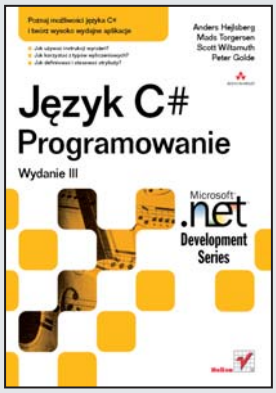
Język C#. Programowanie. Wydanie III. Microsoft .NET Development Series



Autor: Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde
Tłumaczenie: Łukasz Suma
ISBN: 978-83-246-2195-8
Tytuł oryginału: The C# Programming
Format: 170x230, stron: 784

Poznaj możliwości języka C# i twórz wysoko wydajne aplikacje

- Jak używać instrukcji wyrażeń?
- Jak korzystać z typów wyliczeniowych?
- Jak definiować i stosować atrybuty?

Nowoczesny i bezpieczny język programowania C# posiada kilka cech, które ułatwiają opracowywanie solidnych i wydajnych aplikacji – na przykład obsługę wyjątków, wymuszanie bezpieczeństwa typów lub mechanizm odzyskiwania pamięci, czyli automatyczne odzyskiwanie pamięci operacyjnej zajmowanej przez nieużywane obiekty. C# 3.0 oferuje możliwość programowania funkcjonalnego oraz technologię LINQ (zapytań zintegrowanych z językiem), co znacząco poprawia wydajność pracy programisty.

Książka „Język C#. Programowanie. Wydanie III. Microsoft .NET Development Series" zawiera pełną specyfikację techniczną języka programowania C#, opatrzoną najnowszymi zaktualizowanymi informacjami, m.in. na temat inicjalizatorów obiektów i kolekcji, typów anonimowych czy wyrażeń lambda. Dzięki licznym komentarzom i praktycznym poradom, które uzupełniają główną treść podręcznika, szybko nauczysz się posługiwać zmiennymi, przeprowadzać konwersje funkcji i wyznaczać przeładowania. Dowiesz się, jak optymalnie i z fascynującym efektem końcowym wykorzystywać ten nowoczesny język programowania.

- Typy i zmienne
- Klasy i obiekty
- Struktura leksykalna
- Deklaracje struktur
- Składowe
- Konwersje i wyrażenia
- Instrukcje i operatory
- Tablice
- Interfejsy
- Kod nienadzorowany
- Wskaźniki w wyrażeniach
- Bufory o ustalonym rozmiarze
- Dynamiczne alokowanie pamięci

Wykorzystaj wiedzę i doświadczenie najlepszych specjalistów, aby sprawnie posługiwać się językiem C#

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

8. Instrukcje

Język C# oferuje szeroką gamę instrukcji. Większość z nich powinna być dobrze znana programistom, którzy mieli okazję korzystać z języków C i C++.

```
instrukcja:
   instrukcja-oznaczona
   instrukcja-deklaracji
   instrukcja-osadzona

instrukcja-osadzona:
   blok
   instrukcja-pusta
   instrukcja-wyrażenia
   instrukcja-wyboru
   instrukcja-iteracji
   instrukcja-skoku
   instrukcja-try
   instrukcja-checked
   instrukcja-unchecked
   instrukcja-lock
   instrukcja-using
   instrukcja-yield
```

Nieterminalny element *instrukcja-osadzona* jest używany w przypadku instrukcji, które występują w ramach innych instrukcji. Zastosowanie elementu *instrukcja-osadzona* zamiast elementu *instrukcja* wyklucza użycie instrukcji deklaracji i instrukcji oznaczonych w tych kontekstach. Skompilowanie przedstawionego poniżej fragmentu kodu:

```
void F(bool b) {
   if (b)
      int i = 44;
}
```

powoduje zgłoszenie błędu, ponieważ instrukcja `if` wymaga zastosowania w swojej gałęzi elementu *instrukcja-osadzona*, nie zaś elementu *instrukcja*. Gdyby było to dopuszczalne, zmienna `i` mogłaby zostać zadeklarowana, lecz mogłaby nie zostać nigdy użyta. Umieszczenie deklaracji `i` w bloku powoduje, że przykładowy kod staje się poprawny.

8.1. Punkty końcowe i osiągalność

Każda instrukcja ma **punkt końcowy** (ang. *end point*). W intuicyjnym znaczeniu punktem końcowym instrukcji jest miejsce, które znajduje się bezpośrednio za instrukcją. Reguły wykonania instrukcji złożonych (czyli instrukcji zawierających instrukcje osadzone) określają działanie, które jest przeprowadzane, gdy sterowanie osiąga punkt końcowy instrukcji osadzonej. Na przykład gdy sterowanie osiąga punkt końcowy instrukcji w bloku, jest ono przekazywane do kolejnej instrukcji znajdującej się w tym bloku.

Jeśli instrukcja może potencjalnie zostać osiągnięta przez sterowanie, wówczas instrukcja nazywana jest **osiągalną** (ang. *reachable*). Dla odmiany, jeśli nie istnieje możliwość, aby instrukcja została wykonana, nosi ona nazwę **nieosiągalnej** (ang. *unreachable*).

W przedstawionym poniżej przykładzie:

```
void F() {
   Console.WriteLine("osiągalna");
   goto Label;
   Console.WriteLine("nieosiągalna");
   Label:
   Console.WriteLine("osiągalna");
}
```

drugie wywołanie funkcji `Console.WriteLine` jest nieosiągalne, ponieważ nie ma możliwości, aby instrukcja ta została wykonana.

Gdy kompilator stwierdza, że jakaś instrukcja jest nieosiągalna, generowane jest ostrzeżenie. Warto jednak podkreślić, że sytuacja taka *nie* stanowi błędu.

Aby określić, czy pewna instrukcja lub punkt końcowy są osiągalne, kompilator przeprowadza analizę przepływu sterowania zgodnie z regułami osiągalności zdefiniowanymi dla każdej instrukcji. W analizie przepływu sterowania bierze się pod uwagę wartości wyrażeń stałych (opisanych dokładniej w podrozdziale 7.18), które kontrolują zachowanie instrukcji, lecz nie są uwzględniane potencjalne wartości wyrażeń niestałych. Innymi słowy, dla potrzeb analizy przepływu sterowania przyjmuje się, że wyrażenie niestałe danego typu może mieć dowolną wartość, jaką można przedstawić za pomocą tego typu.

W przedstawionym poniżej przykładzie:

```
void F() {
   const int i = 1;
   if (i == 2) Console.WriteLine("nieosiągalna");
}
```

wyrażenie boole'owskie instrukcji `if` jest wyrażeniem stałym, ponieważ obydwa operandy operatora `==` są stałe. Jako takie wyrażenie to jest przetwarzane na etapie kompilacji, dając w wyniku wartość `false`, przez co wywołanie funkcji `Console.WriteLine` zostaje uznane za nieosiągalne. Jeśli jednak `i` zastąpione jest zmienną lokalną, tak jak zostało to pokazane poniżej:

```
void F() {
   int i = 1;
   if (i == 2) Console.WriteLine("osiągalna");
}
```

wywołanie funkcji `Console.WriteLine` zostaje uznane za instrukcję osiągalną, mimo że w rzeczywistości nigdy nie będzie wykonywane.

`Blok` funkcji składowej jest zawsze uważany za osiągalny. Dzięki sukcesywnemu stosowaniu zasad osiągalności dla każdej instrukcji należącej do bloku da się określić osiągalność każdej podanej instrukcji.

W przedstawionym poniżej przykładzie:

```
void F(int x) {
   Console.WriteLine("początek");
   if (x < 0) Console.WriteLine("ujemna");
}
```

osiągalność drugiego wywołania funkcji `Console.WriteLine` jest określana w następujący sposób:

- Pierwsza instrukcja wyrażenia `Console.WriteLine` jest osiągalna, ponieważ osiągalny jest blok metody `F`.
- Punkt końcowy pierwszej instrukcji wyrażenia `Console.WriteLine` jest osiągalny, ponieważ osiągalna jest ta instrukcja.
- Instrukcja `if` jest osiągalna, ponieważ osiągalny jest punkt końcowy instrukcji wyrażenia `Console.WriteLine`.
- Druga instrukcja wyrażenia `Console.WriteLine` jest osiągalna, ponieważ wyrażenie boole'owskie instrukcji `if` nie ma stałej wartości `false`.

Istnieją dwie sytuacje, w których pojawia się błąd czasu kompilacji, gdy punkt końcowy instrukcji jest osiągalny:

- Ponieważ przypadku instrukcji `switch` nie jest dozwolone, aby sekcja „wpadała" do kolejnej sekcji, za błąd czasu kompilacji uważa się przypadek, gdy osiągalny jest punkt końcowy listy instrukcji sekcji `switch`. Wystąpienie takiego błędu oznacza zwykle brak instrukcji `break`.
- Błąd czasu kompilacji pojawia się, gdy osiągalny jest punkt końcowy bloku funkcji składowej, która oblicza wartość. Wystąpienie takiego błędu oznacza zwykle brak instrukcji `return`.

8.2. Bloki

`Blok` umożliwia zapisywanie wielu instrukcji w kontekstach, w których dopuszczalne jest użycie pojedynczej instrukcji:

`blok:`
   { *`lista-instrukcji`*$_{opc}$ }

Blok składa się z ujętej w nawiasy klamrowe opcjonalnej *lista-instrukcji* (o której więcej w punkcie 8.2.1). Jeśli lista instrukcji nie występuje, blok nazywa się pustym.

Blok może zawierać instrukcje deklaracji (o których więcej w podrozdziale 8.5). Zakresem lokalnej zmiennej lub stałej zadeklarowanej w bloku jest ten blok.

W ramach bloku znaczenie nazwy używanej w kontekście wyrażenia zawsze musi być takie samo (o czym więcej w podpunkcie 7.5.2.1).

Blok jest wykonywany w następujący sposób:

- Jeśli blok jest pusty, sterowanie jest przekazywane do punktu końcowego bloku.
- Jeśli blok nie jest pusty, sterowanie jest przekazywane do listy instrukcji. Gdy i jeśli sterowanie osiąga punkt końcowy listy instrukcji, sterowanie jest przekazywane do punktu końcowego bloku.

Lista instrukcji bloku jest osiągalna, jeśli sam blok jest osiągalny.

Punkt końcowy bloku jest osiągalny, jeśli blok ten jest osiągalny i jeśli jest on pusty lub jeśli osiągalny jest punkt końcowy listy wyrażeń.

Blok zawierający jedno lub większą liczbę instrukcji `yield` (o których więcej w podrozdziale 8.14) nosi nazwę bloku iteratora. Bloki iteratorów są używane do implementacji funkcji składowych jako iteratorów (o czym więcej w podrozdziale 10.14). Bloków iteratorów dotyczą pewne dodatkowe ograniczenia:

- Błędem czasu kompilacji jest, gdy w bloku iteratora pojawia się instrukcja `return` (dozwolone są tu jednak instrukcje `yield return`).
- Błędem czasu kompilacji jest, gdy blok iteratora zawiera kontekst nienadzorowany (opisany w podrozdziale 18.1). Blok iteratora zawsze określa kontekst nadzorowany, nawet jeśli jego deklaracja jest zagnieżdżona w kontekście nienadzorowanym.

8.2.1. Lista instrukcji

Lista instrukcji (ang. *statement list*) zawiera jedną lub większą liczbę instrukcji zapisanych w postaci ciągu. Listy instrukcji występują w elementach *blok* (opisanych w podrozdziale 8.2) oraz w elementach *blok-switch* (o których więcej w punkcie 8.7.2):

```
lista-instrukcji:
   instrukcja
   lista-instrukcji instrukcja
```

Lista instrukcji jest wykonywana przez przekazanie sterowania do pierwszej instrukcji. Gdy i jeśli sterowanie osiąga końcowy punkt instrukcji, jest ono przekazywane do następnej. Gdy i jeśli sterowanie osiąga końcowy punkt ostatniej instrukcji, jest ono przekazywane do końcowego punktu listy instrukcji.

Instrukcja znajdująca się na liście instrukcji jest osiągalna, jeśli spełniony jest przynajmniej jeden z następujących warunków:

- Instrukcja jest pierwszą instrukcją i sama lista instrukcji jest osiągalna.
- Końcowy punkt poprzedzającej instrukcji jest osiągalny.
- Instrukcja jest instrukcją oznaczoną, a do jej etykiety odnosi się osiągalna instrukcja `goto`.

> **VLADIMIR RESHETNIKOV** Reguła ta nie ma zastosowania, gdy instrukcja `goto` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a instrukcja oznaczona występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

Punkt końcowy listy instrukcji jest osiągalny, jeśli osiągalny jest końcowy punkt ostatniej instrukcji znajdującej się na liście.

8.3. Instrukcja pusta

`Instrukcja-pusta` nie powoduje wykonania żadnych działań:

```
instrukcja-pusta:
   ;
```

Instrukcja pusta jest używana, gdy nie ma żadnych operacji do wykonania w kontekście, w którym wymagana jest instrukcja.

Wykonanie instrukcji pustej powoduje po prostu przekazanie sterowania do końcowego punktu instrukcji. Z tego powodu końcowy punkt instrukcji pustej jest osiągalny, jeśli osiągalna jest ta instrukcja pusta.

