Jon Skeet

Wydanie IV

Helion

Tytuł oryginału: C# in Depth, Fourth Edition

Tłumaczenie: Tomasz Walczak

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-6029-7

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cshop4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

11 Łączenie danych z użyciem krotek

Zawartość rozdziału

- Używanie krotek do łączenia danych
- Składnia krotek — literały i typy
- Przekształcanie krotek
- W jaki sposób krotki są reprezentowane w środowisku CLR?
- Alternatywy dla krotek i wskazówki dotyczące używania krotek

W C# 3 technologia LINQ zrewolucjonizowała sposób pisania kodu do obsługi kolekcji danych. Między innymi umożliwiła zapisywanie wielu operacji w kategoriach tego, co należy zrobić z każdym pojedynczym elementem: jak przekształcić go z jednej reprezentacji na inną, jak przefiltrować elementy w wynikach lub jak posortować kolekcję na podstawie określonego aspektu każdego elementu. Mimo to w technologii LINQ nie pojawiło się wiele nowych narzędzi do pracy z danymi innymi niż kolekcje.

Typy anonimowe umożliwiają jeden sposób łączenia danych, ale z poważnym ograniczeniem, ponieważ są przydatne tylko w ramach bloku kodu. Nie można np. zadeklarować, że metoda zwraca wartość typu anonimowego. Wynika to z tego, że nie można podać nazwy typu zwracanej wartości.

W C# 7 dodano obsługę krotek, aby ułatwić łączenie danych, a także rozdzielanie typu złożonego na poszczególne komponenty. Jeśli mówisz sobie teraz, że w C# już dostępne są krotki w postaci typów `System.Tuple`, do pewnego stopnia masz rację. Te typy

istnieją już w platformie, ale nie są obsługiwane w języku. Aby jeszcze bardziej skomplikować sytuację, w C# 7 te typy krotek nie są używane na potrzeby krotek obsługiwanych przez język. Zamiast tego wykorzystywane są nowe typy z rodziny `System.ValueTuple`, opisane w podrozdziale 11.4. W punkcie 11.5.1 znajdziesz ich porównanie z typami `System.Tuple`.

11.1. Wprowadzenie do krotek

Krotki umożliwiają utworzenie jednej złożonej wartości na podstawie wielu odrębnych wartości. Są narzędziem do szybkiego łączenia danych bez dodatkowej hermetyzacji. Przydają się, gdy wartości są powiązane ze sobą, ale programista nie chce kłopotać się tworzeniem nowego typu. W C# 7 wprowadzono nową składnię, aby uprościć pracę z krotkami.

Załóżmy np., że dostępna jest sekwencja liczb całkowitych i chcesz znaleźć wartość minimalną oraz maksymalną w jednym przebiegu. Wydaje się, że taki kod powinno dać się umieścić w jednej metodzie. Jaki jednak ma być typ zwracanej przez nią wartości? Mógłbyś zwracać wartość minimalną i użyć parametru `out` dla wartości maksymalnej. Możesz też zastosować dwa parametry `out`. Oba te rozwiązania wydają się jednak dość niezgrabne. Mógłbyś też utworzyć odrębny typ nazwany, jednak oznacza to dużo pracy jak na tylko jeden przykład. Jeszcze inna możliwość to zwrócenie wartości typu `Tuple<int, int>` z użyciem wprowadzonej w .NET 4 klasy `Tuple<,>`, nie da się wtedy łatwo ustalić, która wartość to minimum, a która maksimum. Ponadto trzeba utworzyć obiekt tylko po to, aby zwrócić dwie wartości. Możesz też użyć krotek z C# 7 i zadeklarować następującą metodę:

```
static (int min, int max) MinMax(IEnumerable<int> source)
```

Następnie możesz ją wywoływać tak:

```
int[] values = { 2, 7, 3, -5, 1, 0, 10 };
var extremes = MinMax(values);
Console.WriteLine(extremes.min);
Console.WriteLine(extremes.max);
```

Wywołanie metody obliczającej minimum i maksimum oraz zwracającej te wartości jako krotki.

Wyświetlanie wartości minimalnej (-5).

Wyświetlanie wartości maksymalnej (10).

Dalej przedstawionych jest kilka implementacji metody `MinMax`, jednak ten przykład powinien dać Ci wystarczające wyobrażenie o omawianym mechanizmie, abyś chciał się zapoznać z wszystkimi dość szczegółowymi opisami z tego rozdziału. Jak na mechanizm, który wydaje się prosty, o krotkach można napisać całkiem dużo. Wszystkie te informacje są powiązane ze sobą, dlatego trudno przedstawiać je w jakimś logicznym porządku. Jeśli w trakcie lektury zaczniesz zadawać sobie pytanie: „A co z…?”, zachęcam do tego, byś zapamiętał je do końca rozdziału. Nie ma tu nic skomplikowanego, jednak tekst jest długi — przede wszystkim dlatego, że chcę zaprezentować kompletny opis. Mam nadzieję, że do momentu zakończenia rozdziału znajdziesz odpowiedzi na wszystkie swoje pytania[1].

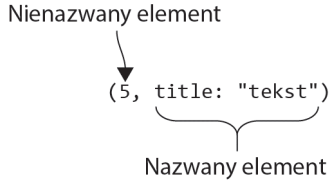
[1] Jeśli tak nie będzie, powinieneś oczywiście poprosić o dodatkowe informacje na forum Author Online lub w serwisie Stack Overflow.

11.2. Literały i typy krotek

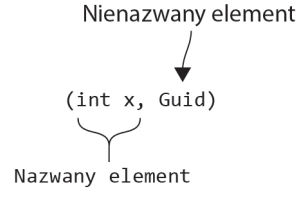
Krotki możesz traktować jak wprowadzenie niektórych typów do środowiska CLR i jako lukier składniowy ułatwiający używanie tych typów — zarówno ich podawanie (na potrzeby zmiennych itd.), jak i tworzenie wartości. Zaczynam od objaśnienia wszystkich kwestii z perspektywy języka C#, bez przejmowania się powiązaniami ze środowiskiem CLR. Później cofam się, aby wyjaśnić wszystko to, co kompilator robi na zapleczu.

11.2.1. Składnia

W C# 7 wprowadzono dwa nowe elementy składniowe: literały krotek i typy krotek. Wyglądają one podobnie. W obu przypadkach należy podać rozdzieloną przecinkami sekwencję dwóch lub więcej elementów w nawiasie. W *literale krotki* każdy element ma wartość i opcjonalną nazwę. W *typie krotki* każdy element ma typ i opcjonalną nazwę. Na rysunku 11.1 pokazany jest przykładowy literał krotki. Na rysunku 11.2 widoczny jest przykładowy typ krotki. Na każdym rysunku pokazany jest jeden element nazwany i jeden nienazwany.



Rysunek 11.1. Literał krotki z elementami o wartościach 5 i "tekst". Nazwa drugiego elementu to title



Rysunek 11.2. Typ krotki z elementami o typach int i Guid. Nazwa pierwszego elementu to x

W praktyce znacznie częściej nazwy podaje się dla wszystkich elementów lub w ogóle się ich nie używa. Używane mogą być np. typy krotek `(int, int)` lub `(int x, int y, int z)` oraz literały krotek `(x: 1, y: 2)` lub `(1, 2, 3)`. Nie jest to jednak wymagane. Używanie nazw nie łączy elementów w dodatkowy sposób. Należy jednak pamiętać o dwóch ograniczeniach dotyczących nazw:

- Nazwy muszą być unikatowe w ramach typu lub literału. Literał krotki `(x: 1, x: 2)` jest niedozwolony i nie ma sensu.
- Nazwy w postaci `ItemN`, gdzie `N` to liczba całkowita, są dozwolone tylko wtedy, gdy wartość `N` pasuje do pozycji elementu w literale lub typie (pierwsza pozycja to `1`). Dlatego krotka `(Item1: 0, Item2: 0)` jest dozwolona, natomiast `(Item2: 0, Item1: 0)` jest nieprawidłowa. W następnym podrozdziale zobaczysz, z czego to wynika.

Typy krotek służą do podawania typów w tych samych miejscach, gdzie używane są inne nazwy typów: w deklaracjach zmiennych, jako typy wartości zwracanych przez metody itd. Literały krotek są używane jak dowolne inne wyrażenia określające wartość. Takie literały jedynie łączą inne elementy w wartość w postaci krotki.

Wartościami elementów w literałach krotek mogą być dowolne wartości inne niż wskaźnik. W większości przykładów z tego rozdziału dla wygody używane są stałe (głównie liczby całkowite i łańcuchy znaków), jednak często jako wartości elementów stosuje się też zmienne. Podobnie typami elementów w krotce mogą być dowolne typy niewskaźnikowe: tablice, parametry określające typ, a nawet inne typy krotek.

Teraz, kiedy wiesz już, jak wyglądają typy krotek, możesz zrozumieć typ zwracany przez metodę `MinMax` — `(int min, int max)`:

- jest to typ krotki o dwóch elementach,
- pierwszy element jest typu `int` i ma nazwę `min`,
- drugi element jest typu `int` i ma nazwę `max`.

Wiesz już też, jak utworzyć krotkę za pomocą literału krotki. Możesz więc napisać kompletną implementację metody. Ilustruje ją listing 11.1.

Listing 11.1. Reprezentowanie wartości minimalnej i maksymalnej z sekwencji za pomocą krotki

```
static (int min, int max) MinMax(
  IEnumerable<int> source)
{
  using (var iterator = source.GetEnumerator())
  {
    if (!iterator.MoveNext())
    {
      throw new InvalidOperationException(
        "Nie można znaleźć minimum i maksimum pustej sekwencji");
    }
    int min = iterator.Current;
    int max = iterator.Current;
    while (iterator.MoveNext())
    {
      min = Math.Min(min, iterator.Current);
      max = Math.Max(max, iterator.Current);
    }
    return (min, max);
  }
}
```

Typem zwracanej wartości jest krotka z nazwanymi elementami.

Uniemożliwia używanie pustych sekwencji.

Używanie zwykłych wartości typu int do śledzenia minimum i maksimum.

Aktualizowanie zmiennych za pomocą nowego minimum lub maksimum.

Tworzenie krotki z użyciem minimum i maksimum.

Jedyne fragmenty listingu 11.1, gdzie używane są nowe funkcje, to objaśniony już typ zwracanej wartości oraz instrukcja `return`, gdzie używany jest literał krotki:

```
return (min, max)
```

Do tej pory nie pisałem nic na temat typu literału krotki. Stwierdziłem jedynie, że takie literały służą do tworzenia wartości krotek, jednak na razie celowo nie doprecyzowuję tej kwestii. Warto zauważyć, że użyty tu literał krotki na razie nie obejmuje żadnych nazw elementów (przynajmniej nie w wersji C# 7.0). Nazwy `min` i `max` określają wartości elementów na podstawie zmiennych lokalnych metody.

Dobre nazwy elementów krotek pasują do dobrych nazw zmiennych

Czy to przypadek, że nazwy zmiennych użyte w literale pasują do nazw używanych w typie wartości zwracanej przez metodę? Dla kompilatora jest to całkowity przypadek. Nie byłoby dla niego żadnym problemem, gdybyś zadeklarował, że metoda zwraca wartość `(waffle: int, iceCream: int)`.

Jednak dla człowieka pasujące nazwy nie są zbiegiem okoliczności. Informują one, że wartości w zwracanej krotce oznaczają to samo co w metodzie. Jeśli zauważysz, że używasz zupełnie innych nazw, sprawdź, czy w kodzie nie występuje błąd lub czy nie można wybrać lepszych nazw.

Skoro jesteśmy już przy definiowaniu nazw, warto zdefiniować *arność* typu lub literału krotki. Arność oznacza liczbę elementów. Na przykład `(int, long)` ma arność 1, a `("a", "b", "c")` ma arność 3. Same typy elementów nie mają znaczenia przy określaniu arności.

UWAGA. Nie jest to nowa terminologia. Arność jest uwzględniana także w typach generycznych, gdzie oznacza liczbę parametrów określających typ. Typ `List<T>` ma arność 1, natomiast typ `Dictionary<TKey, TValue>` ma arność 2.

Wskazówka dotycząca tego, że dobre nazwy elementów pasują do dobrych nazw zmiennych, sugeruje, jaki aspekt literałów krotek usprawniono w C# 7.1.

11.2.2. Wnioskowanie nazw elementów w literałach krotek (C# 7.1)

W C# 7.0 nazwy elementów krotek trzeba jawnie podawać w kodzie. Często prowadzi to do powstawania kodu, który wygląda na nadmiarowy. Nazwy podane w literale krotki odpowiadają nazwom właściwości lub zmiennych lokalnych określających wartości. W najprostszej postaci kod może wyglądać tak:

```
var result = (min: min, max: max);
```

Wnioskowanie jest stosowane nie tylko wtedy, gdy w kodzie używane są proste zmienne. Krotki często są inicjalizowane na podstawie właściwości. Jest to częste zwłaszcza w technologii LINQ w połączeniu z projekcjami.

