O'REILLY®

Java

Podejście funkcyjne

Rozszerzanie obiektowego kodu Javy
o zasady programowania funkcyjnego

Ben Weidig

Helion

Tytuł oryginału: A Functional Approach to Java: Augmenting Object-Oriented Java Code with Functional Principles

Tłumaczenie: Lech Lachowski

ISBN: 978-83-289-0651-8

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Funkcyjna Java

Nic ma w tym nic dziwnego, że kluczem do funkcyjnego podejścia w Javie są **wyrażenia lambda**.

Z tego rozdziału dowiesz się m.in.: jak używać wyrażeń lambda w Javie, dlaczego są one tak ważne, jak je efektywnie wykorzystywać i na czym polega ich wewnętrzny mechanizm działania.

Czym są wyrażenia lambda Javy?

Wyrażenie lambda to pojedyncza linia lub blok kodu Javy, które mają zero lub więcej parametrów i mogą zwracać wartość. W uproszczeniu lambda jest jak **metoda anonimowa**, która nie należy do żadnego obiektu:

```
() -> System.out.println("Witaj, lambdo!")
```

Przyjrzyjmy się szczegółom składni i sposobu implementacji wyrażeń lambda w Javie.

Składnia wyrażeń lambda

Składnia Javy dla wyrażeń lambda jest dość podobna do notacji matematycznej, którą pokazałem w rozdziale 1. dla rachunku lambda:

```
(<parametery>) -> { <ciało> }
```

Ta składnia ma trzy odrębne części:

Parametry

Lista parametrów oddzielonych przecinkami, podobna do listy argumentów metody. Jednak — inaczej niż w przypadku argumentów metody — można pominąć typy argumentów, jeżeli kompilator może je wywnioskować. Mieszanie parametrów typowanych w sposób dorozumiany i bezpośredni jest niedozwolone. Dla pojedynczego parametru nawiasy nie są potrzebne, natomiast są one wymagane w przypadku zera lub więcej niż jednego parametru.

Strzałka

Strzałka (`->`) oddziela parametry od ciała wyrażenia lambda. Jest to odpowiednik znaku λ w rachunku lambda.

Ciało

Pojedyncze wyrażenie lub blok kodu. Wyrażenia jednoliniowe nie wymagają nawiasów klamrowych, a wynik ich wartościowania jest zwracany domyślnie bez konieczności stosowania instrukcji `return`. Jeśli ciało jest reprezentowane przez więcej niż jedno wyrażenie, używany jest typowy blok kodu Javy. Kod musi być umieszczony w nawiasach klamrowych i zawierać instrukcję `return`, jeśli ma być zwracana wartość.

To jest cała definicja składni dla wyrażeń lambda w Javie. Dzięki wielu sposobom deklarowania wyrażeń lambda tę samą lambdę można zapisać z różnymi poziomami szczegółowości, jak pokazałem w listingu 2.1.

Listing 2.1. Różne sposoby zapisywania tej samej lambdy

```
(String input) -> { ❶
  return input !w null;
}

input -> { ❷
  return input != null;
}

(String input) -> input != null ❸

input -> input != null ❹
```

❶ Najbardziej szczegółowy wariant: bezpośrednio typowany parametr w nawiasach i blok ciała.

❷ Pierwszy wariant mieszany: inferencja typu dla parametrów nie wymaga bezpośredniego określania typu, a pojedynczy parametr nie potrzebuje nawiasów. Kontekst, w jakim umieszczona jest deklaracja lambdy pozwala nieznacznie ją skrócić bez usuwania żadnych informacji.

❸ Drugi wariant mieszany: bezpośrednio typowany parametr w nawiasach, ale zamiast bloku ciało pojedynczego wyrażenia; nie są potrzebne nawiasy klamrowe ani instrukcja `return`.

❹ Najbardziej zwięzły wariant, ponieważ ciało można zredukować do pojedynczego wyrażenia.

Wybór wariantu zależy w dużej mierze od kontekstu i osobistych preferencji. Zazwyczaj kompilator może inferować (wnioskować) typy, natomiast osoba czytająca ten kod może mieć problemy ze zrozumieniem jego najkrótszej postaci.

Chociaż zawsze należy dążyć do czystego i zwięzłego kodu, nie oznacza to, że musi on być minimalistyczny. Pewien poziom szczegółowości może pomóc każdemu czytelnikowi kodu — również Tobie — w lepszym zrozumieniu jego działania i mentalnego modelu.

Interfejsy funkcyjne

Do tej pory przyglądaliśmy się ogólnej koncepcji wyrażeń lambda w izolacji. Muszą one jednak istnieć wewnątrz Javy, jej pojęć i reguł językowych.

Java znana jest ze swojej kompatybilności wstecznej. Dlatego nawet jeśli składnia wyrażeń lambda jest przełomową zmianą w składni samej Javy, nadal opiera się na zwykłych interfejsach, co pozwala zapewnić kompatybilność wsteczną i znajome środowisko każdemu programiście tego języka.

Aby wyrażenia lambda w Javie mogły być typami pierwszoklasowymi, wymagają reprezentacji porównywalnej z istniejącymi typami, takimi jak obiekty i typy proste, o czym pisałem w rozdziale 1., w punkcie „Typy pierwszoklasowe i funkcje wyższego rzędu". Dlatego lambdy są reprezentowane przez wyspecjalizowany podtyp interfejsów, tzw. **interfejsy funkcyjne**.

Interfejsy w języku Java

Deklaracja interfejsu składa się z nazwy z opcjonalnymi ograniczeniami typów sparametryzowanych, odziedziczonymi interfejsami i ciałem interfejsu. Ciało może zawierać następujące elementy:

Sygnatury metod

Sygnatury metod nieposiadających ciała, czyli metod abstrakcyjnych (`abstract`), które muszą być implementowane przez dowolną klasę zgodną z interfejsem. Tylko sygnatury tych metod są uwzględniane w ograniczeniach **interfejsów funkcyjnych**, które mogą definiować tylko **pojedynczą metodę abstrakcyjną**.

Metody domyślne

Sygnatury metod mogą mieć implementację „domyślną", oznaczoną słowem kluczowym `default` i blokiem ciała. Każda klasa implementująca dany interfejs **może** nadpisać domyślną implementację metody, ale *nie jest to wymagane*.

Metody statyczne

Metody statyczne, podobnie jak ich odpowiedniki oparte na klasach, są powiązane z samym typem i muszą zapewniać implementację. Jednak — w przeciwieństwie do metod `default` — nie są dziedziczone i nie mogą być nadpisywane.

Wartości stałe

Wartości, które są automatycznie publiczne (`public`), statyczne (`static`) i finalne (`final`).

Dla interfejsów funkcjonalnych nie ma żadnej określonej składni ani słowa kluczowego języka. Wyglądają i zachowują się jak każdy inny interfejs, mogą rozszerzać inne interfejsy lub być przez nie rozszerzane, a klasy mogą je implementować. Skoro są więc jak „normalne" interfejsy, co czyni je interfejsami „funkcyjnymi"? Chodzi o ograniczenie, które pozwala im definiować tylko **pojedynczą metodę abstrakcyjną** (ang. *Single Abstract Method* — SAM).

Jak wskazuje sama nazwa, liczba metod SAM odnosi się tylko do metod `abstract`. Nie ma natomiast ograniczeń dla żadnych dodatkowych metod innych niż `abstract`. Ani metody `default`, ani `static` nie są abstrakcyjne, dlatego nie są liczone jako SAM. Z tego powodu są one często używane do uzupełniania funkcjonalności typu lambda.