Instrukcja pusta może być zastosowana przy pisaniu instrukcji `while` z pustym ciałem, tak jak zostało to zaprezentowane poniżej:

```
bool ProcessMessage() {...}

void ProcessMessages() {
   while (ProcessMessage())
      ;
}
```

Dodatkowo instrukcja pusta może być wykorzystywana do deklarowania etykiety tuż przed zamykającym nawiasem klamrowym (}) bloku, tak jak zostało to przedstawione poniżej:

```
void F() {
   ...
   if (done) goto exit;
      ...
   exit: ;
}
```

8.4. Instrukcje oznaczone

Dzięki elementowi *instrukcja-oznaczona* możliwe jest poprzedzanie instrukcji za pomocą etykiety. Instrukcje oznaczone są dopuszczalne w blokach, nie mogą jednak występować jako instrukcje osadzone:

```
instrukcja-oznaczona:
   identyfikator : instrukcja
```

Za pomocą instrukcji oznaczonej można deklarować etykietę o nazwie określonej przez *identy*
↪fikator. Zakresem etykiety jest cały blok, w którym została ona zadeklarowana, w tym również wszelkie bloki zagnieżdżone. Błędem czasu wykonania jest, gdy dwie etykiety o tej samej nazwie mają zachodzące na siebie zakresy.

Do etykiety mogą odnosić się instrukcje `goto` (opisane w punkcie 8.9.3) występujące w ramach jej zakresu. W konsekwencji instrukcje `goto` mogą przekazywać sterowanie w obrębie bloków oraz poza bloki, nigdy jednak do bloków.

Etykiety mają swoją własną przestrzeń deklaracji i nie kolidują z innymi identyfikatorami. Przedstawiony poniżej przykład:

```
int F(int x) {
   if (x >= 0) goto x;
   x = -x;
   x: return x;
}
```

jest prawidłowym fragmentem kodu, w którym nazwa `x` jest wykorzystywana zarówno w roli parametru, jak i etykiety.

Wykonanie instrukcji oznaczonej odpowiada dokładnie wykonaniu instrukcji, która znajduje się bezpośrednio po niej.

Oprócz osiągalności zapewnianej przez normalny przepływ sterowania instrukcja oznaczona może również zawdzięczać osiągalność odwołującej się do niej osiągalnej instrukcji `goto`. Wyjątkiem jest tu sytuacja, w której instrukcja `goto` znajduje się wewnątrz instrukcji `try` zawierającej blok `finally`, a instrukcja oznaczona występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny; wówczas instrukcja oznaczona nie jest osiągalna za pomocą tej instrukcji `goto`.

ERIC LIPPERT Może się na przykład zdarzyć, że blok `finally` zawsze będzie zgłaszał wyjątek; w takim przypadku będzie istniała osiągalna instrukcja `goto` przekierowująca do potencjalnie nieosiągalnej etykiety.

CHRIS SELLS Proszę nie używać etykiet ani instrukcji `goto`. Nigdy nie zdarzyło mi się ujrzeć kodu, który nie byłby bardziej czytelny bez nich.

8.5. Instrukcje deklaracji

Instrukcja-deklaracji umożliwia deklarowanie lokalnej zmiennej lub stałej. Instrukcji deklaracji można używać w blokach, ale nie są one dozwolone w roli instrukcji zagnieżdżonych:

```
instrukcja-deklaracji:
   deklaracja-zmiennej-lokalnej ;
   deklaracja-stałej-lokalnej ;
```

8.5.1. Deklaracje zmiennych lokalnych

Deklaracja-zmiennej-lokalnej umożliwia deklarowanie jednej lub większej liczby zmiennych lokalnych:

```
deklaracja-zmiennej-lokalnej:
   typ-zmiennej-lokalnej deklaratory-zmiennych-lokalnych

typ-zmiennej-lokalnej:
   typ
   var

deklaratory-zmiennych-lokalnych:
   deklarator-zmiennej-lokalnej
   deklaratory-zmiennych-lokalnych , deklarator-zmiennej-lokalnej

deklarator-zmiennej-lokalnej:
   identyfikator
   identyfikator = inicjalizator-zmiennej-lokalnej

inicjalizator-zmiennej-lokalnej:
   wyrażenie
   inicjalizator-tablicy
```

Typ-zmiennej-lokalnej elementu *deklaracja-zmiennej-lokalnej* określa typ zmiennych wprowadzanych przez deklarację lub wskazuje za pomocą słowa kluczowego `var`, że typ ten powinien zostać wywnioskowany na podstawie inicjalizatora. Po typie występuje lista elementów *deklarator-zmiennej-lokalnej*, z których każdy wprowadza nową zmienną. *Deklarator-zmiennej-lokalnej* składa się z elementu *identyfikator* określającego nazwę zmiennej, po którym może opcjonalnie występować token `=` oraz *inicjalizator-zmiennej-lokalnej*, który nadaje zmiennej wartość początkową.

Gdy *typ-zmiennej-lokalnej* jest określony jako `var`, a w zakresie nie ma typu o takiej nazwie, deklaracja jest **deklaracją zmiennej lokalnej niejawnie typowanej** (ang. *implicitly typed local variable declaration*), której typ jest wnioskowany na podstawie powiązanego elementu *wyrażenie* inicjalizatora. Deklaracje zmiennych lokalnych typowanych niejawnie podlegają ograniczeniom:

- *Deklaracja-zmiennej-lokalnej* nie może zawierać wielu elementów *deklarator-zmiennej-lokalnej*.

ERIC LIPPERT Na wczesnych etapach opracowywania tej możliwości dopuszczalne było korzystanie w tym miejscu z wielu deklaratorów, tak jak zostało to pokazane poniżej:

```
var a = 1, b = 2.5;
```

Gdy programiści C# ujrzeli ten kod, mniej więcej połowa z nich stwierdziła, że zapis ten powinien mieć tę samą semantykę co przedstawiony poniżej:

```
double a = 1, b = 2.5;
```

Druga połowa powiedziała jednak, że powinien mieć taką semantykę jak poniżej:

```
int a = 1; double b = 2.5;
```

Obydwie grupy uważały, że ich interpretacja zagadnienia była „w oczywisty sposób poprawna".

Gdy ma się do czynienia ze składnią, która umożliwia dwie niezgodne, „w oczywisty sposób poprawne" interpretacje, najlepszą rzeczą, jaką można zwykle zrobić, to całkowicie zabronić korzystania z tej składni, zamiast wprowadzać zbędne zamieszanie.

- *Deklarator-zmiennej-lokalnej* musi zawierać *inicjalizator-zmiennej-lokalnej*.
- *Inicjalizator-zmiennej-lokalnej* musi być *wyrażenie*.
- *Wyrażenie* inicjalizatora musi mieć typ czasu kompilacji.
- *Wyrażenie* inicjalizatora nie może odwoływać się do samej deklarowanej zmiennej.

ERIC LIPPERT Podczas gdy w deklaracji zmiennej lokalnej jawnie typowanej może występować odwołanie do niej samej z poziomu inicjalizatora, w przypadku deklaracji zmiennej lokalnej niejawnie typowanej jest to niedopuszczalne.

Na przykład instrukcja `int j = M(out j);` jest dziwna, ale w pełni prawidłowa. Gdyby jednak wyrażenie miało postać `var j = M(out j)`, wówczas mechanizm wyznaczania przeładowania nie mógłby określić typu zwracanego przez metodę `M`, a więc również typu zmiennej `j`, dopóki nie byłby znany typ argumentu. Typ argumentu jest oczywiście dokładnie tym, co próbujemy tu określić.

Zamiast rozwiązywać tu problem „pierwszeństwa kury lub jajka", specyfikacja języka po prostu uniemożliwia stosowanie tego rodzaju konstrukcji.

Poniżej zostały przedstawione przykłady nieprawidłowych deklaracji zmiennych lokalnych typowanych niejawnie:

```
var x;                  // Błąd, brak inicjalizatora, na podstawie którego można wywnioskować typ
var y = {1, 2, 3};      // Błąd, nie jest tu dozwolony inicjalizator tablicowy
var z = null;           // Błąd, null nie ma typu
var u = x => x + 1;     // Błąd, funkcje anonimowe nie mają typu
var v = v++;            // Błąd, inicjalizator nie może odnosić się do samej zmiennej
```

Wartość zmiennej lokalnej jest otrzymywana za pomocą wyrażenia, w którym używa się elementu *nazwa-prosta* (o którym więcej w punkcie 7.5.2). Wartość zmiennej lokalnej modyfikuje się za pomocą elementu *przypisanie* (o którym więcej w podrozdziale 7.16). Zmienna lokalna musi być niewątpliwie ustalona (o czym więcej w podrozdziale 5.3) w każdym miejscu, w którym jej wartość jest uzyskiwana.

CHRIS SELLS Naprawdę uwielbiam deklaracje zmiennych lokalnych typowanych niejawnie, gdy typ jest anonimowy (w którym to przypadku musisz ich używać) lub gdy typ zmiennej jest ujawniony jako część wyrażenia inicjalizującego, a nie dlatego, że jesteś zbyt leniwy, aby pisać!

Przykładem może tu być przedstawiony poniżej fragment kodu:

```
var a = new { Name = "Bob", Age = 42 };   // Dobrze
var b = 1;                                // Dobrze
var v = new Person();                     // Dobrze
var d = GetPerson();                      // ŹLE!
```

Kompilator nie ma najmniejszego problemu z tym, że `d` jest niejawnie typowaną zmienną, za to człowiek, który musi czytać taki kod — wręcz przeciwnie!

Zakresem zmiennej lokalnej deklarowanej w *deklaracja-zmiennej-lokalnej* jest blok, w którym następuje ta deklaracja. Odwoływanie się do zmiennej lokalnej występujące w tekście programu przed elementem *deklarator-zmiennej-lokalnej* jest błędem czasu kompilacji. Błędem czasu kompilacji jest również zadeklarowanie w ramach zakresu zmiennej lokalnej innej lokalnej zmiennej lub stałej o tej samej nazwie.

Deklaracja zmiennej lokalnej, za pomocą której deklaruje się wiele zmiennych, jest równoważna z wieloma deklaracjami pojedynczych zmiennych tego samego typu. Ponadto inicjalizator zmiennej w deklaracji zmiennej lokalnej dokładnie odpowiada instrukcji przypisania, która jest umieszczona bezpośrednio za deklaracją.

Przedstawiony poniżej przykładowy fragment kodu:

```
void F() {
   int x = 1, y, z = x * 2;
}
```

dokładnie odpowiada następującemu fragmentowi:

```
void F() {
   int x; x = 1;
   int y;
   int z; z = x * 2;
}
```

W deklaracji zmiennej lokalnej typowanej niejawnie przyjmuje się, że typ deklarowanej zmiennej lokalnej jest taki sam jak typ wyrażenia użytego do jej zainicjalizowania. Przykładem tego może być fragment kodu pokazany poniżej:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

Deklaracje zmiennych lokalnych typowanych niejawnie przedstawione w powyższym przykładzie są dokładnie równoznaczne z typowanymi jawnie deklaracjami, które zostały zaprezentowane poniżej:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

8.5.2. Deklaracje stałych lokalnych

Deklaracja-stałej-lokalnej umożliwia deklarowanie jednej stałej lokalnej lub większej liczby stałych lokalnych:

```
deklaracja-stałej-lokalnej:
   const typ deklaratory-stałych

deklaratory-stałych:
   deklarator-stałej
   deklaratory-stałych , deklarator-stałej

deklarator-stałej:
   identyfikator = wyrażenie-stałe
```

Typ elementu *deklaracja-stałej-lokalnej* określa typ stałych wprowadzonych przez deklarację. Po typie występuje lista elementów *deklarator-stałej*, z których każdy wprowadza nową stałą. *Deklarator-stałej* składa się z elementu *identyfikator* określającego nazwę stałej, po którym następuje token =, po nim zaś *wyrażenie-stałe* (opisane w podrozdziale 7.18), które określa wartość stałej.

Typ i *wyrażenie-stałe* deklaracji stałej lokalnej muszą stosować się do tych samych reguł, które działają w przypadku odpowiednich elementów deklaracji składowej stałej (o której więcej w podrozdziale 10.4).

Wartość stałej lokalnej jest uzyskiwana w wyrażeniu wykorzystującym element *nazwa-prosta* (opisany w punkcie 7.5.2).

Zakresem stałej lokalnej jest blok, w którym znajduje się ta deklaracja. Odwoływanie się do stałej lokalnej występujące w tekście programu przed elementem *deklarator-stałej* jest błędem czasu kompilacji. Błędem czasu kompilacji jest również zadeklarowanie w ramach zakresu stałej lokalnej innej stałej lub zmiennej lokalnej o tej samej nazwie.

Deklaracja stałej lokalnej, za pomocą której deklaruje się wiele stałych, jest równoważna z wieloma deklaracjami pojedynczych stałych tego samego typu.

8.6. Instrukcje wyrażeń

Instrukcja-wyrażenia powoduje przetworzenie danego wyrażenia. Wartość obliczona przez wyrażenie, jeśli występuje, jest odrzucana:

```
instrukcja-wyrażenia:
   wyrażenie-instrukcji ;

wyrażenie-instrukcji:
   wyrażenie-wywołania
   wyrażenie-tworzenia-obiektu
   przypisanie
   wyrażenie-poinkrementacyjne
   wyrażenie-podekrementacyjne
   wyrażenie-przedinkrementacyjne
   wyrażenie-przeddekrementacyjne
```

Nie wszystkie wyrażenia mogą być stosowane jako instrukcje. W szczególności wyrażenia takie jak `x + y` czy `x == 1`, które jedynie obliczają jakąś wartość (która i tak zostałaby następnie odrzucona), nie mogą występować w roli instrukcji.

