisuje na ekranie pierwiastek, a następnie bez względu na znak liczby (ujemna czy nie) wypisywana jest potęga (kwadrat liczby).

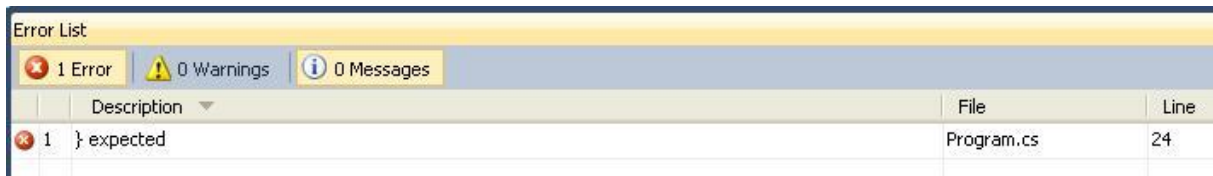
Scenariusz 7.1. a

Zobaczmy co się stanie, gdy usuniemy klamrę, która zamyka instrukcję *if* (w linii oznaczonej komentarzem „Klamra do usunięcia”). Po kliknięciu ikonki *Start Debugging* (lub klawisza *F5*) pojawi się okno z komunikatem o błędzie:



W treści komunikatu jest także pytanie, czy chcemy, aby została uruchomiona poprzednia poprawna wersja programu (jeśli taka była). Jeśli zależy nam na poprawieniu programu należy wybrać odpowiedź „No” (nie). Wówczas w oknie błędów („Error List”) pokaże się lista błędów. Dla powyższego przykładu będzie tylko jeden błąd:

⁶⁸ W przypadku platformy .NET w wyniku kompilacji programu tworzony jest tzw. kod pośredni, który później tłumaczony jest na kod maszynowy.



Treść opisu błędu brzmi: „} expected”, czyli oczekiwany jest nawias klamrowy zamykający. W kolumnie „Line” jest numer linii, można kliknąć w opis błędu i kursor sam ustawi się w linii, w której zgłaszany jest błąd. W tym przypadku kursor ustawi się w ostatniej linii programu, zaraz za ostatnią klamrą zamykającą sekcję *namespace*:



Z układu wcięć w kodzie programu wynika, że brakuje klamry zamykającej instrukcję *if*. Wcięcia jednak są dla programisty (czytelności kodu programu), kompilator je ignoruje. Dla kompilatora zamknięcie instrukcji *if* nastąpiło po napotkaniu pierwszej zamykającej klamry (tej, która jest po instrukcji *Console.ReadKey()*). Kolejna klamra zamknęła ciało metody *Main()*. Ostatnia klamra zamknęła ciało klasy *Program*. Brakło klamry zamykającej sekcję *namespace* – i to dlatego w ostatniej linii zgłaszany jest błąd.

Poprawa błędu składniowego może, ale nie musi być w tej linii, w której zgłaszany jest błąd. W analizowanym programie mamy sytuację, która wymaga zmiany w innym miejscu. Gdybyśmy dopisali klamrę zamykającą na końcu (czyli tam gdzie zgłaszany jest błąd), to wprawdzie mielibyśmy poprawny składniowo program, ale działający inaczej niż oczekiwaliśmy. Po takiej poprawie instrukcja *if* obejmowałaby trzy instrukcje – wypisanie pierwiastka, potęgi oraz metodę *Console.ReadKey()*, czyli program wypisywałby pierwiastek oraz potęgę tylko dla liczb nieujemnych, a dla ujemnych nic by nie robił. Tymczasem potęgą liczby miała być wypisana bez względu na jej znak.

Nie dopisujemy tu zatem klamry na końcu, ale analizujemy program i szukamy właściwego miejsca dla brakującej klamry. Edytor może nam w tym pomóc poprzez

podświetlanie pary nawiasów klamrowych – otwierającego i zamykającego daną sekcję, gdy klikniemy w jeden z nich. Przejrzenie par nawiasów klamrowych pozwoli nam zlokalizować miejsce, gdzie należy wstawić brakujący nawias klamrowy (w tym przypadku zamknięcie instrukcji *if* po wypisaniu wartości pierwiastka dla liczby nieujemnej).

Scenariusz 7.1. b

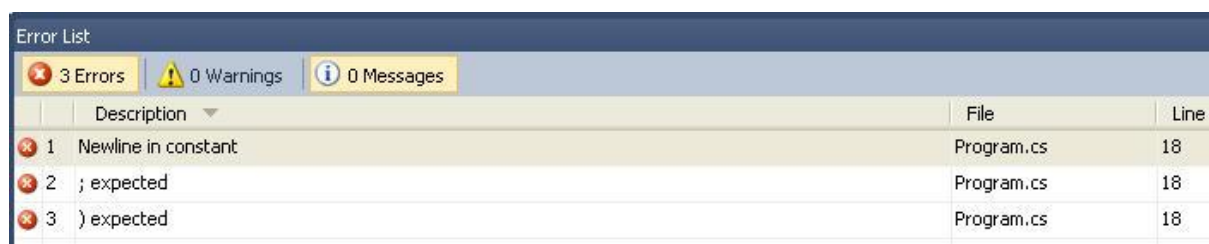
Wróćmy do poprawnej postaci programu z przykładu 7.1, a następnie w linii:

```
Console.WriteLine("Pierwiastek {0}", Math.Sqrt(liczba));
```

usuńmy drugi cudzysłów, czyli po zmianie linia ta powinna wyglądać:

```
Console.WriteLine("Pierwiastek {0}, Math.Sqrt(liczba));
```

Tym razem na liście błędów będą trzy pozycje:



Error List			
3 Errors 0 Warnings 0 Messages			
	Description	File	Line
1	Newline in constant	Program.cs	18
2	; expected	Program.cs	18
3) expected	Program.cs	18

Wszystkie trzy zgłoszone błędy są tak naprawdę skutkiem jednego błędu w programie. Pierwszy zgłoszony błąd dotyczy znaku nowej linii umieszczonego w stałej łańcuchowej, a taka jest konsekwencja nie zamknięcia cudzysłowu, że wszystkie znaki do końca tej linii traktowane są jako zawartość stałej łańcuchowej (literału typu *string*). Kolejne dwa błędy dotyczą braku znaku średnika oraz nawiasu zamykającego. Oba te błędy także wynikają z braku zamknięcia cudzysłowu. Proszę zwrócić uwagę, że żaden z powyższych komunikatów nie mówi wprost „brakuje cudzysłowu” – to musimy sami wywnioskować na podstawie komunikatów i zawartości wskazanej linii programu. Widzimy, że średnik jest, nawias zamykający do metody *Console.WriteLine()* także – to czemu kompilator zgłasza brak? Gdy przesuniemy wzrok w lewo widzimy, że brakuje cudzysłowu, co oznacza, że wszystko w tej linii traktowane jest jako zawartość stałej łańcuchowej, średnik i nawias także.

Scenariusz 7.1. c

Na koniec prześledzimy jeszcze jeden przypadek, tym razem w programie z przykładu 7.1 usuniemy klamrę otwierającą instrukcję *if*. W tej części program będzie wyglądał następująco:

```
if (liczba >= 0)
// Tu brakuje klamry otwierającej
    Console.WriteLine("Pierwiastek {0}", Math.Sqrt(liczba));
}
```

Taki błąd będzie skutkował zgłoszeniem aż pięciu komunikatów o błędach:

Error List				
5 Errors 0 Warnings 0 Messages				
	Description	File	Line	Column
5	Type or namespace definition, or end-of-file expected	Program.cs	24	1
3	Invalid token ',' in class, struct, or interface member declaration	Program.cs	20	60
1	Invalid token '(' in class, struct, or interface member declaration	Program.cs	20	30
2	Invalid token '(' in class, struct, or interface member declaration	Program.cs	20	53
4	Invalid token '(' in class, struct, or interface member declaration	Program.cs	21	28

Klamra zamykająca (po instrukcji *if*) została uznana jako klamra zamykająca ciało metody *Main()*, przez co poniższe linie zawierające wywołanie metod *Console.WriteLine()* oraz *Console.ReadKey()* weszły w obszar definicji klasy *Program*. Dlatego komunikaty o błędach mówią o nieprawidłowych znakach (ang. invalid token) w miejscach, które zdradzają, że analizowane linie programu nie stanowią dozwolonych instrukcji w definicji klasy. Żaden z komunikatów nie mówi, że brakuje klamry otwierającej. Z komunikatu „Nieprawidłowy znak w deklaracji klasy” musimy wywnioskować, że coś jest nie tak z klamrami, skoro program usiłuje traktować zawartość metody *Main()* jako część definicji klasy. Ponadto proszę zwrócić uwagę na mnogość komunikatów spowodowanych jednym błędem. Nie należy się przerażać widząc długą listę błędów, ale po kolei czytać komunikaty i analizować kod.