Większość interfejsów funkcyjnych pakietu JDK oferuje dodatkowe metody `default` i `static` związane z typem. Sprawdzanie deklaracji poszczególnych interfejsów funkcyjnych może ujawnić wiele ukrytych funkcjonalności.

Rozważmy listing 2.2, przedstawiający uproszczoną wersję[1] interfejsu funkcyjnego `java.util.function.Predicate<T>`. Predykat, który jest przeznaczony do testowania warunków; omówię go szerzej w rozdziale 3., w podrozdziale „Cztery główne kategorie interfejsów funkcyjnych”. Oprócz posiadania pojedynczej metody abstrakcyjnej, `boolean test(T t)`, zapewnia on pięć dodatkowych metod (trzy `default` i dwie `static`).

Listing 2.2. Uproszczony interfejs java.util.function.Predicate<T>

```
package java.util.function;

@FunctionalInterface ❶
public interface Predicate<T> {

  boolean test(T t); ❷

  default Predicate<T> and(Predicate<? super T> other) { ❸
    //...
  }

  default Predicate<T> negate() { ❸
    //...
  }

  default Predicate<T> or(Predicate<? super T> other) { ❸
    //...
  }

  static <T> Predicate<T> isEqual(Object targetRef) { ❹
    //...
  }

  static <T> Predicate<T> not(Predicate<? super T> target) { ❹
    //...
  }
}
```

❶ Ten typ używa adnotacji `@FunctionalInterface`, która nie jest bezpośrednio wymagana.

❷ Pojedyncza metoda abstrakcyjna tego typu.

❸ Kilka metod `default` zapewnia wsparcie dla kompozycji funkcyjnej.

❹ Metody pomocnicze `static` są wykorzystywane do upraszczania tworzenia wyrażeń lambda lub opakowywania istniejących.

Każdy interfejs z pojedynczą metodą abstrakcyjną jest automatycznie interfejsem funkcyjnym. Z tego powodu każda z jego implementacji jest reprezentowana przez wyrażenie lambda.

[1] Uproszczona wersja interfejsu `java.util.function.Predicate` została oparta na kodzie źródłowym wersji LTS z tagu Gita `jdk-17+35`, która jest najnowszą wersją, w momencie gdy piszę ten rozdział. Pełny kod źródłowy możesz znaleźć w oficjalnym repozytorium (*https://oreil.ly/Amx25*).

W wersji języka Java 8 dodana została adnotacja znacznikowa `@FunctionalInterface`, aby wyegzekwować wymóg SAM na poziomie kompilatora. Nie jest ona obowiązkowa, ale instruuje kompilator i ewentualnie inne narzędzia bazujące na adnotacjach, że dany interfejs powinien być interfejsem funkcyjnym, a co za tym idzie, trzeba wyegzekwować wymóg pojedynczej metody abstrakcyjnej. Jeżeli dodasz kolejną metodę `abstract`, kompilator Javy nie skompiluje kodu. Dlatego dodawanie adnotacji do interfejsów funkcyjnych ma sens, nawet jeśli nie są one bezpośrednio wymagane. Ułatwia to zrozumienie kodu i intencji takiego interfejsu oraz uodparnia kod na niezamierzone zmiany, które mogą go w przyszłości popsuć.

Opcjonalny charakter adnotacji `@FunctionalInterface` pozwala ponadto zachować wsteczną kompatybilność istniejących interfejsów. Dopóki interfejs spełnia wymagania SAM, jest reprezentowany jako lambda. Interfejsy funkcyjne pakietu JDK omówię dalej w tym rozdziale.

Wyrażenia lambda i zmienne zewnętrzne

W rozdziale 1., w punkcie „Funkcje czyste i transparentność referencyjna", wprowadziłem koncepcję **czystych** — samodzielnych i wolnych od skutków ubocznych — funkcji, które nie mają wpływu na żaden stan zewnętrzny i opierają się wyłącznie na własnych argumentach. Chociaż wyrażenia lambda mają zasadniczo ten sam sens, dopuszczają jednak pewien poziom nieczystości, aby zapewnić większą elastyczność. Mogą one „przechwytywać" stałe i zmienne z zakresu ich utworzenia, w którym zdefiniowana jest dana lambda, co udostępnia jej takie zmienne nawet wtedy, gdy pierwotny zakres już nie istnieje. Pokazałem to w listingu 2.3.

Listing 2.3. Przechwytywanie zmiennych przez lambdę

```
void capture() {
  var theAnswer = 42; ❶

  Runnable printAnswer =
    () -> System.out.println("odpowiedź to " + theAnswer); ❷

  run(printAnswer); ❸
}

void run(Runnable r) {
  r.run();
}

capture();
// Dane wyjściowe:
// odpowiedź to 42
```

❶ Zmienna `theAnswer` jest deklarowana w zakresie metody `capture`.

❷ Lambda `printAnswer` przechwytuje tę zmienną w swoim ciele.

❸ Lambda może zostać uruchomiona w innej metodzie i innym zakresie, ale nadal będzie miała dostęp do zmiennej `theAnswer`.

Lambdy **przechwytujące** i **nieprzechwytujące** stanowią dużą różnicę dla strategii optymalizacji maszyny wirtualnej Javy (ang. *Java Virtual Machine* — JVM). JVM optymalizuje lambdy za pomocą różnych strategii, w zależności od ich rzeczywistego wzorca użycia. Jeśli nie są przechwytywane żadne zmienne, lambda może stać się za kulisami prostą metodą `static`, osiągając lepszą wydajność niż alternatywne podejścia, takie jak klasy anonimowe. Implikacje przechwytywania zmiennych dla wydajności nie są jednak tak jednoznaczne.

Jeżeli kod przechwytuje zmienne, JVM może przełożyć go na wiele sposobów, co może prowadzić do alokacji dodatkowych obiektów i wpływać na wydajność oraz czasy działania mechanizmu odzyskiwania pamięci (ang. *garbage collector*). Nie oznacza to, że przechwytywanie zmiennych jest z natury złym wyborem projektowym. Głównymi celami bardziej funkcyjnego podejścia powinny być poprawa produktywności, prostsze rozumowanie i bardziej zwięzły kod. Mimo to należy unikać niepotrzebnego przechwytywania, zwłaszcza jeśli dąży się do jak najmniejszej liczby alokacji lub najlepszej możliwej wydajności.

Kolejnym powodem do unikania przechwytywania zmiennych jest to, że muszą one być **efektywnie** finalne.

Zmienne efektywnie finalne

JVM musi dołożyć wszelkich starań, aby bezpiecznie korzystać z przechwyconych zmiennych i osiągnąć najlepszą możliwą wydajność. Dlatego zasadniczym wymogiem przechwytywania jest to, że może ono odbywać się tylko w przypadku zmiennych **efektywnie** finalnych.

Mówiąc prościej, każda przechwytywana zmienna musi być referencją niemutowalną, która po zainicjowaniu nie może ulec zmianie. Finalność osiąga się przez bezpośrednie użycie słowa kluczowego `final` lub niezmienianie zmiennej po jej zainicjowaniu — w ten sposób staje się ona efektywnie finalna.

Należy pamiętać, że wymóg ten dotyczy w rzeczywistości **referencji** do zmiennej, a nie samej bazowej struktury danych. Referencja do `List<String>` może być `final` i dlatego może być wykorzystywana w wyrażeniach lambda, ale nadal można dodawać do tej listy nowe elementy, jak pokazałem w listingu 2.4. Zabronione jest jedynie ponowne przypisywanie zmiennej.