Wykonanie elementu *instrukcja-wyrażenia* powoduje przetworzenie zawartego w nim wyrażenia, a następnie przekazanie sterowania do końcowego punktu elementu *instrukcja-wyrażenia*. Końcowy punkt elementu *instrukcja-wyrażenia* jest osiągalny, gdy ta *instrukcja-wyrażenia* jest osiągalna.

8.7. Instrukcje wyboru

Instrukcje wyboru powodują wybranie jednej z wielu możliwych instrukcji do wykonania na podstawie wartości pewnego wyrażenia:

```
instrukcja-wyboru:
    instrukcja-if
    instrukcja-switch
```

8.7.1. Instrukcja if

Instrukcja `if` powoduje wybranie instrukcji do wykonania na podstawie wartości wyrażenia boole'owskiego:

```
instrukcja-if:
  if ( wyrażenie-boole'owskie ) instrukcja-osadzona
  if ( wyrażenie-boole'owskie ) instrukcja-osadzona else instrukcja-osadzona
```

Część `else` jest związana z najbliższą, poprzedzającą ją leksykalnie częścią `if`, która jest dozwolona przez składnię. Z tego powodu instrukcja `if` o przedstawionej poniżej postaci:

```
if (x) if (y) F(); else G();
```

jest równoważna z następującą:

```
if (x) {
   if (y) {
      F();
   }
   else {
      G();
   }
}
```

Instrukcja `if` jest wykonywana w następujący sposób:

- Przetwarzane jest *wyrażenie-boole'owskie* (przedstawione w podrozdziale 7.19).
- Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `true`, sterowanie jest przekazywane do pierwszej osadzonej instrukcji. Gdy sterowanie osiąga jej punkt końcowy, przekazywane jest do punktu końcowego instrukcji `if`.
- Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `false` i jeśli obecna jest część `else`, sterowanie przekazywane jest do drugiej osadzonej instrukcji. Gdy sterowanie osiąga jej punkt końcowy, przekazywane jest do punktu końcowego instrukcji `if`.
- Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `false` i jeśli nie jest obecna część `else`, sterowanie przekazywane jest do punktu końcowego instrukcji `if`.

Pierwsza instrukcja osadzona instrukcji `if` jest osiągalna, jeśli osiągalna jest instrukcja `if` i wyrażenie boole'owskie nie ma stałej wartości `false`.

Druga instrukcja osadzona instrukcji `if`, jeśli obecna, jest osiągalna, jeśli osiągalna jest instrukcja `if` i wyrażenie boole'owskie nie ma stałej wartości `true`.

Punkt końcowy instrukcji `if` jest osiągalny, jeśli osiągalny jest punkt końcowy przynajmniej jednej z jej osadzonych instrukcji. Ponadto punkt końcowy instrukcji `if` pozbawionej części `else` jest osiągalny, jeśli osiągalna jest instrukcja `if` i wyrażenie boole'owskie nie ma stałej wartości `true`.

8.7.2. Instrukcja switch

Instrukcja `switch` powoduje wybranie do wykonania listy instrukcji związanej z etykietą przełącznika, która odpowiada wartości wyrażenia przełączającego:

```
instrukcja-switch:
   switch ( wyrażenie ) blok-switch

blok-switch:
   { sekcje-switch_opc }

sekcje-switch:
   sekcja-switch
   sekcje-switch sekcja-switch

sekcja-switch:
   etykiety-switch lista-instrukcji

etykiety-switch:
   etykieta-switch
   etykiety-switch etykieta-switch

etykieta-switch:
   case wyrażenie-stałe :
   default :
```

Instrukcja-switch składa się ze słowa kluczowego `switch`, po którym występuje ujęte w nawiasy wyrażenie (noszące nazwę wyrażenia przełączającego), po nim zaś *blok-switch*. *Blok-switch* składa się z zera lub większej liczby ujętych w nawiasy klamrowe elementów *sekcja-switch*. Każda *sekcja-switch* zawiera jedną lub większą liczbę elementów *etykiety-switch*, po których występuje *lista-instrukcji* (opisana w punkcie 8.2.1).

Typ rządzący (ang. *governing type*) instrukcji `switch` jest ustalany na podstawie wyrażenia przełączającego w następujący sposób:

- Jeśli typem wyrażenia przełączającego jest `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string` bądź *typ-wyliczeniowy* lub jeśli jest to odpowiadający jednemu z wymienionych typ dopuszczający wartość pustą, wówczas jest on typem rządzącym instrukcji `switch`.
- W innym przypadku musi istnieć dokładnie jedna zdefiniowana przez użytkownika konwersja niejawna (o czym więcej w podrozdziale 6.4) z typu wyrażenia przełączającego na jeden z następujących typów rządzących: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string` lub typ dopuszczający wartość pustą odpowiadający jednemu z wymienionych typów.
- W innym przypadku, jeśli nie istnieje tego rodzaju konwersja niejawna lub jeśli istnieje więcej takich konwersji niejawnych, pojawia się błąd czasu kompilacji.

DON BOX Często przyłapuję się na marzeniu o możliwości używania typu `System.Type` jako typu rządzącego (znanego też jako „typeswitch"). Kocham funkcje wirtualne równie mocno, jak wszyscy wokół, ale niezmiennie pragnę pisać kod, który interpretuje istniejące typy i podejmuje różne działania na podstawie typu pewnej wartości. Dużo radości sprawiłby mi również operator dopasowania w rodzaju ML, który byłby nawet bardziej przydatny.

Wyrażenie stałe każdej etykiety `case` musi oznaczać wartość typu, który da się niejawnie konwertować (o czym więcej w podrozdziale 6.1) na typ rządzący instrukcji `switch`. Jeśli dwie lub większa liczba etykiet należących do tej samej instrukcji `switch` określają tę samą wartość stałą, generowany jest błąd czasu kompilacji.

W instrukcji przełączającej może znajdować się co najwyżej jedna etykieta `default`.

Instrukcja `switch` jest przetwarzana w następujący sposób:

- Wyrażenie przełączające jest przetwarzane i konwertowane na typ rządzący.
- Jeśli jedna ze stałych określonych w etykiecie `case` należącej do tej samej instrukcji `switch` jest równa wartości wyrażenia przełączającego, sterowanie jest przekazywane do listy instrukcji znajdującej się po dopasowanej etykiecie `case`.
- Jeśli żadna ze stałych określonych w etykiecie `case` należącej do tej samej instrukcji `switch` nie jest równa wartości wyrażenia przełączającego i jeśli obecna jest etykieta `default`, sterowanie jest przekazywane do listy instrukcji znajdującej się po etykiecie `default`.
- Jeśli żadna ze stałych określonych w etykiecie `case` należącej do tej samej instrukcji `switch` nie jest równa wartości wyrażenia przełączającego i jeśli nie jest obecna etykieta `default`, sterowanie jest przekazywane do punktu końcowego instrukcji `switch`.

Jeśli osiągalny jest końcowy punkt listy instrukcji sekcji `switch`, generowany jest błąd czasu kompilacji. Zasada ta znana jest jako reguła „niewpadania" (ang. *no-fall-through*). Przedstawiony poniżej fragment kodu:

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
default:
   CaseOthers();
   break;
}
```

jest prawidłowy, ponieważ żadna z sekcji instrukcji przełączającej nie ma osiągalnego punktu końcowego. W przeciwieństwie do języków C i C++ wykonanie sekcji `switch` nie może tu „wpadać" do kolejnej sekcji, a próba skompilowania pokazanego poniżej przykładu:

```
switch (i) {
case 0:
   CaseZero();
case 1:
   CaseZeroOrOne();
default:
   CaseAny();
}
```

powoduje wystąpienie błędu czasu kompilacji. Gdy po wykonaniu pewnej sekcji instrukcji przełączającej ma następować wykonanie innej sekcji, zastosowana musi zostać jawna instrukcja `goto` lub `goto default`, tak jak zostało to zaprezentowane poniżej:

```
switch (i) {
case 0:
   CaseZero();
   goto case 1;
case 1:
   CaseZeroOrOne();
   goto default;
default:
   CaseAny();
   break;
}
```

W elemencie *sekcja-switch* dozwolone jest stosowanie wielu etykiet. Przedstawiony poniżej przykład kodu:

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
case 2:
default:
   CaseTwo();
   break;
}
```

jest prawidłowy. Zaprezentowany kod nie łamie reguły „niewpadania", ponieważ etykiety `case 2:` oraz `default:` są częściami tego samego elementu *sekcja-switch*.

Reguła „niewpadania" zabezpiecza przed występowaniem powszechnej klasy błędów, które zdarzają się w językach C i C++, gdy pominięte zostają przypadkiem instrukcje `break`. Dodatkowo dzięki tej regule sekcje instrukcji przełączającej mogą być dowolnie przestawiane bez wpływu na zachowanie całej instrukcji. Na przykład kolejność sekcji instrukcji `switch` przedstawionej w zaprezentowanym wcześniej fragmencie kodu można odwrócić bez modyfikowania sposobu działania tej instrukcji, tak jak zostało to pokazane poniżej:

```
switch (i) {
default:
   CaseAny();
```

```
   break;
case 1:
   CaseZeroOrOne();
   goto default;
case 0:
   CaseZero();
   goto case 1;
}
```

Lista instrukcji sekcji przełączającej kończy się zwykle instrukcją `break`, `goto case` lub `goto default`, ale dopuszczalna jest tu każda konstrukcja, która sprawia, że końcowy punkt listy instrukcji staje się nieosiągalny. Przykładem może tu być instrukcja `while` sterowana przez wyrażenie boole'owskie `true`, o której wiadomo, że nigdy nie osiąga swojego punktu końcowego. Podobnie jest z instrukcjami `throw` i `return`, które zawsze przekazują sterowanie w inne miejsce i nigdy nie osiągają swojego punktu końcowego. Z tego powodu prawidłowy jest następujący przykładowy fragment kodu:

```
switch (i) {
case 0:
   while (true) F();
case 1:
   throw new ArgumentException();
case 2:
   return;
}
```

DON BOX Kocham regułę „niewpadania", ale aż po dziś dzień stale zapominam o umieszczeniu instrukcji `break` na końcu klauzuli `default` i ciągle musi mi o tym przypominać kompilator.

Typem rządzącym instrukcji `switch` może być typ `string`. Przykład zastosowania tej możliwości został pokazany poniżej:

```
void DoCommand(string command) {
   switch (command.ToLower()) {
   case "run":
      DoRun();
      break;
   case "save":
      DoSave();
      break;
   case "quit":
      DoQuit();
      break;
   default:
      InvalidCommand(command);
      break;
   }
}
```

Podobnie jak operatory równości łańcuchów znakowych (opisane w punkcie 7.9.7), instrukcja `switch` rozróżnia wielkie i małe litery, dlatego dana sekcja zostanie wykonana tylko wówczas, jeśli łańcuch wyrażenia przełączającego będzie dokładnie pasował do stałej etykiety `case`.

Gdy typem rządzącym instrukcji `switch` jest `string`, wartość `null` jest dozwolona jako stała etykiety `case`.

`Lista-instrukcji` lub *`blok-switch`* może zawierać instrukcje deklaracji (przedstawione w podrozdziale 8.5). Zakresem zmiennej lub stałej lokalnej zadeklarowanej w bloku przełączającym jest ten blok przełączający.

BILL WAGNER Wynika z tego, że każdy blok `switch` otaczają niejawne nawiasy klamrowe.

W ramach bloku przełączającego znaczenie nazw użytych w kontekście wyrażenia musi zawsze być takie samo (o czym więcej w podpunkcie 7.5.2.1).

Lista instrukcji danej sekcji instrukcji przełączającej jest osiągalna, jeśli ta instrukcja `switch` jest osiągalna i spełniony jest przynajmniej jeden z przedstawionych poniżej warunków:

- Wyrażenie przełączające jest wartością niestałą.
- Wyrażenie przełączające jest stałą wartością, która pasuje do etykiety `case` znajdującej się w tej sekcji instrukcji przełączającej.
- Wyrażenie przełączające jest wartością stałą, która nie pasuje do żadnej etykiety `case`, a ta sekcja instrukcji przełączającej zawiera etykietę `default`.
- Do etykiety przełączającej tej sekcji instrukcji przełączającej odwołuje się osiągalna instrukcja `goto case` lub `goto default`.

VLADIMIR RESHETNIKOV Reguła ta nie ma zastosowania, gdy instrukcja `goto case` lub `goto default` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a etykieta instrukcji przełączającej występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

Końcowy punkt instrukcji `switch` jest osiągalny, jeśli spełniony jest co najmniej jeden z wymienionych poniżej warunków:

- Instrukcja `switch` zawiera osiągalną instrukcję `break`, która powoduje opuszczenie instrukcji `switch`.

VLADIMIR RESHETNIKOV Reguła ta nie ma zastosowania, gdy instrukcja `break` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a cel instrukcji `break` występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

- Instrukcja `switch` jest osiągalna, wyrażenie przełączające jest wartością niestałą i nie jest obecna etykieta `default`.
- Instrukcja `switch` jest osiągalna, wyrażenie przełączające jest wartością stałą, która nie pasuje do żadnej etykiety `case`, i nie jest obecna etykieta `default`.