W C# 7.1 nazwy elementów krotek zostają wywnioskowane, gdy wartość jest pobierana ze zmiennej lub właściwości. Odbywa się to w taki sam sposób jak wnioskowanie nazw w typach anonimowych. Aby zobaczyć, jak użyteczna jest ta technika, rozważ trzy sposoby zapisu zapytania w technologii LINQ to Objects. To zapytanie złącza dwie kolekcje, aby pobrać nazwiska, stanowiska i działy pracowników. Najpierw podane jest tradycyjne zapytanie w LINQ z użyciem typów anonimowych:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select new { emp.Name, emp.Title, DepartmentName = dept.Name };
```

Dalej pokazane są krotki z jawnie podanymi nazwami elementów:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select (name: emp.Name, title: emp.Title, departmentName: dept.Name);
```

Na końcu używane są wywnioskowane nazwy elementów z C# 7.1:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select (emp.Name, emp.Title, DepartmentName: dept.Name);
```

Powoduje to zmianę wielkości liter w nazwach elementów krotki w porównaniu z poprzednim przykładem, jednak pozwala osiągnąć cel, jakim jest utworzenie krotek o przydatnych nazwach z użyciem zwięzłego kodu.

Choć omawiany mechanizm zaprezentowałem tu w kontekście zapytania w technologii LINQ, można go stosować wszędzie tam, gdzie używane są literały krotek. Na przykład gdy dana jest lista elementów, możesz utworzyć krotkę z liczbą wartości, minimum i maksimum, wykorzystując wnioskowanie nazw elementów dla liczby wartości:

```
ist<int> list = new List<int> { 5, 1, -6, 2 };
var tuple = (list.Count, Min: list.Min(), Max: list.Max());
Console.WriteLine(tuple.Count);
Console.WriteLine(tuple.Min);
Console.WriteLine(tuple.Max);
```

Warto zauważyć, że nadal trzeba podać nazwy elementów `Min` i `Max`, ponieważ te wartości są pobierane za pomocą wywołań metod. Wywołania metod nie pozwalają wywnioskować nazw ani elementów krotek, ani właściwości typów anonimowych.

Niewielką wadą jest to, że jeśli można wywnioskować dwie takie same nazwy, żadna z nich nie zostaje użyta. Jeżeli występuje kolizja między wywnioskowaną nazwą a nazwą podaną jawnie, priorytetowo traktowana jest ta ostatnia, a drugi element pozostaje nienazwany. Wiesz już, jak podawać typy i literały krotek. Co jednak można z nimi zrobić?

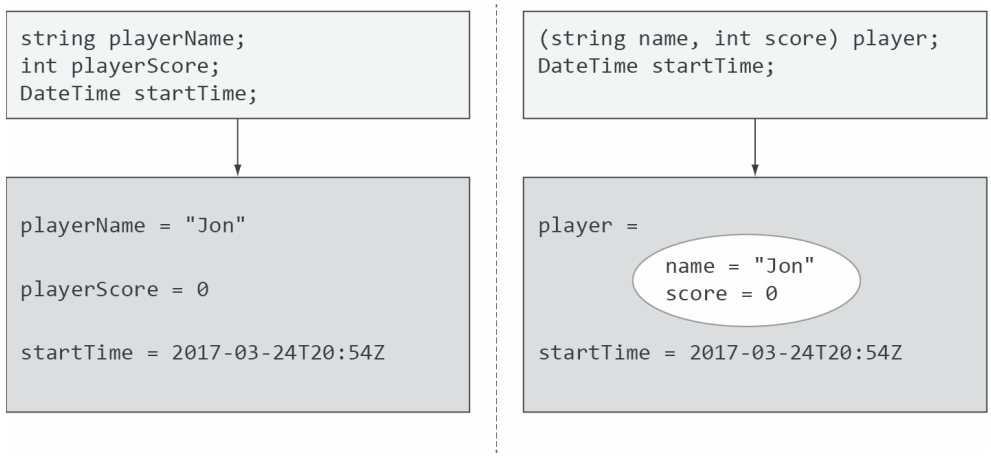
11.2.3. Krotki jako zbiory zmiennych

Następne zdanie może Cię zszokować, więc koniecznie się przygotuj: typy krotek są typami bezpośrednimi z publicznymi polami do odczytu i zapisu. To chyba nie może być prawda?! Zwykle zdecydowanie odradzam używanie modyfikowalnych typów bezpośrednich i zawsze sugeruję, że pola powinny być prywatne. Przeważnie obstaję przy tych rekomendacjach, jednak krotki są nieco odmienne.

Większość typów nie jest tylko danymi. Typy nadają danym znaczenie. Czasem sprawdzają też poprawność danych. Zdarza się, że różne elementy danych łączy określona relacja. Zazwyczaj dostępne są operacje, które mają sens tylko dzięki znaczeniu nadanemu danym.

Krotek nie dotyczy żaden z tych punktów. Działają one jak zbiory zmiennych. Jeśli masz dwie zmienne, możesz modyfikować je niezależnie od siebie. Nie są one z natury połączone ze sobą i nie jest wymuszana relacja między nimi. Krotki pozwalają zrobić dokładnie to samo, ale zapewniają coś jeszcze — umożliwiają przekazywanie całego zbioru zmiennych w jednej wartości. Jest to ważne przede wszystkim w metodach, ponieważ można w nich zwracać tylko jedną wartość.

Na rysunku 11.3 pokazane jest to w formie graficznej. Po lewej stronie widoczny jest kod i model umysłowy związany z deklarowaniem trzech niezależnych zmiennych



Rysunek 11.3. Trzy odrębne zmienne po lewej stronie; dwie zmienne (w tym jedna krotka) po prawej stronie

lokalnych. Po prawej stronie znajduje się podobny kod, jednak dwie z tych zmiennych są zapisane w krotce (w owalu). Po prawej stronie nazwisko (`name`) i liczba punktów (`score`) są połączone w krotce o nazwie `player` (gracz). Gdy chcesz traktować je jako odrębne zmienne, nadal możesz to robić (np. wyświetlając wartość `player.score`), ale możesz też używać ich jak grupy (np. przypisując nową wartość do krotki `player`).

Gdy już zaczniesz myśleć o krotce jak o zbiorze zmiennych, wiele rzeczy nabierze więcej sensu. Jak jednak używać tych zmiennych? Zobaczyłeś już, że gdy w krotce dostępne są elementy nazwane, możesz używać ich za pomocą nazw. Co jednak zrobić, jeśli element nie ma nazwy?

DOSTĘP DO ELEMENTÓW ZA POMOCĄ NAZW I POZYCJI

Może przypominasz sobie, że obowiązuje ograniczenie dla nazw elementów w postaci `ItemN`, gdzie `N` to liczba. Wynika ono z tego, że każdą zmienną w krotce można wskazać zarówno za pomocą pozycji, jak i przy użyciu nazwy. Każdemu elementowi odpowiada jedna zmienna, przy czym można ją wskazać na dwa sposoby. Najłatwiej pokazać to na przykładzie. Ilustruje to listing 11.2.

Listing 11.2. Odczyt i zapis elementów krotek za pomocą nazw i pozycji

```
var tuple = (x: 5, 10);
Console.WriteLine(tuple.x);
Console.WriteLine(tuple.Item1);
Console.WriteLine(tuple.Item2);

tuple.x = 100;
Console.WriteLine(tuple.Item1);
```

Wyświetlanie pierwszego elementu z użyciem nazwy i pozycji.

Drugi element nie ma nazwy; można używać tylko pozycji.

Modyfikowanie pierwszego elementu z użyciem nazwy.

Wyświetlanie pierwszego elementu z użyciem pozycji (wyświetla 100).

Na tym etapie zapewne rozumiesz już, dlaczego zapis `(Item1: 10, 20)` jest poprawny, ale już `(Item2: 10, 20)` jest niedozwolony. W pierwszym przypadku dodawana jest nadmiarowa nazwa elementu, natomiast w drugim powstaje niejednoznaczność co do

tego, czy `Item2` oznacza pierwszy element (podawany za pomocą nazwy), czy drugi element (podawany przy użyciu pozycji). Można by stwierdzić, że zapis `(Item5: 10, 20)` powinien być dopuszczalny, ponieważ krotka obejmuje tylko dwa elementy. Jest to jedna z sytuacji, w których kod wprawdzie nie powoduje technicznie wieloznaczności, ale z pewnością byłby mało zrozumiały, dlatego i tak jest zabroniony.

Teraz wiesz już, że możesz zmodyfikować wartość krotki po jej utworzeniu. Dlatego możesz zmienić metodę `MinMax` i użyć jednej zmiennej lokalnej w postaci krotki do zapisywania dotychczasowych wyników, zamiast oddzielać zmienne `min` i `max`. Nowe rozwiązanie pokazane jest na listingu 11.3.

Listing 11.3. Używanie krotki zamiast dwóch zmiennych w metodzie MinMax

```
static (int min, int max) MinMax(IEnumerable<int> source)
{
  using (var iterator = source.GetEnumerator())
  {
    if (!iterator.MoveNext())
    {
      throw new InvalidOperationException(
        "Nie można znaleźć minimum i maksimum dla pustej sekwencji");
          }
    var result = (min: iterator.Current,
                  max: iterator.Current);
    while (iterator.MoveNext())
    {
      result.min = Math.Min(result.min, iterator.Current);
      result.max = Math.Max(result.max, iterator.Current);
    }
    return result;
  }
}
```

Tworzenie krotki, w której pierwszy element jest ustawiany jako maksimum i minimum.

Modyfikowanie osobno każdego pola krotki.

Bezpośrednie zwracanie krotki.

Listing 11.3 jest bardzo, bardzo podobny do listingu 11.1, jeśli chodzi o działanie kodu. Jedyna różnica to połączenie dwóch z czterech zmiennych lokalnych. Zamiast zmiennych `source`, `iterator`, `min` i `max` używane są teraz zmienne `source`, `iterator` i `result`, przy czym `result` obejmuje elementy `min` i `max`. Ilość zajmowanej pamięci i wydajność są takie same. Różny jest tylko zapis kodu. Czy wersja z krotkami jest lepsza? Ocena jest tu subiektywna, możesz jednak samodzielnie podjąć decyzję. Wybór zapisu to wyłącznie szczegół implementacji.

TRAKTOWANIE KROTEK JAK POJEDYNCZYCH WARTOŚCI

Skoro już jesteśmy przy różnych implementacjach metody, warto rozważyć jeszcze inny zapis. Możesz użyć kodu, który najpierw przypisuje nową wartość do elementu `result.min`, a następnie do elementu `result.max`:

```
result.min = Math.Min(result.min, iterator.Current);
result.max = Math.Max(result.max, iterator.Current);
```

Jeśli przypiszesz wynik bezpośrednio do zmiennej `result`, możesz zastąpić cały zbiór w jednej operacji. Ilustruje to listing 11.4.

Listing 11.4. Ponowne przypisywanie wartości krotki result w jednej instrukcji w metodzie MinMax

```
static (int min, int max) MinMax(IEnumerable<int> source)
{
  using (var iterator = source.GetEnumerator())
  {
    if (!iterator.MoveNext())
    {
      throw new InvalidOperationException(
        "Nie można znaleźć minimum i maksimum w pustej sekwencji");
    }
    var result = (min: iterator.Current, max: iterator.Current);
    while (iterator.MoveNext())
    {
      result = (Math.Min(result.min, iterator.Current),
                Math.Max(result.max, iterator.Current));
    }
    return result;
  }
}
```

Przypisywanie nowej wartości do całej zmiennej result.

Także tu różnica między obiema implementacjami nie jest duża. Na listingu 11.3 oba elementy krotki są aktualizowane niezależnie i sprawdzane są wcześniejsze wartości poszczególnych elementów. Ciekawszym przykładem jest metoda zwracająca ciąg Fibonacciego[2] jako wartość typu `IEnumerable<int>`. C# pomaga napisać taką metodę, ponieważ udostępnia iteratory z instrukcją `yield`, co jednak może okazać się skomplikowane. Na listingu 11.5 pokazano w pełni poprawną implementację z C# 6.

Listing 11.5. Generowanie ciągu Fibonacciego bez krotek

```
static IEnumerable<int> Fibonacci()
{
  int current = 0;
  int next = 1;
  while (true)
  {
    yield return current;
    int nextNext = current + next;
    current = next;
    next = nextNext;
  }
}
```

W trakcie iterowania należy śledzić bieżący i następny element sekwencji. W każdej iteracji kod przechodzi od pary reprezentującej elementy „bieżący i następny" do pary „następny i jeszcze następny". Potrzebna jest do tego zmienna tymczasowa. Nie wystarczy bezpośrednio przypisać nowych wartości do zmiennych `current` i `next` jedna po drugiej, ponieważ po pierwszym przypisaniu utracone zostaną informacje potrzebne w drugim przypisaniu.

[2] Dwa pierwsze elementy tego ciągu to 0 i 1. Następnie każdy element ciągu jest sumą dwóch wcześniejszych.

Krotki umożliwiają wykonanie jednego przypisania modyfikującego oba elementy. Zmienna tymczasowa nadal jest używana w kodzie pośrednim, jednak wynikowy kod źródłowy pokazany na listingu 11.6 jest moim zdaniem piękny.

Listing 11.6. Implementowanie ciągu Fibonacciego z użyciem krotek

static IEnumerable<int> Fibonacci()