Listing 2.4. Zmiana danych kryjących się pod zmienną finalną

```
final List<String> wordList = new ArrayList<>(); ❶

// KOMPILUJE SIĘ BEZ PROBLEMU
Runnable addItemInLambda = () -> wordList.add("dodawanie jest w porządku"); ❷

// NIE KOMPILUJE SIĘ
wordList = List.of("przypisywanie", "kolejnej", "List", "nie", "jest"); ❸
```

❶ Zmienna `wordList` jest bezpośrednio `final`, przez co referencja jest niemutowalna.

❷ Przechwytywanie i używanie tej zmiennej w wyrażeniach lambda działa bez problemów. Słowo kluczowe `final` nie wpływa jednak na samą `List<String>`, umożliwiając dodawanie kolejnych elementów.

❸ Ponowne przypisanie zmiennej jest zabronione ze względu na słowo kluczowe `final` i się nie kompiluje.

Najprostszym sposobem sprawdzenia, czy zmienna jest efektywnie finalna, jest zadeklarowanie jej bezpośrednio jako `final`. Jeśli kod z dodatkowym słowem kluczowym `final` wciąż będzie się kompilował, skompiluje się również bez niego. Dlaczego więc nie zadeklarować każdej zmiennej jako `final`? Skoro kompilator upewnia się, że odwołania „poza ciało" są efektywnie `final`, to słowo kluczowe i tak nie pomoże w kwestii zapewnienia rzeczywistej niemutowalności. Deklarowanie każdej zmiennej jako `final` zmniejszyłoby jedynie przejrzystość kodu bez oferowania większych korzyści. Dodanie modyfikatora, takiego jak `final`, zawsze powinno być świadomą decyzją, której przyświecają określone intencje.

Jeśli uruchomisz w `jshell` którykolwiek z pokazanych przykładów zmiennych efektywnie finalnych, mogą one nie zachowywać się zgodnie z oczekiwaniami. Jest to spowodowane tym, że `jshell` ma specjalną semantykę dotyczącą wyrażeń i deklaracji najwyższego poziomu, która wpływa na wartości `final` lub efektywnie `final` na najwyższym poziomie[2]. Nawet jeśli możesz ponownie przypisać dowolną referencję, czyniąc je nieefektywnie `final`, możesz nadal używać jej w wyrażeniach lambda, o ile nie jesteś w zakresie najwyższego poziomu.

Ponowne finalizowanie referencji

Czasami referencja może nie być efektywnie finalna, ale mimo to możesz chcieć, aby była dostępna w wyrażeniu lambda. Jeśli refaktoryzacja kodu nie wchodzi w grę, istnieje prosta sztuczka **ponownego finalizowania** (refinalizowania). Pamiętaj, że wymóg ten dotyczy tylko referencji, a nie samej bazowej struktury danych.

Można utworzyć nową efektywnie finalną referencję do nieefektywnie finalnej zmiennej; wystarczy odwołać się do pierwotnej zmiennej i nie mutować już tej nowej referencji, jak pokazałem w listingu 2.5.

Listing 2.5. Ponowne finalizowanie referencji

```
var nonEffectivelyFinal = 1_000L; ❶

nonEffectivelyFinal = 9_000L; ❷

var finalAgain = nonEffectivelyFinal; ❸

Predicate<Long> isOver9000 = input -> input > finalAgain;
```

❶ Na tym etapie zmienna `nonEffectivelyFinal` jest nadal efektywnie finalna.

❷ Zmutowanie zmiennej po jej zainicjowaniu sprawia, że nie nadaje się ona do użytku w wyrażeniach lambda.

❸ Przez utworzenie nowej zmiennej i niemutowanie jej po zainicjowaniu „refinalizujemy" odwołanie do bazowej struktury danych.

[2] Oficjalna dokumentacja (*https://oreil.ly/dvmJp*) rzuca nieco światła na specjalną semantykę i wymagania dla wyrażeń i deklaracji najwyższego poziomu.

Pamiętaj, że ponowne finalizowanie referencji to tylko miejscowy opatrunek, a skoro go potrzebujesz, prawdopodobnie otarłeś sobie kolana. Najlepszym podejściem jest więc staranie się, aby ten opatrunek w ogóle nie był potrzebny. Preferowaną opcją powinna być zawsze refaktoryzacja (przeprojektowanie) kodu zamiast naginania go do własnej woli za pomocą sztuczek, takich jak refinalizowanie referencji.

Początkowo zabezpieczenia dotyczące stosowania zmiennych w lambdach, takie jak wymóg efektywnej finalności, mogą wydawać się dodatkowym obciążeniem. Jednak zamiast przechwytywać zmienne „spoza ciała", należy starać się, aby lambdy były samowystarczalne i wymagały podania wszystkich niezbędnych danych jako argumentów. To automatycznie prowadzi do tworzenia bardziej sensownego kodu, zwiększa możliwości wielokrotnego użycia i pozwala na łatwiejsze refaktoryzowanie i testowanie.

A co z klasami anonimowymi?

Poznałeś już wyrażenia lambda i interfejsy funkcyjne, najprawdopodobniej zaczynasz więc dostrzegać ich podobieństwa do **anonimowych klas wewnętrznych**, które stanowią połączenie deklaracji i tworzenia instancji typów. Interfejs lub rozszerzaną klasę można zaimplementować „w locie" i nie jest do tego potrzebna osobna klasa Javy. Czym różni się więc wyrażenie lambda od klasy anonimowej, jeśli jedno i drugie musi implementować konkretny interfejs?

Na pierwszy rzut oka interfejs funkcyjny zaimplementowany przez anonimową klasę wygląda dość podobnie do jego postaci lambda, z wyjątkiem dodatkowego szablonowego kodu (ang. *boilerplate code*), jak pokazałem w listingu 2.6.

Listing 2.6. Porównanie klasy anonimowej i wyrażenia lambda

```
// INTERFEJS FUNKCYJNY (domyślny)
interface HelloWorld {
  String sayHello(String name);
}

// JAKO KLASA ANONIMOWA
var helloWorld = new HelloWorld() {
  @Override
  public String sayHello(String name) {
    return "witaj, " + name + "!";
  }
};

// JAKO LAMBDA
HelloWorld helloWorldLambda = name -> "witaj, " + name + "!";
```

Czy to oznacza, że wyrażenia lambda są jedynie **lukrem syntaktycznym** do implementowania interfejsu funkcyjnego jako klasy anonimowej?

Lukier syntaktyczny

Termin „lukier syntaktyczny” oznacza dodatkowe funkcjonalności języka, które mają „osłodzić” życie programisty — pewne konstrukcje mogą być wyrażane bardziej zwięźle, przejrzyście lub w alternatywny sposób.

Peter J. Landin ukuł ten termin w 1964 r.[3], opisując, w jaki sposób słowo kluczowe `where` zastąpiło λ w języku typu ALGOL.

Instrukcja `import` Javy pozwala na przykład na używanie typów bez ich w pełni kwalifikowanych nazw. Innym przykładem jest wnioskowanie (inferencja) typu przy użyciu `var` dla referencji lub operatora `<>` (tzw. diamentu) dla typów sparametryzowanych. Obie te funkcjonalności upraszczają kod i ułatwiają jego czytanie. Kompilator „ściąga” lukier z kodu i radzi sobie bezpośrednio z jego „goryczą”.

Wyrażenia lambda mogą wyglądać jak lukier syntaktyczny, ale w rzeczywistości znaczą o wiele więcej. **Prawdziwa** różnica — poza długością kodu — polega na generowanym kodzie bajtowym (zobacz listing 2.7) i na tym, jak obsługuje go środowisko uruchomieniowe.

