Pierre-Yves Saumont

Java

Programowanie funkcyjne

Helion

Tytuł oryginału: Functional Programming in Java: How to improve your Java programs using functional techniques

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-3324-6

Wydawnictwo HELION
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javapf.zip

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

Printed in Poland.

Spis treści

2 Użycie funkcji w języku Java

W tym rozdziale:

- działanie funkcji w rzeczywistym świecie;
- sposób reprezentacji funkcji w Javie;
- użycie lambd;
- praca z funkcjami wyższego rzędu;
- rozwijanie funkcji (ang. *currying*);
- programowanie z użyciem interfejsów funkcyjnych.

Aby zrozumieć, jak działa programowanie funkcyjne, moglibyśmy użyć komponentów funkcyjnych zapewnianych przez biblioteczkę stworzoną do tego celu (powstało nawet kilka nakierowanych na Javę 8). Zamiast tego postaramy się wszystko skonstruować samodzielnie i nie korzystać z gotowych komponentów. Po opanowaniu wszystkich elementów będziesz mógł sam wybrać między własnymi funkcjami a tymi zapewnianymi przez Javę 8 czy też przez zewnętrzne biblioteki. W tym rozdziale wykonamy interfejs `Function`, bardzo podobny do interfejsu `Function` z Javy 8. Rozwiązanie będzie uproszczone w kwestii obsługi parametrów typów (unikamy elementów wieloznacznych), aby ułatwić zrozumienie kodu, ale z drugiej strony będzie zawierało kilka funkcjonalności, których brakuje w wersji dostępnej w Javie 8. Poza tymi różnicami oba rozwiązania będą w zasadzie wymienne.

Mogą pojawić się trudności ze zrozumieniem niektórych fragmentów kodu przedstawianych w tym rozdziale. To normalne, ponieważ bardzo trudno wprowadzić funkcje bez korzystania z innych konstrukcji funkcyjnych, takich jak `List`, `Option` itp. Bądź cierpliwy. Wszystkie nieopisane tu elementy zostaną wyjaśnione w następnych rozdziałach.

Wyjaśnię bardzo szczegółowo, czym jest funkcja — zarówno w świecie rzeczywistym, jak i w języku programowania. Funkcje nie są tylko czymś, co istnieje w matematyce czy języku programowania. Funkcje stanowią część codziennego życia. Cały czas modelujemy świat, w którym żyjemy; nie dotyczy to tylko programowania. Tworzymy pewne interpretacje świata wokół nas. Reprezentacje świata bardzo często bazują na obiektach, które modyfikują swój stan wraz ze zmianą czasu. Ten sposób interpretacji leży w naturze człowieka. Przejście od stanu A do stanu B wymaga czasu i ma związany z tym koszt w postaci czasu, wysiłku i pieniędzy.

Weźmy jako przykład dodawanie. Większość z nas traktuje dodawanie jako obliczenie wymagające czasu (a w pewnych sytuacjach nawet wysiłku intelektualnego!). Ma pewien stan początkowy, przejście (obliczenia) i stan końcowy (wynik dodawania).

Aby dodać do siebie 345 765 i 34 524, z pewnością musimy wykonać pewne obliczenia. Niektórym zajmie to tylko chwilkę, innym nieco więcej czasu. Niektórym nigdy się to nie uda lub otrzymają błędny wynik. Niektórzy do obliczeń będą potrzebowali kartki i ołówka, a innym wystarczy głowa. Wszyscy z pewnością w trakcie obliczeń będą zmieniać stan, niezależnie do tego, czy w głowie, czy na papierze. Z drugiej strony, aby dodać 2 do 3, nie potrzebujemy tego wszystkiego. Większość z nas zna odpowiedź na pamięć, więc może jej udzielić bez przeprowadzania jakichkolwiek obliczeń.

Ten przykład pokazuje, że obliczenia nie są elementem niezbędnym. Stanowią jedynie środek do wskazania wyniku funkcji. Wynik istniał, zanim dokonaliśmy obliczeń — po prostu jeszcze go nie znaliśmy.

Programowanie funkcyjne to programowanie z użyciem funkcji. Aby go użyć, musimy najpierw zrozumieć, czym jest funkcja — zarówno w świecie rzeczywistym, jak i w wybranym języku programowania.

2.1. Czym jest funkcja?

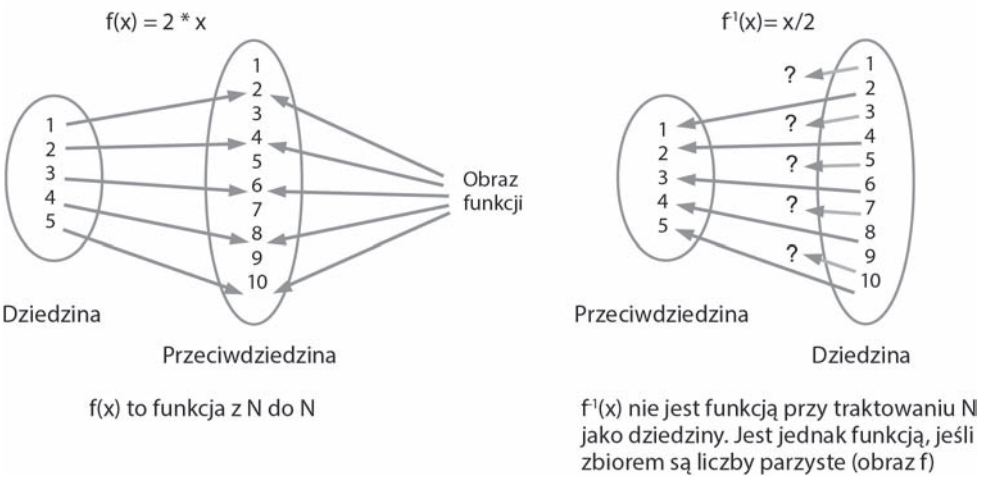
Funkcja znana jest jako pewien byt matematyczny, choć sama koncepcja dotyczy też życia codziennego. Niestety, w życiu codziennym bardzo często mylimy funkcję i efekty. Co gorsza, ten sam błąd popełniamy również w trakcie korzystania z języków programowania.

2.1.1. Funkcje w świecie rzeczywistym

W świecie rzeczywistym funkcja to przede wszystkim koncepcja matematyczna. To związek między zbiorem źródłowym, nazywanym **dziedziną** funkcji, a zbiorem docelowym, nazywanym **przeciwdziedziną** funkcji. Dziedzina i przeciwdziedzina nie muszą się różnić. Na przykład, funkcja może posiadać ten sam zbiór liczb całkowitych w dziedzinie i przeciwdziedzinie.

CO CZYNI RELACJĘ MIĘDZY DWOMA ZBIORAMI FUNKCJĄ?

Relacja, aby byłą funkcją, musi spełnić jeden warunek — wszystkie elementy dziedziny muszą mieć jeden i tylko jeden odpowiadający im element w przeciwdziedzinie, co przedstawia rysunek 2.1.



Rysunek 2.1. Wszystkie elementy dziedziny muszą mieć jeden i tylko jeden odpowiadający im element w przeciwdziedzinie

Ma to pewne interesujące implikacje:

- Nie może istnieć element dziedziny, który nie posiada odpowiadającej mu wartości w przeciwdziedzinie.
- Nie mogą istnieć dwa elementy w przeciwdziedzinie odpowiadające temu samemu elementowi dziedziny.
- Mogą istnieć w przeciwdziedzinie elementy, którym nie odpowiada żaden element zbioru źródłowego.
- Mogą istnieć w przeciwdziedzinie elementy, które odpowiadają kilku elementom ze zbioru źródłowego.
- Zbiór elementów przeciwdziedziny, które posiadają odpowiadające im elementy dziedziny, nazywa się **obrazem** funkcji.

Rysunek 2.1 ilustruje funkcję.

Zdefiniujmy następującą funkcję:

```
f(x) = x + 1
```

w której x to liczba dodatnia. Funkcja reprezentuje związek między każdą liczbą dodatnią i jej następczynią. Możemy funkcji nadać nazwę. W szczególności możemy nadać nazwę, która pozwoli przypomnieć sobie działanie funkcji:

```
następca(x) = x + 1
```

Wydaje się to dobrym pomysłem, ale nie należy ślepo ufać nazwie funkcji. Przecież ktoś mógł nazwać funkcję w poniższy sposób:

```
poprzednik(x) = x + 1
```

Nie jest to błąd, bo nie istnieje żaden oficjalny związek między nazwą funkcji a jej definicją. Oczywiście, użycie takiej nazwy nie jest dobrym pomysłem.

Zauważ, że mówimy tutaj o tym, czym funkcja jest (jej definicja), a nie o tym, co robi. Funkcja nic nie robi. Funkcja `następca` nie dodaje 1 do argumentu. **Ty** możesz dodać 1 do wartości całkowitej i obliczyć następcę, ale **Ty** nie jesteś funkcją. Funkcja:

```
następca(x)
```

nie dodaje `1` do `x`. Jest jedynie równoważna `x + 1`, co oznacza, że za każdym razem, gdy natkniesz się na wyrażenie `następca(x)`, możesz je zamienić na `(x + 1)`.

Nawiasy stosuje się tylko w celu odizolowania wyrażenia. Jeśli wyrażenie stosuje się w odosobnieniu, są zbędne, choć w wielu momentach rozjaśniają sytuację.

FUNKCJE ODWROTNE

Funkcja może posiadać funkcję odwrotną, ale nie musi. Jeśli `f(x)` to funkcja od `A` do `B` (`A` jest dziedziną, a `B` przeciwdziedziną), funkcję odwrotną zapisuje się jako $f^{-1}(x)$ (teraz `B` jest dziedziną, a `A` przeciwdziedziną). Jeśli typ funkcji wyrazimy jako `A -> B`, to funkcja odwrotna (jeśli istnieje) ma typ `B -> A`.

Funkcja odwrotna jest funkcją, jeśli spełni te same warunki, jak każda inna funkcja, czyli jedna i tylko jedna wartość docelowa dla każdej źródłowej. Oznacza to, że funkcją odwrotną dla `następca(x)` będzie relacja `poprzednik(x)` (oczywiście nazwa jest dowolna). Nie jest to jednak funkcja w `N` (zbiór liczb całkowitych dodatnich włącznie z 0), ponieważ dla 0 nie ma w `N` poprzednika. Jeśli jednak funkcję `następca(x)` rozważamy w zbiorze liczb całkowitych ze znakiem (wartości dodatnie i ujemne, oznaczane jako `C`), posiada ona funkcję odwrotną w postaci `poprzednik(x)`.

Niektóre proste funkcje nie posiadają funkcji odwrotnych. Oto przykład:

```
f(x) = (2 * x)
```

Powyższa funkcja nie ma funkcji odwrotnej, jeśli jest definiowana jako przejście z `N` do `N`. Posiada jednak funkcję odwrotną, jeśli stanowi funkcję przejścia z `N` do zbioru liczb całkowitych parzystych.

FUNKCJE CZĘŚCIOWE

Relacja, która nie jest zdefiniowana dla wszystkich elementów dziedziny, ale która spełnia pozostałe wymagania (żaden element dziedziny nie ma więcej niż jednej relacji z elementem z przeciwdziedziny), nazywana jest często **funkcją częściową**. Relacja `poprzednik(x)` jest funkcją częściową w zbiorze `N` (liczby dodatnie i 0), ale jest funkcją pełną w zbiorze `N+` (liczby całkowite dodatnie bez 0). Jej przeciwdziedziną jest `N`.

Funkcje częściowe są bardzo ważne w trakcie programowania, ponieważ wiele błędów wynika z faktu, iż użyto funkcji częściowej w taki sposób, jakby była funkcją pełną. Przykładowo, relacja `f(x) = 1/x` jest funkcją częściową z `N` do `W` (liczby wymierne), ponieważ nie jest zdefiniowana dla 0. Jest funkcją pełną z `N+` do `W`, jak i dla przejścia z `N` do `W` plus błąd. Dodając element do przeciwdziedziny (błąd), możemy przekształcić funkcję częściową w funkcję pełną. Oznacza to jednak, że funkcja potrzebuje jakiegoś sposobu, aby zwrócić błąd. Czy widzisz już analogię do programów komputerowych? Przekonasz się, że zamiana funkcji częściowych na pełne stanowi jeden z istotnych elementów programowania funkcyjnego.

ZŁOŻENIE FUNKCJI

Funkcje to bloczki, które można połączyć w celu zbudowania innych funkcji. Złożenie funkcji `f` i `g` zapisuje się jako `f ∘ g`, który czyta się `f` **od** `g`. Jeśli `f(x) = x + 2` i `g(x) = x * 2`, wtedy:

```
f ∘ g (x) = f(g(x)) = f(x * 2) = (x * 2) + 2
```

Zauważ, że zapisy `f ∘ g (x)` i `f(g(x))` są równoważne. Jednak zapis kompozycji jako `f(g(x))` wskazuje, że używa się `x` jako miejsca dla argumentu. Zapis `f ∘ g` pozwala na określenie złożenia funkcji bez wskazywania elementu tymczasowego.

Jeśli użyjemy tej funkcji dla wartości 5, otrzymamy:

```
f ∘ g (5) = f(g(5)) = f(5 * 2) = 10 + 2 = 12
```

Warto zwrócić uwagę, że `f ∘ g` różni się od `g ∘ f`, choć czasem mogą być sobie równoważne. Oto przykład:

```
g ∘ f (5) = g(f(5)) = g(5 + 2) = 7 * 2 = 14
```

Funkcje stosuje się odwrotnie do kolejności zapisu. Zapis `f ∘ g` oznacza, że najpierw stosuje się `g`, a potem `f`. Standardowe funkcje Javy 8 definiują metodę `compose()` i metodę `andThen()`, aby obsłużyć oba przypadki. W praktyce nie jest to potrzebne, bo `f.andThen(g)` oznacza `g.compose(f)` lub `f ∘ g`.