```
{
  var pair = (current: 0, next: 1);
  while (true)
  {
    yield return pair.current;
    pair = (pair.next, pair.current + pair.next);
  }
}
```

Na tym etapie trudno jest oprzeć się pokusie dodatkowego uogólnienia rozwiązania, aby generować dowolne sekwencje liczb. W tym celu należy cały kod związany z ciągiem Fibonacciego powiązać z argumentami z wywołania metody. Na listingu 11.7 pokazana jest uogólniona metoda `GenerateSequence`, odpowiednia do generowania dowolnych sekwencji na podstawie argumentów.

Listing 11.7. Podział zadań związanych z generowaniem ciągu Fibonacciego

```
static IEnumerable<TResult>
  GenerateSequence<TState, TResult>(
    TState seed,
    Func<TState, TState> generator,
    Func<TState, TResult> resultSelector)
{
  var state = seed;
  while (true)
  {
    yield return resultSelector(state);
    state = generator(state);
  }
}
```

Metoda umożliwiająca generowanie dowolnych sekwencji na podstawie wcześniejszego stanu.

Przykładowe zastosowanie

```
var fibonacci = GenerateSequence(
  (current: 0, next: 1),
  pair => (pair.next, pair.current + pair.next),
  pair => pair.current);
```

Wykorzystanie generatora sekwencji do utworzenia ciągu Fibonacciego.

Podobny efekt można oczywiście uzyskać z użyciem typów anonimowych, a nawet typów nazwanych. Rozwiązanie nie byłoby jednak równie eleganckie. Czytelnicy mający doświadczenie w posługiwaniu się innymi językami programowania mogą nie być pod dużym wrażeniem tego kodu. Nie jest tak, że w C# 7 wprowadzono zupełnie nowy paradygmat programowania. Ekscytująca jest jednak możliwość pisania tak pięknego kodu w C#.

Znasz już podstawy działania krotek. Pora przejść do bardziej zaawansowanych zagadnień. W następnym podrozdziale omawiane są przede wszystkim konwersje, wyjaśniam też jednak, w których miejscach nazwy elementów są ważne, a w których nie.

11.3. Typy krotek i konwersje

Do tego miejsca starannie unikałem szczegółowego objaśniania typów literałów krotek. Dzięki zachowaniu ogólności mogłem pokazać sporo kodu, co pozwoliło Ci zobaczyć, jak używać krotek. Teraz pora uzasadnić człon „od podszewki" z tytułu książki. Najpierw zastanów się nad deklaracjami z użyciem słowa `var` i literałów krotek.

11.3.1. Typy literałów krotek

Niektóre literały krotek mają typ, natomiast inne nie mają. Obowiązuje tu prosta reguła — literał krotki ma typ, gdy wszystkie wyrażenia reprezentujące elementy tej krotki mają typ. Wyrażenie bez typu nie jest niczym nowym w C#. Wyrażenia lambda, grupy metod i literał `null` to wyrażenia niemające typu. Literałów krotek bez typu (podobnie jak innych wymienionych wyrażeń) nie można używać do przypisywania wartości do zmiennych lokalnych z niejawnie określanym typem. Na przykład poniższy kod jest poprawny, ponieważ 10 i 20 to wyrażenia mające typ:

```
var valid = (10, 20);
```

Jednak następny fragment jest błędny, ponieważ literał `null` nie ma typu:

```
var invalid = (10, null);
```

Literał krotki niemający typu, podobnie jak literał `null`, można przekształcić na postać mającą typ. Gdy krotka ma typ, nazwy elementów też są częścią tego typu.

Na przykład we wszystkich poniższych przykładach lewa strona jest odpowiednikiem prawej:

```
var tuple = (x: 10, 20);
```

```
(int x, int) tuple = (x: 10, 20);
```

```
var array = new[] {("a", 10)};
```

```
(string, int)[] array = {("a", 10)};
```

```
string[] input = {"a", "b" };
```

```
string[] input = {"a", "b" };
```

```
var query = input
  .Select(x => (x, x.Length));
```

```
IEnumerable<(string, int)> query =
  input.Select<string, (string, int)>
  (x => (x, x.Length));
```

W pierwszym przykładzie pokazane jest, że nazwy elementów są przenoszone z literałów krotek do typów krotek. W ostatnim przykładzie widać, że wnioskowanie typów działa także w skomplikowanych scenariuszach; typ zmiennej `input` umożliwia, by typ zmiennej `x` w wyrażeniu lambda został określony na stałe jako `string`. Pozwala to odpowiednio powiązać wyrażenie `x.Length`. Powstaje więc literał krotki z elementami typów `string` i `int`, tak więc typ wartości zwracanej przez wyrażenie lambda jest określany w wyniku wnioskowania jako `(string, int)`. Podobne wnioskowanie pokazane jest na listingu 11.7 w implementacji tworzenia ciągu Fibonacciego za pomocą metody generującej sekwencje. Tam jednak nie skupiałem się na używanych typach.

Wiesz już, jak działają literały krotek mające typ. Co jednak z literałami krotek, które nie mają typu? Jak przekształcić literał krotki bez nazw na typ krotki z nazwami? Aby odpowiedzieć na to pytanie, trzeba przyjrzeć się konwersji krotek na ogólnym poziomie.

Trzeba uwzględnić dwa rodzaje konwersji — z literałów krotek na typy krotek i z jednego typu krotki na inny taki typ. Podobne rozróżnienie zostało już opisane w rozdziale 8. Istnieje konwersja z wyrażenia reprezentującego literał tekstowy z interpolacją na typ `FormattableString`, nie można jednak przekształcić typu `string` na `Format` ↪`tableString`. Podobnie jest w omawianym tu scenariuszu. Najpierw przyjrzyj się konwersji literałów.

Parametry wyrażeń lambda mogą wyglądać jak krotki

Wyrażenia lambda z jednym parametrem są oczywiste, jeśli jednak używasz dwóch parametrów, takie wyrażenia mogą przypominać krotki. Przyjrzyj się np. przydatnej metodzie, która używa jednej z wersji metody `Select` z technologii LINQ. Ta metoda zwraca projekcję z użyciem wartości i indeksu elementu. Często przydatne jest przekazywanie indeksu do innych operacji, dlatego oba te komponenty można umieścić w krotce. To oznacza, że należy utworzyć następującą metodę:

```
static IEnumerable<(T value, int index)> WithIndex<T>
  (this IEnumerable<T> source) =>
  source.Select((value, index) => (value, index));
```

Przyjrzyj się teraz następującemu wyrażeniu lambda:

```
(value, index) => (value, index)
```

Pierwsze wystąpienie pary `(value, index)` nie jest literałem krotki. Jest to sekwencja parametrów wyrażenia lambda. Drugie wystąpienie *jest* literałem krotki. Jest to wynik wyrażenia lambda.

Nie jest to żaden problem. Nie chcę jedynie, abyś był zaskoczony, gdy zetkniesz się z podobnym kodem.

11.3.2. Konwersje z literałów krotek na typy krotek

Podobnie jak jest w wielu innych obszarach języka C#, istnieją konwersje jawne i niejawne z literałów krotek na typy krotek. Sądzę, że rzadko będziesz potrzebować konwersji jawnych, co objaśniam dalej. Gdy już zrozumiesz przebieg konwersji niejawnych, konwersje jawne zapewne i tak uznasz za zbędne.

KONWERSJE NIEJAWNE

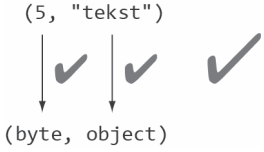
Literał krotki można niejawnie przekształcić na typ krotki, jeśli spełnione są oba wymienione tu warunki:

- literał i typ mają tę samą arność,
- każde wyrażenie w literale można niejawnie przekształcić na powiązany typ elementu.

Pierwszy punkt jest prosty. Dziwne byłoby przekształcanie literału `(5, 5)` na typ `(int, int, int)`. Skąd miałaby pochodzić ostatnia wartość? Drugi punkt jest bardziej skomplikowany, omówię go jednak na przykładach. Najpierw spróbuj przeprowadzić następującą konwersję:

```
(byte, object) tuple = (5, "tekst");
```

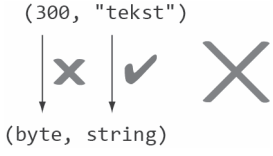
Musisz przyjrzeć się wyrażeniom reprezentującym każdy element źródłowego literału krotki `(5, "tekst")` i sprawdzić, czy istnieje niejawna konwersja na powiązany typ elementu z docelowym typem krotki `(byte, object)`. Jeśli można przekształcić każdy element, konwersja jest prawidłowa:



Choć nie istnieje niejawna konwersja z typu `int` na `byte`, możliwa jest niejawna konwersja ze stałej całkowitoliczbowej `5` na typ `byte` (ponieważ liczba 5 znajduje się w przedziale poprawnych wartości typu `byte`). Ponadto istnieje niejawna konwersja z literału tekstowego na typ `object`. Wszystkie konwersje są poprawne, dlatego cała konwersja też jest prawidłowa. Hura! Teraz spróbuj wykonać inną konwersję:

```
(byte, string) tuple = (300, "tekst");
```

Także teraz spróbuj przeprowadzić niejawną konwersję kolejnych elementów:



W tej sytuacji kod ma przekształcić stałą całkowitoliczbową `300` na typ `byte`. Ta stała wykracza poza przedział poprawnych wartości, dlatego nie istnieje tu niejawna konwersja. Dostępna jest konwersja jawna, nie jest ona jednak pomocna, gdy chcesz uzyskać ogólną niejawną konwersję literału krotki. Istnieje niejawna konwersja z literału tekstowego na typ `string`, jednak ponieważ nie wszystkie konwersje są prawidłowe, cała konwersja jest niedozwolona. Jeśli spróbujesz skompilować taki kod, wystąpi błąd dotyczący wartości 300 w literale krotki:

```
error CS0029: Cannot implicitly convert type 'int' to 'byte'
```

Ten komunikat o błędzie jest nieco mylący. Sugeruje, że wcześniejszy przykład także nie powinien być poprawny. W rzeczywistości kompilator nie próbuje przekształcić wartości typu `int` na typ `byte`; próbuje natomiast przekształcić wyrażenie `300` na typ `byte`.

KONWERSJE JAWNE

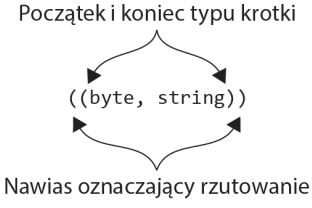
Dla jawnych konwersji literałów krotek używane są te same reguły co dla konwersji niejawnych. Wymagane jest, by możliwa była konwersja jawna każdego wyrażenia reprezentującego element na powiązane typy. Jeśli ten warunek jest spełniony, istnieje jawna konwersja z literału krotki na typ krotki, dlatego można przeprowadzić rzutowanie w standardowy sposób.

WSKAZÓWKA. Każda konwersja niejawna w C# jest też uważana za konwersję jawną, co jest nieco mylące. Aby warunek był bardziej zrozumiały, możesz przyjąć, że oznacza on: „dla każdego elementu musi istnieć konwersja jawna lub niejawna”.

Wróćmy do konwersji wartości `(300, "tekst")`. Można ją jawnie przekształcić na typ krotki `(byte, string)`. Wymaga to jednak użycia kontekstu niekontrolowanego, ponieważ kompilator wykrywa, że stała `300` znajduje się poza zakresem typu `byte`. W bardziej realistycznym przykładzie można użyć pochodzącej z innego miejsca zmiennej typu `int`:

```
int x = 300;
var tuple = ((byte, string)) (x, "tekst");
```

W rzutowaniu `((byte, string))` na pozór używanych jest więcej nawiasów, niż to konieczne. Jednak wszystkie te nawiasy są niezbędne. Wewnętrzne oznaczają typ krotki, natomiast zewnętrzne reprezentują rzutowanie. Na rysunku 11.4 jest to pokazane w formie graficznej.



Rysunek 11.4. Objaśnienie nawiasów w jawnej konwersji krotki

Bardzo nie podoba mi się ten zapis, jednak miło mieć możliwość wykonania takiej konwersji. W wielu sytuacjach prostszym rozwiązaniem jest użycie odpowiedniego rzutowania we wszystkich wyrażeniach reprezentujących elementy w literale krotki. Wtedy nie tylko konwersja krotki jest poprawna, ale też wywnioskowany typ literału jest zgodny z oczekiwaniami programisty. Wcześniejszy przykład zapewne zapisałbym w następujący sposób:

```
int x = 300;
var tuple = ((byte) x, "tekst");
```

Oba te rozwiązania działają tak samo. Gdy konwersja jest stosowana dla całego literału krotki, kompilator i tak stosuje jawną konwersję dla wszystkich wyrażeń reprezentujących elementy. Jednak druga wersja jest moim zdaniem dużo bardziej czytelna. Między innymi jaśniej określa przeznaczenie kodu. Wiesz, że potrzebna jest jawna konwersja z typu `int` na `byte`, a jednocześnie akceptujesz, by typ `string` pozostał niezmieniony. Jeśli próbujesz przekształcić kilka wartości na określony typ krotki (zamiast korzystać z wywnioskowanego typu), druga wersja pozwala jednoznacznie pokazać, które konwersje są jawne i mogą skutkować utratą danych. Chroni to przed przypadkową utratą danych w wyniku jawnej konwersji całej krotki.

ROLA NAZW ELEMENTÓW W KONWERSJACH LITERAŁÓW KROTEK

Może zauważyłeś, że w tym punkcie nic nie piszę o nazwach. Są one prawie zupełnie nieistotne w kontekście konwersji literałów krotek. Najważniejsze jest to, że można przekształcić wyrażenie reprezentujące element bez nazwy na element z typem i nazwą. Kilkakrotnie wykonałeś już taką operację w tym rozdziale i nie zwracałem na to uwagi. Robiłeś to od początku, od pierwszej implementacji metody `MinMax`. W ramach przypomnienia — deklaracja tej metody wyglądała tak:

```
static (int min, int max) MinMax(IEnumerable<int> source)
```

Dalej instrukcja `return` wyglądała tak:

```
return (min, max);
```

Kod próbuje przekształcić literał krotki bez nazw elementów[3] na typ `(int min, int max)`. Jest to oczywiście poprawne. W przeciwnym razie nie pokazywałbym tego kodu. Ponadto jest to wygodne. Nazwy elementów nie są jednak całkowicie bez znaczenia w trakcie konwersji literałów krotek. Gdy nazwa elementu jest bezpośrednio podana w literale krotki, kompilator ostrzega, jeśli w docelowym typie dla danego elementu nie podano nazwy lub gdy podana nazwa jest inna. Oto przykład:

```
(int a, int b, int c, int, int) tuple =
  (a: 10, wrong: 20, 30, pointless: 40, 50);
```

Widoczne są tu wszystkie możliwe kombinacje nazw elementów w następującej kolejności:

1. W docelowym typie i w literale krotki podana jest ta sama nazwa elementu.
2. W docelowym typie i w literale krotki podane są różne nazwy elementu.
3. W docelowym typie określona jest nazwa elementu, jednak nie podano jej w literale krotki.
4. W docelowym typie nie podano nazwy elementu, jednak jest ona określona w literale krotki.
5. Nazwa elementu nie jest podana ani w docelowym typie, ani w literale krotki.

Drugi i czwarty scenariusz skutkują oczywiście ostrzeżeniami w czasie kompilacji. Oto wynik kompilacji takiego kodu:

```
warning CS8123: The tuple element name 'wrong' is ignored because a different
  name is specified by the target type '(int a, int b, int c, int, int)'.
warning CS8123: The tuple element name 'pointless' is ignored because a
  different name is specified by the target type '(int a, int b, int c, int, int)'
```

Drugi z tych komunikatów z ostrzeżeniem nie jest tak przydatny, jak mógłby być, ponieważ w rzeczywistości w docelowym typie w ogóle nie podano nazwy elementu. Mam nadzieję, że i tak potrafisz zrozumieć, gdzie tkwi problem.

Czy opisane rozwiązanie jest przydatne? Jak najbardziej. Nie wtedy, gdy deklarujesz zmienną i tworzysz wartość w jednej instrukcji, ale w sytuacji, gdy deklaracja i tworzenie wartości są rozdzielone. Załóżmy, że metoda `MinMax` z listingu 11.1 jest naprawdę długa i że trudno ją zrefaktoryzować. Czy należy zwrócić wartość `(min, max)`, czy `(max, min)`? Tak, w tej sytuacji nazwa metoda sprawia, że kolejność elementów jest dość oczywista. Jednak w niektórych przypadkach jest inaczej. Wtedy użycie nazw elementów w instrukcji `return` może być przydatne do sprawdzania poprawności kodu. Ten kod skompiluje się bez zgłaszania ostrzeżeń:

```
return (min: min, max: max);
```

[3] Przynajmniej w wersji C# 7.0. W punkcie 11.2.2 zostało napisane, że w C# 7.1 stosowane jest wnioskowanie nazw.

Jeśli jednak odwrócisz kolejność elementów, dla każdego z nich zostanie wyświetlone ostrzeżenie:

```
return (max: max, min: min); <-- Ostrzeżenie CS8123 (dwukrotnie).
```

Zauważ, że dotyczy to tylko jawnie podawanych nazw. Nawet w C# 7.1, gdzie nazwy elementu są wnioskowane na podstawie literału krotki (`max, min`), nie pojawi się ostrzeżenie, jeśli przekształcisz wartość na typ krotki (`int min, int max`).

Zawsze wolę nadawać kodowi taką strukturę, aby program był tak jednoznaczny, że dodatkowe sprawdzanie poprawności nie jest potrzebne. Warto jednak wiedzieć, że opisana technika jest dostępna, jeśli jej potrzebujesz — np. w pierwszym kroku przed refaktoryzacją metody w celu jej skrócenia.

11.3.3. Konwersja między typami krotek

Po zrozumieniu konwersji literałów krotek niejawne i jawne konwersje typów krotek są stosunkowo proste, ponieważ przebiegają podobnie. Nie trzeba się tu przejmować wyrażeniami, ponieważ używane są tylko typy. Istnieje niejawna konwersja ze źródłowego typu krotki na typ docelowy o tej samej arności, jeśli możliwa jest niejawna konwersja każdego elementu z typu źródłowego na powiązany element w typie docelowym. Podobnie istnieje jawna konwersja ze źródłowego typu krotki na typ docelowy o tej samej arności, jeżeli obsługiwana jest jawna konwersja każdego elementu z typu źródłowego na powiązany element w typie docelowym. Oto przykład ilustrujący różne konwersje typu źródłowego (`int, string`):

```
var t1 = (300, "tekst");                    <-- Typ zmiennej t1 zostaje wywnioskowany jako (int, string).
(long, string) t2 = t1;                     <-- Poprawna niejawna konwersja z typu (int, string) na (long, string).
(byte, string) t3 = t1;                     <-- Błąd — brak niejawnej konwersji z typu int na byte.
(byte, string) t4 = ((byte, string)) t1;    <-- Poprawna jawna konwersja z typu (int, string) na (byte, string).
(object, object) t5 = t1;                   <-- Poprawna niejawna konwersja z typu (int, string) na (object, object).
(string, string) t6 = ((string, string)) t1; <-- Błąd — brak konwersji z typu int na string.
```

Tu jawna konwersja z typu (`int, string`) na (`byte, string`) w wierszu 4. sprawi, że wartością elementu `t4.Item1` będzie `44`. Jest to wynik jawnej konwersji wartości 300 typu `int` na typ `byte`.

Inaczej niż w przypadku konwersji literałów krotek tu niedopasowanie nazw elementów nie skutkuje ostrzeżeniami. Mogę przedstawić to na przykładzie podobnym do konwersji literałów krotek o pięciu elementach. Wystarczy zapisać wartość krotki w zmiennej, aby przeprowadzić konwersję z typu na typ zamiast z literału na typ:

```
var source = (a: 10, wrong: 20, 30, pointless: 40, 50);
(int a, int b, int c, int, int) tuple = source;
```

Ten kod skompiluje się bez zgłaszania ostrzeżeń. Ważnym aspektem konwersji typów krotek niezwiązanym z konwersjami literałów jest konwersja tożsamościowa (a nie tylko niejawna).

KONWERSJE TOŻSAMOŚCIOWE TYPÓW KROTEK

Konwersje tożsamościowe występują w C# od czasu powstania tego języka, choć z czasem zostały rozbudowane. Do wersji C# 7 reguły tych konwersji wyglądały tak:

- Istnieje konwersja tożsamościowa z danego typu na ten sam typ.
- Istnieje konwersja tożsamościowa między typami `object` i `dynamic`.
- Istnieje konwersja tożsamościowa między dwoma typami tablicowymi, jeśli możliwa jest konwersja tożsamościowa elementów tych tablic. Występuje np. konwersja tożsamościowa między typami `object[]` i `dynamic[]`.
- Konwersje tożsamościowe są rozszerzane na skonstruowane typy generyczne, gdy istnieje konwersja tożsamościowa między odpowiadającymi sobie argumentami określającymi typ. Na przykład istnieje konwersja tożsamościowa między typami `List<object>` i `List<dynamic>`.

Krotki wprowadzają dodatkowy rodzaj konwersji tożsamościowej — między typami krotek o tej samej arności, gdy istnieje konwersja tożsamościowa między parami odpowiadających sobie elementów (nazwy elementów nie są tu istotne). Oznacza to, że istnieją konwersje tożsamościowe (w obu kierunkach — konwersja tożsamościowa zawsze jest symetryczna) między następującymi typami:

- `(int x, object y)`,
- `(int a, dynamic d)`,
- `(int, object)`.

Dotyczy to również typów skonstruowanych, a typy elementów krotek także mogą być typami skonstruowanymi (o ile konwersja tożsamościowa nadal jest możliwa). Istnieje więc np. konwersja tożsamościowa między dwoma poniższymi typami:

- `Dictionary<string, (int, List<object>)>`,
- `Dictionary<string, (int index, List<dynamic> values)>`.

W kontekście krotek konwersje tożsamościowe są najważniejsze, gdy używane są typy skonstruowane. Irytujące byłoby, gdybyś mógł łatwo przekształcić typ `(int, int)` na `(int x, int y)`, ale już nie typ `IEnumerable<(int, int)>` na `IEnumerable<(int x, int y)>` lub na odwrót.

Konwersje tożsamościowe są też istotne dla wersji przeciążonych metod. W taki sam sposób, jak dwie wersje metody nie mogą różnić się tylko typem zwracanej wartości, tak nie mogą się różnić tylko typami parametrów umożliwiającymi konwersję tożsamościową. Nie można np. umieścić w jednej klasie dwóch następujących metod:

```
public void Method((int, int) tuple) {}
public void Method((int x, int y) tuple) {}
```

Próba utworzenia takich metod spowoduje błąd kompilacji:

```
error CS0111: Type 'Program' already defines a member called 'Method' with
  the same parameter types
```

W języku C# typy użytych tu parametrów nie są takie same, jednak aby komunikat o błędzie był w pełni precyzyjny w kwestii konwersji tożsamościowych, musiałby być dużo bardziej skomplikowany.

Jeśli uważasz, że oficjalne definicje konwersji tożsamościowych są trudne do zrozumienia, możesz myśleć o takich konwersjach w prostszy (choć mniej oficjalny) sposób — dwa typy są identyczne, jeśli nie występuje różnica między nimi w czasie wykonywania programu. To zagadnienie zostanie opisane szczegółowo w podrozdziale 11.4.

BRAK KONWERSJI OPARTEJ NA WARIANCJI GENERYCZNEJ

Po zapoznaniu się z konwersjami tożsamościowymi możesz mieć nadzieję, że da się zastosować typy krotek z użyciem wariancji generycznej w interfejsach i delegatach. Niestety, taka możliwość nie istnieje. Wariancja dotyczy tylko typów referencyjnych, a typy krotek są zawsze typami bezpośrednimi. Wydaje się np., że poniższy kod powinien się skompilować:

```
IEnumerable<(string, string)> stringPairs = new (string, string)[10];
IEnumerable<(object, object)> objectPairs = stringPairs;
```

Tak jednak nie jest. Szkoda. Nie wydaje mi się, aby w praktyce często sprawiało to problem, chcę jednak oszczędzić Ci rozczarowania w sytuacji, gdybyś chciał zastosować takie rozwiązanie i spodziewał się, że zadziała.

11.3.4. Zastosowania konwersji

Wiesz już, jakie możliwości są dostępne. Możliwe, że zastanawiasz się teraz, kiedy warto wykorzystać konwersje krotek. W dużym stopniu zależy to od tego, jak w ogóle stosujesz krotki. Krotki używane w jednej metodzie lub zwracane w metodach prywatnych i wykorzystywane w tej samej klasie rzadko wymagają konwersji. Wystarczy od początku wybrać odpowiedni typ i ewentualnie zrzutować typy w literale krotki w momencie tworzenia początkowej wartości.

Konwersja z jednego typu krotki na inny z większym prawdopodobieństwem będzie potrzebna, gdy używasz metod wewnętrznych lub publicznych przyjmujących albo zwracających krotki. Wtedy masz mniejszą kontrolę nad typami elementów. Im szersze jest zastosowanie danego typu krotki, z tym mniejszym prawdopodobieństwem będzie to *dokładnie* ten typ, jaki jest potrzebny w konkretnych sytuacjach.

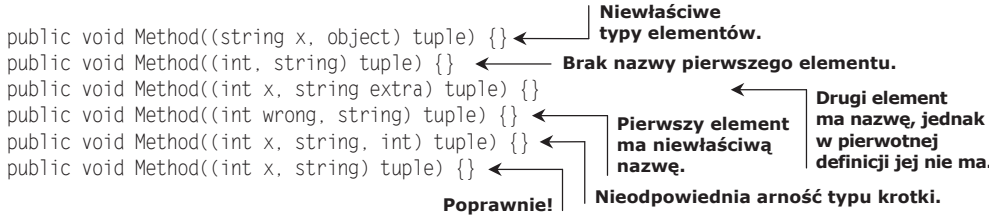
11.3.5. Sprawdzanie nazw elementów przy dziedziczeniu

Choć w konwersjach nazwy elementów nie są istotne, kompilator jest wymagający przy ich używaniu w procesie dziedziczenia. Gdy typ krotki występuje w składowej, którą albo przesłaniasz (a przesłaniana składowa znajduje się w klasie bazowej), albo implementujesz na potrzeby interfejsu, użyte nazwy elementów muszą pasować do tych z pierwotnej definicji. Ponadto jeśli w pierwotnej definicji jakaś nazwa nie występuje, nie można jej zastosować w implementacji. Typy elementów w implementacji muszą umożliwiać konwersję tożsamościową na typy elementów z pierwotnej definicji.

W ramach przykładu rozważ poniższy interfejs `ISample` i kilka metod, które mają być implementacją metody `ISample.Method` (każda z tych wersji powinna się oczywiście znaleźć w odrębnej klasie z implementacją):

