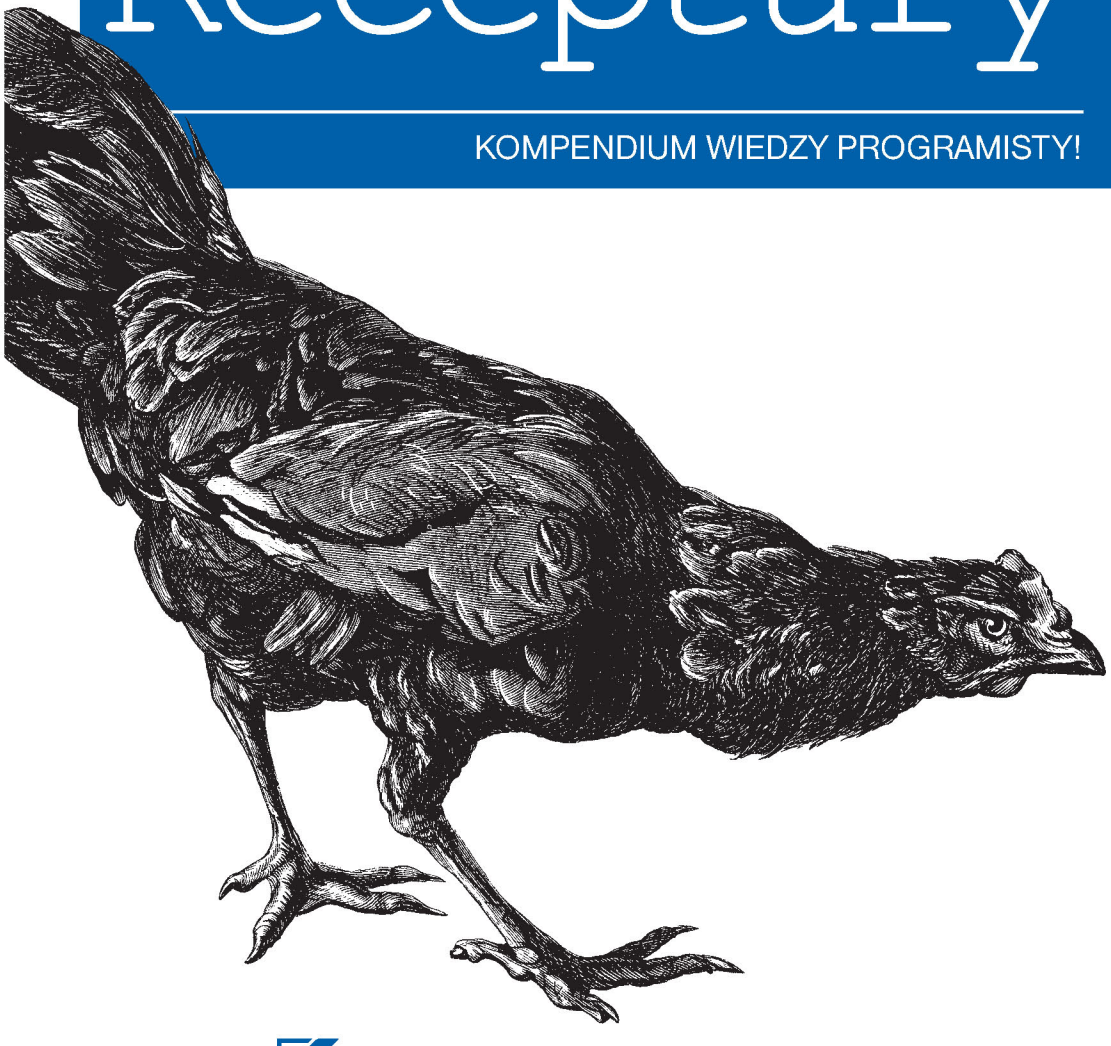
O'REILLY®

Wydanie III

Java Receptury

KOMPENDIUM WIEDZY PROGRAMISTY!



Helion

Ian F. Darwin

Tytuł oryginału: Java Cookbook, Third Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-9570-6

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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jarec3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jarec3.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 9.