Listing 2.7. Różnice między klasami anonimowymi a wyrażeniami lambda w zakresie kodu bajtowego

```
// KLASA ANONIMOWA
0: new #7 // Klasa HelloWorldAnonymous$1 ❶
3: dup
4: invokespecial #9 // Metoda HelloWorldAnonymous$1."<init>":()V ❷
7: astore_1
8: return

// LAMBDA
0: invokedynamic #7, 0 // InvokeDynamic #0:sayHello:()LHelloWorld; ❸
5: astore_1
6: return
```

❶ W zewnętrznej klasie `HelloWorldAnonymous$1` tworzony jest nowy obiekt anonimowej klasy wewnętrznej `HelloWorldAnonymous`.

❷ Wywoływany jest konstruktor klasy anonimowej. W JVM tworzenie obiektów jest procesem dwuetapowym.

❸ Całą logikę stojącą za tworzeniem wyrażenia lambda ukrywa kod operacyjny `invokedynamic`.

Oba warianty mają wspólne wywołania: `astore_1`, które zapisuje referencję w zmiennej lokalnej, i `return`. Z tego powodu żadne z nich nie będzie częścią analizowania kodu bajtowego.

[3] Peter J. Landin, *The Mechanical Evaluation of Expressions*, „The Computer Journal”, t. 6, nr 4 (1964), s. 308 – 320 (*https://oreil.ly/Ee6kW*).

Anonimowa wersja klasy tworzy nowy obiekt o anonimowym typie `HelloWorldAnonymous$1`, czego rezultatem są trzy kody operacyjne:

`new`
: Tworzy nową, niezainicjowaną instancję typu.

`dup`
: Umieszcza wartość na szczycie stosu przez jej zduplikowanie.

`invokespecial`
: Wywołanie metody konstruktora nowo utworzonego obiektu w celu sfinalizowania jego inicjowania.

Z kolei wersja lambda nie musi tworzyć instancji, która musi zostać umieszczona na stosie. Zamiast tego deleguje całe zadanie tworzenia lambdy do JVM za pomocą pojedynczego kodu operacyjnego: `invokedynamic`.

Instrukcja invokedynamic

W Javie 7 wprowadzono nowy kod operacyjny `invokedynamic`[4] maszyny wirtualnej Javy, aby umożliwić bardziej elastyczne wywoływanie metod do obsługi języków dynamicznych, takich jak Groovy (*https://oreil.ly/Db9Q4*) lub JRuby (*https://oreil.ly/gW1Uh*). Kod operacyjny (ang. *opcode*) jest bardziej wszechstronnym wariantem wywołania, ponieważ podczas ładowania klas nieznany jest jego rzeczywisty cel, taki jak wywołanie metody lub wykonanie ciała wyrażenia lambda. Zamiast dokonywać powiązania takiego obiektu docelowego w czasie kompilacji, JVM łączy dynamiczne miejsce wywoływania (`CallSite`) z rzeczywistą metodą docelową.

Następnie środowisko uruchomieniowe przy pierwszym wywołaniu `invokedynamic` używa „metody ładowania początkowego”[5] (ang. *bootstrap method*), aby określić, która metoda powinna zostać faktycznie wywołana.

Możesz potraktować to jak przepis na tworzenie wyrażeń lambda, który wykorzystuje refleksję bezpośrednio w JVM. W ten sposób JVM może optymalizować zadanie tworzenia za pomocą różnych strategii, takich jak dynamiczne proxy, anonimowe klasy wewnętrzne lub `java.lang.invoke.MethodHandle`.

Kolejną dużą różnicą między wyrażeniami lambda a anonimowymi klasami wewnętrznymi jest ich zakres. Klasa wewnętrzna tworzy własny zakres, ukrywając swoje zmienne lokalne przed zakresem zewnętrznym. Dlatego słowo kluczowe `this` odwołuje się do instancji klasy wewnętrznej, a nie zewnętrznego zakresu. Natomiast lambdy są w pełni dostępne w otaczającym je zakresie. Zmienne nie mogą być ponownie deklarowane z tą samą nazwą, a `this` odnosi się do instancji, w której lambda została utworzona, jeśli nie jest statyczna (`static`).

Jak widać, wyrażenia lambda w żaden sposób *nie* są lukrem syntaktycznym.

---

[4] W Java Magazine znajdziesz artykuł (*https://oreil.ly/KrkQo*) autorstwa mistrza Javy Bena Evansa, w którym wyjaśnia on szerzej, na czym polega wywoływanie metod za pomocą `invokedynamic`.

[5] Za tworzenie „metod ładowania początkowego” odpowiedzialna jest klasa `java.lang.invoke.LambdaMetafactory` (*https://oreil.ly/3E-fO*).

Wyrażenia lambda w działaniu

Jak dowiedziałeś się z lektury poprzedniego podrozdziału, lambdy są niezwykłym dodatkiem do Javy, który ma na celu poprawienie jej możliwości programowania funkcyjnego, i są więcej niż tylko lukrem składniowym dla wcześniej dostępnych technik. Ponieważ są typami pierwszoklasowymi, mogą być statycznie typowanymi, zwięzłymi i anonimowymi funkcjami, które przypominają wszystkie pozostałe zmienne. Chociaż składnia strzałkowa może być nowością, ogólny wzorzec użycia powinien być znany każdemu programiście. W tym podrozdziale przejdziemy od razu do używania wyrażeń lambda i zobaczysz je w działaniu.

Tworzenie wyrażeń lambda

Tworzone wyrażenie lambda musi reprezentować pojedynczy interfejs funkcyjny. Rzeczywisty typ może nie być oczywisty, ponieważ wymagany typ jest dyktowany przez argument metody odbierającej lub inferowany przez kompilator, jeśli jest to możliwe.

Aby lepiej zrozumieć to zagadnienie, przyjrzyjmy się jeszcze raz interfejsowi `Predicate<T>`.

Utworzenie nowej instancji wymaga zdefiniowania typu po lewej stronie:

```
Predicate<String> isNull = value -> value == null;
```

Nawet jeśli dla argumentów użyjemy bezpośrednio określonych typów, nadal będzie wymagany typ interfejsu funkcyjnego:

```
// NIE SKOMPILUJE SIĘ
var isNull = (String value) -> value == null;
```

Sygnaturę metody SAM `Predicate<String>` można wywnioskować:

```
boolean test(String input)
```

Kompilator Javy wymaga jednak konkretnego typu dla odniesienia, a nie tylko sygnatury metody. To wymaganie wynika z zasady zapewniania przez Javę kompatybilności wstecznej, o czym wspomniałem już wcześniej. Wykorzystując istniejący system typowania statycznego, wyrażenia lambda idealnie wpasowują się w Javę i oferują takie samo bezpieczeństwo w czasie kompilacji, jak każdy inny typ lub jakiekolwiek wcześniejsze podejście.

Stosowanie się do systemu typów sprawia jednak, że wyrażenia lambda Javy są mniej dynamiczne niż ich odpowiedniki w innych językach. Nawet jeśli dwie lambdy mają tę samą sygnaturę SAM, nie oznacza to, że są wymienne.

Weźmy na przykład następujący interfejs funkcyjny:

```
interface LikePredicate<T> {
  boolean test(T value);
}
```

Mimo że jego metoda SAM jest identyczna z SAM interfejsu `Predicate<T>`, typy te nie mogą być używane zamiennie, co pokazałem w poniższym kodzie:

```
LikePredicate<String> isNull = value -> value == null; ❶

Predicate<String> wontCompile = isNull; ❷
// Error:
// incompatible types: LikePredicate<java.lang.String> cannot be converted
// to java.util.function.Predicate<java.lang.String>
```

❶ Wyrażenie lambda jest tworzone tak jak poprzednio.

❷ Próba przypisania go do interfejsu funkcyjnego z identyczną metodą SAM nie kompiluje się.