Nie umieszczamy tu przypadków prostych, których poprawa jest oczywista, tzn. komunikat o błędzie mówi wprost jaka jest przyczyna i wskazuje właściwą linię, gdzie trzeba poprawić program. Taki przykład umieściliśmy w [punkcie 1.2.2](#). Jeśli mamy dłuższą listę komunikatów lepiej jest zacząć poprawę od pozycji nie budzących wątpliwości (np. treść komunikatu mówi o braku średnika w danej linii i widzimy, że faktycznie tak jest). Po wprowadzeniu poprawek klikamy ikonkę *Start Debugging* ponownie, nowa lista błędów będzie krótsza i łatwiej będzie nam analizować jej zawartość. Pamiętać należy o tym, że niektóre komunikaty mogą być powiązane (tzn. dotyczyć jednej przyczyny).

Z naszego doświadczenia dydaktycznego wynika, że najbardziej uciążliwą przyczyną błędów składniowych w początkowym etapie nauki jest brak prawidłowego umiejscowienia klamry otwierającej lub zamykającej. Uciążliwość polega na tym, że komunikaty o tego typu błędach nie wskazują na ogół numeru linii, gdzie faktycznie jest przyczyna błędu (jak to pokazaliśmy na przykładach), co wymaga analizy większego fragmentu kodu (lub całości).

Do analizy kodu pod kątem takich błędów wręcz nieocenione są wcięcia. Proszę spojrzeć na poniższy program, zapisany niedbale, bez odpowiednich wcięć:


```

namespace Podrecznik
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Wprowadź liczbę");
            int liczba = int.Parse(Console.ReadLine());
            if (liczba >= 0)
                Console.WriteLine("Pierw. {0}", Math.Sqrt(liczba));
            Console.WriteLine("Potęga {0}", Math.Pow(liczba, 2));
            Console.ReadKey();
        }
    }
}

```

Jest to kod ze scenariusza 7.1.c (z brakującą klamrą otwierającą dla instrukcji *if*). W tak niedbale zapisanym kodzie programu nie od razu znajdziemy przyczynę błędu.

Dla porównania spójrzmy na wersję tego samego programu z właściwymi wcięciami:

```

namespace Podrecznik
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Wprowadź liczbę");
            int liczba = int.Parse(Console.ReadLine());
            if (liczba >= 0)
                Console.WriteLine("Pierw. {0}", Math.Sqrt(liczba));
            Console.WriteLine("Potęga {0}", Math.Pow(liczba, 2));
            Console.ReadKey();
        }
    }
}

```

W takim układzie kodu znacznie szybciej znajdziemy brakujący nawias klamrowy.

Na etapie kompilacji zgłaszane są nie tylko błędy składniowe (takie jak brakujące nawiasy, znaki cudzysłowu czy średniki oraz niepoprawne elementy instrukcji). Zgłaszane są wszystkie nieprawidłowe operacje, które mogą być rozpoznane na etapie kompilacji programu, m.in. niezgodność typu danych. Przykładowo w programie z przykładu 7.1 dopisanie przed poleceniem *Console.ReadKey()* instrukcji:

```

liczba = "tekst";

```


spowoduje błąd o treści: “Cannot implicitly convert type 'string' to 'int'” – co oznacza, że nie można dokonać niejawnej konwersji z typu *string* do *int*. Zmienna *liczba* w tym programie ma zadeklarowany typ *int*, a powyższa instrukcja zawiera przypisanie tekstu do tej zmiennej. Podobne błędy są łatwe do poprawy, ponieważ wskazywane są dokładnie linie programu, gdzie użyto zmiennych niezgodnie z ich typem.

7.2 Poprawianie błędów występujących po uruchomieniu programu

Poprawna kompilacja programu jeszcze nie oznacza, że program dobrze działa. Po uruchomieniu programu jego błędne działanie może się objawić na kilka sposobów, m.in.:

- program do pewnego momentu działa, a później się zawiesza (lub zapętla),
- program kończy się prawidłowo (nie zawiesza się), ale wyniki są inne od oczekiwanych (błędne),
- program dobrze działa w określonych warunkach (np. dla niektórych danych, gdy dostępny jest jakiś plik lub inny zasób, itp.), a w innych nie.

W środowisku programistycznym dostępny jest tzw. *debugger* – narzędzie do śledzenia programów i wyszukiwania błędów. Proces śledzenia i poprawiania programu określono debugowaniem (ang. debugging). Większość programów służących do debugowania zawiera podobne elementy:

- Wstrzymanie programu w określonym miejscu (wstawianie tzw. punktów wstrzymania),
- Wykonywanie kodu krok po kroku (do kolejnej instrukcji),
- Możliwość podglądania aktualnych wartości zmiennych (obiektów).

Prześledzimy krótki program zawierający błąd, który po uruchomieniu spowoduje zapętlenie programu.

Przykład 7.2

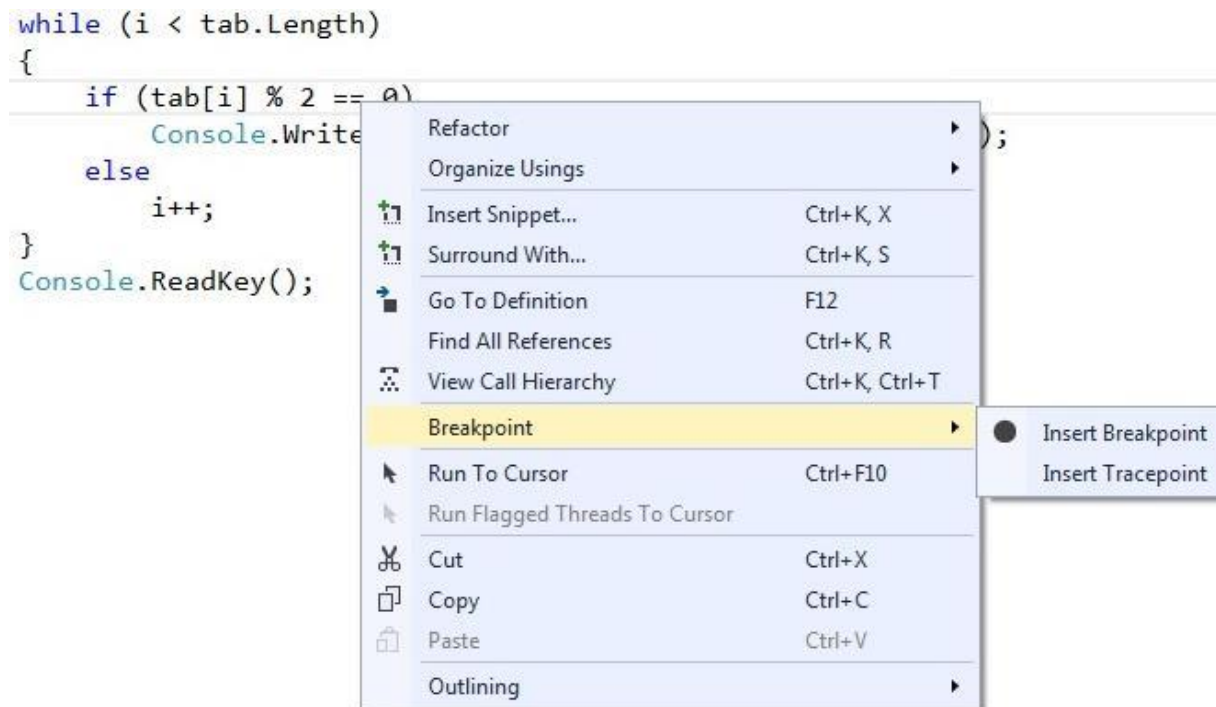
```
static void Main(string[] args)
{
    int[] tab = { 1, 4, 7, 9, 3, 12 };
    int i = 0;
    while (i < tab.Length)
    {
        if (tab[i] % 2 == 0)
            Console.WriteLine("{0} to liczba parzysta", tab[i]);
        else
            i++;
    }
    Console.ReadKey();
}
```