8.8. Instrukcje iteracji

Instrukcje iteracji powodują wielokrotne wykonywanie instrukcji osadzonej:

```
instrukcja-iteracji:
   instrukcja-while
   instrukcja-do
   instrukcja-for
   instrukcja-foreach
```

8.8.1. Instrukcja while

Instrukcja `while` warunkowo wykonuje osadzoną instrukcję zero lub większą liczbę razy:

```
instrukcja-while:
   while ( wyrażenie-boole'owskie ) instrukcja-osadzona
```

Instrukcja `while` jest wykonywana w następujący sposób:

- Przetwarzane jest *`wyrażenie-boole'owskie`* (przedstawione w podrozdziale 7.19).
- Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `true`, sterowanie jest przekazywane do instrukcji osadzonej. Gdy sterowanie osiąga jej punkt końcowy (potencjalnie w wyniku wykonania instrukcji `continue`), przekazywane jest do początku instrukcji `while`.
- Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `false`, sterowanie przekazywane jest do punktu końcowego instrukcji `while`.

W ramach instrukcji osadzonej instrukcji `while` może zostać wykorzystana instrukcja `break` (opisana w punkcie 8.9.1) w celu przekazania sterowania do końcowego punktu instrukcji `while` (i tym samym zakończenia iteracji instrukcji osadzonej), a instrukcja `continue` (opisana w punkcie 8.9.2) może zostać użyta w celu przekazania sterowania do końcowego punktu instrukcji osadzonej (i tym samym przeprowadzenia kolejnej iteracji instrukcji `while`).

Instrukcja osadzona instrukcji `while` jest osiągalna, jeśli osiągalna jest instrukcja `while` i wyrażenie boole'owskie nie ma stałej wartości `false`.

Końcowy punkt instrukcji `while` jest osiągalny, jeśli przynajmniej jeden z wymienionych poniżej warunków jest spełniony:

- Instrukcja `while` zawiera osiągalną instrukcję `break`, która powoduje opuszczenie instrukcji `while`.

> **VLADIMIR RESHETNIKOV** Reguła ta nie ma zastosowania, gdy instrukcja `break` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a cel instrukcji `break` występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

- Instrukcja `while` jest osiągalna i wyrażenie boole'owskie nie ma stałej wartości `true`.

8.8.2. Instrukcja do

Instrukcja `do` warunkowo wykonuje osadzoną instrukcję jeden raz lub większą liczbę razy:

```
instrukcja-do:
   do instrukcja-osadzona while ( wyrażenie-boole'owskie ) ;
```

Instrukcja `do` jest wykonywana w następujący sposób:

- Sterowanie jest przekazywane do instrukcji osadzonej.
- Gdy i jeśli sterowanie osiąga końcowy punkt instrukcji osadzonej (potencjalnie w wyniku wykonania instrukcji `continue`), przetwarzane jest *`wyrażenie-boole'owskie`* (przedstawione w podrozdziale 7.19). Jeśli w wyniku przetwarzania wyrażenia boole'owskiego otrzymana zostaje wartość `true`, sterowanie jest przekazywane do początku instrukcji `do`. W innym przypadku sterowanie przekazywane jest do punktu końcowego instrukcji `do`.

W ramach instrukcji osadzonej instrukcji `do` może zostać wykorzystana instrukcja `break` (opisana w punkcie 8.9.1) do przekazania sterowania do końcowego punktu instrukcji `do` (i tym samym zakończenia iteracji instrukcji osadzonej), a instrukcja `continue` (opisana w punkcie 8.9.2) może zostać użyta do przekazania sterowania do końcowego punktu instrukcji osadzonej.

Instrukcja osadzona instrukcji `do` jest osiągalna, jeśli osiągalna jest instrukcja `do`.

Końcowy punkt instrukcji `do` jest osiągalny, jeśli przynajmniej jeden z wymienionych poniżej warunków jest spełniony:

- Instrukcja `do` zawiera osiągalną instrukcję `break`, która powoduje opuszczenie instrukcji `do`.
- Osiągalny jest końcowy punkt instrukcji osadzonej i wyrażenie boole'owskie nie ma stałej wartości `true`.

> **VLADIMIR RESHETNIKOV** Reguła ta nie ma zastosowania, gdy instrukcja `break` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a cel instrukcji `break` występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

8.8.3. Instrukcja for

Instrukcja `for` powoduje przetworzenie ciągu wyrażeń inicjalizujących, a następnie — dopóki spełniony jest warunek — powtarza wykonanie osadzonej instrukcji i przetwarzanie ciągu wyrażeń iteracji:

```
instrukcja-for:
   for ( inicjalizator-for_opc ; warunek-for_opc ; iterator-for_opc ) instrukcja-
   ↪osadzona

inicjalizator-for:
   deklaracja-zmiennej-lokalnej
   lista-wyrażeń-instrukcji

warunek-for:
   wyrażenie-boole'owskie

iterator-for:
   lista-wyrażeń-instrukcji

lista-wyrażeń-instrukcji:
   wyrażenie-instrukcji
   lista-wyrażeń-instrukcji , wyrażenie-instrukcji
```

Inicjalizator-for, jeśli jest obecny, składa się z elementu *deklaracja-zmiennej-lokalnej* (opisanego w punkcie 8.5.1) lub listy rozdzielonych za pomocą przecinków elementów *wyrażenie-instrukcji* (opisanych w podrozdziale 8.6). Zakres zmiennej lokalnej zadeklarowanej za pomocą elementu *inicjalizator-for* zaczyna się od miejsca występowania *deklaracja-zmiennej-lokalnej* tej zmiennej i rozciąga aż do końca instrukcji osadzonej. Zakres zawiera element *warunek-for* i *iterator-for*.

Warunek-for, jeśli jest obecny, musi być elementem *wyrażenie-boole'owskie* (o którym więcej w podrozdziale 7.19).

Iterator-for, jeśli jest obecny, składa się z listy elementów *wyrażenie-instrukcji* (o którym więcej w podrozdziale 8.6) rozdzielonych za pomocą przecinków.

Instrukcja `for` jest wykonywana w następujący sposób:

- Jeśli obecny jest *inicjalizator-for*, w kolejności, w jakiej je zapisano, są wykonywane inicjalizatory zmiennych lub wyrażenia instrukcji. Krok ten wykonywany jest tylko raz.

- Jeśli obecny jest *warunek-for*, jest on przetwarzany.
- Jeśli nie jest obecny *warunek-for* lub jeśli w wyniku przetwarzania go zostaje uzyskana wartość `true`, sterowanie jest przekazywane do instrukcji osadzonej. Gdy i jeśli sterowanie osiąga końcowy punkt instrukcji osadzonej (potencjalnie w wyniku wykonania instrukcji `continue`), przetwarzane w ciągu są wyrażenia elementu *iterator-for*, jeśli są one obecne, a następnie przeprowadzana jest kolejna iteracja, począwszy od przetworzenia elementu *warunek-for* w poprzednim kroku.
- Jeśli obecny jest *warunek-for* i jeśli w wyniku przetwarzania go zostaje uzyskana wartość `false`, sterowanie jest przekazywane do końcowego punktu instrukcji `for`.

W ramach instrukcji osadzonej instrukcji `for` może zostać użyta instrukcja `break` (opisana w punkcie 8.9.1) w celu przekazania sterowania do końcowego punktu instrukcji `for` (i tym samym zakończenia iteracji instrukcji osadzonej), a instrukcja `continue` (opisana w punkcie 8.9.2) może zostać wykorzystana w celu przekazania sterowania do końcowego punktu instrukcji osadzonej (i tym samym w celu wykonania elementu *iterator-for* i przeprowadzenia kolejnej iteracji instrukcji `for`, począwszy od elementu *warunek-for*).

Instrukcja osadzona instrukcji `for` jest osiągalna, jeśli spełniony jest jeden z przedstawionych poniżej warunków:

- Instrukcja `for` jest osiągalna i nie jest obecny *warunek-for*.
- Instrukcja `for` jest osiągalna i obecny jest *warunek-for*, i nie ma on stałej wartości `false`.

Końcowy punkt instrukcji `for` jest osiągalny, jeśli przynajmniej jeden z wymienionych poniżej warunków jest spełniony:

- Instrukcja `for` zawiera osiągalną instrukcję `break`, która powoduje opuszczenie instrukcji `for`.

> **VLADIMIR RESHETNIKOV** Reguła ta nie ma zastosowania, gdy instrukcja `break` znajduje się wewnątrz bloku `try` lub bloku `catch` instrukcji `try`, która zawiera blok `finally`, a cel instrukcji `break` występuje poza instrukcją `try`, zaś końcowy punkt bloku `finally` jest nieosiągalny.

- Instrukcja `for` jest osiągalna, obecny jest *warunek-for* i nie ma on stałej wartości `true`.

8.8.4. Instrukcja foreach

Instrukcja `foreach` powoduje wyliczenie elementów kolekcji, któremu towarzyszy wykonanie osadzonej instrukcji dla każdego elementu tej kolekcji:

```
instrukcja-foreach:
    foreach ( typ-zmiennej-lokalnej identyfikator in wyrażenie ) instrukcja-
    ↪osadzona
```

Typ i *identyfikator* instrukcji `foreach` stanowią deklarację **zmiennej iteracji** (ang. *iteration variable*) instrukcji. Jeśli jako *typ-zmiennej-lokalnej* podane jest słowo kluczowe `var`, zmienną iteracji nazywa się **niejawnie typowaną zmienną iteracji** (ang. *implicitly typed iteration variable*), a za jej typ przyjmuje się typ elementu instrukcji `foreach`, tak jak zostało to opisane poniżej. Zmienna iteracji odpowiada zmiennej lokalnej tylko do odczytu o zakresie rozciągającym się na instrukcję osadzoną. Podczas wykonywania instrukcji `foreach` zmienna iteracji reprezentuje element kolekcji, dla którego iteracja jest właśnie przeprowadzana. Jeśli instrukcja osadzona próbuje zmodyfikować zmienną iteracji (przez przypisanie lub użycie operatorów ++ oraz--) lub przekazać ją jako parametr `ref` lub `out`, wówczas generowany jest błąd czasu kompilacji.

CHRIS SELLS Ze względu na czytelność kodu powinieneś raczej starać się stosować instrukcje `foreach` zamiast `for`.

VLADIMIR RESHETNIKOV Ze względu na konieczność zapewnienia zgodności wstecznej reguła ta nie jest stosowana, jeśli synonim typu jest wprowadzany za pomocą elementu *dyrektywa-użycia-synonimu* lub jeśli w zakresie znajduje się nieogólny typ o nazwie `var`.

Przetwarzanie czasu kompilacji instrukcji `foreach` zaczyna się od określenia **typu kolekcji** (ang. *collection type*), **typu enumeratora** (ang. *enumeraton type*) oraz **typu elementu** (ang. *element type*) *wyrażenie*. Określanie odbywa się w następujący sposób:

- Jeśli typem `X` elementu *wyrażenie* jest typ tablicowy, wówczas istnieje niejawna konwersja referencyjna z `X` do interfejsu `System.Collections.IEnumerable` (ponieważ typ `System.Array` implementuje ten interfejs). Typem kolekcji jest interfejs `System.Collections.IEnumerable`, typem enumeratora jest interfejs `System.Collections.IEnumerator`, a typem elementu jest typ elementu tablicy `X`.
- W innym przypadku sprawdzane jest, czy typ `X` ma odpowiednią metodę `GetEnumerator`:
  - Przeprowadzane jest odnajdywanie składowej dla typu `X` z identyfikatorem `GetEnumerator` i bez argumentów typu. Jeśli w wyniku tej operacji nie zostaje zwrócone dopasowanie, pojawia się niejednoznaczność lub zostaje zwrócone dopasowanie niebędące grupą metod, należy sprawdzić interfejs enumerowalny, tak jak zostało to opisane poniżej. Zaleca się, aby zgłaszane było ostrzeżenie, jeśli odnajdywanie składowej daje w wyniku coś innego niż grupę metod lub brak dopasowania.

ERIC LIPPERT To oparte na „wzorcu" podejście zostało opracowane wcześniej, niż wprowadzono udostępnienie ogólnego typu `IEnumerable<T>`, dlatego też autorzy kolekcji mogli zapewnić mocniejsze komentarze typów w swoich obiektach enumeratorów.

Implementacje nieogólnego typu `IEnumerable` zawsze kończą się pakowaniem każdej składowej kolekcji liczb całkowitych, ponieważ typem zwracanym przez właściwość `Current` jest `object`. Dostawca kolekcji liczb całkowitych mógłby zapewnić implementacje `GetEnumerator`, `MoveNext` i `Current` niebazujące na interfejsie, tak aby właściwość `Current` zwracała niespakowaną wartość całkowitą.

Oczywiście w świecie ogólnego typu `IEnumerable<T>` cały ten wysiłek staje się niepotrzebny. Ogromna większość iterowanych kolekcji zaimplementuje ten interfejs.