Ze względu na tę niekompatybilność, aby zmaksymalizować interoperacyjność, należy raczej opierać się na interfejsach dostępnych w pakiecie `java.util.function`, które omówię w rozdziale 3. Nadal będziesz napotykać interfejsy z wersji wcześniejszych niż Java 8, takie jak `java.util.con` ↪`current.Callable<V>`, które są identyczne z określonymi interfejsami Javy 8 (i nowszej); w tym przypadku będzie to `java.util.function.Supplier<T>`. Jeśli znajdziesz się w takiej sytuacji, możesz zastosować zgrabny skrót do przełączania lambdy na inny identyczny typ. Więcej informacji na ten temat znajdziesz w rozdziale 3.

Lambdy tworzone *ad hoc* jako argumenty metod i typy zwracane nie cierpią z powodu niezgodności typów, co pokazałem w poniższym przykładzie:

```
List<String> filter1(List<String> values, Predicate<String> predicate) {
  // ...
}

List<String> filter2(List<String> values, LikePredicate<String> predicate) {
  // ...
}

var values = Arrays.asList("a", null, "c");

var result1 = filter1(values, value -> value != null);

var result2 = filter2(values, value -> value != null);
```

Kompilator inferuje typ lambdy *ad hoc* bezpośrednio z sygnatury metody, dzięki czemu możesz skoncentrować się na tym, *co* chcesz osiągnąć za pomocą tego wyrażenia lambda. To samo dotyczy typów zwracanych:

```
Predicate<Integer> isGreaterThan(int value) {
  return compareValue -> compareValue > value;
}
```

Skoro wiesz już, jak tworzyć wyrażenia lambda, musisz nauczyć się je wywoływać.

Wywoływanie wyrażeń lambda

Lambdy są konkretnymi implementacjami odpowiednich interfejsów funkcyjnych. Inne języki, z większą skłonnością do funkcyjności, zazwyczaj traktują lambdy bardziej dynamicznie. Dlatego wzorce użycia Javy mogą różnić się od wzorców takich języków.

W JavaScripcie można na przykład wywołać wyrażenie lambda i przekazywać argument bezpośrednio, jak pokazałem w tym kodzie:

```
let helloWorldJs = name => "witaj, " + name + "!"

let resultJs = helloWorldJs("Ben")
```

Jednak w Javie lambdy zachowują się jak wszystkie inne instancje interfejsu, dlatego trzeba bezpośrednio wywołać ich metody SAM, jak widać poniżej:

```
Function<String, String> helloWorld = name -> "witaj, " + name + "!";

var result = helloWorld.apply("Ben");
```

Wywoływanie **pojedynczej metody abstrakcyjnej** może nie być tak zwięzłe jak w innych, ale zaletą Javy jest jej ciągła kompatybilność wsteczna.

Odwoływanie się do metod

Oprócz wyrażeń lambda w Javie 8 wprowadzono kolejną nową funkcjonalność ze zmianą składni języka jako nowy sposób tworzenia lambd: **odwoływanie się do metod** (referencje do metod). Jest to lukier syntaktyczny, wykorzystujący nowy operator podwójnego dwukropka (`::`) w celu odwoływania się do istniejącej metody zamiast tworzenia na jej podstawie wyrażenia lambda. Pozwala to usprawnić kod funkcyjny.

W listingu 2.8 pokazałem, w jaki sposób poprawia się czytelność potoku strumienia dzięki przekształceniu lambd na referencje do metod. Nie przejmuj się szczegółami! Strumienie omówię w rozdziale 6. Na razie potraktuj to po prostu jako płynne wywoływanie za pomocą metod przyjmujących lambdy.

Listing 2.8. Referencje do metod i strumienie

```
List<Customer> customers = ...;

// LAMBDY
customers.stream()
         .filter(customer -> customer.isActive())
         .map(customer -> customer.getName())
         .map(name -> name.toUpperCase())
         .peek(name -> System.out.println(name))
         .toArray(count -> new String[count]);

// REFERENCJE DO METOD
customers.stream()
         .filter(Customer::isActive)
         .map(Customer::getName)
         .map(String::toUpperCase)
         .peek(System.out::println)
         .toArray(String[]::new);
```

Zastąpienie wyrażeń lambda referencjami do metod usuwa wiele „szumów" bez uszczerbku dla czytelności lub zrozumiałości kodu. Nie ma potrzeby, aby argumenty wejściowe miały rzeczywiste nazwy lub typy ani bezpośrednio wywoływały metodę referencyjną. Ponadto nowoczesne środowiska IDE zwykle zapewniają automatyczną refaktoryzację w celu konwersji wyrażeń lambda na referencje do metod, jeśli ma to zastosowanie.

Dostępne są cztery typy referencji do metody, w zależności od zastępowanego wyrażenia lambda i rodzaju metody, do której musimy się odwołać:

- odwoływanie się do metody statycznej (`static`),
- odwoływanie się do powiązanej metody niestatycznej,
- odwoływanie się do niepowiązanej metody niestatycznej,
- odwoływanie się do konstruktora.

Przyjrzymy się tym kilku rodzajom referencji; dowiesz się, jak i kiedy z nich korzystać.

Odwoływanie się do metody statycznej

Odwoływanie się do metody statycznej tworzy referencję do metody `static` określonego typu, takiej jak metoda `toHexString` dostępna w klasie `Integer`:

```
// FRAGMENT ZACZERPNIĘTY Z java.lang.Integer
public class Integer extends Number {

  public static String toHexString(int i) {
    // ...
  }
}

// LAMBDA
Function<Integer, String> asLambda = i -> Integer.toHexString(i);

// REFERENCJA DO METODY STATYCZNEJ
Function<Integer, String> asRef = Integer::toHexString;
```

Ogólna składnia dla referencji do metod statycznych jest następująca: *`NazwaKlasy::nazwaMetodyStatycznej`*.

Odwoływanie się do powiązanej metody niestatycznej

Aby odwołać się do metody niestatycznej istniejącego obiektu, konieczna jest **referencja do powiązanej metody niestatycznej**. Argumenty wyrażenia lambda są przekazywane do metody referencyjnej tego konkretnego obiektu jako argumenty metody:

```
var now = LocalDate.now();

// LAMBDA BAZUJĄCA NA ISTNIEJĄCYM OBIEKCIE
Predicate<LocalDate> isAfterNowAsLambda = date -> $.isAfter(now);

// REFERENCJA DO POWIĄZANEJ METODY NIESTATYCZNEJ
Predicate<LocalDate> isAfterNowAsRef = now::isAfter;
```

Niepotrzebna jest nawet zmienna pośrednia; wartość zwracaną wywołania kolejnej metody lub dostęp do pola można połączyć bezpośrednio za pomocą operatora podwójnego dwukropka (`::`):

```
// POWIĄZANIE WARTOŚCI ZWRACANEJ
Predicate<LocalDate> isAfterNowAsRef = LocalDate.now()::isAfter;

// POWIĄZANIE POLA STATYCZNEGO
Function<Object, String> castToStr = String.class::cast;
```

Do metod z bieżącej instancji można odwoływać się również za pomocą `this::` lub implementacji `super` przy użyciu `super::` w następujący sposób:

```
public class SuperClass {

  public String doWork(String input) {
    return "super: " + input;
  }
}

public class SubClass extends SuperClass {

  @Override
  public String doWork(String input){
    return "this: " + input;
  }

  public void superAndThis(String input) {
    Function<String, String> thisWorker = this::doWork;
    var thisResult = thisWorker.apply(input);
    System.out.println(thisResult);

    Function<String, String> superWorker = SubClass.super::doWork;
    var superResult = superWorker.apply(input);
    System.out.println(superResult);
  }
}

new SubClass().superAndThis("Witaj, świecie!");
// DANE WYJŚCIOWE:
// this: Witaj, świecie!
// super: Witaj, świecie!
```

Referencje do metody powiązanej to świetny sposób na wykorzystanie metod już istniejących w zmiennych, bieżącej instancji lub implementacji `super`. Pozwala również na refaktoryzację nietrywialnych lub bardziej złożonych wyrażeń lambda do postaci metod i użycie zamiast nich referencji do metod. Na poprawionej czytelności krótkich referencji do metod zyskują w szczególności płynne potoki, takie jak strumienie (zobacz rozdział 6.), lub typy `Optional` (zobacz rozdział 9.).