FUNKCJE Z KILKOMA ARGUMENTAMI

Na razie mówiliśmy jedynie o funkcjach z jednym argumentem. A co z funkcjami posiadającymi kilka argumentów? Tak naprawdę nie istnieje funkcja z kilkoma argumentami. Przypomnę definicję: funkcja to relacja pomiędzy zbiorem źródłowym i zbiorem docelowym. Nie jest to relacja między kilkoma zbiorami wejściowymi i zbiorem docelowym. Funkcja nie może mieć kilku argumentów.

Iloczyn dwóch zbiorów również jest zbiorem, więc funkcja korzystająca z takiego iloczynu zbiorów może się wydawać funkcją przyjmującą kilka argumentów. Rozważmy następującą sytuację:

```
f(x, y) = x + y
```

Jest to relacja między `N × N` i `N`, czyli mamy do czynienia z funkcją. Istnieje jednak tylko jeden argument — jest nim `N × N`.

`N × N` to zbiór wszystkich możliwych par liczb całkowitych. Elementami takiego zbioru są pary liczb całkowitych. Para to specjalny przypadek bardziej ogólnej koncepcji nazywanej **krotką**, która reprezentuje połączenie kilku elementów. Para to dwuelementowa krotka.

Krotki zapisuje się w nawiasach, więc `(3, 5)` to krotka i element zbioru `N × N`. Możemy dla tej krotki użyć funkcji `f`:

```
f((3, 5)) = 3 + 5 = 8
```

W takiej sytuacji przyjmuje się, że zgodnie z konwencją jeden zestaw nawiasów jest zbędny, więc powstaje zapis:

```
f(3, 5) = 3 + 5 = 8
```

Nadal jest to jednak funkcja z jedną krotką, a nie funkcja z dwoma argumentami.

ROZWIJANIE FUNKCJI

Funkcje z krotkami możemy potraktować nieco inaczej. Funkcję `f(5, 3)` możemy zdefiniować jako funkcję z `N` do zbioru funkcji od `N`. Poprzedni przykład moglibyśmy więc zapisać następująco:

```
f(x)(y) = g(y)
```

gdzie:

```
g(y) = x + y
```

W takiej sytuacji możemy napisać:

```
f(x) = g
```

Oznacza to, że wynikiem zastosowania funkcji `f` dla argumentu `x` jest nowa funkcja `g`. Zastosowanie funkcji `g` dla `y` daje wynik:

```
g(y) = x + y
```

W przypadku stosowania `g`, `x` nie jest już dostępne. Nie zależy od argumentu lub czegokolwiek innego. To stała. Stosując to dla przykładu z (3, 5), otrzymujemy:

```
f(3)(5) = g(5) = 3 + 5 = 8
```

Nowym elementem jest jedynie to, że przeciwdziedzina `f` to zbiór funkcji, a nie zbiór liczb. Wynikiem zastosowania `f` dla liczby całkowitej jest funkcja. Wynikiem zastosowania tej funkcji dla liczby całkowitej jest liczba całkowita.

Postać `f(x)(y)` to **rozwinięta** forma funkcji `f(x, y)`. Zastosowanie tego przekształcenia dla funkcji krotki (jeśli chcesz, możesz użyć nazwy „funkcja wieloargumentowa") nazywa się rozwinięciem funkcji (ang. *currying*). Wersja angielska nazwy pochodzi od nazwiska matematyka Haskella Curry'ego (choć to nie on wymyślił to przekształcenie).

CZĘŚCIOWO ZASTOSOWANA FUNKCJA

Postać rozwinięta funkcji w postaci dodatkowej funkcji pośredniej może nie wydawać się naturalna, więc zapewne zastanawiasz się, jak odnieść ją do czegoś ze świata rzeczywistego — przecież w tej wersji każdy z argumentów rozważa się osobno. Najpierw obsługuje się jeden argument, czego efektem jest nowa funkcja. Czy ta nowa funkcja jest użyteczna sama z siebie, czy też stanowi jedynie krok w większych obliczeniach?

W przypadku dodawania faktycznie nie jest zbyt użyteczna. Tak przy okazji — można rozpocząć od któregokolwiek z argumentów. Nie ma to znaczenia, bo choć funkcja pośrednia będzie inna, wynik końcowy nie ulegnie zmianie.

Rozważmy nową funkcję na parze wartości:

```
f(procent, cena) = cena / 100 * (100 + procent)
```

Funkcja ta jest równoważna funkcji:

```
g(cena, procent) = cena / 100 * (100 + procent)
```

Oto obie funkcje po rozwinięciu:

```
f(procent, cena)
g(cena, procent)
```

Wiemy, że `f` i `g` to funkcje. Czym są jednak `f(procent)` i `g(cena)`? Z pewnością są efektem zastosowania `f` dla `procent` i `g` dla `cena`. Jaki to jednak typ wyniku?

`f(procent)` to funkcja zmiany z jednej ceny na inną. Jeśli `procent = 9`, funkcja stosuje podatek wynoszący 9% dla zadanej ceny, czym tworzy nową cenę. Wynikową funkcję można by nazwać `zastosujDziewięcioprocentowyPodatek(cena)`. Byłoby to dosyć użyteczne narzędzie, jeśli stawka podatku nie zmienia się zbyt często.

Z drugiej strony, `g(cena)` to funkcja zmiany z wartości procentowej na cenę. Jeśli cena wynosi 100 złotych, nowa funkcja zastosuje wskazaną stawkę podatku dla ceny wynoszącej 100 złotych. Jak nazwałbyś taką funkcję? Jeśli nie potrafisz wymyślić sensownej nazwy, oznacza to zapewne, że taka postać nie ma sensu (choć wiele zależy od rodzaju rozwiązywanego problemu).

Funkcje takie jak `f(procent)` i `g(cena)` nazywa się często **funkcjami zastosowanymi częściowo**, aby odróżnić je od wersji `f(procent, cena)` i `g(cena, procent)`. Częściowo zastosowane funkcje mogą mieć duży wpływ na obsługę argumentu. Wrócimy do tego tematu w dalszej części książki.

Jeśli masz problem ze zrozumieniem rozwijania funkcji, wyobraź sobie, że podróżujesz do innego kraju i masz przy sobie kalkulator (lub smartfon) pozwalający na konwersję z jednej waluty na inną. Czy chciałbyś wpisywać za każdym razem wartość przelicznika, czy raczej zapisałbyś ją w pamięci kalkulatora? Które z tych rozwiązań byłoby mniej narażone na błędy?

FUNKCJE NIE MAJĄ EFEKTÓW

Pamiętaj, że czyste funkcje jedynie zwracają wartość — nie mają żadnych efektów ubocznych. Nie modyfikują stanu żadnego elementu ze świata zewnętrznego (przez świat **zewnętrzny** rozumiemy elementy poza samą funkcją), nie modyfikują przekazanych argumentów i nie eksplodują (zgłaszają wyjątku), jeśli pojawi się błąd. Mogą jednak zwrócić wyjątek jako wynik wraz z informacją o błędzie. Muszą go jednak zwrócić, a nie zgłosić (rzucić), umieścić w dzienniku zdarzeń lub wypisać na ekranie.

2.2. Funkcje w Javie

W rozdziale 1. używałeś czegoś, co nazwałem **funkcjami**, choć tak naprawdę były to metody. Metody to sposób reprezentacji (do pewnego stopnia) funkcji w języku Java.

2.2.1. Metody funkcyjne

Metoda może być funkcyjna, jeśli stosuje się do zasad obowiązujących czyste funkcje:

- Nie może modyfikować niczego poza funkcją; żadne zmiany wewnątrz funkcji nie mogą wyciekać na zewnątrz.
- Nie może modyfikować argumentów.
- Nie może rzucać wyjątków lub błędów.
- Musi zawsze zwracać wartość.
- Po wywołaniu z tymi samymi argumentami musi zawsze zwrócić ten sam wynik.

Przyjrzyjmy się przykładowi z listingu 2.1.

Listing 2.1. Metody funkcyjne

```
public class FunctionalMethods {

   public int percent1 = 5;
   private int percent2 = 9;
   public final int percent3 = 13;

   public int add(int a, Integer b) {
      return a + b;
   }

   public int mult(int a, Integer b) {
      a = 5;
      b = 2;
     return a * b;
   }

   public int div(int a, int b) {
      return a / b;
   }

   public int applyTax1(int a) {
      return a / 100 * (100 + percent1);
   }

   public int applyTax2(int a) {
      return a / 100 * (100 + percent2);
   }

   public int applyTax3(int a) {
      return a / 100 * (100 + percent3);
   }

   public List<Integer> append(int i, List<Integer> list) {
      list.add(i);
      return list;
   }

   public List<Integer> append2(int i, List<Integer> list) {
      List<Integer> result = new ArrayList<>();
      result.add(i);
      percent2++;
      return result;
   }
}
```

Czy możesz wskazać, które z tych metod reprezentują czyste funkcje? Zastanów się nad tym kilka minut, zanim zaczniesz czytać odpowiedzi umieszczone poniżej. Pomyśl o wszystkich warunkach i całym przetwarzaniu danych umieszczonym w każdej z funkcji. Pamiętaj, że to, co ma znaczenie, dotyczy widoczności na zewnątrz. Nie zapomnij o uwzględnieniu wyjątków.

Przyjrzyjmy się pierwszej metodzie:

```
public int add(int a, int b) {
   return a + b;
}
```

Metoda `add` jest funkcją, ponieważ zawsze zwraca wartość, która zależy tylko od jej argumentów. Nie modyfikuje argumentów i nie wchodzi w żaden sposób w interakcję ze światem zewnętrznym. Metoda może spowodować błąd, jeśli suma `a + b` sprowokuje przepełnienie maksymalnej wartości typu `int`. Nie zgłosi jednak wyjątku. Wynikiem będzie błędna wartość (najczęściej ujemna), ale to inny problem. Wynik musi być taki sam za każdym razem, gdy funkcję wywołamy z tymi samymi argumentami. Nie oznacza to, że wynik musi być dokładny!

> **DOKŁADNOŚĆ** Termin **dokładność** sam w sobie niewiele mówi. Wskazuje jedynie, że odpowiada temu, czego oczekiwano. Aby powiedzieć, że coś jest dokładnie, jak zaplanowano, trzeba znać intencję implementującego. Najczęściej jednak znamy tylko nazwę funkcji, która traktowana jako wyrocznia w tej sprawie może być źródłem nieporozumienia.

Przejdźmy do drugiej metody:

```
public int mult(int a, Integer b) {
   a = 5;
   b = 2;
   return a * b;
}
```

Metoda `mult` jest funkcją czystą z tych samych powodów, co metoda `add`. Może to być dla niektórych nieco dziwne, ponieważ wydaje się modyfikować argumenty. Argumenty w metodach Javy są jednak przekazywane przez wartość, więc zmiana ich wartości wewnątrz funkcji nie powoduje uwidocznienia tej zmiany na zewnątrz. Metoda zawsze zwróci wartość `10`, co nie jest zbyt przydatne, szczególnie że nie zależy to od przekazanych argumentów, ale spełnia to wszystkie wymagania. Kilkukrotne wywołanie metody dla tych samych argumentów spowoduje zwrócenie tej samej wartości.

Tak przy okazji, metoda ta jest równoważna metodzie bez argumentów. To szczególny przypadek funkcji: `f(x) = 10`. To stała.

Przejdźmy do metody `div`:

```
public int div(int a, int b) {
   return a / b;
}
```

Metoda `div` nie jest funkcją czystą, ponieważ zgłosi wyjątek, jeśli nastąpi próba dzielenia przez `0`. Aby uczynić ją funkcją, moglibyśmy testować drugi parametr i zwracać pewną wartość, jeśli wynosi on `0`. Zwróconą wartością musiałby być `int`, więc trudno byłoby znaleźć jakąś sensowną wartość, ale to inny problem.

Przejdźmy do czwartej metody:

```
public int percent1 = 5;

public int applyTax1(int a) {
   return a / 100 * (100 + percent1);
}
```

Metoda `applyTax1` wydaje się nie być funkcją czystą, ponieważ jej wynik zależy od wartości `percent1`, która jest publiczna i może być modyfikowana między dwoma wywołaniami funkcji. W konsekwencji dwa wywołania funkcji używające tego samego argumentu mogą zwrócić różne wyniki. Zmienną `percent1` można traktować jako niejawny parametr, ale nie jest on wyliczany w tym samym momencie co argument metody. Nie będzie to problem, jeśli wartość `percent1` będzie w metodzie użyta tylko raz. Jeśli odczytasz ją dwa razy, może ulec modyfikacji między operacjami odczytu. Jeśli miałaby zostać użyta dwa razy, należałoby ją odczytać i wpisać do zmiennej lokalnej. Po tej operacji `applyTax1` byłoby funkcją czystą dla krotki `(a, percent1)`, ale nie dla samej wartości `a`.

Rozważmy metodę `applyTax2`:

```
private int percent2 = 9;

public int applyTax2(int a) {
   return a / 100 * (100 + percent2);
}
```

Metoda `applyTax2` w zasadzie nie różni się od poprzedniej. Wydawać by się mogło, że jest to funkcja, ponieważ zmienna `percent2` jest prywatna. Jej stan może jednak ulec zmianie dzięki metodzie `setPercent2`. Ponieważ dostęp do `percent2` ma miejsce tylko raz, można traktować `applyTax2` jako funkcję czystą dla krotki `(a, percent2)`. Jeśli rozważamy ją w kontekście samego `a`, nie jest funkcją czystą.