Program powinien wypisać liczby parzyste spośród tych, jakie są w tablicy. Działa jednak inaczej. Program wyświetla komunikat „4 to liczba parzysta” i komunikat ten

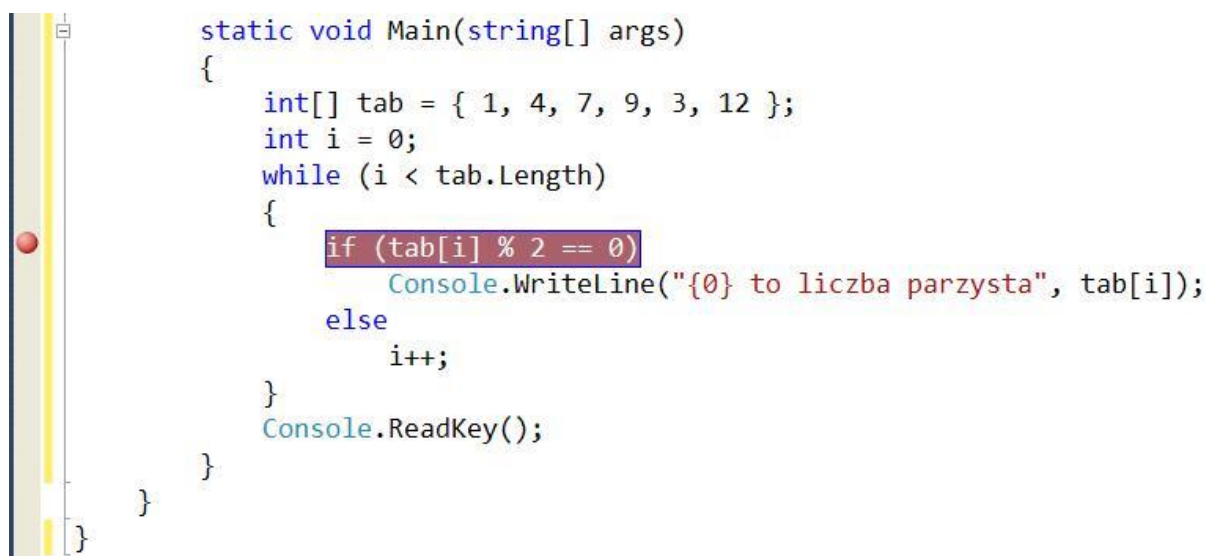
wypełnia cały ekran (jest powielony w kolejnych liniach). Dopóki nie zatrzymamy wykonywania programu wciąż będzie się wyświetlał ten sam komunikat. Proponowany przez nas program może być zbyt prosty, aby wymagał debugowania. Niemniej prześledzimy go, poznając przy okazji możliwości programu do debugowania.

Wstawianie punktu wstrzymania

Program wykonuje się na tyle szybko, że nie jesteśmy w stanie przeanalizować jego działania bez uprzedniego zatrzymania. Podobnie detektyw oglądający film (jako materiał w śledztwie), aby przyjrzeć się szczegółom zatrzymuje film w określonym miejscu, a dalej ogląda w zwolnionym tempie. Punkt wstrzymania (ang. breakpoint) wstawić można na dwa sposoby. Pierwszy sposób polega na kliknięciu myszą w lewy margines w danej linii, a drugi na wybraniu z menu kontekstowego (po kliknięciu prawym klawiszem myszy) pozycji *Breakpoint*. Drugi sposób przedstawia rysunek:



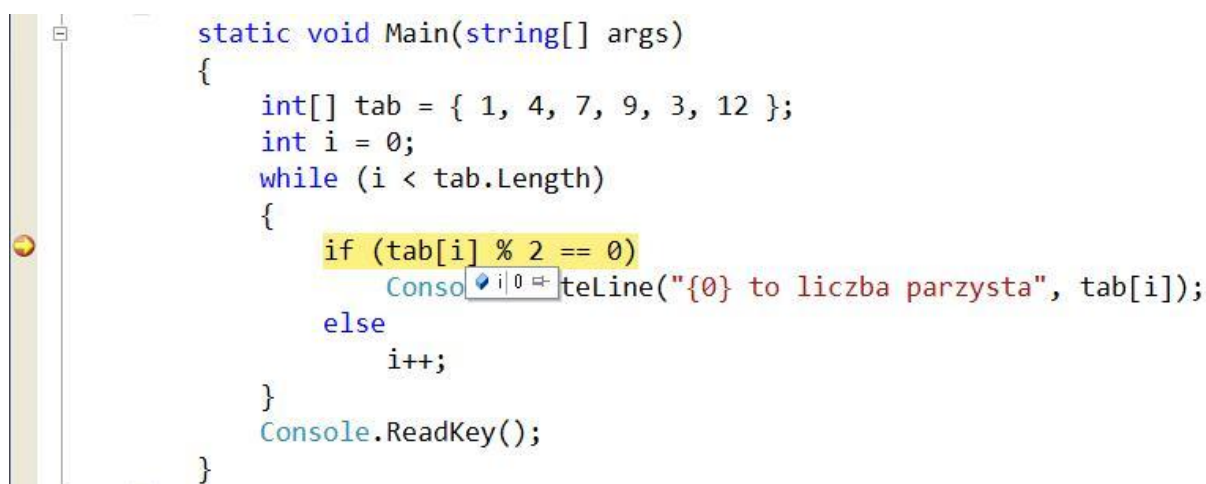
Ponieważ działanie programu wskazuje na problemy z pętlą – punkt wstrzymania najlepiej będzie wstawić we wnętrzu pętli (ewentualnie tuż przed). Po wstawieniu punktu wstrzymania na lewym marginesie okna pojawi się przy wskazanej linii czerwone kółko, a kod w tym miejscu będzie na czerwonym (bordowym) tle:



Po ustawieniu punktu wstrzymania można uruchomić program używając ikonki z zieloną strzałką *Start* (pełna nazwa *Start Debugging*) lub klawisza *F5*. Po uruchomieniu program wykona się do instrukcji, która jest przed punktem wstrzymania. Linia z punktem wstrzymania wyświetla się wówczas na żółtym tle. Zanim przeniesiemy sterowanie do kolejnej instrukcji przypatrzmy się zmiennym – jakie mają wartości przed wykonaniem instrukcji *if*.

Podgląd wartości zmiennych

Aktualną wartość zmiennej możemy podejrzeć po najejaniu myszą na jej nazwę. Przykładowo po najejaniu na zmienną *i* w linii z instrukcją *if* poniżej pokaże się małe okienko z wartością tej zmiennej. Ponieważ jeszcze nie zastosowaliśmy pracy krokowej (co zrobimy wkrótce) widzimy wartość tej zmiennej w pierwszym przebiegu pętli – czyli zero.