Ogólna składnia dla odwołań do powiązanych metod niestatycznych jest następująca:

`nazwaObiektu``::`*`nazwaMetodyInstancji`*

Odwoływanie się do niepowiązanej metody niestatycznej

Jak sugeruje nazwa, **odwołania do niepowiązanej metody niestatycznej** to referencje, które nie są związane z określonym obiektem. Zamiast tego odwołują się do metody instancji danego typu:

```
// FRAGMENT ZACZERPNIĘTY Z java.lang.String
public class String implements ... {

  public String toLowerCase() {
    // ...
  }
}

// LAMBDA
Function<String, String> toLowerCaseLambda = str -> str.toLowerCase();

// REFERENCJA DO NIEPOWIĄZANEJ METODY NIESTATYCZNEJ
Function<String, String> toLowerCaseRef = String::toLowerCase;
```

Ogólna składnia dla odwołań do niepowiązanych metod niestatycznych jest następująca:

```
NazwaKlasy::nazwaMetodyInstancji
```

Ten typ referencji do metody można pomylić z odwołaniem do metody statycznej. Jednak w przypadku **odwołania się do niepowiązanej metody niestatycznej** *NazwaKlasy* oznacza typ instancji, w której zdefiniowana jest referencyjna metoda instancji. Jest to również pierwszy argument wyrażenia lambda. W ten sposób metoda referencyjna jest wywoływana w instancji wejściowej, a nie w bezpośrednim odwołaniu do instancji tego typu.

Odwoływanie się do konstruktora

Ostatni typ odwoływania się do metody to referencja do konstruktora danego typu. Odwołanie do metody konstruktora wygląda następująco:

```
// LAMBDA
Function<String, Locale> newLocaleLambda = language -> new Locale(language);

// ODWOŁANIE DO KONSTRUKTORA
Function<String, Locale> newLocaleRef = Locale::new;
```

Na pierwszy rzut oka odwołania do metody konstruktora wyglądają jak referencje do metod statycznych lub niepowiązanych metod niestatycznych. Metoda referencyjna nie jest rzeczywistą metodą, ale odwołaniem do konstruktora za pomocą słowa kluczowego `new`.

Ogólna składnia odwołań do metody konstruktora to *NazwaKlasy*`::new`.

Koncepcje programowania funkcyjnego w Javie

W rozdziale 1. omówiłem podstawowe koncepcje, które sprawiają, że z teoretycznego punktu widzenia język programowania można określić jako funkcyjny. Przyjrzyjmy się im ponownie z punktu widzenia programisty Javy.

Funkcje czyste i transparentność referencyjna

Koncepcja funkcji czystych opiera się na dwóch gwarancjach, które niekoniecznie są związane z programowaniem funkcyjnym:

- Logika funkcji jest autonomiczna bez żadnego efektu ubocznego.
- Te same dane wejściowe zawsze wygenerują ten sam wynik. Dlatego powtarzające się wywołania można zastąpić przez początkowy wynik, dzięki czemu wywołanie jest referencyjnie transparentne.

Te dwie zasady mają sens nawet w kodzie imperatywnym. Przez swoją autonomiczność kod staje się przewidywalny i prostszy. Jak można osiągnąć te właściwości z perspektywy Javy?

Przede wszystkim należy dokonać sprawdzenia pod kątem niepewności. Czy istnieje logika niepredykcyjna, która nie zależy od argumentów wejściowych? Doskonałymi jej przykładami są generatory liczb losowych lub bieżąca data. Użycie takich danych w funkcji usuwa jej przewidywalność, przez co staje się ona funkcją **nieczystą**.

Następnie trzeba poszukać skutków ubocznych i stanu mutowalnego:

- Czy dana funkcja wpływa na stan poza samą funkcją, taki jak instancja lub zmienna globalna?
- Czy funkcja zmienia wewnętrzne dane swoich argumentów, np. dodając nowe elementy do kolekcji lub zmieniając właściwości obiektu?
- Czy wykonuje inne nieczyste działania, takie jak operacje wejścia-wyjścia?

Skutki uboczne nie są ograniczone wyłącznie do stanu mutowalnego. Efektem ubocznym jest na przykład proste wywołanie `System.out.println`, nawet jeśli może wyglądać nieszkodliwie. Każdy rodzaj operacji wejścia-wyjścia, jak uzyskiwanie dostępu do systemu plików lub wysyłanie żądań sieciowych, jest efektem ubocznym. Rozumowanie jest proste: w takich przypadkach powtarzające się wywołania z tymi samymi argumentami nie mogą zostać zastąpione wynikiem pierwszej ewaluacji. Dobrym wskaźnikiem metody **nieczystej** jest typ zwracany `void`. Jeżeli metoda niczego nie zwraca, generuje jedynie skutki uboczne lub w ogóle nic nie robi.

Funkcje czyste są z natury transparentne pod względem referencyjnym. W związku z tym kolejne wywołania z tymi samymi argumentami można zastąpić wyliczonym wcześniej wynikiem. Ta wymienność pozwala na zastosowanie techniki optymalizacji zwanej **memoizacją**. Nazwa wywodzi się z łacińskiego słowa *memorandum* — do zapamiętania; technika ta opisuje „zapamiętywanie" wartościowanych wcześniej wyrażeń. Umożliwia zaoszczędzanie czasu obliczeniowego kosztem przestrzeni pamięci.

Kompromis między czasem i przestrzenią

Algorytmy zależą od dwóch istotnych czynników: *przestrzeni* (pamięci) i *czasu* (obliczeniowego lub reakcji). W dzisiejszych czasach oba te zasoby mogą być dostępne w ogromnych ilościach, ale nadal są skończone.

Kompromis między czasem i przestrzenią polega na tym, że można zmniejszyć wykorzystanie jednego zasobu przez zwiększenie zużycia drugiego. Jeżeli chcesz zaoszczędzić czas, potrzebujesz więcej pamięci do przechowywania wyników. Potrzebną na bieżąco pamięć możesz też zapisywać przez ciągłe przeliczanie wyników.

Najprawdopodobniej wykorzystujesz już w swoim kodzie ogólną koncepcję przejrzystości referencyjnej w postaci **buforowania**. Bez względu na to, czy mamy do czynienia z dedykowanymi bibliotekami buforowania, takimi jak Ehcache[6], czy z prostymi tablicami wyszukiwania typu `HashMap`, zawsze chodzi o „zapamiętywanie" wartości dla określonych zestawów argumentów wejściowych.

Kompilator Java nie obsługuje automatycznej memoizacji wyrażeń lambda ani wywołań metod. Niektóre frameworki zapewniają adnotacje, takie jak `@Cacheable` frameworku Spring[7] lub `@Cached` w Apache Tapestry[8], i automatycznie generują wewnętrznie wymagany kod.

Tworzenie własnego buforowania wyrażeń lambda również nie jest zbyt trudne, dzięki użytecznym dodatkom do Javy 8 i jej nowszych wersji. Zróbmy to teraz.