```
interface ISample
{
  void Method((int x, string) tuple);
}

public void Method((string x, object) tuple) {}
public void Method((int, string) tuple) {}
public void Method((int x, string extra) tuple) {}
public void Method((int wrong, string) tuple) {}
public void Method((int x, string, int) tuple) {}
public void Method((int x, string) tuple) {}
```



W tym przykładzie opisana jest tylko implementacja interfejsu, jednak te same ograniczenia obowiązują przy przesłanianiu składowych z klasy bazowej. Ponadto w przykładzie używane są tylko parametry, a ograniczenia dotyczą również typów zwracanych wartości. Oznacza to, że dodanie, usunięcie lub zmodyfikowanie nazwy elementu krotki w składowej interfejsu albo składowej klasy wirtualnej lub abstrakcyjnej narusza zgodność z istniejącym kodem. Dlatego dobrze rozważ zastosowanie takiej techniki w publicznym interfejsie API!

UWAGA. Pod niektórymi względami jest to niespójne rozwiązanie, ponieważ kompilator nigdy wcześniej nie uwzględniał zmiany nazw parametrów metody przez autora klasy, który przesłaniał metodę lub implementował interfejs. Możliwość podawania nazw argumentów oznacza, że mogą wystąpić problemy, gdy kod jednostki wywołującej zmieni się i użyty zostanie interfejs zamiast implementacji lub na odwrót. Podejrzewam, że gdyby projektanci języka C# zaczynali pracę od początku, zakazaliby modyfikowania nazw argumentów w tym kontekście.

W C# 7.3 do krotek dodano nowy mechanizm języka — porównywanie ich za pomocą operatorów == i !=.

11.3.6. Operatory równości i nierówności (C# 7.3)

W punkcie 11.4.5 zobaczysz, że w środowisku CLR od początku można było porównywać krotki za pomocą metody `Equals`. W tym środowisku nie istniały jednak przeciążone wersje operatorów == i != dla krotek. W C# 7.3 kompilator udostępnia implementacje operatorów == i != do porównywania krotek, jeśli istnieje konwersja tożsamościowa między typami krotek obu operandów. Oznacza to, że nazwy elementów nie są tu istotne.

Kompilator rozwija operatory == i != jako porównania par wartości odpowiadających elementów za pomocą operatorów == i !=. Prawdopodobnie najprościej przedstawić to za pomocą przykładu (zobacz listing 11.8).

Listing 11.8. Operatory równości i nierówności

```
var t1 = (x: "x", y: "y", z: 1);
var t2 = ("x", "y", 1);

Console.WriteLine(t1 == t2); ← Operator równości.
Console.WriteLine(t1.Item1 == t2.Item1 &&
                  t1.Item2 == t2.Item2 &&
                  t1.Item3 == t2.Item3);
```

Analogiczny kod generowany przez kompilator.

```
Console.WriteLine(t1 != t2);          <-- Operator nierówności.
Console.WriteLine(t1.Item1 != t2.Item1 &&
                  t1.Item2 != t2.Item2 &&   Analogiczny kod generowany przez kompilator.
                  t1.Item3 != t2.Item3);
```

Na listingu 11.8 pokazane są dwie krotki (jedna z nazwami elementów i druga bez). Kod sprawdza, czy są one równe, czy nierówne. W obu scenariuszach pokazany jest też kod generowany przez kompilator na podstawie operatora. Należy zauważyć, że w wygenerowanym kodzie używane są wersje operatorów udostępniane przez typy elementów. Typy ze środowiska CLR nie potrafią udostępniać takich możliwości bez korzystania z mechanizmu refleksji. Dlatego to zadanie lepiej obsługiwać za pomocą kompilatora.

To już wszystkie potrzebne informacje na temat reguł używania krotek w języku. Dokładne szczegóły przekazywania nazw elementów w ramach wnioskowania typów i podobne wiadomości najlepiej opisano w specyfikacji języka. Nawet w tej książce występuje granica potrzebnej szczegółowości omówień. Choć mógłbyś wykorzystać zaprezentowane tu informacje i zignorować obsługę krotek w środowisku CLR, będziesz potrafił lepiej używać krotek i zrozumieć ich działanie, jeśli zrobisz dodatkowy krok i dowiesz się, jak kompilator przetwarza opisane reguły w kod pośredni.

Otrzymałeś już bardzo dużą dawkę wiedzy. Jeśli jeszcze nie próbowałeś pisać kodu z użyciem krotek, jest to dobry moment, aby to zrobić. Zrób sobie przerwę od książki i przed przejściem do omówienia implementacji krotek sprawdź, czy radzisz sobie z korzystaniem z nich.

11.4. Krotki w środowisku CLR

Choć w teorii język C# nie jest powiązany z platformą .NET, w praktyce autorzy każdej implementacji, jaką znam, przynajmniej próbują w jakimś zakresie upodobnić ją do zwykłej platformy .NET (nawet jeśli stosowana jest kompilacja AOT i kod działa w urządzeniu innym niż komputer PC). Specyfikacja języka C# stawia określone wymagania środowisku docelowemu. M.in. dostępne muszą być ustalone typy. W czasie, gdy powstaje ta książka, nie istnieje specyfikacja języka C# 7. Podejrzewam jednak, że gdy się pojawi, będzie wymagała dostępności opisanych w tym podrozdziale typów, aby móc używać krotek.

W odróżnieniu od typów anonimowych, gdzie każda unikatowa sekwencja nazw właściwości w podzespole powoduje wygenerowanie przez kompilator nowego typu, krotki nie wymagają generowania przez kompilator dodatkowych typów. Zamiast tego używany jest nowy zestaw typów z platformy. Pora się z nimi zapoznać.

11.4.1. Wprowadzenie do typów System.ValueTuple<...>

Krotki w C# 7 są implementowane przy użyciu rodziny typów `System.ValueTuple`. Te typy znajdują się w podzespole `System.ValueTuple.dll`, który jest częścią specyfikacji .NET Standard 2.0, ale nie występuje w żadnych starszych wersjach platformy .NET. Aby używać tych typów w starszych platformach, należy dodać zależność w postaci pakietu NuGet `System.ValueTuple`.

Istnieje dziewięć struktur `ValueTuple` o generycznej arności od 0 do 8:

- `System.ValueTuple` (niegeneryczny)
- `System.ValueTuple<T1>`
- `System.ValueTuple<T1, T2>`
- `System.ValueTuple<T1, T2, T3>`
- `System.ValueTuple<T1, T2, T3, T4>`
- `System.ValueTuple<T1, T2, T3, T4, T5>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>`

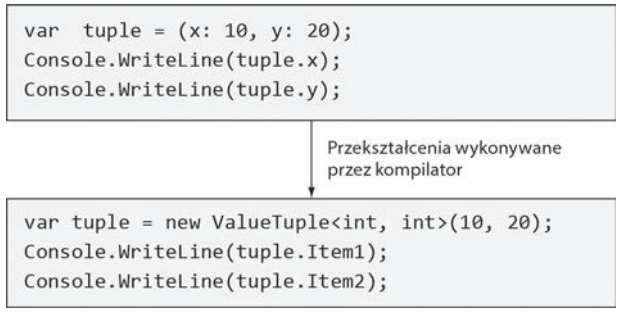
Na razie pomiń pierwsze dwa i ostatni z tych typów (ten ostatni jest opisany w punktach 11.4.7 i 11.4.8). Tu do omówienia pozostają typy o generycznej arności od 2 do 7. W praktyce to z nich będziesz zapewne korzystał najczęściej.

Opis każdego typu `ValueTuple<…>` bardzo przypomina wcześniejsze opisy typów krotek. Typy `ValueTuple<…>` to typy bezpośrednie z polami publicznymi. Nazwy tych pól to `Item1`, `Item2` itd. (do `Item7`). Ostatnie pole w krotce o arności 8 ma nazwę `Rest`.

Za każdym razem, gdy używasz typu krotki w C#, jest on odwzorowywany na typ `ValueTuple<…>`. To odwzorowanie jest oczywiste, gdy w typie krotki w C# nie są używane nazwy elementów. Na przykład typ `(int, string, byte)` jest odwzorowywany na typ `ValueTuple<int, string, byte>`. Co jednak z opcjonalnymi nazwami elementów z typów krotek w C#? Typy generyczne są generyczne tylko ze względu na parametry określające typ. Nie można w magiczny sposób nadać dwóm skonstruowanym typom różnych nazw pól. Jak kompilator sobie z tym radzi?

11.4.2. Obsługa nazw elementów

Kompilator języka C# ignoruje nazwy w kontekście odwzorowywania typów krotek z C# na typy `ValueTuple<…>` ze środowiska CLR. Choć w języku C# `(int, int)` i `(int x, int y)` to odmienne typy, oba są odwzorowywane na typ `ValueTuple<int, int>`. Następnie kompilator odwzorowuje wszystkie przypadki użycia nazw elementów na odpowiednie nazwy `ItemN`. Na rysunku 11.5 pokazane jest, jak kod C# z literałem krotki jest przekształcany na kod C# używający tylko typów ze środowiska CLR.



Rysunek 11.5. Dokonywane przez kompilator przekształcenia typu krotki na typ ValueTuple

Warto zauważyć, że w dolnej połowie rysunku 11.5 znajduje się wiele nazw. Nazwy zmiennych lokalnych, takie jak w tym kodzie, są używane tylko na etapie kompilacji.

Jedyny ślad po nich w czasie wykonywania programu znajduje się w pliku PDB tworzonym po to, by zapewnić debugerowi dodatkowe informacje. A co z nazwami elementów widocznymi poza stosunkowo niewielkim kontekstem metody?

NAZWY ELEMENTÓW W METADANYCH

Wróć do używanej kilkakrotnie w tym rozdziale metody `MinMax`. Załóżmy, że chcesz utworzyć ją jako metodę publiczną w całym pakiecie metod agregujących, które wspomagają technologię LINQ to Objects. Szkoda byłoby tracić czytelność zapewnianą przez nazwy elementów krotek. Wiesz jednak, że typ CLR wartości zwracanej przez metodę nie będzie obejmować tych nazw. Na szczęście kompilator może wykorzystać technikę pomocną także dla innych mechanizmów, które nie są bezpośrednio obsługiwane w środowisku CLR, np. dla parametrów `out` i domyślnych wartości parametrów. Tym wybawieniem są atrybuty.

W tym scenariuszu kompilator używa atrybutu `TupleElementNamesAttribute` (znajduje się on w tej samej przestrzeni nazw co wiele podobnych atrybutów — `System.Runtime.Compiler.Services`), aby zakodować nazwy elementów w podzespole. Na przykład publiczną deklarację metody `MinMax` można zapisać w C# 6 tak:

```
[return: TupleElementNames(new[] {"min", "max"})]
public static ValueTuple<int, int> MinMax(IEnumerable<int> numbers)
```

Kompilator języka C# 7 nie pozwala skompilować tego kodu. Zgłasza błąd z informacją, że należy bezpośrednio zastosować składnię dla krotek. Jednak jeśli skompilujesz ten sam kod za pomocą kompilatora dla C# 6, otrzymasz podzespół, który możesz wykorzystać w C# 7, a elementy zwracanej krotki będą dostępne za pomocą nazw.

Omawiany atrybut staje się bardziej skomplikowany, gdy używane są zagnieżdżone typy krotek. Jest jednak mało prawdopodobne, abyś kiedykolwiek musiał bezpośrednio interpretować ten atrybut. Warto jedynie wiedzieć, że taki atrybut istnieje i że pozwala on przekazywać nazwy elementów także poza zmiennymi lokalnymi. Takie atrybuty są generowane przez kompilator języka C# nawet na potrzeby składowych prywatnych, choć w tym przypadku te atrybuty prawdopodobnie nie byłyby potrzebne. Podejrzewam jednak, że prościej jest traktować wszystkie składowe w ten sam sposób niezależnie od modyfikatorów dostępu.

BRAK NAZW ELEMENTÓW W CZASIE WYKONYWANIA PROGRAMU

Jeśli nie jest to oczywiste na podstawie wcześniejszego tekstu, warto dodać, że w czasie wykonywania programu w wartości krotki nie występują nazwy elementów. Gdy wywołasz `GetType()` dla wartości krotki, otrzymasz typ `ValueTuple<…>` z odpowiednimi typami elementów, jednak nazwy elementów z kodu źródłowego będą nieobecne. Jeżeli wykonasz kod w trybie kroczenia i debuger wyświetli nazwy, wynika to z tego, że debuger używa dodatkowych informacji do ustalenia pierwotnych nazw elementów. Te nazwy nie są czymś, co środowisko CLR zna bezpośrednio.

UWAGA. To podejście może wydawać się znajome programistom używającym Javy. Java w podobny sposób obsługuje typy generyczne z informacjami o typie, które są niedostępne w czasie wykonywania programu. W Javie nie istnieje coś takiego jak obiekt typu `ArrayList<Integer>` lub `ArrayList<String>`. Używane są tylko obiekty typu `ArrayList`. W Javie oka-

zało się to problemem, jednak nazwy elementów krotek nie są równie ważne jak argumenty określające typ w typach generycznych, dlatego można mieć nadzieję, że nie będą powodować podobnych kłopotów.

Nazwy elementów istnieją w krotkach w C#, ale już nie w środowisku CLR. A co z konwersjami?

11.4.3. Implementacje konwersji krotek

Typy z rodziny `ValueTuple` nie obsługują *żadnych* konwersji w środowisku CLR. Takie konwersje byłyby niemożliwe. Konwersje dostępne w języku C# nie mogą być zapisane w informacjach o typie. Zamiast tego kompilator języka C# tworzy nową wartość, gdy jest to konieczne, i przeprowadza odpowiednie konwersje każdego elementu. Poniżej pokazane są dwie przykładowe konwersje: jedna niejawna (z wykorzystaniem niejawnej konwersji z typu `int` na `long`) i druga jawna (z użyciem jawnej konwersji z typu `int` na `byte`):