Kolejnym sposobem podglądania wartości zmiennych jest dodanie ich do okna *Watch*. W tym celu najlepiej jest zaznaczyć myszą nazwę zmiennej, a następnie prawym klawiszem myszy wybrać opcję *Add Watch*. W oknie tym możemy nie tylko sprawdzić bieżącą wartość

zmiennej, ale ją zmienić (np. chcąc sprawdzić zachowanie programu od konkretnego stanu). Jeżeli zaznaczymy zmienną *i* oraz wybierzemy opcję *Add Watch* pojawi się okno:

Watch 1		
Name	Value	Type
i	0	int

Zmienne możemy dodawać do tego okna także bezpośrednio, wpisując w kolejnych wierszach. Można wpisać także wyrażenia, np. wpisanie w kolumnie *Name* wyrażenia *i + 2* pokaże wartość 2 ($0+2=2$). Natomiast wpisanie *tab[i]* pokaże wartość 1 ($tab[0] = 1$).

Wartość zmiennych możemy sprawdzić także w oknach *Locals* (opcja *Debug / Windows/ Locals*) oraz *Autos* (opcja *Debug / Windows/ Autos*). Okno *Locals* pokazuje wszystkie zmienne dostępne w bieżącym zakresie. Dla naszego przykładu są to:

Locals		
Name	Value	Type
args	{string[0]}	string[]
tab	{int[6]}	int[]
i	0	int

Przy pozycji dla tablicy *tab* można kliknąć w ikonkę z plusem i rozwinąć szczegóły (zobaczyć zawartość tablicy).

Natomiast okno *Autos* zawiera tylko te zmienne, które występują w bieżącej instrukcji (gdzie jest wstrzymany program) oraz w instrukcji poprzedniej. Można powiedzieć, że okno *Autos* jest podzbiorem okna *Locals*, ale nie do końca tak jest. Ponieważ w przypadku okna *Autos* pokazane mogą być dodatkowe szczegóły. I tak dla analizowanego przykładu i wstrzymania w instrukcji *if* zawartość okna *Autos* jest następująca:

Autos	
Name	Value
i	0
tab	{int[6]}
tab.Length	6
tab[i]	1

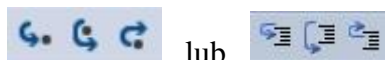
Okno *Autos* zawiera nie tylko zmienne (*i* oraz zmienną tablicową *tab*), ale także dodatkowo wartość elementu tablicy *tab[i]* oraz właściwość *tab.Length*. Wszystkie te elementy występują w obu instrukcjach – w instrukcji *if* (gdzie jest punkt wstrzymania) oraz w poprzedniej (z pętlą *while*).




W ramach testowanego przykładu możemy wykorzystać okno *Watch* do wyświetlenia dwóch pozycji: wartości zmiennej *i* oraz elementu tablicy *tab[i]*.

Praca krokowa

Program zatrzymujemy w określonym miejscu, po to, aby od tego miejsca działał „w zwolnionym tempie”, czyli krok po kroku. Nad oknem z kodem programu dostępne są trzy

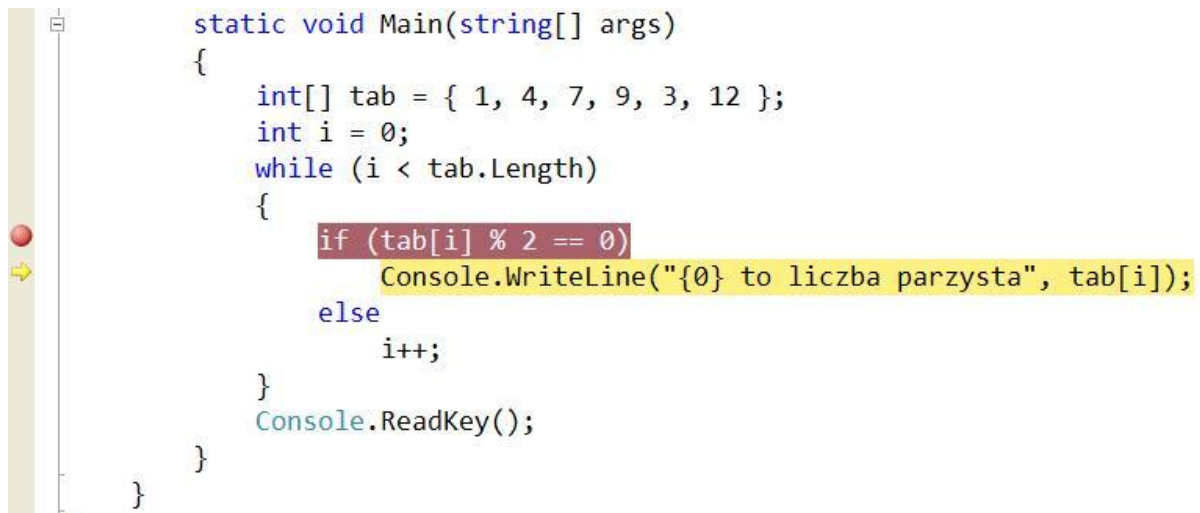
ikonki dotyczące pracy krokowej. Umieszczamy poniżej dwa warianty graficzne tych ikon, spośród których dostępny jest jeden (w zależności od zainstalowanej wersji Visual Studio):



-  *Step Into* (krok do wewnątrz) – przejście do kolejnego kroku (instrukcji), przy czym jeśli kolejną instrukcją jest wywołanie metody, to śledzenie przechodzi do wnętrza tej metody. Skrót klawiszowy – *F11*.
-  *Step Over* (krok z pominięciem) – przejście do kolejnego kroku (instrukcji), przy czym jeśli kolejną instrukcją jest wywołanie metody, to śledzenie nie przechodzi do wnętrza tej metody (tzn. wykonuje ją jako całość i przechodzi do instrukcji, która jest zaraz za jej wywołaniem). Skrót klawiszowy – *F10*.
-  *Step Out* (krok z wyjściem) – wyjście z aktualnie wykonywanej metody, sterowanie przechodzi do instrukcji, która jest zaraz za wywołaniem danej metody. Skrót klawiszowy – *Shift + F11*.

Ponieważ w analizowanym przykładzie nie ma definiowanej metody (a przejście *Step Into* dotyczy tylko metod definiowanych w programie), to oba przejścia *Step Into* oraz *Step Over* będą działać tak samo – po prostu przenosić sterowanie do kolejnej instrukcji.

Przypominamy, że program wstrzymany jest na instrukcji *if*. Zmienna *i* ma w tym miejscu wartość 0, a *tab[i]* ma wartość 1. Przechodzimy do następnego kroku (klikając ikonkę *Step Into* lub *Step Over*). Oznacza to, że wykona się zaznaczona na żółto instrukcja: *if(tab[i] % 2 == 0)*, a program przejdzie dalej. Ponieważ *tab[i]*, czyli 1 nie ma reszty z dzielenia przez 2 równej zero, to sterowanie zostanie przeniesione do instrukcji będącej po słowie kluczowym *else*, tzn. *i++*; . Wartość zmiennych w oknie *Watch* jeszcze się nie zmienia w tym momencie. Wykonanie kolejnego kroku zatrzymuje program w linii z klamrą zamykającą pętlę *while*. Wówczas widzimy już zmiany w oknie *Watch* – zmienna *i* jest równa 1, a *tab[i]* ma wartość 4. Kolejny krok przenosi nas do linii z warunkiem dla pętli *while*, ponieważ warunek ten jest spełniony dla *i* równego 1 przewidujemy, że kolejny krok przeniesie nas do wnętrza i tak się dzieje. Przechodząc kolejne kroki widzimy, że element *tab[i]* (czyli liczba 4) jest podzielny przez 2, więc tym razem program wchodzi do instrukcji wyświetlającej komunikat o tym, że 4 to liczba parzysta. Program mamy jak na razie w tym miejscu (na żółtym tle wyświetla się bieżąca instrukcja):



```
static void Main(string[] args)
{
    int[] tab = { 1, 4, 7, 9, 3, 12 };
    int i = 0;
    while (i < tab.Length)
    {
        if (tab[i] % 2 == 0)
            Console.WriteLine("{0} to liczba parzysta", tab[i]);
        else
            i++;
    }
    Console.ReadKey();
}
```