Przejdźmy do szóstej metody:

```
public final int percent3 = 13;

public int applyTax3(int a) {
   return a / 100 * (100 + percent3);
}
```

Metoda `applyTax3` jest szczególna. Dla tego samego argumentu zawsze zwróci tę samą wartość, ponieważ zależy tylko do tego argumentu i właściwości finalnej `percent3`, która nie może ulec zmianie. Wydawać by się mogło, że nie jest to funkcja czysta, ponieważ jej wynik nie zależy tylko do argumentu metody (wynik funkcji czystych musi zależeć tylko od argumentu). Nie będzie jednak sprzeczności, jeśli potraktujemy `percent3` jako argument pomocniczy. W zasadzie całą klasę można potraktować jako jeden argument pomocniczy, ponieważ metody mają dostęp do wszystkich właściwości klasy.

To istotne spostrzeżenie. Wszystkie metody instancji można zastąpić metodami statycznymi poprzez dodanie argumentu o typie takim jak klasa, w której się znajdują. Metodę `applyTax3` można więc zapisać jako:

```
public static int applyTax3(FunctionalMethods x, int a) {
   return a / 100 * 100 + x.percent3;
}
```

Metodę tę można wywołać z wnętrza klasy, przekazując referencję do `this` jako argument, np. `applyTax3(this, a)`. Można ją też wywołać z zewnątrz, ponieważ jest publiczna. Wystarczy jedynie referencja do instancji klasy `FunctionalMethods`. Metoda `applyTax3` jest więc funkcją czystą dla krotki `(this, a)`.

Doszliśmy do ostatniej metody:

```
public List<Integer> append(int i, List<Integer> list) {
   list.add(i);
   return list;
}
```

Metoda `append` modyfikuje argument przed jego zwróceniem, a zmiana jest widoczna na zewnątrz funkcji, więc nie jest to funkcja czysta.

NOTACJA OBIEKTOWA KONTRA NOTACJA FUNKCYJNA

Przedstawiłem sytuację, w której metody instancji korzystające z właściwości klasy można potraktować tak, jakby instancja klasy była ich niejawnym parametrem. Metody korzystające z zawartości instancji mogą zostać zamienione na metody statyczne, jeśli ich wcześniej niejawny parametr (instancję) przekaże się jawnie.

Rozważmy klasę `Payment` z rozdziału 1.:

```
public class Payment {

   public final CreditCard cc;
   public final int amount;

   public Payment(CreditCard cc, int amount) {
      this.cc = cc;
      this.amount = amount;
   }

   public Payment combine(Payment other) {
      if (cc.equals(other.cc)) {
         return new Payment(cc, amount + other.amount);
      } else {
         throw new IllegalStateException("Karty nie pasują do siebie.");
      }
   }
}
```

Metoda `combine` korzysta z pól `cc` i `amount` klasy ją zawierającej. Z tego powodu nie może być ustatyczniona. Metoda stosuje zawierającą ją klasę jako niejawny parametr.

Gdyby jednak parametr uczynić jawnym, moglibyśmy przekształcić metodę na jej statyczny odpowiednik:

```
public class Payment {

   public final CreditCard cc;
   public final int amount;

   public Payment(CreditCard cc, int amount) {
      this.cc = cc;
      this.amount = amount;
   }

   public static Payment combine(Payment payment1, Payment payment2) {
      if (payment1.cc.equals(payment2.cc)) {
         return new Payment(payment1.cc, payment1.amount + payment2.amount);
      } else {
         throw new IllegalStateException("Karty nie pasują do siebie.");
      }
   }
}
```

Metoda statyczna daje pewność, że w zastosowanym kodzie nie istnieje żaden niechciany dostęp do kontekstu zawierającego metodę. Zmienia to jednak sposób korzystania z metody.

Wewnątrz klasy metodę statyczną można wywołać, przekazując jej referencję `this`:

```
Payment newPayment = combine(this, otherPayment);
```

Jeśli metodę wywołujemy spoza klasy, trzeba użyć nazwy klasy:

```
Payment newPayment = Payment.combine(payment1, payment2);
```

Różnica jest niewielka, ale wszystko ulega zmianie, gdy trzeba połączyć wywołania metod. Jeśli musimy połączyć kilka płatności, metoda instancji zapisana jako:

```
public Payment combine(Payment payment) {
   if (this.cc.equals(payment.cc)) {
      return new Payment(this.cc, this.amount + payment.amount);
   } else {
      throw new IllegalStateException("Karty nie pasują do siebie.");
   }
}
```

może skorzystać z notacji obiektowej:

```
Payment newPayment = p0.combine(p1).combine(p2).combine(p3);
```

To znacznie bardziej przejrzysta wersja niż:

```
Payment newPayment = p0.combine(p1).combine(p2).combine(p3);
```

Co więcej, dodanie jeszcze jednej płatności jest znacznie łatwiejsze w pierwszej z przedstawionych sytuacji.

2.2.2. Interfejsy funkcyjne Javy i klasy anonimowe

Metody można uczynić funkcyjnymi, ale brakuje im czegoś, co pozwalałoby im reprezentować funkcje w programowaniu funkcyjnym — nie można ich zmieniać poza przypisywaniem argumentów. Nie można przekazać metody jako argumentu do innej metody.

W konsekwencji nie można tworzyć kompozycji metod bez ich wykonywania — dopuszczalne są tylko kompozycje wykonań metod, ale nie samych metod. Metoda Javy należy do klasy, w której została zdefiniowana, i cały czas tam pozostaje.

Możemy tworzyć kompozycje metod, wywołując je z innych metod, ale trzeba to czynić, pisząc program. Jeśli niezbędne są różne kompozycje w zależności od konkretnych warunków, trzeba wszystko przewidzieć w trakcie pisania kodu. Nie można napisać programu w taki sposób, aby sam zmieniał się w trakcie wykonywania. Czy aby na pewno?

Ależ można! Czasami rejestruje się procedury obsługi w trakcie wykonywania programu, aby obsłużyć konkretne przypadki. Procedury obsługi trafiają na listę, z której mogą być usunięte. Nic też nie stoi na przeszkodzie, aby zmienić kolejność ich wykonywania. W jaki sposób realizuje się to zadanie? Używając klas zawierających metody z właściwą obsługą zadania.

W interfejsach graficznych bardzo często używa się elementów nasłuchujących konkretne zdarzenia, na przykład przemieszczenie kursora myszy, zmianę rozmiaru okna czy pisanie tekstu. Kod obsługujący zdarzenia umieszcza się najczęściej w klasach anonimowych implementujących konkretny interfejs. Dokładnie ten sam mechanizm warto wykorzystać do utworzenia funkcji.

Przypuśćmy, że tworzymy metodę, która potraja przekazaną liczbę całkowitą. Najpierw musimy zdefiniować interfejs z jedną metodą:

```
public interface Function {
   int apply(int arg);
}
```

Następnie implementujemy tę metodę, aby utworzyć funkcję:

```
Function triple = new Function() {
   @Override
   public int apply(int arg) {
      return arg * 3;
   }
};
```

Funkcję możemy teraz zastosować dla zadanego argumentu:

```
System.out.println(triple.apply(2));
6
```

Muszę przyznać, że nie jest to zbyt spektakularne. Stara, dobra metoda byłaby z pewnością łatwiejsza w użyciu. Jeśli chcemy wykonać inną funkcję, wystarczy postąpić dokładnie tak samo:

```
Function square = new Function() {
   @Override
   public int apply(int arg) {
      return arg * arg;
   }
};
```

Idzie nam dobrze, ale jakie są tego zalety?

2.2.3. Złożenie funkcji

Jeśli potraktuje się funkcje jak metody, ich złożenie wydaje się proste:

```
System.out.println(square.apply(triple.apply(2)));
36
```

To jednak nie jest złożenie funkcji. To złożenie zastosowań funkcji. Złożenie funkcji to operacja binarna na funkcjach, podobnie jak dodawanie to operacja binarna na liczbach. Możemy złożyć funkcje programowo, używając do tego metody:

```
Function compose(final Function f1, final Function f2) {
   return new Function() {
      @Override
      public int apply(int arg) {
         return f1.apply(f2.apply(arg));
      }
   };
}

System.out.println(compose(triple, square).apply(3));
27
```

Zapewne zaczynasz rozumieć, jak duże daje to możliwości! Pozostają jeszcze do rozwiązania dwa duże problemy. Po pierwsze, funkcja może przyjmować i zwracać tylko liczby całkowite (typ `int`). Przystąpmy od razu do rozwiązania tej kwestii.

2.2.4. Funkcje polimorficzne

Aby funkcja stała się bardziej użyteczna, zmieńmy ją na funkcję polimorficzną poprzez parametryzację typów. W Javie zadanie to realizują typy generyczne:

```
public interface Function<T, U> {
   U apply(T arg);
}
```

Stosując nowy interfejs, wcześniejsze funkcje możemy zapisać następująco:

```
Function<Integer, Integer> triple = new Function<Integer, Integer>() {
   @Override
   public Integer apply(Integer arg) {
      return arg * 3;
   }
};

Function<Integer, Integer> square = new Function<Integer, Integer>() {
   @Override
   public Integer apply(Integer arg) {
      return arg * arg;
   }
};
```

Zauważ, że zmieniliśmy typ z `int` na `Integer`, ponieważ `int` nie może być używane w kontekście parametryzacji typów. Na szczęście, automatyczne pakowanie i rozpakowywanie typów podstawowych czyni całą konwersję transparentną.

ĆWICZENIE 2.1

Napisz metodę `compose` wykorzystującą dwie nowe funkcje.

> **UWAGA** Odpowiedzi do ćwiczeń znajdują się tuż po każdym ćwiczeniu, więc polecam próbę zmierzenia się z ćwiczeniem przed zaglądaniem do odpowiedzi. Odpowiedź znajdziesz również w kodzie źródłowym dołączonym do książki. Pierwsze ćwiczenie jest proste, ale niektóre późniejsze będą naprawdę trudne, więc niełatwo będzie się oprzeć pokusie zajrzenia do rozwiązania. Pamiętaj, że im więcej zastanawiania się i poszukiwania rozwiązania, tym więcej się nauczysz.

ROZWIĄZANIE ĆWICZENIA 2.1

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                          Function<Integer, Integer> f2) {
   return new Function<Integer, Integer>() {
      @Override
      public Integer apply(Integer arg) {
         return f1.apply(f2.apply(arg));
      }
   };
}
```

Problem ze składaniem funkcji

Złożenie funkcji to bardzo ważny element, ale implementowanie go w Javie obarczone jest dużym ryzykiem. Złożenie kilku funkcji nie jest szkodliwe. Pomyśl jednak o liście 10 tysięcy funkcji i ich złożeniu do jednej funkcji. (Można to zrobić operacją zwinięcia, o której więcej napiszę w rozdziale 3.).

W programowaniu imperatywnym każda z funkcji jest wyliczana, zanim wynik zostanie przekazany na wejście następnej funkcji. W programowaniu funkcyjnym złożenie funkcji oznacza zbudowanie wynikowej funkcji bez wyliczania czegokolwiek. To właśnie ta cecha czyni cały mechanizm wyjątkowo użytecznym, bo jeszcze nic nie liczymy. W konsekwencji próba zastosowania takiego złożenia funkcji spowoduje wywołanie wielu osadzonych metod, co może zakończyć się przepełnieniem bufora. Można to zademonstrować za pomocą prostego przykładu (wykorzystując lambdy, czyli funkcje anonimowe, które omówię w następnym punkcie).