Zbudowanie własnej memoizacji przez utworzenie tabeli wyszukiwania „na żądanie" wymaga udzielenia odpowiedzi na dwa pytania:

- W jaki sposób jednoznacznie identyfikujemy funkcję i jej argumenty wejściowe?
- W jaki sposób możemy zapisywać ewaluowane wyniki?

Jeśli wywołanie funkcji lub metody ma tylko jeden argument ze stałą wartością `hashCode` lub inną wartością deterministyczną, możemy utworzyć prostą tabelę wyszukiwania opartą na typie `Map`. W przypadku wywołań wieloargumentowych należy najpierw zdefiniować sposób tworzenia klucza wyszukiwania.

W Javie 8 wprowadzono wiele funkcjonalnych dodatków do typu `Map`. Jednym z tych dodatków jest metoda `computeIfAbsent`, niezwykle pomocna w łatwym implementowaniu memoizacji, jak pokazałem w listingu 2.9.

Listing 2.9. Memoizacja z wykorzystaniem Map#computeIfAbsent

```
Map<String, Object> cache = new HashMap<>(); ❶

<T> T memoize(String identifier, Supplier<T> fn) { ❷
  return (T) cache.computeIfAbsent(identifier, key -> fn.get());
}

Integer expensiveCall(String arg0, int arg1) { ❸
  // ...
}

Integer memoizedCall(String arg0, int arg1) { ❹
  var compoundKey = String.format("expensiveCall:%s-%d", arg0, arg1);
  return memoize(compoundKey, () -> expensiveCall(arg0, arg1));
}

var calculated = memoizedCall("witaj, świecie!", 42); ❺

var cached = memoizedCall("witaj, świecie!", 42); ❻
```

[6] Ehcache (*https://oreil.ly/5E0BT*) to szeroko stosowana biblioteka buforowania Javy.

[7] Wewnętrzne działanie i mechanizmy kluczy adnotacji `@Cacheable` wyjaśnia oficjalna dokumentacja (*https://oreil.ly/30P7w*).

[8] Adnotacja `@Cached` frameworku Tapestry (*https://oreil.ly/taX6J*) nie obsługuje buforowania opartego na kluczach, ale można ją powiązać z polem.

❶ Wyniki są buforowane w prostej strukturze `HashMap<String, Object>`, która pozwala buforować każdy rodzaj wywołania na podstawie identyfikatora. W zależności od wymagań konieczne może być uwzględnienie specjalnych kwestii, takich jak buforowanie wyników na żądania z aplikacji internetowej lub zastosowanie koncepcji „czasu życia pakietu" (ang. *time-to-live* — TTL). Ten przykład ma pokazać najprostszą formę tablicy wyszukiwania.

❷ Metoda `memoize` przyjmuje identyfikator oraz `Supplier<T>` na wypadek, gdyby pamięć podręczna nie miała jeszcze zapisanego wyniku.

❸ `expensiveCall` to metoda, która zostaje zapamiętana.

❹ Dla wygody istnieje wyspecjalizowana metoda zapamiętywanych wywołań, więc nie trzeba ręcznie budować identyfikatora przy każdym wywołaniu `memoize`. Ma ona te same argumenty co metoda obliczania i deleguje rzeczywisty proces memoizacji.

❺ Ta metoda pomocnicza pozwala zastąpić nazwę metody wywołania, aby zamiast wersji oryginalnej użyć tej zapamiętanej.

❻ Drugie wywołanie natychmiast zwraca zbuforowany wynik bez dodatkowego wartościowania.

Ta implementacja jest dość uproszczona i nie jest rozwiązaniem uniwersalnym. Daje jednak ogólne wyobrażenie o koncepcji przechowywania wyniku wywołania za pomocą metody pośredniej, która wykonuje rzeczywistą memoizację.

Na tym nie kończą się jednak funkcyjne dodatki do typu `Map`. Oferuje on narzędzia, które m.in. umożliwiają tworzenie powiązań „w locie" i zapewniają precyzyjniejszą kontrolę nad tym, czy określona wartość jest już obecna. Szerzej omówię te kwestie w rozdziale 11.

Niemutowalność

Klasyczne podejście do Javy w programowaniu obiektowym opiera się na mutowalnym stanie programu, najdobitniej reprezentowanym przez obiekty JavaBeans i POJO, którym przyjrzymy się w rozdziale 4. Nie ma sprecyzowanej definicji, w jaki sposób powinien być obsługiwany w OOP stan programu, a niemutowalność nie jest warunkiem wstępnym ani unikatową cechą programowania funkcyjnego. Tak czy inaczej, mutowalny stan jest cierniem w oku wielu koncepcji programowania funkcyjnego, ponieważ w celu zapewnienia integralności danych i ogólnego bezpiecznego ich używania wymagane są niemutowalne struktury danych.

POJO (ang. *Plain Old Java Object*) to tzw. zwykłe stare obiekty Javy, które nie są związane żadnymi specjalnymi ograniczeniami, innymi niż te narzucone przez język Java. JavaBeans to szczególny rodzaj obiektów POJO. Więcej informacji na temat wspomnianych obiektów znajdziesz w rozdziale 4., w podrozdziale „Mutowalność i struktury danych w OOP".

W porównaniu z innymi językami w Javie obsługa niemutowalności jest dość ograniczona. Dlatego Java musi egzekwować konstrukcje takie jak zmienne efektywnie finalne, które omówiłem wcześniej w tym rozdziale. Do obsługi „pełnej" niemutowalności trzeba zaprojektować od podstaw własne struktury danych jako niemutowalne, co może być kłopotliwe i podatne na błędy.

Aby zminimalizować wymagany kod boilerplate'owy i oprzeć się na przetestowanych w boju implementacjach, często wybiera się zewnętrzne biblioteki. W celu wypełnienia tej luki w Javie 14 wprowadzono w końcu niemutowalne klasy danych — rekordy (ang. *records*). Przyjrzymy się im w rozdziale 5.

Niemutowalność to złożony temat i omówię go szerzej w rozdziale 4., z którego dowiesz się więcej o znaczeniu i sposobie właściwego stosowania niemutowalności za pomocą wbudowanych narzędzi lub metodą „zrób to sam" (ang. *Do It Yourself* — DIY).

Typy pierwszoklasowe

Ponieważ wyrażenia lambda są konkretnymi implementacjami interfejsów funkcyjnych, stają się typami pierwszoklasowymi i są użyteczne jako zmienne, argumenty i wartości zwracane, jak pokazałem w listingu 2.10.

Listing 2.10. Pierwszoklasowe lambdy Javy

```
// PRZYPISANIE ZMIENNEJ

UnaryOperator<Integer> quadraticFn = x -> x * x; ❶
quadraticFn.apply(5); ❷
// => 25

// ARGUMENT METODY

public Integer apply(Integer input, UnaryOperator<Integer> operation) {
  return operation.apply(input); ❸
}

// WARTOŚĆ ZWRACANA

public UnaryOperator<Integer> multiplyWith(Integer multiplier) {
  return x -> multiplier * x; ❹
}

UnaryOperator<Integer> multiplyWithFive = multiplyWith(5);
multiplyWithFive.apply(6);
// => 30
```

❶ Przypisywanie wyrażenia lambda Javy do zmiennej `quadraticFn`.

❷ Może być ona używana jak każda inna „normalna" zmienna Javy przez wywołanie metody `apply` jej interfejsu.

❸ Jak każdy inny typ, wyrażenia lambda są użyteczne jako argumenty.