  - Przeprowadzane jest wyznaczanie przeładowania dla otrzymanej grupy metod i pustej listy argumentów. Jeśli operacja ta nie daje w wyniku żadnych możliwych do zastosowania metod, uzyskany wynik jest niejednoznaczny lub otrzymana zostaje pojedyncza najlepsza metoda, lecz jest to metoda statyczna lub niepubliczna, wówczas należy sprawdzić interfejs enumerowalny, tak jak zostało to opisane poniżej. Zaleca się, aby zgłaszane było ostrzeżenie, jeśli wyznaczanie przeładowania daje w wyniku coś innego niż jednoznaczną publiczną metodę instancji lub brak możliwych do zastosowania metod.
  - Jeśli typ zwracany `E` metody `GetEnumerator` nie jest typem klasy, struktury lub interfejsu, generowany jest błąd i nie są podejmowane żadne dalsze kroki.
  - Przeprowadzane jest odnajdywanie składowej dla `E` z identyfikatorem `Current` i bez argumentów typu. Jeśli proces ten nie zwraca dopasowania, wynikiem jest błąd lub cokolwiek innego niż publiczna właściwość instancji umożliwiająca odczyt, generowany jest błąd i nie są podejmowane żadne dalsze kroki.
  - Przeprowadzane jest odnajdywanie składowej dla `E` z identyfikatorem `MoveNext` i bez argumentów typu. Jeśli proces ten nie zwraca dopasowania, wynikiem jest błąd lub cokolwiek innego niż publiczna właściwość instancji umożliwiająca odczyt, generowany jest błąd i nie są podejmowane żadne dalsze kroki.
  - Przeprowadzane jest wyznaczanie przeładowania dla grupy metod z pustą listą argumentów. Jeśli w wyniku tej operacji nie uzyskuje się żadnych możliwych do zastosowania metod, wynik operacji jest niejednoznaczny lub otrzymuje się pojedynczą najlepszą metodę, lecz jest to metoda statyczna lub niepubliczna, lub też zwracanym przez nią typem nie jest `bool`, generowany jest błąd i nie są wykonywane żadne dalsze kroki.
  - Typem kolekcji jest `X`, typem enumeratora jest `E`, a typem elementu jest typ właściwości `Current`.
- W innym przypadku sprawdzany jest interfejs enumerowalny:
  - Jeśli jest dokładnie jeden typ `T`, taki że istnieje niejawna konwersja z `X` na interfejs `System.Collections.Generic.IEnumerable<T>`, wówczas typem kolekcji jest ten interfejs, typem enumeratora jest `System.Collections.Generic.IEnumerator<T>`, a typem elementu jest `T`.
  - W innym przypadku, jeśli istnieje więcej niż jeden taki typ `T`, wówczas generowany jest błąd i nie są podejmowane żadne dalsze kroki.

– W innym przypadku, jeśli istnieje niejawna konwersja z `X` do interfejsu `System.Collections.IEnumerable`, wówczas typem kolekcji jest ten interfejs, typem enumeratora jest `System.Collections.IEnumerator`, a typem elementu jest `object`.

– W innym przypadku generowany jest błąd i nie są podejmowane żadne dalsze kroki.

W wyniku podjęcia przedstawionych tu kroków, jeśli działanie zakończy się sukcesem, otrzymuje się typ kolekcji `C`, typ enumeratora `E` oraz typ elementu `T`. Instrukcja `foreach` o postaci:

```
foreach (V v in x) instrukcja-osadzona
```

jest rozwijana do postaci:

```
{
   E e = ((C)(x)).GetEnumerator();
   try {
      V v;
      while (e.MoveNext()) {
         v = (V)(T)e.Current;
         instrukcja-osadzona
      }
   }
   finally {
      ...    // Zwolnienie zmiennej e
   }
}
```

Zmienna `e` nie jest widoczna ani dostępna dla wyrażenia `x`, instrukcji osadzonej ani jakiegokolwiek innego kodu źródłowego programu. Zmienna `v` jest tylko do odczytu w instrukcji osadzonej. Jeśli nie istnieje jawna konwersja (o której więcej w podrozdziale 6.2) z typu `T` (typu elementu) na typ `V` (typu *`typ-zmiennej-lokalnej`* w instrukcji `foreach`), generowany jest błąd i nie są wykonywane żadne dalsze kroki. Jeśli zmienna `x` ma wartość `null`, w czasie wykonania zgłaszany jest wyjątek `System.NullReferenceException`.

Implementacja może definiować dany element *`instrukcja-foreach`* na różne sposoby — na przykład z powodów związanych z wydajnością — jeśli tylko jego zachowanie jest zgodne z przedstawionym powyżej rozwinięciem.

Ciało bloku `finally` jest konstruowane zgodnie z przedstawionymi poniżej krokami:

- Jeśli istnieje niejawna konwersja z typu `E` na interfejs `System.IDisposable`, wówczas
  – Jeśli `E` jest typem wartościowym niedopuszczającym wartości pustej, klauzula `finally` jest rozwijana do postaci równoznacznej z zapisem:

```
finally {
   ((System.IDisposable)e).Dispose();
}
```

  – W innym przypadku klauzula `finally` jest rozwijana do postaci równoznacznej z zapisem:

```
finally {
   if (e != null) ((System.IDisposable)e).Dispose();
}
```

z wyjątkiem tego, że jeśli `E` jest typem wartościowym lub parametrem typu zrealizowanym do typu wartościowego, wówczas rzutowanie na interfejs `System.IDisposable` nie powoduje wystąpienia operacji pakowania.

- W innym przypadku, jeśli `E` jest typem ostatecznym, klauzula `finally` jest rozwijana do pustego bloku:

```
finally {
}
```

- W innym przypadku klauzula `finally` jest rozwijana do postaci:

```
finally {
   System.IDisposable d = e as System.IDisposable;
   if (d != null) d.Dispose();
}
```

Zmienna lokalna `d` nie jest widoczna ani dostępna dla jakiegokolwiek fragmentu kodu użytkownika. W szczególności nie powoduje konfliktu z jakąkolwiek inną zmienną, której zakres obejmuje blok `finally`.

Kolejność, w jakiej instrukcja `foreach` przechodzi przez elementy tablicy, jest następująca: w przypadku tablic jednowymiarowych elementy są przetwarzane zgodnie z porządkiem rosnących indeksów, zaczynając od indeksu `0`, a kończąc na indeksie `Lenght - 1`. W przypadku tablic wielowymiarowych elementy są przetwarzane w taki sposób, że indeksy wymiaru znajdującego się najbardziej na prawo są zwiększane najpierw, a następnie zwiększane są indeksy następnego wymiaru na lewo i tak dalej, aż do wymiaru znajdującego się najbardziej na lewo.

Wykonanie przedstawionego poniżej przykładowego programu powoduje wyświetlenie na ekranie każdej wartości dwuwymiarowej tablicy z zachowaniem kolejności elementów:

```
using System;
class Test
{
   static void Main() {
      double[,] values = {
         {1.2, 2.3, 3.4, 4.5},
         {5.6, 6.7, 7.8, 8.9}
      };
      foreach (double elementValue in values)
      Console.Write("{0} ", elementValue);
      Console.WriteLine();
   }
}
```

W wyniku skompilowania i uruchomienia programu na ekranie komputera pojawiają się następujące dane:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

W przedstawionym poniżej przykładzie:

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

typ `n` jest wnioskowany jako `int`, bowiem właśnie taki jest typ elementu tablicy liczb.

8.9. Instrukcje skoku

Instrukcje skoku powodują bezwarunkowe przekazanie sterowania:

```
instrukcja-skoku:
   instrukcja-break
   instrukcja-continue
   instrukcja-goto
   instrukcja-return
   instrukcja-throw
```

Miejsce, do którego instrukcja skoku przekazuje sterowanie, nosi nazwę **celu** (ang. *target*) instrukcji skoku.

Gdy instrukcja skoku występuje w bloku, a jej cel znajduje się poza tym blokiem, mówi się, że instrukcja skoku **opuszcza** (ang. *exit*) blok. Mimo że instrukcja skoku może przekazywać sterowanie *poza* blok, nigdy nie jest w stanie przekazać go *do* bloku.

Wykonanie instrukcji skoków jest komplikowane przez obecność ingerujących instrukcji `try`. W przypadku braku takich instrukcji `try` instrukcja skoku bezwarunkowo przekazuje sterowanie z instrukcji skoku do jej celu. Jeśli są obecne ingerujące instrukcje `try`, wykonanie jest bardziej skomplikowane. Jeśli instrukcja skoku opuszcza jeden lub większą liczbę bloków `try` z powiązanymi blokami `finally`, sterowanie jest początkowo przekazywane do bloku `finally` najbardziej wewnętrznej instrukcji `try`. Gdy i jeśli sterowanie osiąga końcowy punkt bloku `finally`, jest ono przekazywane do bloku `finally` kolejnej obejmującej instrukcji `try`. Proces ten jest powtarzany aż do momentu wykonania bloków `finally` wszystkich ingerujących instrukcji `try`.

W przedstawionym poniżej przykładzie:

```
using System;
class Test
{
   static void Main() {
      while (true) {
         try {
            try {
               Console.WriteLine("Przed instrukcją break");
               break;
            }
            finally {
               Console.WriteLine("Najbardziej wewnętrzny blok finally");
            }
         }
         finally {
            Console.WriteLine("Najbardziej zewnętrzny blok finally ");
```

```
            }
        }
        Console.WriteLine("Po instrukcji break");
    }
}
```

bloki `finally` związane z dwoma instrukcjami `try` są wykonywane przed przekazaniem sterowania do celu instrukcji skoku.

W wyniku skompilowania i uruchomienia przedstawionego powyżej programu na ekranie komputera pojawiają się następujące dane:

```
Przed instrukcją break
Najbardziej wewnętrzny blok finally
Najbardziej zewnętrzny blok finally
Po instrukcji break
```

8.9.1. Instrukcja break

Instrukcja `break` powoduje opuszczenie najbliższej obejmującej instrukcji `switch`, `while`, `do`, `for` lub `foreach`.

```
instrukcja-break:
    break ;
```

Celem instrukcji `break` jest końcowy punkt najbliższej obejmującej instrukcji `switch`, `while`, `do`, `for` lub `foreach`. Jeśli instrukcji `break` nie obejmuje instrukcja `switch`, `while`, `do`, `for` lub `foreach`, generowany jest błąd czasu kompilacji.

Gdy wiele instrukcji `switch`, `while`, `do`, `for` lub `foreach` jest zagnieżdżonych w sobie, instrukcja `break` jest stosowana jedynie do najbardziej wewnętrznej instrukcji obejmującej. Aby przekazać sterowanie przez wiele zagnieżdżonych poziomów, trzeba skorzystać z instrukcji `goto` (opisanej w punkcie 8.9.3).

Instrukcja `break` nie może spowodować opuszczenia bloku `finally` (o czym więcej w podrozdziale 8.10). Gdy instrukcja `break` występuje w ramach bloku `finally`, cel tej instrukcji musi znajdować się w tym samym bloku `finally`. W innym przypadku generowany jest błąd czasu kompilacji.

Instrukcja `break` jest wykonywana w następujący sposób:

- Jeśli instrukcja `break` powoduje opuszczenie jednego lub większej liczby bloków `try` z powiązanymi blokami `finally`, sterowanie jest początkowo przekazywane do bloku `finally` najbardziej wewnętrznej instrukcji `try`. Gdy i jeśli sterowanie osiąga końcowy punkt bloku `finally`, jest ono przekazywane do bloku `finally` kolejnej obejmującej instrukcji `try`. Proces ten jest powtarzany aż do momentu wykonania bloków `finally` wszystkich ingerujących instrukcji `try`.
- Sterowanie jest przekazywane do celu instrukcji `break`.

Ponieważ instrukcja `break` bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

8.9.2. Instrukcja continue

Instrukcja `continue` powoduje rozpoczęcie nowej iteracji najbliższej obejmującej instrukcji `while`, `do`, `for` lub `foreach`:

```
instrukcja-continue:
   continue ;
```

Celem instrukcji `continue` jest końcowy punkt najbliższej instrukcji osadzonej najbliższej obejmującej instrukcji `while`, `do`, `for` lub `foreach`. Jeśli instrukcji `continue` nie obejmuje instrukcja `while`, `do`, `for` lub `foreach`, generowany jest błąd czasu kompilacji.

Gdy wiele instrukcji `while`, `do`, `for` lub `foreach` jest zagnieżdżonych w sobie, instrukcja `continue` jest stosowana jedynie do najbardziej wewnętrznej instrukcji obejmującej. Aby przekazać sterowanie przez wiele zagnieżdżonych poziomów, trzeba skorzystać z instrukcji `goto` (opisanej w punkcie 8.9.3).

Instrukcja `continue` nie może spowodować opuszczenia bloku `finally` (o którym więcej w podrozdziale 8.10). Gdy instrukcja `continue` występuje w ramach bloku `finally`, cel tej instrukcji musi znajdować się w tym samym bloku `finally`. W innym przypadku generowany jest błąd czasu kompilacji.

Instrukcja `continue` jest wykonywana w następujący sposób:

- Jeśli instrukcja `continue` powoduje opuszczenie jednego lub większej liczby bloków `try` z powiązanymi blokami `finally`, sterowanie jest początkowo przekazywane do bloku `finally` najbardziej wewnętrznej instrukcji `try`. Gdy i jeśli sterowanie osiąga końcowy punkt bloku `finally`, jest ono przekazywane do bloku `finally` kolejnej obejmującej instrukcji `try`. Proces ten jest powtarzany aż do momentu wykonania bloków `finally` wszystkich ingerujących instrukcji `try`.
- Sterowanie jest przekazywane do celu instrukcji `continue`.