```
int fnum = 10_000; Function<Integer, Integer> g = x -> x;
Function<Integer, Integer> f = x -> x + 1;
for (int i = 0; i < fnum; i++) {
   g = Function.compose(f, g);
};
System.out.println(g.apply(0));
```

Przepełnienie bufora nastąpi, gdy `fnum` osiągnie wartość około 7500. Mam nadzieję, że nie dokonujesz składania tysięcy funkcji przy byle okazji, ale warto zdawać sobie z sprawę z tego ograniczenia.

2.2.5. Upraszczanie kodu za pomocą funkcji anonimowych

Drugim problemem, o którym wspomniałem, jest to, że funkcje zdefiniowane za pomocą klas anonimowych są mało wygodne w stosowaniu. Używając Javy od wersji 5. do 7., nie można nic z tym zrobić. Na szczęście, Java 8 wprowadziła lambdy, czyli funkcje anonimowe.

Funkcje anonimowe nie zmieniają sposobu definiowania interfejsu `Function`, ale czynią implementację znacznie przyjemniejszą:

```
Function<Integer, Integer> triple = x -> x * 3;
Function<Integer, Integer> square = x -> x * x;
```

Funkcje anonimowe to nie tylko uproszczenie składni. Mają bowiem pewne konsekwencje również w kwestii kompilacji kodu. Jedną z głównych różnic między funkcjami anonimowymi a tradycyjnym sposobem pisania klas anonimowych jest to, że można pominąć typy po prawej stronie znaku równości. Stało się to możliwe tylko dlatego, że w Javie 8 wprowadzono usprawnienia dotyczące wnioskowania na temat typów.

Do Javy 7 jedyne wnioskowanie co do typu było możliwe w sytuacji tworzenia łańcucha dereferencji identyfikatorów:

```
System.out.println();
```

Nie musimy podawać typu dla `out`, bo Java potrafi sama go odgadnąć. Gdyby tworzenie łańcucha wywołań nie było możliwe, musielibyśmy napisać:

```
PrintStream out = System.out;
out.println();
```

Java 7 wprowadziła pewne drobne usprawnienie w postaci **operatora <>**:

```
List<String> list = new ArrayList<>();
```

Nie trzeba ponawiać parametru typu `String` w `ArrayList`, ponieważ Java potrafi sama go wywnioskować na podstawie deklaracji. Dokładnie taki sam proces zachodzi w funkcjach anonimowych:

```
Function<Integer, Integer> triple = x -> x * 3;
```

W przedstawionym przykładzie Java potrafi wywnioskować typ `x`. Nie zawsze jest to jednak możliwe. Jeżeli Java zgłosi informację, że nie potrafi odgadnąć typu, musisz zapisać go jawnie. Wymaga to użycia nawiasów:

```
Function<Integer, Integer> triple = (Integer x) -> x * 3;
```

OKREŚLANIE TYPU FUNKCJI

Choć Java 8 wprowadziła funkcje anonimowe ułatwiające implementację prawdziwych funkcji, nie zawiera żadnego narzędzia upraszczającego pisanie typów funkcji. Typem funkcji z `Integer` na `Integer` jest:

```
Function<Integer, Integer>
```

Implementację funkcji zapisuje się następująco:

```
x -> wyrażenie
```

Byłoby miło, gdybyśmy mogli zastosować takie samo uproszczenie dla typu, co pozwoliłoby na zapisanie całości jako:

```
Integer -> Integer square = x -> x * x;
```

Niestety, takiego zapisu Java 8 nie dopuszcza. Nie można go też dodać samodzielnie.

ĆWICZENIE 2.2

Napisz nową wersję metody `compose`, która korzysta z funkcji anonimowych (lambdy).

ROZWIĄZANIE 2.2

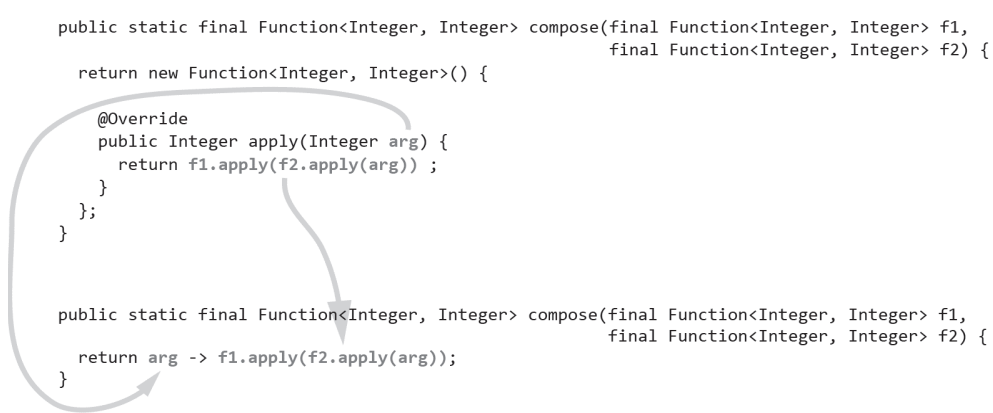
Zamiana klas anonimowych na funkcje anonimowe jest bardzo prosta. Oto pierwsza wersja metody `compose`:

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                          Function<Integer, Integer> f2) {
   return new Function<Integer, Integer>() {
      @Override
      public Integer apply(Integer arg) {
         return f1.apply(f2.apply(arg));
      }
   };
}
```

Wystarczy tylko zastąpić wartość zwracaną przez metodę `compose` argumentem metody `apply` klasy anonimowej, po której pojawia się operator strzałki (`->`) oraz wartość zwracana przez metodę `apply`:

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                          Function<Integer, Integer> f2) {
   return arg -> f1.apply(f2.apply(arg));
}
```

Nazwa argumentu jest całkowicie dowolna. Rysunek 2.2 przedstawia cały proces:



Rysunek 2.2. Zastąpienie klasy anonimowej funkcją anonimową

2.3. Zaawansowane funkcjonalności funkcji

Przedstawiłem, jak wykonać funkcje `apply` i `compose`. Wskazałem, że funkcja może być reprezentowana przez metody lub przez obiekty. Nie odpowiedziałem jednak jeszcze na podstawowe pytanie: dlaczego obiekty funkcji są potrzebne? Czy nie łatwiej byłoby po prostu użyć metod? Zanim jednak odpowiem na to pytanie, musimy zastanowić się na inną kwestią — funkcyjną reprezentacją metod wieloargumentowych.

2.3.1. Co z funkcjami dotyczącymi kilku argumentów?

W punkcie 2.1.1 wskazałem, że nie istnieją funkcje z kilkoma argumentami. Istnieją tylko funkcje z jedną krotką argumentów. To, ile krotka zawiera elementów, jest już bez znaczenia. Często stosuje się dla takich krotek specjalne nazwy, jak para, tercet, kwartet itp. Można też użyć nazw z informacją liczbową, np. krotka2, krotka3, krotka4 itd. Wskazałem również, że argumenty można aplikować jeden po drugim. Każda taka aplikacja zwraca nową funkcję (poza oczywiście ostatnią).

Spróbujmy zdefiniować funkcję dodającą dwie liczby całkowite. Zastosujemy funkcję dla pierwszego argumentu i uzyskamy w efekcie nową funkcję. Typ będzie miał postać:

```
Function<Integer, Function<Integer, Integer>>
```

Może się to wydawać nieco skomplikowane, szczególnie jeśli myślisz, że zapis mógłby mieć postać:

```
Integer -> Integer -> Integer
```

Zgodnie z prawem łączności, jest to równoważne:

```
Integer -> (Integer -> Integer)
```

gdzie lewy `Integer` to typ argumentu, a element w nawiasach to zwracany typ, którym oczywiście jest funkcja. Gdy usuniemy słowo `Function` z `Function<Integer, Function<` ↪`Integer, Integer>>`, otrzymamy:

```
<Integer, <Integer, Integer>>
```

Uzyskaliśmy ten sam wynik. Sposób pisania typów funkcji w Javie jest bardzo rozwlekły, ale nieskomplikowany.

ĆWICZENIE 2.3

Napisz funkcję dodającą dwa obiekty `Integer`.

ROZWIĄZANIE 2.3

Funkcja przyjmie `Integer` jako argument i zwróci funkcję z `Integer` na `Integer`, więc typem będzie `Function<Integer, Function<Integer, Integer>>`. Nadajmy jej nazwę `add`. Zaimplementujmy ją za pomocą funkcji anonimowych. Wynik końcowy ma postać:

```
Function<Integer, Function<Integer, Integer>> add = x -> y -> x + y;
```

Łatwo zauważyć, że wkrótce pojawią się problemy z długością wiersza kodu! Java nie posiada aliasów typów, ale podobny efekt można uzyskać dziedziczeniem. Jeśli wiele funkcji definiuje się z tym samym typem, można zamienić go na znacznie krótszy identyfikator:

```
public interface BinaryOperator extends
                Function<Integer, Function<Integer, Integer>> {}
BinaryOperator add = x -> y -> x + y;
BinaryOperator mult = x -> y -> x * y;
```

Liczba argumentów nie ma ograniczenia. Można zdefiniować funkcję z dowolną liczbą argumentów. Jak wskazałem w pierwszej części tego rozdziału, funkcje takie jak `add` lub `mult` zdefiniowane powyżej są **częściowo rozwiniętym** równoważnikiem funkcji krotek.

2.3.2. Zastosowanie funkcji z częściowym rozwinięciem

Przedstawiłem sposób pisania funkcji i ich implementacji, ale jak je zastosować? W taki sam sposób, jak każdą inną funkcję. Stosuje się funkcję dla pierwszego argumentu, a następnie stosuje wynik dla następnego argumentu i tak dalej aż do ostatniego argumentu. Zastosowanie funkcji `add` dla wartości `3` i `5` ma postać:

```
System.out.println(add.apply(3).apply(5));
8
```

Tutaj ponownie brakuje nieco lukru składniowego, bo z pewnością przejrzyściej i wygodniej byłoby napisać nazwę funkcji, a po niej jej argumenty. Taki właśnie zapis możliwy jest w języku Scala:

```
add(3)(5)
```

Jeszcze bardziej przejrzystą wersję dopuszcza Haskell:

```
add 3 5
```

Być może takie uproszczenie pojawi się w przyszłych wersjach Javy.

2.3.3. Funkcje wyższego rzędu

W punkcie 2.1.4 powstała metoda łącząca funkcje. Metoda ta była funkcyjna — przyjmowała krotkę dwóch funkcji jako swój argument i zwracała funkcję. Zamiast metody moglibyśmy jednak użyć funkcji! Ten szczególny rodzaj funkcji, przyjmujący funkcje jako argumenty i zwracający funkcje, nazywa się **funkcjami wyższego rzędu**.

ĆWICZENIE 2.4

Napisz funkcję, która łączy ze sobą dwie funkcje `square` i `triple` używane w ćwiczeniu 2.2.

ROZWIĄZANIE 2.4

Ćwiczenie okaże się bardzo proste, o ile zastosuje się odpowiednią procedurę. W pierwszej kolejności musimy napisać typ. Funkcja będzie działała na dwóch argumentach, czyli musi stosować częściowe rozwinięcie. Dwoma argumentami i zwracanym typem będą funkcję od `Integer` do `Integer`:

```
Function<Integer, Integer>
```

Nadajmy tej części nazwę `T`. Chcemy utworzyć funkcję przyjmującą argument typu `T` (pierwszy argument) i zwracającą funkcję od `T` (drugi argument) do `T` (zwracana wartość). Typem funkcji jest więc:

```
Function<T, Function<T, T>>
```

Zastępując `T` właściwą wartością, otrzymamy prawdziwy typ:

```
Function<Function<Integer, Integer>,
         Function<Function<Integer, Integer>,
                  Function<Integer, Integer>>>
```

Głównym problemem w tym przypadku jest długość wiersza kodu. Implementacja okaże się znacznie prostsza niż typ:

```
x -> y -> z -> x.apply(y.apply(z));
```

Oto pełna postać kodu:

```
Function<Function<Integer, Integer>,
         Function<Function<Integer, Integer>,
                  Function<Integer, Integer>>> compose =
                                          x -> y -> z -> x.apply(y.apply(z));
```

Oczywiście, kod można zapisać w jednym wierszu! Sprawdźmy go za pomocą funkcji `square` i `triple`:

```
Function<Integer, Integer> triple = x -> x * 3;
Function<Integer, Integer> square = x -> x * x;
Function<Integer, Integer> f = compose.apply(square).apply(triple);
```

W tym kodzie rozpoczynamy od zastosowania pierwszego argumentu. Otrzymujemy w efekcie nową funkcję, w której możemy zastosować drugi argument. Wynikiem jest funkcja stanowiąca połączenie dwóch argumentów funkcyjnych. Stosując nową funkcję względem wartości 2, spowodujemy, że najpierw 2 zostanie użyte dla `triple`, a następnie `square` będzie użyte dla wyniku (odpowiada to definicji kompozycji funkcji).

```
System.out.println(f.apply(2));
36
```

Zwróć uwagę na kolejność parametrów: najpierw jest używane `triple` i dopiero jego wynik stanowi podstawę dla `square`.

2.3.4. Polimorficzne funkcje wyższego rzędu

Przedstawiona funkcja `compose` działa, ale dokonuje kompozycji tylko i wyłącznie funkcji z `Integer` do `Integer`. Czy nie byłoby bardziej interesujące, gdyby można było realizować kompozycję dowolnych rodzajów funkcji, na przykład z `String` na `Double` lub z `Boolean` na `Long`? Jednak to tylko początek. W pełni polimorficzna funkcja `compose` umożliwiałaby kompozycję dla `Function<Integer, Function<Integer, Integer>>`, czyli funkcji `add` i `mult` napisanych w ćwiczeniu 2.3. Powinna również umożliwić kompozycję funkcji różnych typów, o ile tylko typ zwracany przez jedną z nich pasuje do typu przyjmowanego przez następną.

ĆWICZENIE 2.5 (TRUDNE)

Napisz polimorficzną wersję funkcji `compose`.

WSKAZÓWKA

Próbując wykonać to ćwiczenie, natkniesz się na dwa problemy. Pierwszym jest brak właściwości polimorficznych w Javie. W Javie można tworzyć polimorficzne klasy, interfejsy i metody, ale nie polimorficzne właściwości. Rozwiązaniem jest przechowywanie funkcji w metodzie, klasie lub interfejsie, a nie we właściwości.

Drugim problemem jest to, że Java nie obsługuje różnych wariantów, więc próba rzutowania na przykład `Function<Integer, Integer>` na `Function<Object, Object>` zakończy się błędem kompilacji. Trzeba wspomóc Javę i wszystkie typy wskazywać bardzo dokładnie.

ROZWIĄZANIE 2.5

Pierwszym krokiem wydaje się próba „uogólnienia" przykładu z ćwiczenia 2.4:

```
<T, U, V> Function<Function<T, U>,
                   Function<Function<V, T>,
                            Function<V, U>>> higherCompose =
                                        f -> g -> x -> f.apply(g.apply(x));
```

Nie jest to jednak możliwe, ponieważ Java nie obsługuje samodzielnych, generycznych właściwości. Aby właściwość była generyczna, musi powstać w obrębie zakresu definiującego typy parametrów. Jedynie klasy, interfejsy i metody mogą definiować typy parametrów, więc właściwość trzeba zdefiniować wewnątrz jednego z tych elementów. Najbardziej praktyczna będzie metoda statyczna:

```
static <T, U, V> Function<Function<U, V>,
                          Function<Function<T, U>,
                                   Function<T, V>>> higherCompose() {
   return f -> g -> x -> f.apply(g.apply(x));
}
```

Warianty

Warianty opisują, w jaki sposób sparametryzowane typy zachowują się w kwestii podtypów. **Kowariancja** oznacza, że `Matcher<Red>` będzie traktowane jako podtyp `Matcher<Color>`, jeśli `Red` jest podtypem `Color`. W takiej sytuacji mówi się, że `Matcher<T>` jest kowariantem `T`. Jeśli z drugiej strony `Matcher<Color>` jest traktowany jako podtyp `Matcher<Red>`, wtedy mówimy, że `Matcher<T>` jest kontrwariantem `T`. Choć w Javie `Integer` jest podtypem `Object`, `List<Integer>` nie jest podtypem `List<Object>`. Może się to wydawać dziwne, ale `List<Integer>` jest `Object`, ale nie jest `List<Object>`. Z tego powodu `Function<Integer, Integer>` nie jest `Function<Object, Object>` (po tym wyjaśnieniu wniosek ten nie zaskakuje!).

W Javie wszystkie typy parametryzowane są niezmienne względem ich parametrów.

Zauważ, że metoda o nazwie `higherCompose()` nie przyjmuje parametrów i zawsze zwraca tę samą wartość. To stała. To, że została zdefiniowana jako metoda, jest z tego punktu widzenia nieistotne. To nie jest metoda zapewniająca kompozycję funkcji. To jedynie metoda zwracająca funkcję do kompozycji funkcji.

Uważaj na kolejność parametrów typów i na ich implementację jako parametrów funkcji anonimowych. Wszystko przedstawia rysunek 2.3.