Dalsze śledzenie (klikanie ikonki *Step Into*) prowadzi nas do linii z klamrą zamykającą, później do warunku pętli *while*, dalej do instrukcji *if*, w końcu ponownie do instrukcji zawierającej komunikat, że liczba 4 jest parzysta. Kolejne kroki będą już tylko „w kółko” – wciąż wyświetla się ten sam komunikat. Widzimy, że dla liczb parzystych ten program nie przewiduje zwiększania zmiennej *i*, przez co od chwili napotkania pierwszej liczby parzystej nie potrafi wyjść z pętli. W celu poprawienia tego programu najlepiej jest usunąć polecenie „else”, wówczas zmienna *i* będzie zwiększana bez względu na to, czy w tablicy o danym indeksie była liczba parzysta czy nie. Poprawiony kod tego programu ma postać:

```
static void Main(string[] args)
{
    int[] tab = { 1, 4, 7, 9, 3, 12 };
    int i = 0;
    while (i < tab.Length)
    {
        if (tab[i] % 2 == 0)
            Console.WriteLine("{0} to liczba parzysta", tab[i]);
        i++;
    }
    Console.ReadKey();
}
```

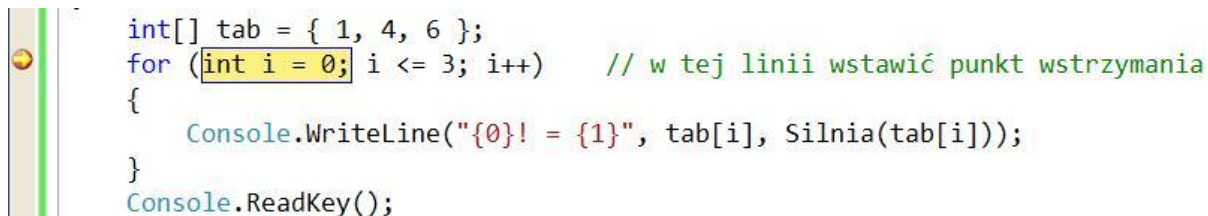
Prześledzimy jeszcze jeden program.

Przykład 7.3

```
static int Silnia(int n)
{
    int s = 1;
    for (int i = 1; i <= n; i++)
        s = s * i;
    return (s);
}

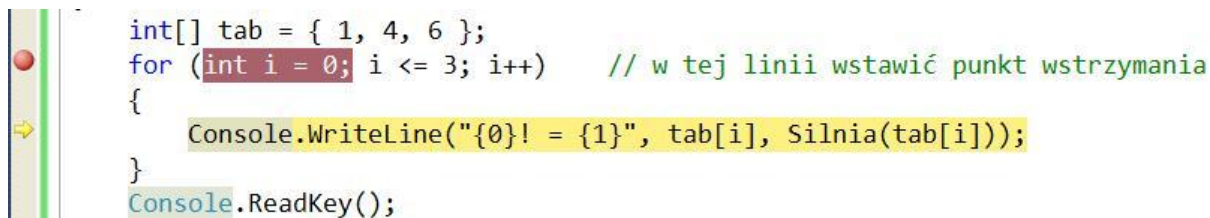
static void Main(string[] args)
{
    int[] tab = { 1, 4, 6 };
    for (int i = 0; i <= 3; i++)    // w tej linii wstawić punkt wstrzymania
    {
        Console.WriteLine("{0}! = {1}", tab[i], Silnia(tab[i]));
    }
    Console.ReadKey();
}
```

Program powinien dla każdej liczby w tablicy pokazać jej wartość oraz wartość obliczonej silni. Po uruchomieniu widzimy jednak, że program podaje komunikat o przekroczonym zakresie dla indeksu tablicy. Ustawimy punkt wstrzymania w linii z deklaracją pętli *for*. Po uruchomieniu programu (ikonka *Start Debugging* lub klawisz *F5*) program zatrzyma się w zadanej linii:



```
int[] tab = { 1, 4, 6 };
for (int i = 0; i <= 3; i++)    // w tej linii wstawić punkt wstrzymania
{
    Console.WriteLine("{0}! = {1}", tab[i], Silnia(tab[i]));
}
Console.ReadKey();
```

Po przejściu kolejnego kroku (*Step Into* lub *Step Over*) podświetlona zostanie część z warunkiem, a w kolejnym kroku klamra otwierająca ciało pętli *for*. Następnie sterowanie będzie umieszczone w linii, w której wypisywana jest liczba i jej silnia:



```
int[] tab = { 1, 4, 6 };
for (int i = 0; i <= 3; i++)    // w tej linii wstawić punkt wstrzymania
{
    Console.WriteLine("{0}! = {1}", tab[i], Silnia(tab[i]));
}
Console.ReadKey();
```

W tej linii jest wywołanie metody *Silnia()*, jeśli klikniemy ikonkę *Step Into* (lub klawisz *F11*) to program przejdzie do pierwszej linii wewnątrz metody *Silnia()*. Jeśli jednak klikniemy ikonkę *Step Over* (lub klawisz *F10*) to program przejdzie do linii z klamrą zamykającą pętlę *for* (w następnych krokach wykona się nowy przebieg pętli). Proponujemy dla pierwszego z wywołań kliknąć *F10*, a dla pozostałych *F11* (aby sprawdzić zachowanie

debuggera w obu przypadkach). W oknie *Locals* lub *Watch* podpatrujemy aktualną wartość zmiennej *i*. Program wypisze wszystkie trzy liczby wraz z policzonymi dla nich silniami. Po wypisaniu trzeciej liczby zmienna *i* przyjmuje wartość 3, program sprawdza wyrażenie logiczne zawarte w pętli *for* i ponieważ jest ono spełnione ($i \leq 3$) program, wchodzi do wnętrza pętli po raz kolejny. Spójrzmy na wartości zmiennych w chwili, gdy program zatrzymany jest na linii z instrukcją *Console.WriteLine()*:

Locals	
Name	Value
args	{string[0]}
i	3
tab	{int[3]}
[0]	1
[1]	4
[2]	6

Zmienna *i* równa się 3, a ponieważ tablica o rozmiarze 3 ma indeksy 0, 1 i 2 to program się zakończy w tym miejscu komunikatem o przekroczeniu zakresu – nie ma w tej tablicy elementu *tab[3]*.

Poprawa błędu w tym przypadku polegać będzie na zmianie wyrażenia logicznego w pętli *for* (powinien być użyty operator relacji „<” zamiast „<=”), czyli instrukcja ta powinna wyglądać:

```
for (int i = 0; i < 3; i++)
```

lub jeszcze lepiej tak:

```
for (int i = 0; i < tab.Length; i++)
```

Omówione przykłady wykorzystano do przedstawienia podstawowych możliwości debuggera, które są wystarczające w początkowym etapie nauki. Zachęcamy do korzystania z tego narzędzia, nie tylko podczas problemów z danym programem. Debugger i jego możliwość pracy krokowej można wykorzystać także do lepszego rozumienia programów, które działają poprawnie, a które zawarte są w podręcznikach, na stronach internetowych lub zostały napisane przez inną osobę w zespole programistycznym. Debugowanie wykorzystuje się także do optymalizacji wydajności programu.

Zakończenie, czyli początek

Mamy nadzieję, że udało się nam osiągnąć założony cel i pomóc Czytelnikowi postawić w programowaniu pierwsze kroki. Gdyby porównać naukę programowania do nauki pływania, to po przebrnięciu przez ten podręcznik, Czytelnik – jako początkujący „pływak” – powinien z zadowoleniem zauważyć, że nie tylko unosi się na wodzie, ale i potrafi także przepłynąć pewną, niedużą odległość. Jego technika wymaga jednak długiego jeszcze doskonalenia, zanim będzie mógł wypłynąć na otwarte i szerokie wody.