Ponieważ instrukcja `continue` bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

8.9.3. Instrukcja goto

Instrukcja `goto` powoduje przekazanie kontroli do instrukcji oznaczonej za pomocą etykiety:

```
instrukcja-goto:
   goto identyfikator ;
   goto case wyrażenie-stałe ;
   goto default ;
```

Celem instrukcji `goto` *`identyfikator`* instrukcja oznaczona za pomocą danej etykiety. Jeśli etykieta o podanej nazwie nie istnieje w bieżącej funkcji składowej lub jeśli instrukcja `goto` nie znajduje się

w zakresie tej etykiety, generowany jest błąd czasu kompilacji. Reguła ta dopuszcza używanie instrukcji `goto` w celu przekazywania kontroli *poza* zagnieżdżony zakres, nie da się jednak zrobić tego *do* zagnieżdżonego zakresu. W przedstawionym poniżej przykładzie:

```
using System;
class Test
{
   static void Main(string[] args) {
      string[,] table = {
         {"Czerwony", "Niebieski", "Zielony"},
         {"Poniedziałek", "Środa", "Piątek"}
      };
      foreach (string str in args) {
         int row, colm;
         for (row = 0; row <= 1; ++row)
            for (colm = 0; colm <= 2; ++colm)
               if (str == table[row,colm])
                  goto done;
         Console.WriteLine("{0} nieznaleziony", str);
         continue;
  done:
         Console.WriteLine("Znaleziony {0} na pozycji [{1}][{2}]", str, row, colm);
      }
   }
}
```

instrukcja `goto` jest używana do przekazywania sterowania poza zagnieżdżony zakres.

Celem instrukcji `goto case` jest lista instrukcji znajdująca się w bezpośrednio obejmującej instrukcji `switch` (o której więcej w punkcie 8.7.2), która zawiera etykietę `case` z odpowiednią wartością stałą. Jeśli żadna instrukcja `switch` nie obejmuje instrukcji `goto case`, jeśli *`wyrażenie-stałe`* nie jest niejawnie konwertowalne (o czym więcej w podrozdziale 6.1) na typ rządzący najbliższą obejmującą instrukcją `switch` lub jeśli najbliższa obejmująca instrukcja `switch` nie zawiera etykiety `case` z odpowiednią wartością stałą, generowany jest błąd czasu kompilacji.

Celem instrukcji `goto default` jest lista instrukcji znajdująca się w bezpośrednio obejmującej instrukcji `switch` (o której więcej w punkcie 8.7.2), która zawiera etykietę `default` z odpowiednią wartością stałą. Jeśli żadna instrukcja `switch` nie obejmuje instrukcji `goto case` lub jeśli najbliższa obejmująca instrukcja `switch` nie zawiera etykiety `case default`, generowany jest błąd czasu kompilacji.

Instrukcja `goto` nie może spowodować opuszczenia bloku `finally` (opisanego w podrozdziale 8.10). Gdy instrukcja `goto` występuje w ramach bloku `finally`, cel tej instrukcji musi znajdować się w obrębie tego samego bloku `finally`, a w innym przypadku generowany jest błąd czasu kompilacji.

BILL WAGNER Reguły te sprawiają, że instrukcja `goto` staje się w C# czymś nieco tylko mniejszym niż czysta złośliwość, ja jednak nadal widzę dla niej dobre zastosowanie.

Instrukcja goto jest wykonywana w następujący sposób:

- Jeśli instrukcja goto powoduje opuszczenie jednego bloku lub większej liczby bloków try z powiązanymi blokami finally, sterowanie jest początkowo przekazywane do bloku finally najbardziej wewnętrznej instrukcji try. Gdy i jeśli sterowanie osiąga końcowy punkt bloku finally, jest ono przekazywane do bloku finally kolejnej obejmującej instrukcji try. Proces ten jest powtarzany aż do momentu wykonania bloków finally wszystkich ingerujących instrukcji try.
- Sterowanie jest przekazywane do celu instrukcji goto.

Ponieważ instrukcja goto bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

KRZYSZTOF CWALINA Ogólnie rzecz biorąc, uwielbiam prace Dijkstry, ale nie zgadzam się z opinią, że instrukcja goto jest szkodliwa. W rzeczywistości może ona bowiem w wielu przypadkach znacznie uprościć kod. Podobnie jak każda inna możliwość, może ona oczywiście być nadużywana, ale szkodzi tak naprawdę zły programista, nie zaś sama instrukcja jako taka. Mimo to instrukcja goto przydaje mi się tak rzadko, że bez kłopotu mógłbym sobie bez niej poradzić.

8.9.4. Instrukcja return

Instrukcja return powoduje zwrócenie sterowania do miejsca wywołania funkcji składowej, w której występuje ta instrukcja:

```
instrukcja-return:
    return wyrażenie_opc ;
```

Instrukcja return bez elementu *wyrażenie* może być używana wyłącznie w funkcji składowej, która nie oblicza wartości, a więc w metodzie z typem zwracanym void, akcesorze set właściwości lub indeksatora, akcesorach add lub remove zdarzenia, konstruktorze instancji, konstruktorze statycznym lub destruktorze.

Instrukcja return z elementem *wyrażenie* może być używana wyłącznie w funkcji składowej, która oblicza wartość, a więc w metodzie z typem zwracanym innym niż void, akcesorze get właściwości lub indeksatora lub operatorze definiowanym przez użytkownika. Musi istnieć niejawna konwersja (opisana w podrozdziale 6.1) z typu elementu *wyrażenie* na typ zwracany przez funkcję składową zawierającą tę instrukcję.

VLADIMIR RESHETNIKOV Jeśli instrukcja return występuje w ramach funkcji anonimowej, zamiast przedstawionych tutaj stosowane są reguły zaprezentowane w podrozdziale 6.5.

Gdy instrukcja `return` pojawia się w bloku `finally` (o którym więcej w podrozdziale 8.10), generowany jest błąd czasu kompilacji.

Instrukcja `return` jest wykonywana w następujący sposób:

- Jeśli w instrukcji `return` podano *`wyrażenie`*, jest ono przetwarzane i otrzymana w wyniku tego wartość jest konwertowana na zwracany typ zawierającej funkcji składowej za pomocą konwersji niejawnej. Wynik tej konwersji staje się wartością zwracaną do miejsca wywołania.
- Jeśli instrukcja `return` jest obejmowana przez jeden blok lub większą liczbę bloków `try` z powiązanymi blokami `finally`, sterowanie jest początkowo przekazywane do bloku `finally` najbardziej wewnętrznej instrukcji `try`. Gdy i jeśli sterowanie osiąga końcowy punkt bloku `finally`, jest ono przekazywane do bloku `finally` kolejnej obejmującej instrukcji `try`. Proces ten jest powtarzany aż do momentu wykonania bloków `finally` wszystkich ingerujących instrukcji `try`.
- Sterowanie jest zwracane do miejsca wywołania zawierającej je funkcji składowej.

Ponieważ instrukcja `return` bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

8.9.5. Instrukcja throw

Instrukcja `throw` powoduje zgłoszenie wyjątku:

`instrukcja-throw:`
`throw` *`wyrażenie`*$_{opc}$ `;`

Instrukcja `throw` z elementem *`wyrażenie`* powoduje zgłoszenie wartości otrzymanej w wyniku przetwarzania tego elementu *`wyrażenie`*. *`Wyrażenie`* musi oznaczać wartość typu klasy `System.`
`↪Exception`, typu klasy, która dziedziczy po `System.Exception`, lub typu parametru typu, którego skuteczną klasą bazową jest `System.Exception` (bądź też jej podklasa). Jeśli w wyniku przetwarzania elementu *`wyrażenie`* otrzymana zostaje wartość `null`, zamiast wspomnianego wcześniej zgłaszany jest wyjątek `System.NullReferenceException`.

Instrukcja `throw` bez elementu *`wyrażenie`* może być używana jedynie w blokach `catch`. W tego rodzaju przypadkach instrukcja `throw` ponownie zgłasza wyjątek, który jest w danej chwili obsługiwany przez blok `catch`.

VLADIMIR RESHETNIKOV Instrukcja `throw` bez elementu *`wyrażenie`* nie może być stosowana w bloku `finally` ani w funkcji anonimowej, która jest zagnieżdżona wewnątrz najbliższego obejmującego bloku `catch`, tak jak zostało to pokazane w przedstawionym poniżej fragmencie kodu:

```
delegate void F();
class Program {
   static void Main() {
      try {
      }
      catch {
         F f = () => { throw; };    // Błąd CS0156
         try {
         }
         finally {
            throw;                  // Błąd CS0724
         }
      }
   }
}
```

Ponieważ instrukcja `throw` bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

Gdy zgłaszany jest wyjątek, sterowanie przekazywane jest do pierwszej klauzuli `catch` w obejmującej instrukcji `try`, która może go obsłużyć. Proces, który się odbywa od momentu zgłoszenia wyjątku do momentu przekazania sterowania do odpowiedniego kodu obsługi, znany jest pod nazwą **propagacji wyjątku** (ang. *exception propagation*). Propagacja wyjątku polega na powtarzaniu przeprowadzania przedstawionych poniżej kroków aż do momentu znalezienia klauzuli `catch`, która do niego pasuje. W zaprezentowanym opisie **punkt zgłoszenia** (ang. *throw point*) jest początkowo miejscem, w którym zgłaszany jest wyjątek.

- W bieżącej funkcji składowej sprawdzana jest każda instrukcja `try` obejmująca punkt zgłoszenia. Dla każdej instrukcji `S`, począwszy od najbardziej wewnętrznej instrukcji `try`, a skończywszy na najbardziej zewnętrznej instrukcji `try`, przeprowadzane są następujące działania:
  - Jeśli blok `try` instrukcji `S` obejmuje punkt zgłoszenia i jeśli `S` ma jedną lub większą liczbę klauzul `catch`, klauzule te są sprawdzane w kolejności występowania w celu odnalezienia odpowiedniego kodu obsługi wyjątku. Za dopasowanie uważana jest pierwsza klauzula `catch`, w której określono typ wyjątku lub typ bazowy typu wyjątku. Ogólna klauzula `catch` (o której więcej w podrozdziale 8.10) jest uznawana za dopasowanie dla każdego typu wyjątku. Jeśli znaleziona zostaje pasująca klauzula `catch`, propagacja wyjątku jest kończona przez przekazanie sterowania do bloku tej klauzuli.
  - W innym przypadku, jeśli blok `try` lub blok `catch` instrukcji `S` obejmuje punkt zgłoszenia i jeśli `S` ma blok `finally`, sterowanie jest przekazywane do tego bloku `finally`. Jeśli blok `finally` zgłasza kolejny wyjątek, przetwarzanie bieżącego wyjątku jest przerywane. W innym przypadku, gdy sterowanie osiąga końcowy punkt bloku `finally`, przetwarzanie bieżącego wyjątku jest kontynuowane.

- Jeśli kod obsługi wyjątku nie został znaleziony w bieżącym wywołaniu funkcji składowej, wywołanie to jest przerywane. Kroki przedstawione powyżej są powtarzane dla miejsca wywołania funkcji składowej z punktem zgłoszenia odpowiadającym instrukcji, w której wywołana została ta funkcja składowa.
- Jeśli przetwarzanie wyjątku przerywa wszystkie wywołania funkcji składowych w bieżącym wątku, wskazując tym samym, że w wątku tym nie ma odpowiedniego kodu obsługi dla tego wyjątku, wówczas przerywany jest sam wątek. Wpływ takiego przerwania na otoczenie zależy od implementacji.

BILL WAGNER Z zaprezentowanej tu procedury wynika, że powinieneś przeprowadzać pewne podstawowe czynności czyszczące w znajdujących się na samym szczycie hierarchii metodach wszystkich swoich wątków. W innym przypadku zachowanie Twojej aplikacji w zderzeniu z błędami będzie nieprzewidywalne.

8.10. Instrukcja try

Instrukcja `try` zapewnia mechanizm wychwytywania wyjątków, które zdarzają się podczas wykonywania bloku. Ponadto instrukcja ta umożliwia określenie bloku kodu, który jest wykonywany zawsze, gdy sterowanie opuszcza instrukcję `try`:

```
instrukcja-try:
   try blok klauzule-catch
   try blok klauzula-finally
   try blok klauzule-catch klauzula-finally

klauzule-catch:
   sprecyzowane-klauzule-catch ogólna-klauzula-catch_opc
   sprecyzowane-klauzule-catch_opc ogólna-klauzula-catch

sprecyzowane-klauzule-catch:
   sprecyzowana-klauzula-catch
   sprecyzowane-klauzule-catch sprecyzowana-klauzula-catch

sprecyzowana-klauzula-catch:
   catch ( typ-klasy identyfikator_opc ) blok

ogólna-klauzula-catch:
   catch blok

klauzula-finally:
   finally blok
```

Możliwe są trzy postacie instrukcji `try`:

- Blok `try`, po którym występuje jeden lub większa liczba bloków `catch`.
- Blok `try`, po którym występuje blok `finally`.
- Blok `try`, po którym występuje jeden lub większa liczba bloków `catch`, po nich zaś blok `finally`.

Gdy w klauzuli `catch` podany jest *`typ-klasy`*, musi nim być `System.Exception`, typ dziedziczący po `System.Exception` lub typ parametru typu, którego skuteczną klasą bazową jest `System.Exception` (bądź też jej podklasa).