```
static <T, U, V> Function<Function<U, V>,
                  Function<Function<T, U>,
                           Function<T, V>>> higherCompose()  {
    return x -> y -> z -> x.apply(y.apply(z));
}
```

Function<U, V> Function<T, U> T

Rysunek 2.3. Zwróć uwagę na kolejność parametrów typów

Moglibyśmy nadać parametrom funkcji anonimowych bardziej znaczące nazwy, na przykład `uvFunction` i `tuFunction` albo po prostu `uv` i `tu`, ale nie warto tego robić. Nazwy są zawodne. Pokazują jedynie intencję (programisty) i nic więcej. Można podmienić nazwy bez zauważenia żadnej różnicy:

```
static <T, U, V> Function<Function<U, V>,
                  Function<Function<T, U>,
                           Function<T, V>>> higherCompose() {
   return tuFunc -> uvFunc -> t -> tuFunc.apply(uvFunc.apply(t));
}
```

W tym przykładzie `tuFunc` to funkcja od `U` do `V`, `uvFunc` to funkcja od `T` do `U`.

Jeśli potrzebne są dodatkowe informacje na temat typów, można zapisać je na początku każdego parametru funkcji anonimowej, zamykając typ i parametr w nawiasach:

```
static <T, U, V> Function<Function<U, V>,
                          Function<Function<T, U>,
                                   Function<T, V>>> higherCompose() {
   return (Function<U, V> f) -> (Function<T, U > g) -> (T x)
                                                    -> f.apply(g.apply(x));
}
```

Funkcji możemy użyć w następujący sposób:

```
Integer x = Function.higherCompose().apply(square).apply(triple).apply(2);
```

Otrzymamy jednak błąd kompilacji:

```
Error:(39, 48) java: incompatible types:
...Function<java.lang.Integer,java.lang.Integer>
cannot be converted to ...Function<java.lang.Object,java.lang.Object>
```

Kompilator wskazuje, że nie potrafił określić prawdziwych typów dla parametrów typów `T`, `U` i `V`, więc dla wszystkich trzech użył `Object`. Wiemy jednak, że funkcje `square` i `triple` używają typów `Function<Integer, Integer>`. Jeśli sądzisz, że to wystarczające informacje, aby wywnioskować typy dla `T`, `U` i `V`, to jesteś mądrzejszy od Javy! Java próbuje iść w drugą stronę i rzutuje `Function<Integer, Integer>` na `Function<Object, Object>`. Choć `Integer` to `Object`, `Function<Integer, Integer>` to nie `Function<Object, Object>`. Te dwa typy nie są powiązane, ponieważ w Javie typy są niezmienne. Aby rzutowanie zadziałało, typy powinny być kowariancjami, ale Java nie wie o tym związku.

Rozwiązaniem jest powrót do oryginalnego problemu i wspomożenie kompilatora informacją o prawdziwych typach `T`, `U` i `V`. Możemy to zrobić, wstawiając informację o typie między kropkę i nazwę metody:

```
Integer x = Function.<Integer, Integer, Integer>higherCompose().apply(...
```

To mało praktyczne, ale to nawet nie główny problem. Często grupuje się funkcje takie jak `higherCompose` w bibliotekę klas, a następnie chce się zastosować import statyczny, aby uprościć kod:

```
import static com.fpinjava. ... .Function.*;
...
Integer x = <Integer, Integer, Integer>higherCompose().apply(...;
```

Niestety, nie uda się tego skompilować!

ĆWICZENIE 2.6 (TYM RAZEM ŁATWE!)

Napisz funkcję `higherAndThen`, która realizuje kompozycję, ale odwrotnie, czyli `higherCom`
`↪pose(f, g)` jest równoważne `higherAndThen(g, f)`.

ROZWIĄZANIE 2.6

```
public static <T, U, V> Function<Function<T, U>, Function<Function<U, V>,
                                 Function<T, V>>> higherAndThen() {
   return f -> g -> x -> g.apply(f.apply(x));
}
```

Testowanie parametrów funkcji

Jeśli masz wątpliwości co do kolejności parametrów, warto przetestować funkcje wyższego rzędu za pomocą funkcji o innych typach. Testując funkcje od `Integer` do `Integer`, można doprowadzić do dwuznaczności, ponieważ wszystko będzie działać w obu kierunkach, więc trudno wykryć pomyłkę. Oto przykład testu sprawdzającego funkcje różnych typów:

```
public void TestHigherCompose() {

   Function<Double, Integer> f = a -> (int) (a * 3);
   Function<Long, Double> g = a -> a + 2.0;

   assertEquals(Integer.valueOf(9), f.apply((g.apply(1L))));
   assertEquals(Integer.valueOf(9),
      Function.<Long, Double, Integer>higherCompose().apply(f).apply(g).apply(1L));
}
```

Zauważ, że Java nie jest w stanie odgadnąć typów, więc trzeba je wskazać w momencie wywoływania funkcji `higherCompose`.

2.3.5. Użycie funkcji anonimowych

Do tej pory korzystaliśmy z funkcji nazwanych. Funkcje były implementowane jako klasy anonimowe, ale instancje miały nazwy i przypisane jawnie typy. Często nie nadaje się funkcjom nazw. Używa się ich jako instancji anonimowych. Przyjrzyjmy się przykładowi.

Zamiast pisać:

```
Function<Double, Double> f = x -> Math.PI / 2 - x;
Function<Double, Double> sin = Math::sin;
Double cos = Function.compose(f, sin).apply(2.0);
```

możemy użyć funkcji anonimowej:

```
Double cos = Function.compose(x -> Math.PI / 2 - x, Math::sin).apply(2.0);
```

Powyżej używamy metody `compose` zdefiniowanej statycznie w klasie `Function`. Nic nie stoi na przeszkodzie, aby użyć funkcji wyższego rzędu:

```
Double cos = Function.<Double, Double, Double>higherCompose()
            .apply(z -> Math.PI / 2 - z).apply(Math::sin).apply(2.0);
```

Referencje do metod
Poza funkcjami anonimowymi Java 8 wprowadza również referencję do metody, która jest składnią pozwalającą na zastąpienie funkcji anonimowej, jeśli miałaby się składać z wywołania metody z pojedynczym argumentem. Na przykład:

```
Function<Double, Double> sin = Math::sin;
```

jest równoważne:

```
Function<Double, Double> sin = x -> Math.sin(x);
```

Tutaj `sin` jest metodą statyczną klasy `Math`. Gdyby była metodą instancji aktualnej klasy, moglibyśmy napisać:

```
Function<Double, Double> sin = this.sin(x);
```

Przedstawiony zapis pojawi się w książce bardzo często, bo pozwala utworzyć z metody funkcję.

KIEDY KORZYSTAĆ Z FUNKCJI ANONIMOWYCH, A KIEDY Z NAZWANYCH?

Poza specjalnymi przypadkami, w których funkcji anonimowych nie można zastosować, to od użytkownika zależy wybór miedzy funkcją anonimową a nazwaną. Ogólna zasada jest następująca: jeśli funkcja jest stosowana tylko raz, wersja anonimowa jest odpowiednia. **Stosowana tylko raz** oznacza, że jest napisana tylko raz. Nie oznacza to, że zostanie wywołana tylko raz.

W poniższym przykładzie definiujemy metodę, która oblicza kosinus wartości typu `Double`. Metoda używa dwóch funkcji anonimowych, ponieważ korzysta z wyrażenia lambda i referencji do metody:

```
Double cos(Double arg) {
   return Function.compose(z -> Math.PI / 2 - z, Math::sin).apply(arg);
}
```

Nie martw się tworzeniem anonimowych instancji. Java nie zawsze tworzy nowe obiekty, gdy zostanie wywołana funkcja. Poza tym tworzenie egzemplarzy takich obiektów jest tanie. Zamiast tego należy zdecydować, czy użyć funkcji anonimowej czy nazwanej, na podstawie czytelności kodu. Jeśli zależy Ci na wydajności i poprawie wielokrotności użycia, stosuj jak najczęściej referencje do metod.

ZGADYWANIE TYPU

Zgadywanie typu potrafi być problemem w przypadku funkcji anonimowych. W poprzednim przykładzie typy dwóch funkcji anonimowych mogły być odgadnięte przez kompilator, ponieważ wie, że metoda `compose` przyjmuje jako argumenty dwie funkcje:

```
static <T, U, V> Function<V, U> compose(Function<T, U> f, Function<V, T> g)
```

Nie zawsze taka sztuczka się powiedzie. Jeśli zastąpi się drugi argument wyrażeniem lambda zamiast referencją do metody:

```
Double cos(Double arg) {
   return Function.compose(z -> Math.PI / 2 - z,
                           a -> Math.sin(a)).apply(arg);
}
```

kompilator się pogubi i wyświetli następujący komunikat o błędzie:

```
Error:(64, 63) java: incompatible types: java.lang.Object cannot be converted
to double
Error:(64, 44) java: bad operand types for binary operator '-' first type: double
second type: java.lang.Object
Error:(64, 72) java: incompatible types: java.lang.Object cannot be converted
to java.lang.Double
```

Kompilator tak mocno się pogubił, że wskazuje nawet na nieistniejący błąd w kolumnie 44! Błąd w kolumnie 63 jest jednak prawdziwy. Choć wydaje się to dziwne, Java nie jest w stanie odgadnąć typu drugiego argumentu. Aby kod udało się skompilować, musimy dodać adnotacje o typie:

```
Double cos(Double arg) {
   return Function.compose(z -> Math.PI / 2 - z,
                   (Function<Double, Double>) (a) -> Math.sin(a)).apply(arg);
}
```

To dobry powód, by polecać stosowanie referencji do metod.

2.3.6. Funkcje lokalne

Przekonaliśmy się, że możemy definiować funkcje lokalnie w metodach, ale nie możemy definiować metod wewnątrz metod.

Z drugiej strony, funkcje można definiować wewnątrz funkcji bez najmniejszych problemów, jeśli stosuje się funkcje anonimowe (lambdy). Najczęstszym formatem, który się widuje, są osadzone lambdy:

```
public <T> Result<T> ifElse(List<Boolean> conditions, List<T> ifTrue) {
   return conditions.zip(ifTrue)
      .flatMap(x -> x.first(y -> y._1))
      .map(x -> x._2);
}
```

Nie przejmuj się, jeśli nie rozumiesz, co robi ten kod. Dowiesz się wszystkiego w następnych rozdziałach. Ważne jest tylko to, że metoda `flatMap` przyjmuje funkcję jako swój argument (w postaci lambdy), a implementacja tej funkcji (kod po `->`) definiuje nową lambdę, która odpowiada za lokalnie osadzoną funkcję.

Funkcje lokalne nie zawsze są anonimowe. Jeśli są to **funkcje pomocnicze**, często się je nazywa. W tradycyjnej Javie korzystanie z metod pomocniczych to częsta praktyka. Metody te umożliwiają uproszczenie kodu przez abstrakcję pewnych fragmentów (przeniesienie rozwlekłej logiki w inne miejsce). To samo rozwiązanie pojawia się dla funkcji, ale można go nie zauważyć, bo dzięki funkcjom anonimowym nie jest jawne. Możemy zawsze użyć jawnie zadeklarowanych funkcji lokalnych, jak w poniższym przykładzie, który jest w zasadzie równoważny poprzedniemu:

```
public <T> Result<T> ifElse_(List<Boolean> conditions, List<T> ifTrue) {
   Function<Tuple<Boolean, T>, Boolean> f1 = y -> y._1;
   Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>> f2 =
                                                          x -> x.first(f1);
   Function<Tuple<Boolean, T>, T> f3 = x -> x._2;
   return conditions.zip(ifTrue)
      .flatMap(f2)
      .map(f3);
}
```

Jak wcześniej wspomniałem, obie postacie (z lokalnie nazwanymi funkcjami i bez nich) nieco się różnią, co czasem może mieć znaczenie. Zastosowanie nazwanych funkcji jawnie wskazuje typy, co może okazać się niezbędne, jeśli kompilator nie potrafi właściwie odgadnąć typów.

Jest to nie tylko użyteczne dla kompilatora, ale stanowi także sporą pomoc dla programisty próbującego odgadnąć stosowane typy. Jawne zapisanie oczekiwanych typów pomaga odnaleźć miejsce, gdzie nie spełniono oczekiwań.

2.3.7. Domknięcia

Pokazaliśmy, że czyste funkcje nie mogą zależeć od niczego innego poza ich argumentami, gdy wyliczają wynik. Metody Javy często korzystają ze składowych klasy, odczytując je, a czasem nawet zapisując. Metody mogą nawet korzystać ze składowych statycznych innych klas. Powiedziałem, że metody **funkcyjne** to metody szanujące transparentność referencyjną, czyli nie mają żadnych obserwowalnych efektów poza zwróceniem wartości. To samo dotyczy funkcji. Funkcje są czyste, jeśli nie mają żadnych obserwowalnych efektów ubocznych.

Ale co z funkcjami (i metodami), które zwracają wartości zależne nie tylko od argumentów, ale również od elementów znajdujących się w otaczającym je środowisku? Już widzieliśmy taką sytuację. Elementy środowiska otaczającego można potraktować jako niejawne parametry używających ich funkcji lub metod.

Funkcje anonimowe stawiają dodatkowy wymóg: mogą mieć dostęp tylko do lokalnych zmiennych oznaczonych jako finalne. Nie jest to wymóg specyficzny dla funkcji anonimowych. To samo wymaganie istniało wcześniej dla klas anonimowych przed Javą 8. Funkcje anonimowe również muszą się do tego warunku dostosować, choć ograniczenie nieco złagodzono. Począwszy od Javy 8, elementy dostępne z poziomu klas anonimowych lub funkcji anonimowych mogą być finalne w sposób niejawny. Nie trzeba ich jawnie oznaczać jako `final`, wystarczy ich nie modyfikować. Oto przykład:

```
public void aMethod() {
   double taxRate = 0.09;
   Function<Double, Double> addTax = price -> price + price * taxRate;
   ...
}
```

W tym przykładzie funkcja `addTax` „domyka się" nad zmienną lokalną `taxRate`. Kompilacja kodu powiedzie się, jeśli tylko zmienna `taxRate` nie będzie modyfikowana, choć nie wskazano jawnie, że jest to zmienna typu `final`.

Poniższy przykład nie skompiluje się poprawnie, ponieważ zmienna `taxRate` nie jest dłużej finalna w sposób niejawny:

```
public void aMethod() {
   double taxRate = 0.09;
   Function<Double, Double> addTax = price -> price + price * taxRate;
   ...
   taxRate = 0.13;
   ...
}
```

Pamiętaj, że wymóg dotyczy tylko zmiennych lokalnych. Poniższy kod skompiluje się bez problemów:

```
double taxRate = 0.09;

public void aMethod() {
   Function<Double, Double> addTax = price -> price + price * taxRate;
   taxRate = 0.13;
   ...
}
```

W przedstawionym przykładzie trzeba jawnie podkreślić, że `addTax` nie jest funkcją względem `price`, ponieważ nie gwarantuje uzyskiwania tego samego wyniku dla tego samego argumentu. Można ją jednak traktować jako funkcję krotki `(price, taxRate)`.

Domknięcia są zgodne z czystymi funkcjami, jeśli potraktuje się je jako dodatkowe, niejawne argumenty. Mogą jednak sprawić problemy w trakcie refektoryzacji kodu lub gdy funkcje są przekazywane jako parametry do innych funkcji. W konsekwencji powstały program może okazać się trudny w analizie i konserwacji.

Jednym ze sposobów uczynienia programów bardziej modularnymi jest użycie funkcji z krotkami argumentów:

```
double taxRate = 0.09;

Function<Tuple<Double, Double>, Double> addTax
   = tuple -> tuple._2 + tuple._2 * tuple._1;

System.out.println(addTax.apply(new Tuple<>(taxRate, 12.0)));
```

Korzystanie z krotek nie jest jednak wygodne, ponieważ Java nie oferuje jeszcze prostej składni dla takich sytuacji. Wyjątkiem są tylko argumenty funkcji, gdzie mamy dostęp do notacji z nawiasami. Dla funkcji z krotką musimy zdefiniować specjalny interfejs:

```
interface Function2<T, U, V> {
   V apply(T t, U u);
}
```

Interfejs stosujemy następnie dla funkcji anonimowych:

```
Function2<Double, Double, Double> addTax = (taxRate, price) -
   > price + price * taxRate;
double priceIncludingTax = addTax.apply(0.09, 12.0);
```

Zauważ, że funkcje anonimowe to jedyne miejsce, gdzie Java umożliwia zastosowanie notacji (x, y) dla krotek. Niestety, nie można jej użyć do zwrócenia krotki z funkcji.

Można także użyć klasy `BiFunction` zdefiniowanej w Javie 8, która symuluje funkcję krotki dwóch argumentów. Jest też `BinaryOperator`, która odpowiada funkcji krotki dwóch argumentów tego samego typu, oraz `DoubleBinaryOperator`, dotycząca funkcji krotki dwóch wartości typu `double`. Wszystkie te możliwości są dobre, ale co zrobić, jeśli potrzebujemy trzech lub więcej argumentów? Moglibyśmy zdefiniować `Function3`, `Function4` itd., ale rozwijanie funkcji to znacznie lepsze rozwiązanie. Właśnie z tego powodu nauczenie się użycia rozwijania funkcji jest tak ważne. Na szczęście, sam mechanizm, jak mogłeś się przekonać, jest bardzo prosty:

```
double tax = 0.09;

Function<Double, Function<Double, Double>> addTax
   = taxRate -> price -> price + price * taxRate;

System.out.println(addTax.apply(tax).apply(12.00));
```

2.3.8. Częściowe zastosowanie funkcji i automatyczne rozwijanie

Wersje z domknięciem i rozwijaniem funkcji z poprzedniego przykładu dają ten sam wynik i mogą wydawać się równoważne. W zasadzie „semantycznie" są bardzo różne. Jak wcześniej wskazałem, oba parametry grają różne role. Procent podatku nie powinien się często zmieniać, ale cena będzie ulegała zmianie przy każdym wywołaniu. Widać to wyraźnie w wersji z domknięciem. Funkcja domyka się nad parametrem, który się nie zmienia (jest finalny). W wersji z rozwijaniem oba argumenty mogą się zmieniać przy każdym wywołaniu, choć w praktyce procent podatku nie będzie zmieniał się częściej niż w wersji z domknięciem.

Często zdarzy się, że będziemy potrzebować różnych procentowo podatków, ponieważ część produktów będzie wymagała jednej stawki, a część innej. W tradycyjnej Javie stosuje się w takiej sytuacji klasę działającą trochę jak sparametryzowany „kalkulator podatkowy":

```
public class TaxComputer {

   private final double rate;

   public TaxComputer(double rate) {
      this.rate = rate;
   }
```

```
    public double compute(double price) {
        return price * rate + price;
    }
}
```

Klasa umożliwia utworzenie kilku różnych instancji `TaxComputer` dla kilku różnych stawek podatku. Z poszczególnych instancji można już korzystać tak często, jak to potrzebne:

```
TaxComputer tc9 = new TaxComputer(0.09);
double price = tc9.compute(12);
```

Ten sam efekt uzyskamy, wykonując częściowe zastosowanie funkcji:

```
Function<Double, Double> tc9 = addTax.apply(0.09);
double price = tc9.apply(12.0);
```

Funkcja `addTax` pochodzi z końca punktu 2.3.7.

Wyraźnie widać, że rozwinięcie i częściowe zastosowanie funkcji są ze sobą blisko powiązane. Rozwinięcie funkcji zastępuje funkcję, która korzysta z krotki, nową funkcją, którą można zastosować częściowo, argument po argumencie. To główna różnica między funkcją rozwiniętą a funkcją krotki. W funkcji krotki wszystkie argumenty wylicza się przed zastosowaniem funkcji. W wersji rozwiniętej wszystkie argumenty muszą być znane, zanim funkcja zostanie w pełni zastosowana, ale jeden argument można wyliczyć przed częściowym zastosowaniem. Nie trzeba stosować pełnego rozwinięcia. Funkcję trzech argumentów można zmienić na funkcję krotki, która zwraca funkcję wymagającą tylko jednego argumentu.

W programowaniu funkcyjnym rozwinięcie i częściowe zastosowanie stosuje się tak często, że warto dokonać ich abstrakcji, tak aby zapewnić jak największą automatyzację. W poprzednich częściach pojawiły się tylko funkcje rozwinięte, a nie funkcje krotek. Ma to pewną zaletę — częściowe zastosowanie takiej funkcji jest bardzo proste.

ĆWICZENIE 2.7 (BARDZO ŁATWE)

Napisz metodę funkcyjną, która zastosuje częściowo funkcję rozwiniętą, czyli zamieni wersję dwuargumentową na jednoargumentową.

ROZWIĄZANIE 2.7

Nie musisz nic robić! Sygnatura tej metody ma postać:

```
<A, B, C> Function<B, C> partialA(A a, Function<A, Function<B, C>> f)
```

Od razu widać, że częściowe zastosowanie pierwszego argumentu jest równie proste co zastosowanie drugiego argumentu (funkcji) dla pierwszego:

```
<A, B, C> Function<B, C> partialA(A a, Function<A, Function<B, C>> f) {
    return f.apply(a);
}
```

(Jeśli chcesz się dowiedzieć, jak użyć `partialA`, zajrzyj do testu sprawdzającego powyższy zapis w kodzie dołączonym do książki).

Zauważ, że oryginalna funkcja była typu `Function<A, Function<B, C>>`, co oznacza $A \rightarrow B \rightarrow C$. A co, jeśli chcielibyśmy częściowo zastosować funkcję dla drugiego argumentu?

ĆWICZENIE 2.8

Napisz metodę, która zastosuje częściowo funkcję rozwiniętą dwóch argumentów, ale dla drugiego argumentu.

ROZWIĄZANIE 2.8

Dla wcześniejszej funkcji odpowiedzią byłaby metoda o następującej sygnaturze:

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f)
```

Ćwiczenie jest nieco trudniejsze, ale nadal proste, jeśli zastanowimy się nad typami. Pamiętaj, zawsze ufaj typom! Nie zapewnią natychmiastowego rozwiązania w każdej sytuacji, ale doprowadzą do niego. Ta funkcja ma tylko jedną możliwą implementację, więc jeśli uda się skompilować kod, znalazłeś właściwe rozwiązanie!

Wiemy, że musimy zwrócić funkcję od `A` do `C`. Możemy rozpocząć implementację, pisząc poniższy kod:

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
   return a ->
```

W kodzie `a` to zmienna typu `A`. Po strzałce musimy napisać wyrażenie, które składa się z funkcji `f` i zmiennych `a` oraz `b`, ale musi przekształcić się w funkcję od `A` do `C`. Funkcja `f` to funkcja od `A` do `B -> C`, więc zaczniemy od zastosowania jej względem `A`:

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
   return a -> f.apply(a)
```

Otrzymujemy funkcję od `B` do `C`. Potrzebujemy `C`, ale mamy już `B`, więc odpowiedź jest prosta:

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
   return a -> f.apply(a).apply(b);
}
```

Mamy to! W zasadzie nie musieliśmy robić nic więcej poza podążaniem za typami.

Jak wcześniej wspomniałem, najważniejszą rzeczą jest posiadanie rozwiniętej wersji funkcji. Zapewne bardzo szybko nauczysz się pisać funkcje rozwinięte w sposób bezpośredni. Zadaniem, które powraca jak bumerang, gdy zaczyna się pisać programy funkcyjne w Javie, jest zamiana metod z kilkoma argumentami na funkcje rozwinięte. Zadanie jest wyjątkowo proste.

ĆWICZENIE 2.9 (BARDZO ŁATWE)

Skonwertuj następującą metodę na jej rozwinięty odpowiednik:

```
<A, B, C, D> String func(A a, B b, C c, D d) {
   return String.format("%s, %s, %s, %s", a, b, c, d);
}
```

(Wiem, że ta metoda jest całkowicie bezużyteczna, ale to tylko ćwiczenie).

ROZWIĄZANIE 2.9

Ponownie nie pozostaje nam nic innego niż zastąpienie przecinków odpowiednimi strzałkami. Pamiętaj jednak, aby zdefiniować funkcję w strefie, która przyjmuje parametry typów, co dla właściwości nie jest dostępne. Oznacza to konieczność zdefiniowania klasy, interfejsu lub metody ze wszystkimi wymaganymi parametrami typu.

Wykorzystamy w tym celu metodę. Najpierw zapisz parametry typów:

```
<A,B,C,D>
```

Następnie dodaj zwracany typ. Początkowo wydaje się to trudne, ale dotyczy to głównie czytania zapisu. Napisz słowo `Function<`, a później pierwszy parametr i przecinek:

```
<A,B,C,D> Function<A,
```

Następnie zrób to samo, ale z drugim typem parametru:

```
<A,B,C,D> Function<A, Function<B,
```

Kontynuuj, aż nie zostanie żaden parametr:

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D,
```

Dodaj zwracany typ i zamknij wszystkie otwarte nawiasy:

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>>
```

Dodaj nazwę funkcji oraz nawiasy:

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>> f() {
}
```

W części implementacyjnej wymień wszystkie parametry, oddzielając je strzałkami w prawo (zakończ wymienianie na strzałce):

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, E>>>> f() {
   return a -> b -> c -> d ->
}
```

Na końcu dodaj implementację, która jest taka sama, jak oryginalnej metody:

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>> f() {
   return a -> b -> c -> d -> String.format("%s, %s, %s, %s", a, b, c, d);
}
```

Te same zasady można zastosować dla rozwijania funkcji krotki.