Miejsce, w którym kończy się ten podręcznik jest dopiero początkiem drogi do profesjonalnego programowania. Przeczytanie tej książki, analiza zawartych w niej przykładów, napisanie pierwszych własnych programów – to wszystko powinno Czytelnikowi pomóc usamodzielnąć się w dalszym kształceniu. Ponieważ najtrudniejsze bywają pierwsze kroki, może się okazać, że mimo tego, iż obszar, jaki pozostał Czytelnikowi do zgłębienia jest znacznie większy niż ten dotąd poznany, to jednak „najgorsze ma już za sobą”.

Pod koniec rozdziału szóstego, przy omawianiu cech programowania obiektowego, wskazaliśmy te elementy języka C#, które zostały do poznania. Dokonaliśmy tam krótkiego wstępu do tematu dziedziczenia, od którego począwszy, Czytelnik powinien naszym zdaniem, kontynuować dalszą naukę.

Z ważnych zagadnień nieomówionych w tym podręczniku można by także wymienić: obsługę wyjątków, obsługę strumieni wejścia-wyjścia, kolekcje, metody wirtualne, przeciążanie operatorów, interfejsy, klasy abstrakcyjne, delegaty i zdarzenia oraz programowanie wizualne (Windows Forms i WPF).

Dodatki – podręczny bryk

Tabela 1. Typy wbudowane w C#

Typ	Zakres wartości	Rozmiar / Precyzja	Rodzaj typu
bool	Dwie wartości: <i>true</i> lub <i>false</i>		Typ wartości
sbyte	od -128 do 127	8-bitowa liczba ze znakiem	Typ wartości
byte	od 0 do 255	8-bitowa liczba bez znaku	Typ wartości
char	od 0 do 65535	16-bitowy znak Unicode	Typ wartości
short	-32 768 do 32 767	16-bitowa liczba ze znakiem	Typ wartości
ushort	0 do 65 535	16-bitowa liczba bez znaku	Typ wartości
int	-2 147 483 648 do 2 147 483 647	32-bitowa liczba ze znakiem	Typ wartości
uint	0 do 4 294 967 295	32-bitowa liczba bez znaku	Typ wartości
long	-9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	64-bitowa liczba ze znakiem	Typ wartości
ulong	0 to 18 446 744 073 709 551 615	64-bitowa liczba bez znaku	Typ wartości
float	od $\pm 1,5 \cdot 10^{-45}$ do $\pm 3,4 \cdot 10^{38}$	7 miejsc po przecinku	Typ wartości
double	od $\pm 5,0 \cdot 10^{-324}$ do $\pm 1,7 \cdot 10^{308}$	15-16 cyfr	Typ wartości
decimal	od $\pm 1,0 \cdot 10^{-28}$ do $\pm 7,9 \cdot 10^{28}$	18-29 cyfr	Typ wartości
string	Łańcuch znaków ujęty w cudzysłowy, np. "Nauka programowania"		Typ referencyjny

Więcej o typach w dokumentacji MSDN:

Typy wbudowane: <http://msdn.microsoft.com/en-us/library/ya5y69ds.aspx>

Typy wartości wraz z wykazem modyfikatorów typów (sufiksów): <http://msdn.microsoft.com/en-us/library/bfft1t3c.aspx>

Tabela 2. Operatory w C#

Rodzaj	Operator	Nazwa	Opis / Przykłady użycia
Arytmetyczne	+ -	Dodawanie i odejmowanie	<code>x = a + b + c - d;</code>
	* /	Mnożenie i dzielenie	<code>double x; int a=1,b=1;</code> <code>x = a*(b/2); // wynik będzie 0 (dzielenie całkowite)</code> <code>x = a*((double)b/2); // w tej wersji wynik będzie 0,5</code>
	%	Reszta z dzielenia (dzielenie modulo)	<code>R = 10 % 3; // R będzie równe 1</code>
	++	Inkrementacja (operator może być przed lub po zmiennej)	<code>int x, y=0;</code> <code>x = 2* y++; // x będzie równe 0, y =1 (po zmiennej)</code>
	--	Dekrementacja (operator może być przed lub po zmiennej)	<code>int x, y=5;</code> <code>x = 2 *--y; // x będzie równe 8, y =4 (przed zmienną)</code>
Przypisania	=	Przypisanie	<code>x = y;</code>
	+=	Przypisanie ze zwiększeniem	<code>x= x + y;</code>
	-=	Przypisanie ze zmniejszeniem	<code>x= x - y;</code>
	*=	Przypisanie z mnożeniem	<code>x=x * y;</code>
	/=	Przypisanie z dzieleniem	<code>x=x / y;</code>
	%=	Przypisanie z dzieleniem modulo	<code>x=x % y;</code>
Relacji	==	Równe	<code>x == y;</code>
	!=	Różne (w matematyce \neq)	<code>x != y;</code>
	< <=	Mniejsze, mniejsze lub równe	<code>x < y;</code> <code>x <= y;</code>
	> >=	Większe, większe lub równe	<code>x > y;</code> <code>x >= y;</code>
Logiczne	!	Negacja	<code>bool y = false;</code> <code>bool x = !y; // x będzie równe true</code>
	&&	Koniunkcja warunkowa	<code>int x = 0, y = 1; bool z;</code> <code>z = x > 5 && y++ > 10;</code> Po wykonaniu tych instrukcji z przyjmie false, a y nie ulegnie zmianie (do sprawdzania tej części warunku nie dojdzie), czyli nadal będzie równe 1.
	&	Koniunkcja	<code>int x = 0, y = 1; bool z;</code> <code>z = x > 5 & y++ > 10;</code> Po wykonaniu tych instrukcji z przyjmie false, a y będzie równe 2.
	 	Alternatywa warunkowa	<code>int x = 10, y = 1; bool z;</code> <code>z = x > 5 y++ > 10;</code> Po wykonaniu tych instrukcji z przyjmie true, a y nie ulegnie zmianie (do sprawdzania tej części warunku nie dojdzie), czyli nadal będzie równe 1.
	 	Alternatywa	<code>int x = 10, y = 1; bool z;</code> <code>z = x > 5 y++ > 10;</code> Po wykonaniu tych instrukcji z przyjmie true, a y będzie równe 2.

Więcej o operatorach w bibliotece MSDN: [http://msdn.microsoft.com/en-us/library/6a71f45d\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/6a71f45d(v=vs.71).aspx)

Tabela 3. Priorytety operatorów w C#

Nazwa	Symbol
Nadrzędne	., (), [], new, x++, x--
Pozostałe jednoargumentowe	(typ), !, ++x, --x
Mnożenie, dzielenie, dzielenie modulo	*, /, %
Dodawanie i odejmowanie	+, -
Relacyjne	<, <=, >, >=
Porównania	==, !=
Koniunkcja (iloczyn logiczny)	&&
Alternatywa (suma logiczna)	
Przypisania	=, *=, /=, %=, +=, -=

Wykaz zawiera tylko wybrane operatory (wystarczający w początkowym okresie nauki programowania), pełny wykaz operatorów znajduje się w dokumentacji MSDN: [http://msdn.microsoft.com/en-us/library/6a71f45d\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/6a71f45d(v=vs.80).aspx).