```
(int, string) t1 = (300, "tekst");
(long, string) t2 = t1;
(byte, string) t3 = ((byte, string)) t1;
```

Kompilator generuje tu taki kod, jakbyś użył następujących instrukcji:

```
var t1 = new ValueTuple<int, string>(300, "tekst");
var t2 = new ValueTuple<long, string>(t1.Item1, t1.Item2);
var t3 = new ValueTuple<byte, string>((byte) t1.Item1, t1.Item2));
```

W tym przykładzie uwzględniane są tylko konwersje między typami krotek, które już poznałeś. Jednak konwersje z literałów krotek na typy krotek przebiegają w identyczny sposób. Każda potrzebna konwersja z wyrażenia reprezentującego element na docelowy typ elementu jest wykonywana w ramach wykonywania odpowiedniego konstruktora typu `ValueTuple<…>`.

Dowiedziałeś się już, czego kompilator potrzebuje do obsługi składni dla krotek. Jednak typy `ValueTuple<…>` udostępniają dodatkowe techniki, aby ułatwić pracę z tymi typami. Typy te są bardzo ogólne i nie oferują zbyt wielu możliwości, jednak metoda `ToString()` wyświetla czytelne dane wyjściowe oraz dostępnych jest kilka sposobów porównywania wartości tych typów. Zapoznaj się teraz z dostępnymi technikami.

11.4.4. Tekstowe reprezentacje krotek

Tekstowa reprezentacja krotki wygląda podobnie do literału krotki z kodu źródłowego w języku C#. Wyświetlana jest sekwencja wartości rozdzielonych przecinkami i umieszczonych w nawiasie. Niedostępne są mechanizmy do precyzyjnego kontrolowania takich danych wyjściowych. Jeśli np. używasz krotki typu `(DateTime, DateTime)` do zapisywania przedziału czasu, nie możesz przekazać łańcucha znaków formatowania, aby określić, że elementy mają być formatowane jako daty. Metoda `ToString()` krotki wywołuje metodę `ToString()` każdego elementu różnego od `null` (dla `null` używany jest pusty łańcuch znaków).

Warto przypomnieć, że nazwy nadane elementom krotki nie są znane w czasie wykonywania programu, dlatego nie mogą pojawiać się w wynikach wywołania `ToString()`. Dlatego takie wyniki są nieco mniej użyteczne niż tekstowa reprezentacja typów anonimowych, choć jeśli wyświetlasz wiele krotek tego samego typu, docenisz brak powtórzeń nazw. Jeden krótki przykład wystarczy, aby zademonstrować wszystkie opisane wcześniej informacje:

```
var tuple = (x: (string) null, y: "tekst", z: 10);
Console.WriteLine(tuple.ToString());
```

Rzutowanie wartości null na łańcuch znaków, co pozwala wywnioskować typ krotki.

Zapisywanie wartości krotki w konsoli.

Oto dane wyjściowe tego fragmentu kodu:

```
(, tekst, 10)
```

Metoda `ToString()` jest tu wywoływana bezpośrednio, aby udowodnić, że nie są wykonywane żadne dodatkowe działania. Takie same dane wyjściowe uzyskasz po wywołaniu `Console.WriteLine(tuple)`.

Tekstowa reprezentacja krotek jest oczywiście przydatna w celach diagnostycznych, jednak rzadko nadaje się do bezpośredniego wyświetlania w aplikacjach komunikujących się z użytkownikiem końcowym. Zapewne zechcesz udostępnić dodatkowy kontekst, podać informacje na temat formatowania niektórych typów i zapewne bardziej przejrzyście obsługiwać wartości `null`.

11.4.5. Standardowe porównania na potrzeby sprawdzania równości i sortowania

Każdy typ `ValueTuple<…>` zawiera implementacje interfejsów `IEquatable<T>` i `IComparable<T>`, gdzie `T` to dany typ. Na przykład typ `ValueTuple<T1, T2>` zawiera implementacje interfejsów `IEquatable<ValueTuple<T1, T2>>` i `IComparable<ValueTuple<T1, T2>>`.

W każdym typie w naturalny sposób zaimplementowany jest też niegeneryczny interfejs `IComparable` i przesłonięta jest metoda `object.Equals(object)`. Wywołanie `Equals(object)` zwraca `false`, jeśli argumentem jest obiekt innego typu, a wywołanie `CompareTo(object)` zgłasza w takiej sytuacji wyjątek typu `ArgumentException`. W innych sytuacjach każda z tych metod deleguje zadanie do swojego odpowiednika z interfejsu `IEquatable<T>` lub `IComparable<T>`.

Testy równości są wykonywane element po elemencie z użyciem domyślnego mechanizmu sprawdzania równości z typu każdego elementu. Podobnie skróty elementów są obliczane za pomocą domyślnego mechanizmu sprawdzania równości, po czym następuje łączenie tych skrótów w sposób zależny od implementacji, aby uzyskać ogólny skrót dla krotki. Porównania krotek na potrzeby sortowania także odbywają się element po elemencie, przy czym początkowe elementy są uznawane za ważniejsze niż dalsze. Dlatego np. krotka `(1, 5)` jest uznawana za mniejszą niż `(3, 2)`.

Te porównania sprawiają, że z krotek łatwo jest korzystać w technologii LINQ. Załóżmy, że masz kolekcję krotek typu `(int, int)` reprezentujących współrzędne `(x, y)`. Możesz użyć znanych operacji z technologii LINQ, aby znaleźć na liście różne punkty i je uporządkować. Ilustruje to listing 11.9.

Listing 11.9. Znajdowanie i porządkowanie różnych punktów

```
var points = new[]
{
  (1, 2), (10, 3), (-1, 5), (2, 1),
  (10, 3), (2, 1), (1, 1)
};

var distinctPoints = points.Distinct();
Console.WriteLine($"Liczba różnych punktów: {distinctPoints.Count()}");
Console.WriteLine("Uporządkowane punkty:");
foreach (var point in distinctPoints.OrderBy(p => p))
{
  Console.WriteLine(point);
}
```

Wywołanie `Distinct()` powoduje, że w danych wyjściowych krotka `(2, 1)` występuje tylko raz. Jednak ponieważ równość jest sprawdzana element po elemencie, krotka `(2, 1)` jest różna od `(1, 2)`.

Ponieważ pierwszy element krotki jest uważany w czasie sortowania za najważniejszy, punkty są sortowane według współrzędnej `x`. Jeśli kilka punktów ma tę samą wartość współrzędnej `x`, punkty te są sortowane według współrzędnej `y`. Dlatego dane wyjściowe wyglądają tak:

```
Liczba różnych punktów: 5
Uporządkowane punkty:
(-1, 5)
(1, 1)
(1, 2)
(2, 1)
(10, 3)
```

Zwykłe porównania nie umożliwiają zdefiniowania, jak należy porównywać poszczególne elementy. Oczywiście, możesz stosunkowo łatwo utworzyć własne niestandardowe implementacje interfejsów `IEqualityComparer<T>` i `IComparer<T>` dla określonych typów krotek, jednak wtedy warto rozważyć, czy nie lepiej byłoby zaimplementować kompletny niestandardowy typ, który chcesz reprezentować, i całkowicie zrezygnować wtedy z krotek. Inna możliwość to zastosowanie porównań strukturalnych, które w niektórych sytuacjach są prostsze.

11.4.6. Strukturalne porównania na potrzeby sprawdzania równości i sortowania

Oprócz standardowych interfejsów `IEquatable` i `IComparable` każda struktura `ValueTuple` bezpośrednio implementuje interfejsy `IStructuralEquatable` i `IStructuralComparable`. Te interfejsy istnieją od wersji .NET 4.0 i są implementowane w tablicach oraz niemodyfikowalnych klasach z rodziny `Tuple`. Wprawdzie sam nigdy nie używałem tych interfejsów, nie oznacza to jednak, że nie można ich stosować i to w przydatny sposób. Odzwierciedlają one zwykłe interfejsy API do porównywania i sortowania elementów, jednak każda metoda przyjmuje obiekt porównujący przeznaczony dla poszczególnych elementów:

```
public interface IStructuralEquatable
{
  bool Equals(Object, IEqualityComparer);
  int GetHashCode(IEqualityComparer);
}

public interface IStructuralComparable
{
  int CompareTo(Object, IComparer);
}
```

Omawiane interfejsy mają umożliwiać porównywanie złożonych obiektów na potrzeby sprawdzania równości i sortowania w wyniku porównywania par elementów za pomocą danego obiektu porównującego. Zwykłe generyczne porównania zaimplementowane w typach `ValueTuple` są statycznie bezpieczne ze względu na typ, ale stosunkowo mało elastyczne, ponieważ zawsze używany jest domyślny sposób porównywania elementów. Porównania strukturalne są mniej bezpieczne ze względu na typ, ale zapewniają dodatkową swobodę. Na listingu 11.10 pokazane jest to z użyciem łańcuchów znaków i obiektu porównującego ignorującego wielkość liter.

Listing 11.10. Porównania strukturalne z użyciem obiektu porównującego ignorującego wielkość liter

```
static void Main()
{
  var Ab = ("A", "b");
  var aB = ("a", "B");
  var aa = ("a", "a");
  var ba = ("b", "a");

  Compare(Ab, aB);
  Compare(aB, aa);
  Compare(aB, ba);
}

static void Compare<T>(T x, T y)
  where T : IStructuralEquatable, IStructuralComparable
{
  var comparison = x.CompareTo(
    y, StringComparer.OrdinalIgnoreCase);
  var equal = x.Equals(
    y, StringComparer.OrdinalIgnoreCase);

  Console.WriteLine(
    $"{x} i {y} - porównanie: {comparison}; równość: {equal}");
}
```

Nietypowe nazwy zmiennych odzwierciedlające wartości.

Wykonywanie wybranych interesujących porównań.

Sortowanie i sprawdzanie równości bez uwzględniania wielkości liter.

Dane wyjściowe z listingu 11.10 pokazują, że porównania rzeczywiście są przeprowadzane parami bez uwzględniania wielkości liter:

```
(A, b) i (a, B) - porównanie: 0; równość: True
(a, B) i (a, a) - porównanie: 1; równość: False
(a, B) i (b, a) - porównanie: -1; równość: False
```

Zaletą porównań tego rodzaju jest to, że wszystko sprowadza się do łączenia operacji. Obiekt porównujący wie tylko tyle, jak porównać wszystkie poszczególne elementy, a implementacja krotki deleguje wszystkie porównania do tego obiektu. Przypomina to nieco działanie technologii LINQ, gdzie zapisywane są operacje na poszczególnych elementach, jednak żądane jest wykonanie ich na kolekcjach.

Wszystko działa świetnie, jeśli krotki zawierają elementy tego samego typu. Jeżeli chcesz wykonywać porównania strukturalne na krotkach z elementami różnych rodzajów, np. porównywać wartości `(string, int, double)`, musisz się upewnić, że obiekt porównujący potrafi porównywać łańcuchy znaków, liczby całkowite i liczby zmiennoprzecinkowe o podwójnej precyzji. Jednak w każdym porównaniu trzeba uwzględnić tylko dwie wartości tego samego typu.

Implementacje typów `ValueTuple` umożliwiają porównywanie wyłącznie krotek z takimi samymi argumentami określającymi typ. Jeśli np. spróbujesz porównać krotkę typu `(string, int)` z krotką typu `(int, string)`, wyjątek zostanie zgłoszony natychmiast, przed porównaniem jakichkolwiek elementów. Omawianie przykładowego obiektu porównującego wykracza poza zakres tej książki, jednak w przykładowym kodzie źródłowym dołączonym do książki znajdziesz zarys takiego obiektu (typ `CompoundEqualityComparer`), który powinien być dobrym punktem wyjścia, gdybyś kiedyś potrzebował zaimplementować podobne rozwiązanie w kodzie produkcyjnym.

To kończy omawianie typów `ValueTuple<…>` o arności od 2 do 7. Wspomniałem jednak, że wrócę do trzech pozostałych typów wspomnianych w punkcie 11.4.1. Najpierw przyjrzyj się typom `ValueTuple<T1>` i `ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>`, które są ze sobą bardziej powiązane, niż może Ci się wydawać.

11.4.7. Krotki jednowartościowe i duże krotki

Krotek jednowartościowych (`ValueTuple<T1>`), nazywanych przez zespół projektujący C# *womple*, nie można tworzyć jako samodzielnych obiektów za pomocą składni dla krotek. Takie krotki muszą być częścią innej krotki. Wcześniej opisane zostało, że istnieją generyczne struktury typu `ValueTuple` przyjmujące tylko do ośmiu parametrów. Co ma zrobić kompilator C#, gdy natrafi na literał krotki obejmujący więcej niż osiem elementów? Używa wtedy typu `ValueTuple<…>` o arności 8, gdzie pierwszych siedem argumentów odpowiada pierwszym siedmiu typom z literału krotki, a ostatni element to zagnieżdżony typ krotki zawierający pozostałe elementy. Jeśli literał krotki ma dokładnie osiem elementów typu `int`, użyte zostaną następujące typy:

```
ValueTuple<int, int, int, int, int, int, int, ValueTuple<int>>
```

Występuje tu krotka jednowartościowa. Jest ona wyróżniona pogrubieniem. Typ `ValueTuple<…>` o arności 8 jest zaprojektowany specjalnie w tym celu. Ostatni argument określający typ (`TRest`) ma ograniczenie wymagające użycia typu bezpośredniego. Ponadto, o czym wspomniałem na początku punktu 11.4.1, nie istnieje pole `Item8`. Zamiast tego występuje pole `Rest`.

Ważne jest to, że ostatni element w typie `ValueTuple<…>` o arności 8 zawsze powinien być krotką wieloelementową, a nie pojedynczą wartością (aby uniknąć wieloznaczności). Spójrz na poniższy typ krotki:

```
ValueTuple<A, B, C, D, E, F, G, ValueTuple<H, I>>
```

Zgodnie ze składnią języka C# można go traktować jak typ `(A, B, C, D, E, F, G, H, I)` o arności 9 lub typ `(A, B, C, D, E, F, G, (H, I))` o arności 8, gdzie ostatni element jest typu krotki.

Programiści nie muszą się martwić tymi zagadnieniami, ponieważ kompilator C# umożliwia używanie nazw `ItemX` dla *wszystkich* elementów krotki niezależnie od ich liczby i tego, czy używasz składni dla krotek, czy bezpośrednio stosujesz typ `ValueTuple`. Przyjrzyj się np. tej dość długiej krotce:

```
var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16);
Console.WriteLine(tuple.Item16);
```

Jest to w pełni prawidłowy kod, przy czym wyrażenie `tuple.Item16` jest przekształcane przez kompilator na postać `tuple.Rest.Rest.Item2`. Jeśli chcesz stosować rzeczywiste nazwy pól, oczywiście możesz to robić, jednak nie zalecam tego. Przejdźmy teraz od długich krotek do ich całkowitego przeciwieństwa.

11.4.8. Niegeneryczna struktura ValueTuple

Jeśli krotki jednowartościowe wydają się zbędne, to krotki niegeneryczne niezawierające żadnych elementów (ang. *nuple*) można uznać za zupełnie bezsensowne. Możliwe, że oczekujesz, iż niegeneryczny typ `ValueTuple` jest klasą statyczną, podobną do niegenerycznej klasy `Nullable`. Jednak `ValueTuple` to struktura traktowana w innym kodzie w taki sam sposób jak pozostałe struktury krotek, jednak niezawierająca żadnych danych. Zaimplementowane są w niej wszystkie interfejsy opisane wcześniej w tym podrozdziale, ale każda wartość tego typu jest równa wszystkim pozostałym wartościom tego typu (zarówno ze względu na równość, jak i na potrzeby sortowania), co jest zrozumiałe, ponieważ nic nie różni takich krotek od siebie.

Omawiany typ udostępnia metody statyczne, które byłyby przydatne do tworzenia wartości typu `ValueTuple<…>`, gdyby nie istniały literały krotek. Te metody są pomocne zwłaszcza w sytuacji, jeśli chcesz używać typów krotek z C# 6 lub z innego języka bez wbudowanej obsługi krotek oraz gdy chcesz, by typy elementów zostały wywnioskowane (pamiętaj, że w wywołaniu konstruktora zawsze musisz podać wszystkie argumenty określające typ, co bywa irytujące). Na przykład aby w C# 6 utworzyć krotkę typu `(int, int)`, używając wnioskowania typów, możesz posłużyć się następującym zapisem:

```
var tuple = ValueTuple.Create(5, 10);
```

Zespół odpowiedzialny za C# sugeruje, że w przyszłości krotki niegeneryczne bez elementów mogą być przydatne do dopasowywania wzorców i dekompozycji. Jednak obecnie nie mają one praktycznych zastosowań.

11.4.9. Metody rozszerzające

Klasa statyczna `System.TupleExtensions` jest dostępna w tym samym podzespole co typy z rodziny `System.ValueTuple`. Zawiera ona metody rozszerzające dla typów `System.Tuple` i `System.ValueTuple`. Są trzy rodzaje takich metod:

- `Deconstruct` — rozszerzająca typy `Tuple`,
- `ToValueTuple` — rozszerzająca typy `Tuple`,
- `ToTuple` — rozszerzająca typy `ValueTuple`.

Każda z tych przeciążonych metod ma 21 wersji związanych z arnością, opartych na tym samym wzorcu, jaki poznałeś na potrzeby obsługi arności równej 8 i większej. Metody `Deconstruct` są opisane w rozdziale 12., a metody `ToValueTuple` i `ToTuple` działają dokładnie tak, jak możesz tego oczekiwać — przekształcają obiekty między niemodyfikowalnymi referencyjnymi typami krotek z czasów platformy .NET 4.0 a nowymi modyfikowalnymi bezpośrednimi typami krotek. Uważam, że te typy są przydatne przede wszystkim do pracy ze starszym kodem, gdzie używane są typy `Tuple`.

Uff! To już wszystko, co moim zdaniem warto wiedzieć na temat typów używanych do implementowania krotek w środowisku CLR. Dalej opisane zostaną inne możliwości. Jeśli zastanawiasz się nad użyciem krotek, powinieneś wiedzieć, że są one tylko jednym z narzędzi w Twoim przyborniku — i nie zawsze okazują się najbardziej odpowiednie.

11.5. Alternatywy dla krotek

Może wydać się to banalne, jednak każde rozwiązanie, które stosowałeś w przeszłości do zbiorów zmiennych, nadal jest akceptowalne. *Nie musisz* wszędzie używać krotek z C# 7. W tym podrozdziale pokrótce opisuję wady i zalety innych technik.

11.5.1. System.Tuple<...>

Typy `System.Tuple<…>` z platformy .NET 4 to niemodyfikowalne typy referencyjne (choć typy elementów mogą być modyfikowalne). Możesz uznać, że takie typy są niemodyfikowalne w „płytki" sposób, podobnie jak pola `readonly`.

Największą wadą tych typów jest brak integracji z językiem. Krotki w starszym stylu są trudniejsze do tworzenia, ich specyfikacje zajmują więcej miejsca, nie są dostępne konwersje opisane w podrozdziale 11.3 i, co najważniejsze, można stosować tylko nazwy w formacie `ItemX`. Choć nazwy stosowane w krotkach z C# 7 są używane tylko w czasie kompilacji, znacznie zwiększają użyteczność krotek.

Referencyjne typy krotek przypominają kompletne obiekty, a nie zbiory wartości. W zależności od kontekstu jest to wadą lub zaletą. Zwykle ich używanie jest mniej wygodne, jednak kopiowanie jednej referencji do dużego obiektu typu `Tuple<…>` jest dużo wydajniejsze niż kopiowanie obiektu typu `ValueTuple<…>`, co wymaga skopiowania wartości wszystkich elementów. Używanie typu referencyjnego korzystnie wpływa na pracę w środowisku wielowątkowym. Kopiowanie referencji odbywa się atomowo, natomiast kopiowanie krotek typu bezpośredniego — nie.

11.5.2. Typy anonimowe

Typy anonimowe wprowadzono w ramach technologii LINQ. Według mojego doświadczenia nadal stosuje się je głównie w tej technologii. Można używać ich dla zwykłych zmiennych w metodzie, jednak nie przypominam sobie, abym kiedykolwiek zetknął się z takim rozwiązaniem w kodzie produkcyjnym.

Większość zalet typów anonimowych (np. nazwane elementy, naturalne sprawdzanie równości i przejrzysta reprezentacja tekstowa) jest dostępna także w krotkach w C# 7. Główny problem z typami anonimowymi związany jest z ich anonimowością — wartości takich typów nie mogą być zwracane przez metody lub właściwości bez utraty bezpieczeństwa ze względu na typ. Trzeba byłoby wtedy użyć typu `object` lub `dynamic`. Informacje o typie byłyby dostępne w czasie wykonywania programu, jednak kompilator nie miałby do nich dostępu. Krotki z C# 7 nie mają tej wady. Jak już zobaczyłeś, dozwolone jest zwracanie krotek przez metody.

Dostrzegam jednak cztery zalety typów anonimowych w porównaniu z krotkami:

- W C# 7 inicjalizatory kolekcji pozwalają podać zarówno nazwę, jak i wartość za pomocą jednego identyfikatora, są więc prostsze niż krotki. Porównaj np. zapisy `new { p.Name, p.Age }` i `(name: p.Name, age: p.Age)`. Problem ten został rozwiązany w C# 7.1, ponieważ nazwy elementów krotek mogą zostać wywnioskowane, co pozwala uzyskać zwięzłą reprezentację, np. `(p.Name, p.Age)`.
- Używanie nazw w tekstowej reprezentacji typów anonimowych może być wygodne na potrzeby diagnostyki.
- Typy anonimowe są obsługiwane przez zewnętrznych dostawców w technologii LINQ (komunikujących się z bazami danych itd.). Literałów krotek nie można obecnie używać w drzewach wyrażeń, co zmniejsza ich atrakcyjność.
- Typy anonimowe w niektórych sytuacjach mogą się okazać bardziej wydajne, ponieważ w potoku operacji przekazywana jest tylko jedna referencja. Moim zdaniem w większości scenariuszy nie jest to jednak istotne, a to, że krotki nie powodują tworzenia obiektów, które musiałyby zostać usunięte przez mechanizm przywracania pamięci, oznacza oczywiście plus po stronie krotek.

Sądzę, że w technologii LINQ to Objects krotki będą powszechnie używane — przede wszystkim w połączeniu z wersją C# 7.1 i wnioskowaniem nazw elementów krotek.

11.5.3. Typy nazwane

Krotki to tylko zbiory zmiennych. Nie zapewniają hermetyzacji ani nie określają znaczenia zmiennych (sam decydujesz, do czego ich używasz). Czasem potrzebujesz właśnie takiego rozwiązania, jednak uważaj, aby nie posunąć się za daleko. Rozważ typ `(double, double)`. Można go używać dla:

- współrzędnych kartezjańskich w układzie dwuwymiarowym (x, y),
- współrzędnych biegunowych w układzie dwuwymiarowym (promień, kąt),
- pary początek/koniec na linii jednowymiarowej,
- rozmaitych innych danych.

Każde z tych zastosowań wymagałoby innych operacji, gdyby w modelu użyto typu w postaci klasy. Nie musiałbyś się wtedy martwić, że nazwy nie będą uwzględniane lub że przypadkowo użyjesz współrzędnych kartezjańskich zamiast biegunowych.

Jeśli chcesz tymczasowo pogrupować wartości lub tworzysz prototyp i nie jesteś pewien, czego potrzebujesz, krotki świetnie się sprawdzą. Jeżeli jednak zauważysz, że używasz krotek w tym samym kształcie w kilku miejscach kodu, zalecam zastąpienie ich typem nazwanym.

UWAGA. Fantastycznie byłoby móc użyć analizatora kodu Roslyn do zautomatyzowania takich zadań i wykrywać różne zastosowania krotek na podstawie nazw elementów. Niestety, nie znam żadnego narzędzia, które to potrafi.

Po tym wprowadzeniu do różnych możliwości zakończmy rozdział szczegółowymi zaleceniami na temat tego, gdzie krotki mogą być przydatne.

11.6. Zastosowania i rekomendacje

Przede wszystkim należy pamiętać, że obsługa krotek w języku jest nowością w C# 7. Wszelkie sugestie z tego podrozdziału są wynikiem *przemyśleń* na temat krotek, a nie długiego ich *użytkowania*. Analizy są wartościowe, jednak nie dają wglądu w praktykę. W przeszłości zdarzało mi się mylnie zakładać, do czego wykorzystam nowe funkcje języka w przyszłości, dlatego zachęcam do tego, by z dystansem traktować wszystko, co tu opisuję. Mimo to mam nadzieję, że ten materiał przynajmniej zachęci Cię do przemyśleń.

11.6.1. Niepubliczne interfejsy API i kod, który można łatwo modyfikować

Dopóki cała społeczność nie nabierze większego doświadczenia w używaniu krotek i dopóki nie powstaną dobre praktyki sprawdzone w boju, unikałbym stosowania krotek w publicznych interfejsach API, w tym jako składowych chronionych w typach, które mogą być klasami bazowymi w innych podzespołach. Jeśli masz to szczęście, że możesz kontrolować (i dowolnie modyfikować) cały kod, który komunikuje się z Twoim kodem, możesz pozwolić sobie na więcej eksperymentów. Unikaj jednak zwracania krotek w metodach publicznych tylko dlatego, że jest to łatwe rozwiązanie; w przyszłości możesz odkryć, że warto zapewnić większą hermetyzację zwracanych wartości. Typy nazwane wymagają więcej pracy w zakresie projektowania i implementowania, jednak uzyskany efekt zapewne nie będzie trudniejszy w użyciu dla jednostki wywołującej. Krotki są wygodne głównie dla piszących je osób, a nie dla autorów wywołującego je kodu.

Obecnie preferuję jeszcze bardziej radykalne rozwiązanie i używam krotek tylko jako szczegółu implementacji w typach. Bez obaw zwracam krotki w metodach prywatnych, ale nie robię tego w metodach wewnętrznych w kodzie produkcyjnym. Zwykle im bardziej ograniczony jest zakres działania danego kodu, tym łatwiej jest zmienić decyzję i wymaga to mniej zastanawiania się.

11.6.2. Zmienne lokalne

Krotki zaprojektowano głównie po to, aby umożliwić zwracanie przez metodę wartości wielu typów bez używania parametrów `out` lub tworzenia specjalnego typu dla zwracanej wartości. Nie oznacza to jednak, że są to jedyne zastosowania krotek.

Nierzadko się zdarza, że w metodzie znajduje się naturalna grupa zmiennych. Często świadczy o tym wspólny przedrostek w nazwach takich metod. Na przykład na listingu 11.11 znajduje się metoda, która może pochodzić z kodu gry i wyświetlać gracza z najwyższym osiągniętym do danej daty wynikiem. Choć w technologii LINQ to Objects znajduje się metoda `Max`, która zwraca najwyższą wartość na potrzeby projekcji, nie ma metody zwracającej element sekwencji powiązany z tą wartością.

UWAGA. Inna możliwość to użycie metody `OrderByDescending(…).FirstOrDefault()`, jednak wymaga to sortowania danych, podczas gdy potrzebna jest tylko jedna wartość. Pakiet MoreLinq udostępnia metodę `MaxBy`, która zapełnia opisaną lukę. Jeszcze inna możliwość (oprócz przechowywania dwóch zmiennych) polega na utworzeniu jednej zmiennej `highestGame` i używaniu w porównaniach jej właściwości `Score`. W bardziej złożonych scenariuszach takie rozwiązanie może okazać się nieakceptowalne.

Listing 11.11. Wyświetlanie gracza, który do danej daty uzyskał najwyższy wynik