Gdy w klauzuli `catch` podane są zarówno *`typ-klasy`*, jak i identyfikator, deklarowana jest **zmienna wyjątku** (ang. *exception variable*) o podanej nazwie. Zmienna wyjątku odpowiada zmiennej lokalnej o zakresie rozciągającym się na blok `catch`. W czasie wykonywania tego bloku `catch` zmienna wyjątku reprezentuje wyjątek, który jest właśnie obsługiwany. Na potrzeby sprawdzania niewątpliwego ustalenia przyjmuje się, że zmienna wyjątku jest niewątpliwie ustalona w całym swoim zakresie.

Jeśli klauzula `catch` nie zawiera nazwy zmiennej wyjątku, nie jest możliwe uzyskanie dostępu do obiektu wyjątku w tym bloku `catch`.

Klauzula `catch`, w której nie jest określony ani typ wyjątku, ani nazwa zmiennej wyjątku, jest nazywana ogólną klauzulą `catch`. Instrukcja `try` może mieć tylko jedną ogólną klauzulę `catch`; jeśli jest ona obecna, musi być ostatnią klauzulą `catch`.

Niektóre języki programowania mogą obsługiwać wyjątki, których nie da się przedstawić jako obiektów klas pochodnych wobec `System.Exception`, choć tego rodzaju wyjątki nigdy nie powinny być generowane przez kod C#. Do wychwytywania takich wyjątków może zostać zastosowana ogólna klauzula `catch`. Z tego powodu różni się ona pod względem składni od takiej, w której podano typ `System.Exception`, tym, że ogólna klauzula `catch` może wychwytywać wyjątki zgłaszane w innych językach programowania.

ERIC LIPPERT W bieżącej implementacji C# i CLR firmy Microsoft domyślnie jest tak, że zgłoszony obiekt, który nie jest typu pochodnego wobec `Exception`, jest konwertowany na obiekt `RuntimeWrappedException`. Dzięki temu instrukcja `catch (Exception e)` jest w stanie wychwycić wszystkie wyjątki.

Jeśli chcesz zmienić to zachowanie i używać tradycyjnej semantyki C# 1.0, w której niebędące klasy `Exception` obiekty zgłaszane przez inne języki nie są przechwytywane w ten sposób, wówczas powinieneś zastosować następujący atrybut podzespołu:

```
[assembly:System.Runtime.CompilerServices.RuntimeCompatibility
   (WrapNonExceptionThrows = false)]
```

W celu odnalezienia kodu obsługi wyjątku klauzule `catch` są sprawdzane w kolejności leksykalnej. Jeśli w klauzuli `catch` podano typ, który jest taki sam jak typ podany we wcześniejszej klauzuli `catch` tej samej instrukcji `try` lub jest wobec niego pochodny, generowany jest błąd czasu kompilacji. Gdyby nie istniało to ograniczenie, możliwe byłoby tworzenie nieosiągalnych klauzul `catch`.

W ramach bloku `catch` można korzystać z instrukcji `throw` (o której więcej w punkcie 8.9.5) bez elementu *`wyrażenie`* w celu ponownego zgłoszenia wyjątku, który został przechwycony przez ten blok `catch`. Przypisania do zmiennej wyjątku nie zmieniają wyjątku, który jest ponownie zgłaszany.

W przedstawionym poniżej przykładzie:

```
using System;
class Test
{
   static void F() {
      try {
         G();
      }
      catch (Exception e) {
         Console.WriteLine("Wyjątek w F: " + e.Message);
         e = new Exception("F");
         throw;    // Ponowne zgłoszenie
      }
   }
   static void G() {
      throw new Exception("G");
   }
   static void Main() {
      try {
         F();
      }
      catch (Exception e) {
         Console.WriteLine("Wyjątek w Main: " + e.Message);
      }
   }
}
```

metoda `F` przechwytuje wyjątek, wyświetla na ekranie pewien komunikat diagnostyczny, modyfikuje zmienną wyjątku, a następnie zgłasza go ponownie. Zgłoszony ponownie wyjątek jest oryginalnym wyjątkiem, dlatego na ekranie komputera pojawiają się następujące dane:

```
Wyjątek w F: G
Wyjątek w Main: G
```

Gdyby w pierwszym bloku `catch` zgłosić wyjątek `e`, zamiast ponownie zgłaszać bieżący wyjątek, na ekranie komputera pojawiłyby się następujące dane:

```
Wyjątek w F: G
Wyjątek w Main: F
```

Próba przekazania sterowania poza blok `finally` za pomocą instrukcji `break`, `continue` lub `goto` stanowi błąd czasu kompilacji. Gdy w bloku `finally` występuje instrukcja `break`, `continue` lub `goto`, jej cel musi znajdować się w obrębie tego samego bloku `finally`; w innym przypadku generowany jest błąd czasu kompilacji.

Wystąpienie w bloku `finally` instrukcji `return` jest błędem czasu kompilacji.

Instrukcja `try` jest wykonywana w następujący sposób:

- Sterowanie jest przekazywane do bloku `try`.
- Gdy i jeśli sterowanie osiągnie końcowy punkt bloku `try`:
  - Jeśli instrukcja `try` ma blok `finally`, wykonywany jest ten blok `finally`.
  - Sterowanie jest przekazywane do końcowego punktu instrukcji `try`.
- Jeśli wyjątek jest propagowany do instrukcji `try` w czasie wykonywania bloku `try`:
  - Klauzule `catch`, jeśli jakieś występują, są sprawdzane w kolejności występowania w celu odnalezienia odpowiedniego kodu obsługi wyjątku. Pierwsza klauzula `catch`, w której określono typ wyjątku lub typ bazowy typu wyjątku jest uważana za dopasowanie. Ogólna klauzula `catch` jest uznawana za dopasowanie dla każdego typu wyjątku. Jeśli znaleziona zostaje pasująca klauzula `catch`:
    - Jeśli w pasującej klauzuli `catch` zadeklarowano zmienną wyjątku, przypisywany jest do niej obiekt wyjątku.
    - Sterowanie jest przekazywane do pasującego bloku `catch`.
    - Gdy i jeśli sterowanie osiąga końcowy punkt bloku `catch`:
      - Jeśli instrukcja `try` ma blok `finally`, wykonywany jest ten blok `finally`.
      - Sterowanie jest przekazywane do końcowego punktu instrukcji `try`.
    - Jeśli wyjątek jest propagowany do instrukcji `try` podczas wykonywania bloku `catch`:
      - Jeśli instrukcja `try` ma blok `finally`, wykonywany jest ten blok `finally`.
      - Sterowanie jest przekazywane do kolejnej obejmującej instrukcji `try`.
  - Jeśli instrukcja `try` nie ma klauzul `catch` lub jeśli żadna z klauzul `catch` nie pasuje do wyjątku:
    - Jeśli instrukcja `try` ma blok `finally`, wykonywany jest ten blok `finally`.
    - Sterowanie jest przekazywane do kolejnej obejmującej instrukcji `try`.

ERIC LIPPERT Jeśli stos wywołania zawiera kod chroniony za pomocą bloków `try-catch` napisanych w innych językach programowania (takich jak Visual Basic), środowisko uruchomieniowe może zastosować „filtr wyjątków", aby sprawdzić, czy dany blok `catch` jest odpowiedni dla zgłaszanego wyjątku. W wyniku tego kod użytkownika może zostać wykonany po zgłoszeniu wyjątku, lecz przed wykonaniem powiązanego bloku `finally`. Jeśli Twój

kod obsługi wyjątków zależy od stanu globalnego, który jest spójny dzięki temu, że blok `finally` jest uruchamiany przed jakimkolwiek innym kodem użytkownika, wówczas powinieneś w odpowiedni sposób sprawdzić, czy Twój kod wychwytuje wyjątek, zanim środowisko uruchomieniowe stosuje definiowane przez użytkownika filtry wyjątków, które mogą znajdować się na stosie.

Instrukcje bloku `finally` są wykonywane zawsze, gdy sterowanie opuszcza instrukcję `try`. Jest tak niezależnie od tego, czy przepływ sterowania jest wynikiem normalnego wykonania, wynikiem wykonania instrukcji `break`, `continue`, `goto` lub `return`, czy też wynikiem propagacji wyjątku poza tę instrukcję `try`.

Jeśli wyjątek jest zgłaszany podczas wykonywania bloku `finally` i nie jest przechwytywany w obrębie tego bloku, jest on propagowany do następnej obejmującej instrukcji `try`. Jeśli kolejny wyjątek występuje w czasie propagacji, wyjątek ten zostaje utracony. Proces propagacji wyjątku został dokładniej przedstawiony w opisie instrukcji `throw` (znajdującym się w punkcie 8.9.5).

BILL WAGNER Zachowanie to sprawia, że bardzo ważne jest pisanie klauzul `finally` w sposób bezpieczny w celu uniknięcia ryzyka zgłoszenia drugiego wyjątku.

Blok `try` instrukcji `try` jest osiągalny, jeśli osiągalna jest ta instrukcja `try`.

Blok `catch` instrukcji `try` jest osiągalny, jeśli osiągalna jest ta instrukcja `try`.

Blok `finally` instrukcji `try` jest osiągalny, jeśli osiągalna jest ta instrukcja `try`.

Końcowy punkt instrukcji `try` jest osiągalny, jeśli spełnione są obydwa podane poniżej warunki:

- Osiągalny jest punkt końcowy bloku `try` lub osiągalny jest punkt końcowy przynajmniej jednego bloku `catch`.
- Jeśli obecny jest blok `finally`, osiągalny jest punkt końcowy tego bloku `finally`.

8.11. Instrukcje checked i unchecked

Instrukcje `checked` i `unchecked` są używane do sterowania **kontekstem kontroli przekroczenia zakresu** (ang. *overflow checking context*) w przypadku konwersji i operacji arytmetycznych na typach całkowitych:

```
instrukcja-checked:
   checked blok

instrukcja-unchecked:
   unchecked blok
```

Instrukcja checked powoduje, że wszystkie wyrażenia znajdujące się w elemencie *blok* są przetwarzane w kontekście kontrolowanym, zaś instrukcja unchecked powoduje, że wszystkie wyrażenia znajdujące się w elemencie *blok* są przetwarzane w kontekście niekontrolowanym.

Instrukcje checked i unchecked dokładnie odpowiadają operatorom checked i unchecked (opisanym w punkcie 7.5.12), z wyjątkiem tego, że operują one na blokach wyrażeń.

8.12. Instrukcja lock

Instrukcja lock umożliwia nałożenie blokady wzajemnie wykluczającej na dany obiekt, wykonanie instrukcji, a następnie zdjęcie tej blokady:

```
instrukcja-lock:
   lock ( wyrażenie ) instrukcja-osadzona
```

Wyrażenie instrukcji lock musi oznaczać wartość typu, o którym wiadomo, że jest to *typ-refe*
↪*rencyjny*. Dla elementu *wyrażenie* instrukcji lock nigdy nie jest przeprowadzana niejawna konwersja pakowania (o której więcej w punkcie 6.1.7). Z tego powodu pojawia się błąd czasu kompilacji, jeśli *wyrażenie* oznacza wartość *typ-wartościowy*.

Instrukcja lock o postaci:

```
lock (x) ...
```

gdzie x jest wyrażeniem *typ-referencyjny*, jest dokładnie równoważna z kodem:

```
System.Threading.Monitor.Enter(x);
try {
   ...
}
finally {
   System.Threading.Monitor.Exit(x);
}
```

z wyjątkiem tego, że x jest przetwarzane tylko raz.

ERIC LIPPERT Nieszczęśliwą konsekwencją wyboru tego sposobu generowania kodu jest to, że w przypadku kompilacji przeznaczonych do debugowania pomiędzy operacją Enter i blokiem try może występować „pusta" instrukcja IL (te bezużyteczne w innych sytuacjach instrukcje są często generowane po to, aby narzędzie diagnostyczne miało do dyspozycji wyraźnie określone miejsce, w którym można w razie potrzeby umieścić punkt przerwania podczas procesu diagnozowania).

Jeśli uruchamiasz kompilację przeznaczoną do diagnozowania i nastąpi przełączenie wątku podczas wykonywania tej operacji pustej, może się zdarzyć, że wyjątek przerwania wątku zostaje zgłoszony w innym wątku. Przerwanie wątku może wystąpić przed wejściem do

bloku `try`, dlatego blok `finally` nigdy nie jest wykonywany i blokada nigdy nie zostaje zwolniona. Zachowanie takie może prowadzić do bardzo trudnych do zdiagnozowania sytuacji zakleszczenia.

Problem ten zostanie być może wyeliminowany w przyszłych wersjach C# za pomocą różnych postaci operacji `Enter`, które będzie się dało bezpiecznie wywołać w ramach bloku `try`.

BILL WAGNER Użycie instrukcji `lock()` wiąże się również z dodatkowymi przeprowadzanymi przez kompilator kontrolami tego, czy blokowanie nie dotyczy elementu typu wartościowego.

Podczas utrzymywania blokady wzajemnie wykluczającej możliwe jest również nałożenie i zwolnienie blokady przez kod przetwarzany w ramach tego samego wątku wykonania. W przeciwieństwie do niego kod wykonywany w innych wątkach nie może nałożyć blokady, zanim bieżąca blokada nie zostanie zwolniona.