Tabela 4. Wykaz słów kluczowych w C#

abstract	as	base	bool	break	byte	case	catch
char	checked	class	const	continue	decimal	default	delegate
do	double	else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto	if	implicit
in	in (generic modifier)	int	interface	internal	is	lock	long
namespace	new	null	object	operator	out	out (generic modifier)	override
params	private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct	switch
this	throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while	

Wykaz słów kluczowych z opisem: [http://msdn.microsoft.com/en-us/library/x53a06bb\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/x53a06bb(v=vs.71).aspx)

Tabela 5. Tabela ASCII dla znaków z zakresu 32-127

Kod dziesiętny	Znak		Kod dziesiętny	Znak
32	<i>spacja</i>		80	P
33	!		81	Q
34	"		82	R
35	#		83	S
36	\$		84	T
37	%		85	U
38	&		86	V
39	'		87	W
40	(88	X
41)		89	Y
42	*		90	Z
43	+		91	[
44	,		92	\
45	-		93]
46	.		94	^
47	/		95	_
48	0		96	`
49	1		97	a
50	2		98	b
51	3		99	c
52	4		100	d
53	5		101	e
54	6		102	f
55	7		103	g
56	8		104	h
57	9		105	i
58	:		106	j
59	;		107	k
60	<		108	l
61	=		109	m
62	>		110	n
63	?		111	o
64	@		112	p
65	A		113	q
66	B		114	r
67	C		115	s
68	D		116	t
69	E		117	u
70	F		118	v
71	G		119	w
72	H		120	x
73	I		121	y
74	J		122	z
75	K		123	{
76	L		124	
77	M		125	}
78	N		126	~
79	O		127	DEL

Wszystkie znaki na stronie [http://msdn.microsoft.com/en-us/library/4z4t9ed1\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/4z4t9ed1(v=vs.71).aspx)

Tabela 6. Formatowanie wartości liczbowych

Specyfikator formatu	Opis
C lub c	Liczba zostanie wyświetlona jako waluta (domyślnie wg ustawień regionalnych). Przykład: <pre>double y = 24.56; Console.WriteLine("{0:C}", y); // wyświetli się 24,56 zł Console.WriteLine("{0,10:C}", y); // wyświetli się 24,56 zł na 10 znakach, równane do prawej</pre>
D lub d	Dla wartości całkowitoliczbowych. Precyzja określa liczbę cyfr (prezentowana wartość będzie dopełniana zerami poprzedzającymi). Przykład: <pre>int suma = 10; Console.WriteLine("{0:D4}", suma); // wyświetli się 0010 Console.WriteLine("{0,-8:D4}", suma); // 0010 na 8 znakach, do lewej</pre>
E lub e	Konwersja do notacji naukowej. Precyzja określa liczbę miejsc po przecinku. Przykład: <pre>Console.WriteLine("{0:E2}", 1052.0329112756); // 1,05E+003;</pre>
F lub f	Wartości stałopozycyjne. Precyzja określa liczbę miejsc po przecinku. Przykład: <pre>double x = 12.345678; Console.WriteLine("{0:F4}", x); // 12,3457 Console.WriteLine("{0,10:F4}", x); // 12,3457, na 10 znakach do prawej</pre>
P lub p	Wartości procentowe. Liczba jest mnożona przez 100 i wyświetlana ze znakiem procenta. Precyzja określa liczbę miejsc po przecinku (domyślnie dwa). Przykład: <pre>Console.WriteLine("Wskaźnik: {0:P}", 0.45); // 45,00% Console.WriteLine("Wskaźnik: {0,8:P0}", 0.45); // 45% na 8 znakach do prawej</pre>

Wykaz zawiera tylko wybrane specyfikatory formatu dla wartości liczbowych. Pełny wykaz znajduje się w dokumentacji MSDN: <http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx>. Wykaz specyfikatorów dla dat: <http://msdn.microsoft.com/en-us/library/az4se3k1.aspx>.

Tabela 7. Znaki specjalne

Znak specjalny	Opis
<code>\'</code>	Znak apostrofu (')
<code>\"</code>	Znak cudzysłowu (")
<code>\\</code>	Ukośnik lewy (ang. backslash) (\)
<code>\n</code>	Nowa linia
<code>\t</code>	Tabulacja pozioma
<code>\0</code>	Znak pusty

Wykaz zawiera tylko wybrane znaki specjalne. Pełny wykaz znajduje się w dokumentacji MSDN: [http://msdn.microsoft.com/en-us/library/aa691087\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa691087(v=vs.71).aspx). Przykłady użycia: [http://msdn.microsoft.com/en-us/library/aa691090\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa691090(v=vs.71).aspx).

Tabela 8. Instrukcje warunkowe

Instrukcja	Opis
if	<p>if (wyrażenie logiczne) instrukcja; Jeśli wyrażenie logiczne ma wartość <i>true</i> to wykonywana jest <i>instrukcja</i>.</p> <p>lub</p> <p>if (wyrażenie logiczne) instrukcja1 else instrukcja2; Jeśli wyrażenie logiczne ma wartość <i>true</i> to wykonywana jest <i>instrukcja1</i> w przeciwnym razie <i>instrukcja2</i>.</p> <p>W obu przypadkach zamiast pojedynczych instrukcji mogą być bloki instrukcji ujęte w klamry. Instrukcje <i>if</i> można zagnieżdżać.</p> <p>Przykład 1 (sprawdzenie warunku):</p> <pre>if(x > 10 && x < 45) { y++; s = s * y; }</pre> <p>Przykład 2 (sprawdzenie warunku oraz jego alternatywy):</p> <pre>if(x > 0) y = ++x; else y = --x;</pre> <p>Przykład 3 (postać wielowarunkowa instrukcji <i>if</i>):</p> <pre>if (cyfra == 0) slownie = "zero"; else if (cyfra == 1) slownie = "jeden"; else if (cyfra == 2) slownie = "dwa";</pre>
operator warunkowy ()?:	<p>(wyrażenie logiczne) ? wyrażenie1 : wyrażenie2; Jeśli wyrażenie logiczne ma wartość <i>true</i>, to operator warunkowy zwraca <i>wyrażenie1</i>, w przeciwnym wypadku zwraca <i>wyrażenie2</i>.</p> <p>Przykłady</p> <pre>int x = 1; int y = (x > 0) ? 5 : 6; // y będzie równe 5 int y = (x > 0) ? ++x : --x; // y będzie równe 2 (kod równoważny z kodem przykładu 2 dla instrukcji if)</pre>
switch..case	<p>Przykład (wyświetli się napis "Wybrano 1")</p> <pre>int x = 1; switch (x) { case 1: Console.WriteLine("Wybrano 1"); break; case 2: Console.WriteLine("Wybrano 2"); break; default: Console.WriteLine("Domyślny wybór"); break; }</pre>

Opis składni w dokumentacji MSDN: [http://msdn.microsoft.com/en-us/library/5011f09h\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/5011f09h(v=vs.90).aspx) (if) oraz [http://msdn.microsoft.com/en-us/library/06tc147t\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/06tc147t(v=vs.90).aspx) (switch..case).

Tabela 9. Instrukcje iteracyjne (pętle)

Instrukcja	Opis	
for	<pre>for(inicjalizacja; warunek; iteracja) // ciało pętli</pre> <p>Ciało pętli jest wykonywane dopóki spełniony jest warunek (wyrażenie logiczne). Pęta operuje na zmiennej sterującej (liczniku pętli).</p> <p>Przykład 1 (wypisanie liczb 1, 2, 3, 4 jedna pod drugą)</p> <pre>for (int i = 1; i < 5; i++) { Console.WriteLine(i); }</pre>	<p>Przykład 2 (wypisanie liczb 6,5,4,3 jedna pod drugą)</p> <pre>for (int i = 6; i > 2; i--) { Console.WriteLine(i); }</pre> <p>Przykład 3 (zagnieżdżone pętle <i>for</i>, wypisanie trzech wierszy, jeden pod drugim, w pierwszym cyfry 1, 1, 1, w drugim 2, 2, 2, a w trzecim 3, 3, 3)</p> <pre>for (int i = 1; i < 4; i++) { for (int j = 1; j < 4; j++) Console.Write(i); Console.WriteLine(); }</pre>
while	<pre>while (wyrażenie logiczne) { // ciało pętli }</pre> <p>Ciało pętli jest wykonywane dopóki wyrażenie logiczne ma wartość <i>true</i>.</p> <p>Przykład (wypisanie liczb 1,2,3,4 jedna pod drugą)</p> <pre>int i = 1; while (i < 5) { Console.WriteLine(i); i++; }</pre>	
do while	<pre>do { // ciało pętli } while (wyrażenie logiczne);</pre> <p>Ciało pętli jest wykonywane dopóki wyrażenie logiczne ma wartość <i>true</i>. Wyrażenie logiczne sprawdzane jest po pierwszym przebiegu pętli (czyli pętla wykona się przynajmniej raz).</p>	<p>Przykład (w pętli oczekiwanie na wprowadzenie tekstu, dopóki tekst jest różny od tekstu „koniec”)</p> <pre>do { Console.WriteLine("Wpisz"); a = Console.ReadLine(); } while (a != "koniec");</pre>
foreach	<p>Pętla <i>foreach</i> jest przeznaczona dla kolekcji (np. tablicy). Ma tyle przebiegów, ile jest elementów w kolekcji. Przy użyciu tej pętli nie można zmieniać elementów kolekcji (są dostępne tylko „do odczytu”).</p> <p>Przykład (wypisanie na ekranie imion Ala, Ola, Ela, jedno pod drugim)</p> <pre>string[] imiona = { "Ala" , "Ola" , "Ela" }; foreach (string x in imiona) Console.WriteLine(x);</pre>	