Techniki programowania funkcyjnego: interfejsy funkcyjne, strumienie i kolekcje równoległe

9.0. Wprowadzenie ⑧

Java jest językiem programowania obiektowego. Doskonale o tym wiemy. Obecnie coraz większym zainteresowaniem cieszy się programowanie funkcyjne (ang. *functional programming*, FP). Być może nie ma aż tak wielu definicji programowania funkcyjnego jak języków, które umożliwiają stosowanie tego stylu programowania, choć ich liczba może być podobna. A oto co na temat programowania funkcyjnego napisano w Wikipedii:

> „(…) paradygmat programowania, styl tworzenia struktury i elementów programów komputerowych, traktujący obliczenia jako przetwarzanie funkcji matematycznych i unikający przechowywania stanu oraz danych podlegających zmianom. Programowanie funkcyjne kładzie nacisk na funkcje, których wyniki zależą wyłącznie od danych wejściowych, a nie od stanu programu, innymi słowy, na funkcje o charakterze matematycznym. Jest to paradygmat programowania deklaratywnego, co oznacza, że programowanie bazuje na wykorzystaniu wyrażeń. W kodzie funkcyjnym wyniki zwracane przez funkcję są zależne wyłącznie od przekazanych do niej argumentów, a zatem dwukrotne wywołanie funkcji *f* z tym samym argumentem *x* w obu przypadkach spowoduje zwrócenie tego samego wyniku *f*(*x*). Wyeliminowanie efektów ubocznych, takich jak zmiany stanu, które nie zależą od danych przekazanych do funkcji, znacznie ułatwia zrozumienie i określenie sposobu działania programu i stanowi jeden z kluczowych powodów rozwoju programowania funkcyjnego (…)”.
>
> — *http://en.wikipedia.org/wiki/Functional_programming*
>
> z października 2014

W jaki sposób możemy wykorzystać zasady programowania funkcyjnego? Jednym z nich mogłoby być użycie odpowiedniego funkcyjnego języka programowania, a do wiodących języków tego typu należą Haskell OCaml, Erlang oraz rodzina języków LISP. Jednak wymagałoby to odejścia z ekosystemu języka Java. Można by się ewentualnie zastanowić nad wykorzystaniem języków Scala (*http://www.scala-lang.org/*) lub Clojure (*http://clojure.org/*), które działają w oparciu o wirtualną maszynę Javy i zapewniają wsparcie dla programowania funkcyjnego w kontekście języka obiektowego.

Jednak niniejsza książka jest poświęcona Javie, zatem można sobie wyobrazić, że będziemy dążyć do skorzystania z zalet, jakie daje programowanie funkcyjne z wykorzystaniem wyłącznie tego języka. Do cech programowania funkcyjnego należą:

- Funkcje czyste (ang. *pure functions*), czyli funkcje, które nie mają żadnych efektów ubocznych i których wyniki zależą wyłącznie od przekazanych argumentów, a nie od stanu programu, który może ulegać zmianom.
- Funkcje pierwszej klasy, czyli możliwość traktowania funkcji jako danych.
- Dane niezmienne.
- Częste stosowanie rekurencji i przetwarzania leniwego.

Funkcje czyste są całkowicie niezależne; ich działanie zależy wyłącznie od przekazanych danych wejściowych oraz ich wewnętrznej logiki, a nie od zmiennego „stanu" innych części programu — w rzeczywistości w programowaniu funkcyjnym nie ma czegoś takiego jak „zmienne globalne", a jedynie „stałe globalne". Choć dla osób przyzwyczajonych do stosowania języków imperatywnych, takich jak Java, może to być sporym zaskoczeniem, to jednak takie rozwiązania mogą znacznie ułatwić testowanie programów oraz zapewnienie prawidłowości ich działania! Oznacza to bowiem, że niezależnie do tego, co się dzieje w pozostałych częściach programu (nawet w niezależnie działających wątkach), wywołanie metody, takie jak `computeValue(27)`, zawsze i bezwarunkowo zwróci tę samą wartość (wyjątkami są tu funkcje zwracające elementy stanu globalnego, np. aktualną datę i godzinę, wartość losową itd.).

W tym rozdziale terminów *funkcja* oraz *metoda* będę używał wymiennie, choć zapewne nie jest to do końca poprawne. Osoby związane z programowaniem funkcyjnym używają terminu „funkcja", mając na myśli matematyczną definicję funkcji, natomiast w języku Java „metody" są jedynie „kodem, który można wywołać" (z obiektowego punktu widzenia „wywołanie metody" w Javie określa się także jako *przesłanie sygnału* do obiektu).

„Traktowanie funkcji jako danych" oznacza, że można utworzyć obiekt będący funkcją, przekazać go do innej funkcji, napisać funkcję, która będzie zwracać inną funkcję, i tak dalej — a to wszystko bez konieczności stosowania jakiejkolwiek szczególnej składni, gdyż funkcje *są* danymi.