```
public void DisplayHighScoreForDate(LocalDate date)
{
  var filteredGames = allGames.Where(game => game.Date == date);
  string highestPlayer = null;
  int highestScore = -1;
  foreach (var game in filteredGames)
  {
    if (game.Score > highestScore)
    {
      highestPlayer = game.PlayerName;
      highestScore = game.Score;
    }
  }
  Console.WriteLine(highestPlayer == null
    ? "Nie rozegrano żadnej gry"
    : $"Najwyższy wynik to {highestScore} osięgnięty przez {highestPlayer}");
}
```

Używane są tu cztery zmienne lokalne (włącznie z parametrem):

- `date`
- `filteredGames`
- `highestPlayer`
- `highestScore`

Przynajmniej dwie z tych zmiennych są ze sobą ściśle powiązane. Są razem inicjowane i modyfikowane. To sugeruje, że mógłbyś *rozważyć* zastosowanie zmiennej w postaci krotki, tak jak na listingu 11.12.

Listing 11.12. Refaktoryzacja w celu użycia zmiennej lokalnej w postaci krotki

```
public void DisplayHighScoreForDate(LocalDate date)
{
  var filteredGames = allGames.Where(game => game.Date == date);
  (string player, int score) highest = (null, -1);
  foreach (var game in filteredGames)
  {
```

```
    if (game.Score > highest.score)
    {
      highest = (game.PlayerName, game.Score);
    }
  }
  Console.WriteLine(highest.player == null
    ? "Nie rozegrano żadnej gry"
    : $"Najwyższy wynik to {highest.score} osiągnięty przez {highest.player}");
}
```

Zmiany zostały wyróżnione pogrubieniem. Czy nowa wersja jest lepsza? Możliwe. Na poziomie „filozoficznym" jest to dokładnie ten sam kod, jeśli potraktować krotkę jak kolekcję zmiennych. Ta wersja wydaje mi się bardziej przejrzysta, ponieważ zmniejsza liczbę jednostek używanych w metodzie na ogólnym poziomie. Oczywiście w tego rodzaju uproszczonych przykładach odpowiednich dla książek różnice w przejrzystości są zwykle niewielkie. Gdybyś jednak używał skomplikowanej metody, której nie da się podzielić na kilka mniejszych metod, zmienne lokalne w postaci krotek mogłyby poprawić czytelność w większym stopniu. Podobne rozważania dotyczą także pól.

11.6.3. Pola

Pola, podobnie jak zmienne lokalne, czasem w naturalny sposób są ze sobą powiązane. Oto przykład z klasy `PrecalculatedDateTimeZone` z biblioteki Noda Time:

```
private readonly ZoneInterval[] periods;
private readonly IZoneIntervalMapWithMinMax tailZone;
private readonly Instant tailZoneStart;
private readonly ZoneInterval firstTailZoneInterval;
```

Nie zamierzam omawiać znaczenia wszystkich tych pól. Mam jednak nadzieję, że jest widoczne, iż trzy ostatnie są związane z końcową strefą czasową (ang. *tail zone*). Można rozważyć przekształcenie czterech pokazanych pól na dwa, z których jedno będzie krotką:

```
private readonly ZoneInterval[] periods;
private readonly
  (IZoneIntervalMapWithMinMax intervalMap,
   Instant start,
   ZoneInterval firstInterval) tailZone;
```

W dalszym kodzie można wtedy używać pól `tailZone.start`, `tailZone.intervalMap` itd. Ponieważ zmienna `tailZone` jest zadeklarowana z modyfikatorem `readonly`, wartości do jej elementów można przypisywać wyłącznie w konstruktorze. Występuje tu kilka ograniczeń i zastrzeżeń:

- Wartości pojedynczych elementów krotki można przypisywać w konstruktorze, jednak gdy zainicjalizujesz tylko niektóre elementy (a nie wszystkie), nie jest zgłaszane ostrzeżenie. Na przykład jeśli w pierwotnym kodzie zapomnisz zainicjalizować pole `tailZoneStart`, pojawi się ostrzeżenie. Nie zobaczysz jednak analogicznego komunikatu, jeśli zapomnisz zainicjalizować pole `tailZone.start`.
- Albo wszystkie elementy pola z krotką są tylko do odczytu, albo żaden z tych elementów nie ma tej cechy. Jeśli istnieje grupa powiązanych pól i jedne z nich

są tylko do odczytu, a inne nie, to albo musisz pominąć ten aspekt, albo zrezygnować z używania krotki. Ja zwykle rezygnuję wtedy z krotki.
- Jeśli niektóre pola są automatycznie generowane i powiązane z automatycznie implementowanymi właściwościami, trzeba napisać kompletną właściwość, aby móc używać krotki. W takiej sytuacji zrezygnowałbym z używania krotek.

Jednym z mniej oczywistych aspektów krotek jest ich współdziałanie z typowaniem dynamicznym.

11.6.4. Krotki i typowanie dynamiczne nie współdziałają dobrze ze sobą

Rzadko używam typu `dynamic` i podejrzewam, że przydatne zastosowania typowania dynamicznego i właściwe zastosowania krotek pokrywają się w niewielkim stopniu. Warto jednak poznać dwa problemy związane z dostępem do elementów.

DYNAMICZNY BINDER NIE ZNA NAZW ELEMENTÓW

Warto pamiętać, że nazwy elementy są uwzględniane głównie w czasie kompilacji. Jeśli połączysz to z wiedzą, że wiązanie dynamiczne ma miejsce tylko w czasie wykonywania programu, podejrzewam, że domyślisz się skutków. W ramach prostego przykładu rozważ następujący kod:

```
dynamic tuple = (x: 10, y: 20);
Console.WriteLine(tuple.x);
```

Można oczekiwać, że ten kod wyświetli liczbę 10. W rzeczywistości zgłaszany jest wyjątek:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
  'System.ValueTuple<int,int>' does not contain a definition for 'x'
```

Choć jest to niefortunne, zachowywanie informacji o nazwach elementów dla dynamicznego bindera wymagałoby dużo zachodu. Nie spodziewam się zmian w tym obszarze. Jeśli zmodyfikujesz ten fragment tak, aby wyświetlał element `tuple.Item1`, kod zadziała poprawnie — a przynajmniej dla pierwszych siedmiu elementów.

DYNAMICZNY BINDER (OBECNIE) NIE ZNA WYSOKICH NUMERÓW ELEMENTÓW

W punkcie 11.5.4 zobaczyłeś, w jaki sposób kompilator traktuje krotki mające więcej niż siedem elementów. Kompilator używa typu `ValueTuple<…>` o arności 8, gdzie ostatni element zawiera krotkę dostępną za pomocą pola `Rest` zamiast pola `Item8`. Oprócz modyfikowania samego typu kompilator zmienia sposób dostępu do numerowanych elementów. Na przykład jeśli w kodzie źródłowym używany jest element `tuple.Item9`, w wygenerowanym kodzie pośrednim jest to element `tuple.Rest.Item2`.

W czasie, gdy powstaje ta książka, dynamiczny binder nie zna tych możliwości. Dlatego wystąpi wyjątek, choć ten sam kod wiązany w czasie kompilacji działałby poprawnie. W ramach przykładu możesz łatwo przetestować tę technikę i poeksperymentować z nią:

```
var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9);
Console.WriteLine(tuple.Item9); ←——— Działa (używany jest element tuple.Rest.Item2).
dynamic d = tuple;
Console.WriteLine(d.Item9); ←——— Błąd w czasie wykonywania programu.
```

Ten problem (w odróżnieniu od poprzedniego) można rozwiązać, usprawniając dynamiczny binder. Jednak działanie programu będzie wtedy zależeć od używanej wersji dynamicznego bindera. Zwykle wyraźnie określone jest, z której wersji kompilatora korzystasz oraz jakich wersji podzespołu i platformy używasz. Wymóg stosowania określonej wersji dynamicznego bindera nieco komplikuje sytuację.

Podsumowanie

- Krotki pełnią funkcje zbioru elementów bez hermetyzacji.
- Krotki w C# 7 mają inne reprezentacje w języku i w środowisku CLR.
- Krotki są typami bezpośrednimi z publicznymi i modyfikowalnymi polami.
- Krotki w C# umożliwiają nadawanie nazw elementom.
- W strukturach `ValueTuple<…>` w środowisku CLR nazwy elementów to zawsze `Item1`, `Item2` itd.
- C# umożliwia konwersje typów krotek i literałów krotek.

od podszewki

Wydanie IV

C# liczy sobie około dwudziestu lat. Jest niestrudzenie rozwijany i doskonalony przez Microsoft, a dzięki swojej wszechstronności znajduje zastosowanie w wielu dziedzinach: pisaniu gier komputerowych, tworzeniu skalowalnych i niezawodnych aplikacji internetowych oraz aplikacji mobilnych, a nawet niskopoziomowym programowaniu komponentów większych systemów. Twórcy C# postawili na obiektowość, ścisłą kontrolę typów, a przede wszystkim na prostotę w stosowaniu. W tym celu wykorzystano wyniki badań akademickich i połączono je z praktycznymi technikami rozwiązywania problemów. W efekcie C# stał się ulubionym językiem profesjonalistów.

To czwarte wydanie podręcznika przeznaczonego dla programistów C#, którzy znają podstawy tego języka, jednak zależy im na dogłębnym zrozumieniu ważnych pojęć i przyswojeniu różnych sposobów myślenia o pozornie znanych zagadnieniach. W książce skrótowo opisano wersje C# od 2 do 5, a wyczerpująco omówiono wersje od 6 do 7.3. Zaprezentowano również niektóre informacje o projektowanych nowych elementach języka C# 8, takich jak typy referencyjne przyjmujące wartość `null`, wyrażenia `switch`, usprawnienia dopasowywania wzorców, a także dalsza integracja asynchroniczności z podstawowymi mechanizmami języka. Poszczególne treści zilustrowano licznymi przykładami kodu źródłowego.

W tej książce między innymi:

- wyrażenia lambda, inicjalizatory zapytań, asynchroniczność
- składowe z ciałem w postaci wyrażenia
- zaawansowane techniki pracy z ciągami znaków
- zagadnienia integracji krotek z językiem
- dekonstruktory i dopasowywanie wzorców
- nowe techniki stosowania referencji i powiązanych mechanizmów

Jon Skeet jest starszym inżynierem oprogramowania w firmie Google. Odpowiada za tworzenie bibliotek klienckich dla .NET na platformie Google Cloud. Bierze również udział w tworzeniu standardu języka C# w organizacji ECMA. Jest programistą o niemal legendarnych umiejętnościach tworzenia kodu, przy tym chętnie dzieli się wiedzą i doświadczeniem z innymi. Ma nietypowe hobby: interesuje się zagadnieniami pomiaru czasu i dat.

C#. Programowanie na najwyższym poziomie!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-6029-7

9 788328 360297

Cena: 99,00 zł