ĆWICZENIE 2.10

Napisz metodę do rozwijania funkcji `Tuple<A, B>` na `C`.

ROZWIĄZANIE 2.10

Ponownie podążaj za typami. Wiesz, że metoda przyjmie parametr typu `Function<Tuple<A, B>, C>` i zwróci `Function<A, Function<B, C>>`, więc sygnatura jest następująca:

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f)
```

Przejdźmy do implementacji. Musimy zwrócić rozwiniętą funkcję dwóch argumentów, więc zacznijmy od:

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f) {
   return a -> b ->
}
```

Na końcu trzeba przeprowadzić określenie zwracanego typu. W tym celu użyj funkcji `f` i zastosuj ją dla nowego `Tuple` zbudowanego z parametrów `a` i `b`:

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f) {
   return a -> b -> f.apply(new Tuple<>(a, b));
}
```

Ponownie, jeśli uda się skompilować taki kod, nie będzie on błędny. Ta pewność to jedna z wielu zalet programowania funkcyjnego! (Nie zawsze jest to prawdą, ale w następnym rozdziale dowiesz się, jak sprawić, aby taka pewność pojawiała się częściej).

2.3.9. Zamiana argumentów częściowo zastosowanych funkcji

Jeśli mamy funkcję dwóch argumentów, możemy zechcieć zastosować tylko pierwszy argument i otrzymać częściowo zastosowaną funkcję. Załóżmy, że funkcja ma postać:

```
Function<Double, Function<Double, Double>> addTax = x -> y -> y + y / 100 * x;
```

Prawdopodobnie chcemy najpierw zastosować stawkę podatku, aby później móc stosować dla niego dowolną cenę:

```
Function<Double, Double> add9percentTax = addTax.apply(9.0);
```

Aby dodać podatek do ceny, można użyć kodu:

```
Double priceIncludingTax = add9percentTax.apply(price);
```

Wszystko działa dobrze, ale co, jeśli początkowa funkcja miała poniższą postać?

```
Function<Double, Function<Double, Double>> addTax = x -> y -> x + x / 100 * y;
```

W tym przypadku cena to pierwszy argument. Zastosowanie tylko ceny nie ma sensu, więc jak w takiej sytuacji zastosować tylko stawkę podatku? (Zakładamy, że nie mamy dostępu do oryginalnej implementacji funkcji).

ĆWICZENIE 2.11

Napisz metodę, która zamienia argumenty rozwijanej funkcji.

ROZWIĄZANIE 2.11

Poniższa metoda zwraca rozwiniętą funkcję z argumentami w odwrotnej kolejności. Rozwiązanie można uogólnić do dowolnej liczby argumentów i dowolnego ich ułożenia:

```
public static <T, U, V> Function<U, Function<T, V>> reverseArgs(Function<T,
   Function<U, V>> f) {
   return u -> t -> f.apply(t).apply(u);
}
```

Dzięki tej metodzie można zastosować częściowo dowolny z dwóch argumentów. Jeśli na przykład mamy funkcję, która wylicza miesięczną ratę dla pożyczki o wskazanym oprocentowaniu i kwocie:

```
Function<Double, Function<Double, Double>> payment = amount -> rate -> ...
```

możemy bardzo łatwo utworzyć funkcję o jednym argumencie, która wylicza płatność dla stałej kwoty i zmiennego oprocentowana, lub też funkcję dla stałego oprocentowania i zmiennej kwoty.

2.3.10. Funkcje rekurencyjne

Funkcje rekurencyjne to wszechobecny element większości funkcyjnych języków programowania, choć rekurencja i programowanie i funkcyjne nie są powiązane. Niektórzy programiści języków funkcyjnych twierdzą nawet, że rekurencja jest jak polecenie `goto` w innych językach programowania i powinno się go unikać za wszelką cenę. Niemniej, jako programista funkcyjny powinieneś opanować rekurencję do perfekcji, nawet jeśli później będziesz jej celowo unikał.

Jak zapewne wiesz, Java jest ograniczona w kwestii rekurencji. Metody mogą wywoływać siebie same rekurencyjnie, ale oznacza to, że stan obliczeń jest każdorazowo umieszczany na stosie aż do osiągnięcia warunku granicznego, po którym następuje przywracanie wartości ze stosu aż do zwrócenia wyniku. Rozmiar stosu można zdefiniować, ale wszystkie wątki stosują ten sam rozmiar. Rozmiar domyślny zależy od implementacji Javy: od 320 kB w wersji 32-bitowej do 1064 kB w wersji 64-bitowej. Obie wartości są stosunkowo małe, jeśli porównać je z rozmiarem sterty służącej do przechowywania obiektów. Oznacza to, że liczba możliwych do wykonania kroków referencyjnych jest dosyć niewielka.

Określenie, ile kroków rekurencyjnych może obsłużyć Java, jest trudne, ponieważ zależy to od rozmiaru danych umieszczanych na stosie i rozmiaru stosu w momencie rozpoczynania części rekurencyjnej. Ogólnie można przyjąć, że Java obsłuży od 5 do 6 tysięcy kroków.

Możemy sztucznie zwiększyć ten rozmiar, ponieważ Java używa wewnętrznie tzw. mechanizmu memoizacji. Technika ta polega na zapamiętaniu wyniku funkcji lub metody w pamięci w celu przyspieszenia późniejszego dostępu do niej. Zamiast ponownie wyliczać wynik, Java pobiera go po prostu z pamięci. Poza przyspieszeniem związanym z natychmiastowym dostępem do wyniku unikamy również części procesu rekurencyjnego. Powrócimy jeszcze do tego tematu w rozdziale 4., w którym nauczymy się tworzenia w Javie rekurencji bazującej na stosie. W dalszej części tego rozdziału załóżmy jednak, że rekurencja w Javie działa w pełni poprawnie.

Metodę rekurencyjną łatwo zdefiniować. Metodę `factorial(int n)` zdefiniujemy jako metodę zwracającą `1`, jeśli argumentem jest `0`, i `n * factorial(n - 1)` w pozostałych sytuacjach:

```
public int factorial(int n) {
   return n == 0 ? 1 : n * factorial(n - 1);
}
```

Przypomnę, że funkcja spowoduje przepełnienie bufora dla `n` równego od `5000` do `6000`, więc nie używaj tego kodu w systemie produkcyjnym.

Pisanie metod rekurencyjnych jest proste, ale co z funkcjami rekurencyjnymi?

ĆWICZENIE 2.12

Napisz rekurencyjną funkcję dla silni.

WSKAZÓWKA

Nie próbuj napisać anonimowej funkcji rekurencyjnej, ponieważ aby funkcja mogła wywołać samą siebie, musi posiadać nazwę, która jest znana przed ponownym wywołaniem. Ponieważ powinna być zdefiniowana, gdy wywołuje samą siebie, oznacza to, że powinna być zdefiniowana, zanim spróbuje się ją zdefiniować!

ROZWIĄZANIE 2.12

Odstawmy na chwilę na bok ten problem typu, co było pierwsze — kura czy jajko. Konwersja jednoargumentowej metody na funkcję jest prosta. Typem jest `Function<Integer, Integer>`. Implementacja powinna być taka sama jak dla metody:

```
Function<Integer, Integer> factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
```

A teraz trudniejszy kawałek. Kodu nie uda się skompilować, ponieważ kompilator będzie narzekał na `Illegal self reference`. Co to oznacza? W dużym skrócie: kompilator analizuje kod definiujący funkcję `factorial`, w trakcie tego procesu zauważa wywołanie funkcji `factorial`, która jeszcze nie została zdefiniowana.

W konsekwencji definicja lokalnie rekurencyjnej funkcji nie jest możliwa. Czy jednak można zdefiniować taką funkcję jako zmienną składową lub jako zmienną statyczną? Nie rozwiąże to problemu z referencją do samego siebie, bo byłoby to równoważne definicji zmiennej liczbowej w poniższy sposób:

```
int x = x + 1;
```

Problem trzeba rozwiązać przez zadeklarowanie zmiennej i dopiero późniejsze przypisanie jej wartości. Można to zrobić w konstruktorze lub dowolnej innej metodzie, ale najwygodniej wykorzystać w tym celu inicjalizację. Oto przykład:

```
int x;
{
   x = x + 1;
}
```

Teraz wszystko działa poprawnie, ponieważ składowe są definiowane przez inicjalizacją (w trakcie definiowania zmienna otrzyma domyślną wartość — `0` dla `int`, `null` dla funkcji). Fakt, że zmienna jest równa `null`, przez pewien czas nie będzie stanowił problemu, ponieważ chwilę później wykona się konstruktor (wyjątkiem może być sytuacja, gdy inny kod stara się wykorzystać w międzyczasie taką zmienną). Użyjmy opisanej sztuczki do zdefiniowania funkcji:

```
public Function<Integer, Integer> factorial;
```

```
{
   factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
}
```

Sztuczka zadziała także dla statycznie definiowanych funkcji:

```
public static Function<Integer, Integer> factorial;

static {
   factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
}
```

Jedyną wadą tego podejścia jest to, że pola nie można zadeklarować jako `final`, co nie jest szczególnie dobre z uwagi na fakt, iż programiści funkcyjni uwielbiają niezmienność. Na szczęście, mamy do dyspozycji jeszcze jedną sztuczkę:

```
public final Function<Integer, Integer> factorial =
   n -> n <= 1 ? n : n * this.factorial.apply(n - 1);
```

Dodając `this.` przed nazwą zmiennej, możemy odnieść się do niej i jednocześnie użyć modyfikatora `final`. W implementacji statycznej zastąp `this` nazwą dołączanej klasy:

```
public static final Function<Integer, Integer> factorial =
   n -> n <= 1 ? n : n * FunctionExamples.factorial.apply(n - 1);
```

2.3.11. Funkcja tożsamościowa

W programowaniu funkcyjnym funkcje traktuje się podobnie jak dane. Mogą być przekazywane jako argumenty do innych funkcji, mogą być przez inne funkcje zwracane, a także mogą być wykorzystywane w różnych operacjach, podobnie jak liczby całkowite i zmiennoprzecinkowe. W przyszłych programach będziemy dla funkcji używać operatorów, co wymagać będzie pewnego neutralnego elementu. Będzie on działał podobnie jak 0 dla dodawania, 1 dla mnożenia lub pusty tekst dla łączenia tekstów.

Dodajmy funkcję tożsamościową do definicji klasy `Function` w postaci metody o nazwie `identity`, która zwraca właśnie tego typu funkcję:

```
static <T> Function<T, T> identity() {
   return t -> t;
}
```

Po dodaniu tej metody ukończyliśmy tworzenie interfejsu `Function`, który przedstawia listing 2.2.

Listing 2.2. Pełna wersja interfejsu Function

```
public interface Function<T, U> {

   U apply(T arg);

   default <V> Function<V, U> compose(Function<V, T> f) {
      return x -> apply(f.apply(x));
   }

   default <V> Function<T, V> andThen(Function<U, V> f) {
```

```
        return x -> f.apply(apply(x));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }

    static <T, U, V> Function<V, U> compose(Function<T, U> f,
                                            Function<V, T> g) {
        return x -> f.apply(g.apply(x));
    }

    static <T, U, V> Function<T, V> andThen(Function<T, U> f,
                                            Function<U, V> g) {
        return x -> g.apply(f.apply(x));
    }

    static <T, U, V> Function<Function<T, U>,
                              Function<Function<U, V>,
                                       Function<T, V>>> compose() {
        return x -> y -> y.compose(x);
    }

    static <T, U, V> Function<Function<T, U>,
                              Function<Function<V, T>,
                                       Function<V, U>>> andThen() {
        return x -> y -> y.andThen(x);
    }

    static <T, U, V> Function<Function<T, U>,
                              Function<Function<U, V>,
                                       Function<T, V>>> higherAndThen() {
        return x -> y -> z -> y.apply(x.apply(z));
    }