Nie zaleca się blokowania obiektów `System.Type` w celu zsynchronizowania dostępu do danych statycznych. Inny fragment kodu może zablokować ten sam typ, co może spowodować zakleszczenie. Lepszym sposobem jest synchronizowanie dostępu do danych statycznych przez blokowanie prywatnego obiektu statycznego. Przykład takiej operacji został przedstawiony poniżej:

```
class Cache
{
   private static object synchronizationObject = new object();

   public static void Add(object x) {
      lock (Cache.synchronizationObject) {
         ...
      }
   }
   public static void Remove(object x) {
      lock (Cache.synchronizationObject) {
         ...
      }
   }
}
```

JOSEPH ALBAHARI Dobrą taktyką przy pisaniu bibliotek przeznaczonych do publicznego wykorzystania jest zabezpieczanie funkcji statycznych przed ubocznymi efektami działania wielu różnych wątków (co można zwykle osiągnąć przez implementowanie blokady *w obrębie* funkcji, tak jak zostało to przedstawione na powyższym przykładzie). Znacznie trudniej jest użytkownikom kodu stosować blokadę *wokół* wywołania Twoich statycznych metod lub właściwości (a często jest to wręcz niemożliwe), ponieważ nie będą oni wiedzieli, z jakich innych miejsc wywoływane są Twoje funkcje.

8.13. Instrukcja using

Instrukcja `using` umożliwia uzyskanie jednego lub większej liczby zasobów, wykonanie instrukcji, a następnie zwolnienie tych zasobów:

```
instrukcja-using:
   using ( pozyskanie-zasobu ) instrukcja-osadzona

pozyskanie-zasobu:
   deklaracja-zmiennej-lokalnej
   wyrażenie
```

Zasób (ang. *resource*) jest klasą lub strukturą implementującą interfejs `System.IDisposable`, który zawiera pojedynczą bezparametrową metodę o nazwie `Dispose`. Kod, w którym używany jest zasób, może wywołać metodę `Dispose`, aby wskazać, że zasób nie jest już potrzebny. Jeśli metoda `Dispose` nie zostaje wywołana, wówczas automatyczne zwolnienie zasobu następuje w końcu jako efekt odzyskiwania pamięci.

> **JOSEPH ALBAHARI** Wywołanie metody `Dispose` nie wpływa w żaden sposób na działanie mechanizmu odzyskiwania pamięci: obiekt zaczyna nadawać się do odzyskania, gdy (i tylko gdy) nie odwołują się do niego żadne inne obiekty. Również odzyskiwanie pamięci nie wpływa na zwalnianie zasobu: mechanizm odzyskiwania nie wywoła metody `Dispose`, dopóki nie napiszesz finalizatora (destruktora), w którym jawnie znajdzie się to wywołanie.
>
> Dwie czynności przeprowadzane najczęściej w ramach metody `Dispose` to zwalnianie niezarządzanych zasobów i wywołanie metod `Dispose` na rzecz innych obiektów, do których odwołuje się obiekt bieżący, lub takich, w których jest „posiadaniu". Zwolnienie niezarządzanych zasobów możliwe jest również z poziomu samego finalizatora, ale operacja taka oznacza oczekiwanie nieokreśloną ilość czasu na uruchomienie mechanizmu odzyskiwania pamięci. I właśnie dlatego istnieje `IDisposable`.

Jeśli postacią elementu *`pozyskanie-zasobu`* jest *`deklaracja-zmiennej-lokalnej`*, wówczas typem elementu *`deklaracja-zmiennej-lokalnej`* musi być interfejs `System.IDisposable` lub typ, który może być jawnie konwertowany do `System.IDisposable`. Jeśli *`pozyskanie-zasobu`* ma postać *`wyrażenie`*, wówczas wyrażenie to musi być typu `System.IDisposable` lub typu, który można niejawnie konwertować do `System.IDisposable`.

Zmienne lokalne zadeklarowane w elemencie *`pozyskanie-zasobu`* są tylko do odczytu i muszą zawierać inicjalizator. Jeśli instrukcja osadzona próbuje zmodyfikować te zmienne lokalne (za pomocą operatorów `++` oraz `--`), pobrać ich adresy lub przekazać je jako parametry `ref` lub `out`, generowany jest błąd czasu kompilacji.

Instrukcja `using` jest przekształcana przez podział na trzy części: pozyskanie, użycie i zwolnienie. Użycie zasobu jest niejawnie ujęte w instrukcję `try` zawierającą klauzulę `finally`. Ta klauzula `finally` zwalnia używany zasób. Jeśli pozyskany zostaje zasób `null`, wówczas nie jest przeprowadzane wywołanie `Dispose` i nie jest zgłaszany wyjątek.

Instrukcja `using` o postaci:

```
using (ResourceType resource = expression) statement
```

odpowiada jednemu z dwóch możliwych rozwinięć. Gdy `ResourceType` jest typem wartościowym, rozwinięcie instrukcji jest następujące:

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      ((IDisposable)resource).Dispose();
   }
}
```

W innym przypadku, gdy `ResourceType` jest typem referencyjnym, rozwinięcie ma postać:

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      if (resource != null) ((IDisposable)resource).Dispose();
   }
}
```

W każdym z tych rozwinięć zmienna `resource` w instrukcji osadzonej jest tylko do odczytu.

Implementacja może definiować dany element *`instrukcja-using`* na różne sposoby — co może być na przykład spowodowane względami wydajnościowymi — jeśli tylko jego zachowanie jest zgodne z przedstawionymi powyżej rozwinięciami.

Instrukcja `using` o postaci:

```
using (expression) statement
```

ma te same dwa rozwinięcia, ale w tym przypadku `ResourceType` jest niejawnie typem czasu kompilacji elementu `expression`, a zmienna `resource` jest niedostępna w instrukcji osadzonej i jest dla niej niewidoczna.

Gdy *`pozyskanie-zasobu`* przyjmuje postać *`deklaracja-zmiennej-lokalnej`*, możliwe jest pozyskanie wielu zasobów danego typu. Instrukcja `using` o postaci:

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

jest dokładnie równoważna z następującą sekwencją zagnieżdżonych instrukcji `using`:

```
using (ResourceType r1 = e1)
   using (ResourceType r2 = e2)
      ...
         using (ResourceType rN = eN)
            statement
```

Wykonanie przedstawionego poniżej przykładowego kodu powoduje utworzenie pliku o nazwie *log.txt* i zapisanie w nim dwóch linii tekstu. Program otwiera następnie ten sam plik do odczytu i kopiuje znajdujące się w nim linie tekstu na ekran:

```
using System;
using System.IO;
class Test
{
   static void Main() {
      using (TextWriter w = File.CreateText("log.txt")) {
         w.WriteLine("To jest pierwsza linia");
         w.WriteLine("To jest druga linia ");
      }
      using (TextReader r = File.OpenText("log.txt")) {
         string s;
         while ((s = r.ReadLine()) != null) {
            Console.WriteLine(s);
         }
      }
   }
}
```

Dzięki temu, że klasy `TextWriter` i `TextReader` implementują interfejs `IDisposable`, w przykładzie można zastosować instrukcje `using` w celu zapewnienia, że używany plik jest prawidłowo zamykany po wykonaniu operacji zapisu i odczytu.

> **CHRIS SELLS** Niemal zawsze powinieneś owijać za pomocą bloków `using` wszelkie pozyskiwane zasoby, które implementują interfejs `IDisposable` (chyba że przetrzymujesz ten obiekt pomiędzy wywołaniami metod). Mimo że mechanizm odzyskiwania pamięci platformy .NET świetnie radzi sobie ze zwalnianiem zasobów pamięciowych, wszystkimi innymi zasobami powinieneś zarządzać samodzielnie. Również kompilator doskonale radzi sobie z generowaniem dla Ciebie odpowiedniego kodu, ale tylko wówczas, gdy umieszczasz alokacje zasobów w blokach `using`.

8.14. Instrukcja yield

Instrukcja `yield` jest używana w bloku iteratora (o którym więcej w podrozdziale 8.2) w celu zwrócenia wartości do obiektu enumeratora (o którym więcej w punkcie 10.4.4) lub obiektu enumerowalnego (o którym więcej w punkcie 10.14.5) iteratora lub w celu zasygnalizowania końca iteracji:

```
instrukcja-yield:
   yield return wyrażenie ;
   yield break ;
```

Zwróć uwagę na to, że `yield` nie jest słowem zarezerwowanym; ma ono specjalnie znaczenie wyłącznie wtedy, gdy jest używane bezpośrednio przed słowem kluczowym `return` lub `break`. W innych kontekstach `yield` może być wykorzystywane jako identyfikator.

Miejsce, w którym może wystąpić instrukcja `yield`, wyznaczane jest przez kilka wymienionych poniżej ograniczeń:

- Gdy instrukcja `yield` (w którejkolwiek z postaci) występuje poza elementem *`ciało-metody`*, *`ciało-operatora`* lub *`ciało-akcesora`*, pojawia się błąd czasu kompilacji.
- Gdy instrukcja `yield` (w którejkolwiek z postaci) występuje wewnątrz funkcji anonimowej, pojawia się błąd czasu kompilacji.
- Gdy instrukcja `yield` (w którejkolwiek z postaci) występuje w klauzuli `finally` instrukcji `try`, pojawia się błąd czasu kompilacji.
- Gdy instrukcja `yield return` występuje gdziekolwiek w instrukcji `try`, która zawiera klauzule `catch`, pojawia się błąd czasu kompilacji.

Przedstawiony poniżej fragment kodu zawiera kilka przykładów prawidłowych i nieprawidłowych zastosowań instrukcji `yield`:

```
delegate IEnumerable<int> D();
IEnumerator<int> GetEnumerator() {
   try {
      yield return 1;    // W porządku
      yield break;       // W porządku
   }
   finally {
      yield return 2;    // Błąd, instrukcja yield w klauzuli finally
      yield break;       // Błąd, instrukcja yield w klauzuli finally
   }
   try {
      yield return 3;    // Błąd, instrukcja yield return w bloku try...catch
      yield break;       // W porządku
   }
   catch {
      yield return 4;    // Błąd, instrukcja yield return w bloku try...catch
      yield break;       // W porządku
   }
   D d = delegate {
      yield return 5;    // Błąd, instrukcja yield w funkcji anonimowej
   };
}
int MyMethod() {
   yield return 1;       // Błąd, niewłaściwy typ zwracany dla bloku iteratora
}
```

Musi istnieć niejawna konwersja (opisana w podrozdziale 6.1) z typu wyrażenia znajdującego się w instrukcji `yield return` na typ zwracany (o czym więcej w punkcie 10.14.3) przez iterator.

Instrukcja `yield return` jest wykonywana w następujący sposób:

- Wyrażenie podane w instrukcji jest przetwarzane, niejawnie konwertowane na typ zwracany i przypisywane do właściwości `Current` obiektu enumeratora.
- Zawieszane jest wykonanie bloku iteratora. Jeśli instrukcja `yield return` znajduje się w obrębie jednego bloku lub większej liczby bloków `try`, powiązane bloki `finally` *nie* są wykonywane w tym momencie.
- Metoda `MoveNext` obiektu enumeratora zwraca do miejsca wywołania wartość `true`, wskazując, że obiektowi enumeratora udało się przejść do kolejnego elementu.

Następne wywołanie metody `MoveNext` obiektu enumeratora powoduje ponowne podjęcie wykonania bloku iteratora w tym miejscu, w którym zostało ono ostatnio zawieszone.

Instrukcja `yield break` jest wykonywana w następujący sposób:

- Jeśli instrukcja `yield break` powoduje opuszczenie jednego lub większej liczby bloków `try` z powiązanymi blokami `finally`, sterowanie jest początkowo przekazywane do bloku `finally` najbardziej wewnętrznej instrukcji `try`. Gdy i jeśli sterowanie osiąga końcowy punkt bloku `finally`, jest ono przekazywane do bloku `finally` kolejnej obejmującej instrukcji `try`. Proces ten jest powtarzany aż do momentu wykonania bloków `finally` wszystkich ingerujących instrukcji `try`.
- Sterowanie jest przekazywane do miejsca wywołania bloku iteratora. Jest nim metoda `MoveNext` lub metoda `Dispose` obiektu enumeratora.

Ponieważ instrukcja `yield break` bezwarunkowo przekazuje sterowanie w jakieś inne miejsce, końcowy punkt tej instrukcji nigdy nie jest osiągalny.

CHRIS SELLS Czasami zdarza mi się zapomnieć, że instrukcja `yield return` nie działa jak instrukcja `return` pod tym względem, że kod znajdujący się po `yield return` może zostać wykonany. Na przykład w przedstawionym poniżej fragmencie programu kod występujący po pierwszym `return` nigdy nie może zostać wykonany:

```
int F() {
   return 1;
   return 2;    // Nigdy nie może być wykonane
}
```

W przeciwieństwie do tego kod po pierwszej instrukcji `yield return` widocznej w poniższym fragmencie może być wykonany:

```
IEnumerable<int> F() {
   yield return 1;
   yield return 2;    // Może być wykonane
}
```

Często boleśnie odczuwam to w przypadku instrukcji `if`:

```
IEnumerable<int> F() {
   if(...) { yield return 1; }    // Chcę, żeby było to jedyną wartością zwracaną
   yield return 2;                // Ups!
}
```

W takich przypadkach pomocne może się okazać zapamiętanie, że instrukcja `yield return` nie jest „końcowa", tak jak instrukcja `return`.

BILL WAGNER W podrozdziale tym przedstawione zostało działanie instrukcji `yield` w teorii. W praktyce instrukcje `yield` będą powodowały tworzenie zagnieżdżonych klas, które implementują wzorzec iteratora.