Opis składni w dokumentacji MSDN: [http://msdn.microsoft.com/en-us/library/32dbftby\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/32dbftby(v=vs.90).aspx)

Tabela 10. Tablice

Rodzaj	Opis
Tablica jedno-wymiarowa	<pre>Typ[] nazwa = new Typ[rozmiar];</pre> <p>lub</p> <pre>Typ[] nazwa = {Lista inicjalizacyjna};</pre> <p>Przykład 1 (deklaracja tablicy i w kolejnych liniach inicjalizacja elementów)</p> <pre>int[] tablica = new int[3]; tablica[0] = 19; tablica[1] = 34; tablica[2] = 23;</pre> <p>Przykład 2 (deklaracja i inicjalizacja tablicy w jednej linii – zapis równoważny z zapisem z przykładu 1)</p> <pre>int[] tablica = {19, 34, 23};</pre> <p>Przykład 3 (wyświetlenie wszystkich elementów tablicy przy użyciu pętli <i>for</i>)</p> <pre>int[] tablica = {10, 15, 25}; for(int i = 0; i < tablica.Length; i++) Console.WriteLine(tablica[i]);</pre>
Tablica dwu-wymiarowa prostokątna	<p>Przykład 1 (deklaracja tablicy dwuwymiarowej o rozmiarze 3 x 3, a następnie w pętli wpisanie liczby 2 do każdego jej elementu)</p> <pre>int[,] tab = new int [3,3]; for (int i = 0; i < tab.GetLength(0); i++) for (int j = 0; j < tab.GetLength(1); j++) tab[i,j] = 2;</pre> <p>Przykład 2 (deklaracja tablicy dwuwymiarowej wraz z inicjalizacją, a następnie wypisanie elementów wiersz po wierszu)</p> <pre>int[,] tab = { { 1, 2 }, { 3, 4 }, { 5, 6 } }; for (int i = 0; i < tab.GetLength(0); i++) { for (int j = 0; j < tab.GetLength(1); j++) { Console.Write(tab[i, j]); } Console.WriteLine(); }</pre>
Tablica postrzępiona	<p>Przykład (deklaracja tablicy z inicjalizacją i wyświetlenie elementów wierszami przy użyciu zagnieżdżonych pętli <i>foreach</i>)</p> <pre>int[][] tab = { new int[] { 1, 2 }, new int[] { 3, 4, 5, 6 } }; foreach (int[] podtablica in tab) { foreach (int x in podtablica) Console.Write(x); Console.WriteLine(); }</pre>

Opis składni w dokumentacji MSDN: <http://msdn.microsoft.com/en-us/library/vstudio/9b9dty7d.aspx>.

Tabela 11. Wybrane metody klasy String

Metoda	Opis
Substring()	<p>Metoda zwraca podłańcuch pobrany z łańcucha, dla którego jest wołana. Występuje w dwóch przeciążonych wariantach, przedstawionych na poniższych przykładach.</p> <p>Przykład 1 (wyświetli podłańcuch pobrany z łańcucha <i>s1</i> od podanej pozycji (7) do końca. Uwaga: Pozycje (indeksy) w łańcuchach numerowane są od 0)</p> <pre>string s1 = "Ala ma kota"; Console.WriteLine(s1.Substring(7)); // wyświetli "kota"</pre> <p>Przykład 2 (Wyświetli podłańcuch pobrany z łańcucha <i>s1</i> od pozycji podanej jako pierwszy argument (7), tyle znaków, ile wynika z drugiego argumentu (2)).</p> <pre>string s1 = "Ala ma kota"; Console.WriteLine(s1.Substring(7, 2)); // wyświetli "ko"</pre>
IndexOf()	<p>Metoda ma kilka przeciążonych wariantów, omawiamy tu tylko ten, który ma jeden argument typu <i>string</i>. Metoda zwraca numer pozycji pierwszego wystąpienia tekstu (podanego jako argument metody) w łańcuchu znakowym, dla którego jest wołana. Jeśli tekst nie występuje zwraca -1.</p> <p>Przykład (wyświetli numer pozycji znalezionej tekstu w łańcuchu <i>s1</i> lub wartość -1 w przypadku, gdy tekst nie zostanie znaleziony)</p> <pre>string s1 = "Ala ma kota"; Console.WriteLine(s1.IndexOf("kota")); // wyświetli liczbę 7 Console.WriteLine(s1.IndexOf("psa")); // wyświetli liczbę -1</pre> <p>Podobne działanie ma metoda <i>LastIndexOf()</i>, która zwraca pozycję ostatniego wystąpienia szukanego tekstu.</p>
Insert	<p>Wstawianie tekstu do łańcucha znakowego. Metoda ma dwa argumenty – numer pozycji, gdzie należy wstawić tekst oraz tekst do wstawienia.</p> <p>Przykład (dodanie tekstu "psa i " w miejscu, gdzie zaczyna się podłańcuch "kota", czyli wyświetli się "Ala ma psa i kota")</p> <pre>string s1 = "Ala ma kota"; Console.WriteLine(s1.Insert(7, "psa i "));</pre>
Compare	<p>Metoda ma kilka przeciążonych wariantów, omawiamy tu ten, który przyjmuje dwa argumenty typu <i>string</i> (ten wariant uwzględnia wielkość liter przy porównywaniu). Metoda zwraca wartość ujemną, gdy pierwszy podłańcuch jest „mniejszy” niż drugi. Wartość dodatnią w odwrotnym przypadku. Zwraca zero, gdy oba łańcuchy są równe. Mniejszy i większy w przypadku łańcuchów oznacza uporządkowanie wg alfabetu (np. w poniższym przykładzie tekst <i>s1</i> wg porządku alfabetycznego jest wcześniejszy (mniejszy) niż <i>s2</i>). Uwaga: jest to metoda statyczna, w jej wywołaniu podajemy nazwę klasy.</p> <p>Przykład (porównanie łańcuchów)</p> <pre>string s1 = "brat", s2 = "kot", s3 = "as"; Console.WriteLine(String.Compare(s1, s2)); // wyświetli -1 Console.WriteLine(String.Compare(s1, s3)); // wyświetli 1</pre>

Literatura

Biblioteka MSDN - scentralizowana baza oficjalnych dokumentów dla programistów i deweloperów, zawierająca dokumentacje techniczne <http://msdn.microsoft.com/library> (podstawowe źródło).

Albahari B., Drayton P., Merrill B., *C#. Leksykon*, Wydawnictwo Helion 2001.

Grębosz J., *Symfonia C++ Standard*, Edition 2000, Kraków 2005 (najnowsze wydanie tej książki jest z 2010 r. – polecamy tę książkę wszystkim zainteresowanym językiem C++).

Griffiths I., Adams M., Liberty J., *C#. Programowanie*, Wydawnictwo Helion 2012.

Kubiak M.J., *C#. Zadania z programowania z przykładowymi rozwiązaniami*, Wydawnictwo Helion 2012.

Lee W.M., *C# 2008. Warsztat programisty*, Helion Gliwice 2010.

Liberty J., *C#. Programowanie*, Wydawnictwo Helion 2005.

Lis M., *C#. Praktyczny kurs*, Wydawnictwo Helion 2012.

Perry S.C., *Core C# i .NET*, Helion Gliwice 2006.