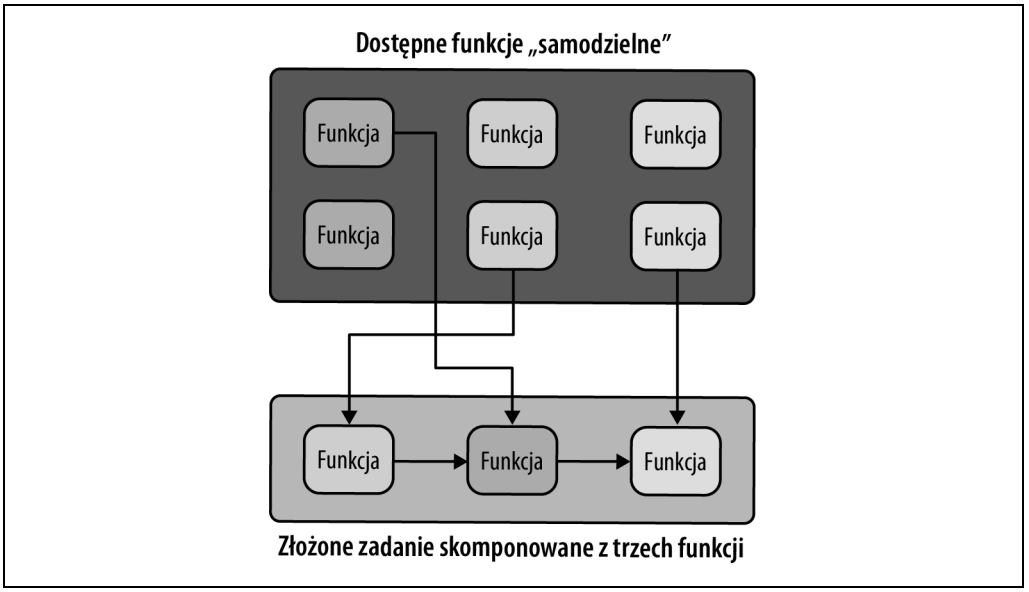
❹ Zwracanie lambdy jest jak zwracanie jakiejkolwiek innej zmiennej Javy.

Przyjmowanie lambd jako argumentów i zwracanie lambd jest niezbędne dla następnej koncepcji: kompozycji funkcyjnej.

Kompozycja funkcyjna

Idea tworzenia złożonych systemów przez komponowanie mniejszych komponentów jest kamieniem węgielnym programowania, niezależnie od stosowanego paradygmatu. W OOP obiekty mogą składać się z mniejszych obiektów, budując bardziej złożone interfejsy API. W programowaniu funkcyjnym dwie funkcje są łączone w celu zbudowania nowej funkcji, która potem może stać się elementem kolejnej kompozycji.

Kompozycja funkcyjna jest prawdopodobnie jednym z najważniejszych aspektów programowania funkcyjnego. Pozwala budować złożone systemy przez komponowanie mniejszych funkcji wielokrotnego użytku w większe łańcuchy, które wykonują bardziej złożone zadanie, jak pokazałem na rysunku 2.1.



Rysunek 2.1. Komponowanie złożonych zadań z wielu funkcji

Możliwości kompozycji funkcjonalnej Javy zależą w dużym stopniu od konkretnych typów. Łączenie różnych interfejsów funkcyjnych dostarczanych przez JDK omówię w rozdziale 3., w podrozdziale „Kompozycja funkcyjna".

Ewaluacja leniwa

Chociaż — przynajmniej co do zasady — Java jest językiem ewaluowanym nieleniwie (gorliwie), obsługuje wiele leniwych konstrukcji, do których należą:

- operatory logiczne ewaluacji minimalnej,
- operator `if-else` i operator warunkowy (trójargumentowy) `:?`,
- pętle `for` i `while`.

Prostym przykładem ewaluacji leniwej są operatory logiczne wartościowania minimalnego (ang. *short-circuit*):

```
var result1 = simple() && complex();

var result2 = simple() || complex();
```

Ewaluacja metody `complex` (złożonej) zależy od wyniku klasy `simple` (prostej) i operatora logicznego użytego w ogólnym wyrażeniu. Dlatego JVM może odrzucać wyrażenia, które nie wymagają wartościowania, co omówię szczegółowo w rozdziale 11.

Podsumowanie

- Interfejsy funkcyjne są konkretnymi typami i reprezentują wyrażenia lambda Javy.
- Składnia wyrażeń lambda Javy jest zbliżona do podstawowej notacji matematycznej rachunku lambda.
- Lambdy mogą być wyrażane na wielu poziomach szczegółowości, w zależności od kontekstu i wymagań. Krótszy kod nie zawsze jest tak ekspresyjny, jak powinien, zwłaszcza jeśli Twój kod czytają inne osoby.
- Wyrażenia lambda nie są lukrem syntaktycznym dzięki temu, że JVM wykorzystuje kod operacyjny `invokedynamic`. Pozwala to na stosowanie wielu technik optymalizacji, aby uzyskać lepszą wydajność, podobną do alternatywnych konstrukcji, takich jak klasy anonimowe.
- Aby zmienne zewnętrzne mogły być wykorzystywane w wyrażeniach lambda, muszą być efektywnie finalne (`final`), ale to sprawia, że niemutowalne są tylko referencje, a nie bazowe struktury danych.
- Odwoływanie się do metod jest zwięzłą alternatywą dla dopasowania sygnatur metod i definicji wyrażeń lambda. Zapewnia nawet prosty sposób użycia „identycznych, ale niekompatybilnych” typów interfejsów funkcyjnych.

To obowiązkowa lektura dla każdego programisty, który chce poprawić swoje umiejętności i pozostać na bieżąco z trendami w programowaniu!

A. N. M. Bazlur Rahman, inżynier oprogramowania i mistrz Javy

Aby napisać dobry kod, programiści muszą wybrać optymalny sposób rozwiązania danego problemu. Java jest znana ze skutecznego i przetestowanego podejścia obiektowego (OOP), jednak ten paradygmat nie zawsze okazuje się wystarczający. Zamiast wymuszać OOP w każdym wypadku, warto wdrożyć w swoim kodzie zasady programowania funkcyjnego (FP), aby zapewnić sobie najlepsze korzyści płynące ze stosowania obydwóch paradygmatów.

Dzięki tej książce zrozumiesz bazowe koncepcje programowania funkcyjnego i przekonasz się, że możesz włączać je do kodu bez rezygnacji z paradygmatu obiektowego. Dowiesz się również, kiedy w swojej codziennej pracy używać takich opcji jak niemutowalność i funkcje czyste i dlaczego warto to robić. Poznasz różne aspekty FP: kompozycję, ekspresyjność, modułowość, wydajność i efektywne manipulowanie danymi. Nauczysz się korzystać z FP w celu zapewnienia wyższego bezpieczeństwa i łatwiejszego utrzymywania kodu. Te wszystkie cenne umiejętności ułatwią Ci pisanie bardziej zwięzłego, rozsądnego i przyszłościowego kodu.

W książce między innymi:

- zasady programowania funkcyjnego
- przegląd typów programowania funkcyjnego dostępnych w Javie
- różne koncepcje FP i sposoby ich zastosowania
- rozszerzanie kodu Javy o aspekty FP bez przechodzenia na pełną funkcyjność
- doskonalenie własnego stylu programowania niezależnie od języka lub paradygmatu

Ben Weidig jest programistą samoukiem. Od niemal 20 lat tworzy strony internetowe i różne aplikacje w kilku językach programowania. Jest autorem licznych artykułów o Javie, programowaniu funkcyjnym i najlepszych praktykach kodowania. Bierze również udział w projektach *open source*.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0651-8
9 788328 906518
Cena: 87,00 zł