    static <T, U, V> Function<Function<U, V>,
                              Function<Function<T, U>,
                                       Function<T, V>>> higherCompose() {
        return (Function<U, V> x) -> (Function<T, U> y) -> (T z) -> x.apply(y.apply(z));
    }
}
```

2.4. Interfejsy funkcyjne Javy 8

Funkcji anonimowych używa się w miejscach, gdzie dozwolony jest określony interfejs. W ten sposób Java wie, jakie metody może wywołać. Java nie wprowadza żadnych ograniczeń co do nazewnictwa, co zdarza się w innych językach. Jedynym ograniczeniem jest, aby interfejs nie był wieloznaczny, czyli posiadał tylko jedną metodę abstrakcyjną. (Rzeczywistość jest nieco bardziej złożona, bo niektóre metody się nie liczą). Interfejsy tego typu nazywa się **SAM** (skrót od *Single Abstract Method*) lub **interfejsami funkcyjnymi**.

Pamiętaj, że wyrażeń lambda używa się nie tylko do tworzenia funkcji. W standardowej Javie 8 dostępnych jest wiele interfejsów funkcyjnych, choć nie wszystkie są związane z funkcjami. Najważniejsze z nich wymieniłem poniżej.

- Interfejs `java.util.function.Function` najbardziej przypomina interfejs `Function` tworzony w tym rozdziale. Dodaje element wieloznaczny do typów parametrów metod, co czyni je bardziej użytecznymi.
- Interfejs `java.util.function.Supplier` jest równoważny funkcji bez argumentów. W programowaniu funkcyjnym taki element to stała, więc na pierwszy rzut oka może nie wydawać się interesujący, ale ma dwa bardzo konkretne sposoby użycia. Po pierwsze, jeśli nie jest transparentny referencyjnie (nie jest funkcją czystą), może posłużyć do przekazania zmiennych danych (na przykład czasu lub liczb losowych). (Nie będziemy korzystać z takich niefunkcyjnych elementów!). Po drugie, co będzie dla nas interesujące, umożliwia leniwe wyliczanie wartości. Powrócimy do tego tematu w następnych rozdziałach.
- Interfejs `java.util.function.Consumer` nie dotyczy funkcji, ale efektów. (Nie jest to efekt **uboczny**, ponieważ efekt jest jedynym wynikiem działania `Consumer`, bo niczego nie zwraca).
- Interfejs `java.lang.Runnable` pozwala na użycie dla efektów, które dodatkowo nie przyjmują parametrów. Choć lepiej do tego celu wykonać osobny interfejs, bo `Runnable` zbyt mocno kojarzy się z wątkami i niektóre narzędzia do analizy statycznej kodu mogą zgłaszać ostrzeżenia, jeśli będzie używany w innym kontekście.

Java definiuje wiele innych interfejsów funkcyjnych (43 w pakiecie `java.util.function`), ale z punktu widzenia programowania funkcyjnego są one bezużyteczne. Wiele z nich dotyczy typów podstawowych, a inne funkcji dwuargumentowych. Istnieją też wersje specjalne dla operacji (funkcji o dwóch argumentach tego samego typu).

W książce nie będę poruszał zbyt często tematu standardowych funkcji Javy 8. To celowe działanie. To nie jest książka o Javie 8. To książka o programowaniu funkcyjnym, która do prezentacji przykładów wykorzystuje język Java. Nauka polega na konstruowaniu elementów, a nie użyciu gotowych klocków. Jeśli dobrze poznasz wszystkie elementy, będziesz mógł sam zdecydować, czy korzystać z własnych funkcji, czy z rozwiązań proponowanych przez Javę 8. Tworzony w książce interfejs `Function` przypomina interfejs o tej samej nazwie z Javy 8. Nie używa elementów wieloznacznych dla argumentów, aby nie gmatwać kodu prezentowanego w książce. Z drugiej strony, `Function` z Javy 8 nie definiuje `compose` i `andThen` jako funkcji wyższego rzędu, ale jako metody. Poza tymi różnicami obie implementacje `Function` są wymienne.

2.5. Debugging funkcji anonimowych

Wykorzystanie funkcji anonimowych promuje nowy sposób pisania kodu. Kod, który wcześniej pisany był jako kilka krótkich wierszy kodu, zastępowany jest długimi jednowierszowcami, takimi jak:

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {
   return conditions.zip(ifTrue).flatMap(x -> x.first(y -> y._1))
                                .map(x -> x._2).getOrElse(ifFalse);
}
```

(Implementacja metody `ifElse` została podzielona na dwa wiersze tylko z powodu długości wiersza w książce. W kodzie aplikacji będzie to jeden długi wiersz kodu).

W Javie od wersji 5 do 7 ten sam kod można zapisać bez funkcji anonimowych w sposób przedstawiony na listingu 2.3.

Listing 2.3. Jednowierszowa metoda z funkcjami anonimowymi zapisana w starszej wersji języka Java

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {

   Function<Tuple<Boolean, T>, Boolean> f1 =
      new Function<Tuple<Boolean, T>, Boolean>() {
         public Boolean apply(Tuple<Boolean, T> y) {
            return y._1;
         }
      };

   Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>> f2 =
      new Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>>() {
         public Result<Tuple<Boolean, T>> apply(List<Tuple<Boolean, T>> x) {
            return x.first(f1);
         }
      };

   Function<Tuple<Boolean, T>, T> f3 =
      new Function<Tuple<Boolean, T>, T>() {
         public T apply(Tuple<Boolean, T> x) {
            return x._2;
         }
      };

   Result<List<Tuple<Boolean, T>>> temp1 = conditions.zip(ifTrue);
   Result<Tuple<Boolean, T>> temp2 = temp1.flatMap(f2);
   Result<T> temp3 = temp2.map(f3);
   T result = temp3.getOrElse(ifFalse);
   return result;
}
```

Oczywiście, napisanie i późniejsze czytanie wersji z funkcjami anonimowymi jest znacznie prostsze. Wersje sprzed Javy 8 były często zbyt złożone, aby udawało się je zaakceptować. Gdy jednak dochodzimy do debugowania, wersja z funkcjami anonimowymi zaczyna stanowić wyzwanie. Jeśli jeden wiersz kodu jest równoważny wcześniejszym 20, jak umieścić w nim punkt wstrzymania, aby znaleźć potencjalny błąd? Nie wszystkie debuggery są przygotowane, aby działać efektywnie z funkcjami anonimowymi. Prostym rozwiązaniem jest podział wersji jednowierszowej na kilka osobnych wierszy:

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {
   return conditions.zip(ifTrue)
      .flatMap(x -> x.first(y -> y._1))
      .map(x -> x._2)
      .getOrElse(ifFalse);
}
```

W ten sposób punkty wstrzymania można określić osobno dla każdego fizycznego wiersza. To z pewnością rozwiązanie ułatwiające czytanie kodu (i umieszczanie go w książkach). Niestety, nie rozwiązuje naszego głównego problemu, bo każdy wiersz nadal zawiera wiele elementów, które niełatwo sprawdzić w tradycyjnych debuggerach.

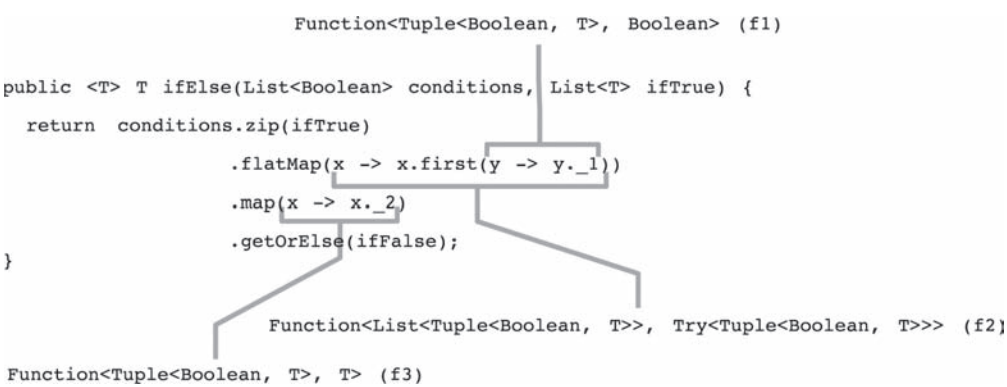
Aby zmniejszyć znaczenie tej kwestii, warto intensywnie testować testami jednostkowymi każdy komponent. Oznacza to sprawdzenie każdej metody i każdej funkcji jako argumentu do każdej z metod. To nic trudnego. W powyższym kodzie użyliśmy po kolei: `List.zip`, `Option.flatMap`, `List.first`, `Option.map` i `Option.getOrElse`. To, co robi każda z tych metod, można dobrze przetestować. Choć jeszcze tego nie wiesz, zbudujemy w następnych rozdziałach komponenty `Option` i `List`, a także napiszemy implementacje metod `map`, `flatMap`, `first`, `zip` i `getOrElse` (oraz wielu innych). Jak się przekonasz, metody te są czysto funkcyjne. Nie mogą zgłosić żadnych wyjątków i zawsze zwracają oczekiwany wynik bez dodatkowych działań ubocznych. Jeśli więc są w pełni przetestowane, nie może stać się nic złego.

Od strony funkcji przedstawiony kod używa trzech różnych:

- `x x.first`
- `y y._1`
- `x x._2`

Pierwsza funkcja nie może zgłosić żadnego wyjątku, ponieważ x nie może być równe `null` (przyczyny poznasz w rozdziale 5.), a metoda `first` również nie zgłasza wyjątków.

Druga i trzecia funkcja nie może zgłosić wyjątku `NullPointerException`, ponieważ `Tuple` nie można utworzyć z argumentu z wartością `null`. (Kod klasy `Tuple` znajduje się w rozdziale 1.). Rysunek 2.4 przedstawia funkcje w ich anonimowej formie.



Rysunek 2.4. Funkcje w anonimowej formie

To jeden z obszarów, w których programowanie funkcyjne bryluje — jeśli żaden z komponentów nie może zawieść, cały program również nie zawiedzie. W programowaniu imperatywnym komponenty mogą działać poprawnie w testach, ale nie działać w systemie produkcyjnym z powodu niedeterministycznego zachowania. Jeśli zachowanie komponentu zależy od warunków zewnętrznych, nie można go w pełni przetestować. Nawet jeśli

komponent nie wykazuje żadnych problemów testowany jednostkowo, nie wiadomo, czy kompozycja wielu komponentów nie spowoduje błędnego działania. W programowaniu funkcyjnym nie ma takich sytuacji. Jeśli komponenty zachowują się deterministycznie, podobnie będzie z ich kompozycją.

Oczywiście, nadal pozostaje wiele miejsc mogących doprowadzić do błędów. Program może nie realizować zadania, które powinien, bo komponenty złożono w niewłaściwy sposób. Niemniej błędy implementacyjne nie mogą doprowadzić do nieoczekiwanego zawieszenia się aplikacji. Wyłączenie się aplikacji może spowodować przekazanie referencji o wartości `null` do konstruktora `Tuple`. Aby wychwycić taki błąd, nie potrzeba jednak debuggera.

Podsumowując: debugowanie programów funkcyjnych wykorzystujących funkcje anonimowe jest trudniejsze niż programów imperatywnych, ale z drugiej strony potrzeba znacznie mniej debugowania, jeśli sprawdzono poprawność działania elementów składowych. Pamiętaj, że wszystkie stwierdzenia pozostaną prawdziwe tylko wtedy, gdy zgłoszony wyjątek wyłącza program. Zajmiemy się tą kwestią w rozdziale 6. Przypomnę, że domyślnie zgłoszenie wyjątku lub błędu spowoduje wyłączenie tylko i wyłącznie wątku, w którym to miało miejsce, a nie całej aplikacji. Nawet błąd `OutOfMemoryError` może nie wyłączyć całej aplikacji, więc odpowiednia obsługa takiej sytuacji spoczywa na programiście.

2.6. Podsumowanie

- Funkcja to relacja między zbiorem źródłowym i zbiorem docelowym. Określa związek między elementami zbioru źródłowego (dziedzina) i elementami zbioru docelowego (przeciwdziedzina).
- Funkcje czyste nie mają żadnych widocznych efektów poza zwróceniem wartości.
- Funkcje przyjmują tylko jeden argument, który może być krotką z kilkoma wartościami.
- Funkcje krotek można rozwinąć, aby móc stosować je dla pojedynczych elementów krotki.
- Gdy dla rozwiniętej funkcji zastosujemy część argumentów, mówimy o jej częściowym zastosowaniu.
- W Javie funkcje mogą być reprezentowane przez metody, funkcje anonimowe, referencje do metod i klasy anonimowe.
- Referencje do metod to zalecany sposób reprezentacji funkcji.
- Można tworzyć kompozycje funkcji, aby generować nowe funkcje.
- Funkcje mogą wywoływać same siebie w sposób rekurencyjny, ale głębia rekurencji jest ograniczona rozmiarem stosu.
- Referencje do metod i funkcje anonimowe mogą pojawić się w miejscach, w których Java oczekuje interfejsu funkcyjnego.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Programowanie funkcyjne — pisz kod funkcjonalny!

Większość programistów pracuje zgodnie z paradygmatem programowania imperatywnego, który polega na tworzeniu ciągu instrukcji zmieniających stan programu. Najpoważniejszą wadą tej metody pracy jest podatność kodu na błędy, które później jest trudno wykryć i usunąć. Alternatywą jest programowanie funkcyjne — metodyka kładąca największy nacisk na stałe i funkcje. Takie programowanie polega na konstruowaniu funkcji oraz na obliczaniu wartości wyrażeń. W ten sposób otrzymuje się kod odporny na błędy. Niestety, nie zawsze można skorzystać z języków do programowania funkcyjnego.

Ta książka stanowi znakomite wprowadzenie do programowania funkcyjnego na przykładzie Javy. Przedstawiono tu zasady programowania funkcyjnego i metody budowania funkcyjnych struktur danych. Poprzez poznanie paradygmatu funkcyjnego możliwe jest pisanie lepszych programów, a tworzony kod zawiera mniej błędów i staje się zdecydowanie bardziej niezawodny. W każdym rozdziale znalazły się przykłady kodu, a także ćwiczenia, instrukcje i wskazówki, dzięki którym opanowanie poszczególnych koncepcji będzie o wiele łatwiejsze. Wyczerpująco omówiono tu m.in. transparentność referencyjną, niezmienność, trwałość i leniwe obliczanie wartości.

Najważniejsze zagadnienia: sterowanie wykonaniem programu / różne rodzaje funkcji w Javie / rekurencja i jej zastosowania / operacje wejścia-wyjścia / obsługa błędów / Java 8 a programowanie funkcyjne

Pierre-Yves Saumont jest doświadczonym programistą Javy. Od trzydziestu lat tworzy oprogramowanie wykorzystywane w przedsiębiorstwach. Pracuje jako inżynier do spraw badań i rozwoju w firmie Alcatel-Lucent Submarine Networks. W 1999 r. napisał pierwszą francuskojęzyczną książkę traktującą o programowaniu w Javie (*Le guide du developpeur Java*).

ISBN 978-83-283-3324-6

9 788328 333246

Informatyka w najlepszym wydaniu

cena: 89,00 zł