Jednym z rozwiązań wprowadzonych w nowej wersji języka — Java 8 — mającym na celu udostępnienie możliwości programowania funkcyjnego są „interfejsy funkcyjne". *Interfejsem funkcyjnym* w Javie nazywamy interfejs, który definiuje tylko jedną metodę. Przykładami takich interfejsów mogą być bardzo popularny interfejs `Runnable` definiujący metodę `run()` oraz powszechnie używany w bibliotece Swing interfejs `ActionListener`, który definiuje metodę `actionPerformed(ActionEvent)`. Okazuje się, że także te nowe interfejsy języka Java 8 mogą posiadać metody zadeklarowane za pomocą słowa kluczowego `default`, którego użycie w tym kontekście jest nowością. Takie domyślne metody stają się dostępne i mogą być używane w każdej klasie implementującej dany interfejs. Jeśli się nad tym zastanowimy, to stanie się jasne, że działanie takich metod nie może zależeć od stanu konkretnej klasy, gdyż nie miałyby one możliwości odwołania się do tego stanu w czasie kompilacji programu.

A zatem nieco precyzyjniej rzecz ujmując, interfejs funkcyjny jest interfejsem definiującym jedną, niedomyślną metodę. W języku Java można korzystać z funkcyjnego stylu programowania, jeśli zastosujemy interfejsy funkcyjne oraz jeśli kod umieszczany w metodach będzie korzystał wyłącznie ze sfinalizowanych zmiennych oraz pól obiektów. Jednym ze sposobów spełnienia tych wymagań jest korzystanie z metod domyślnych. Kilka pierwszych receptur tego rozdziału jest poświęconych właśnie interfejsom funkcyjnym.

Kolejnym nowym rozwiązaniem umożliwiającym stosowanie funkcyjnego stylu programowania są „wyrażenia lambda". Lambda to wyrażenie, którego typem jest interfejs funkcyjny i które może być używane jako dana (czyli można je przypisywać zmiennym lub zwracać jako wynik wywołania metody itd.). Poniżej podałem dwa krótkie przykłady wyrażeń lambda.

```
ActionListener x = (e -> System.out.println("Uaktywniono " + e.getSource());

public class RunnableLambda {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("Witam w wątku")).start();
    }
}
```

Kolejną nowością wprowadzoną w Java 8 są klasy `Stream`. Przypominają one nieco potok, w którym można coś umieścić, następnie wykonać na nim jakieś operacje, po czym przekazać jego zawartość dalej — czyli coś, co można by uznać za połączenie uniksowych potoków oraz opracowanego przez Google modelu programowania rozproszonego MapReduce (którego przykładem może być projekt Hadoop, *http://hadoop.apache.org/*), lecz działające w obrębie jednej wirtualnej maszyny Javy, czyli jednego programu. Obiekty `Stream` mogą działać szeregowo lub równolegle; przy czym te drugie zostały zaprojektowane w celu wykorzystania możliwości przetwarzania równoległego, jakie zapewniają platformy sprzętowe (zwłaszcza serwery, które bardzo często są wyposażane w procesory dysponujące dwunastoma lub szesnastoma rdzeniami). Także klasom `Stream` poświęciłem kilka receptur zamieszczonych w tym rozdziale.

Z klasami `Stream` powiązany jest interfejs `Spliterator`, stanowiący pochodną (pod względem logicznym, a nie dziedziczenia) iteratorów, lecz zaprojektowany w celu wykorzystania w przetwarzaniu równoległym. *Większość osób nie będzie musiała tworzyć własnych implementacji tego interfejsu, a nawet nie będzie musiała zbyt często jawnie wywoływać jego metod*, dlatego też nie poświęcę mu w tej książce wiele uwagi.

Patrz także

Ogólne informacje na temat programowania funkcyjnego można znaleźć w książce *Functional Thinking* (*http://shop.oreilly.com/product/0636920029687.do*).

Dostępna jest także książka Richarda Warburtona *Java 8 Lambdas* (*http://shop.oreilly.com/product/0636920030713.do*), która została w całości poświęcona wyrażeniom lambda oraz związanym z nimi narzędziom.

9.1. Stosowanie wyrażeń lambda lub domknięć zamiast klas wewnętrznych

Problem ⑧

Chcemy uniknąć pisania rozbudowanego kodu, którego wymaga stosowanie klas wewnętrznych.

Rozwiązanie

Należy skorzystać z wyrażeń lambda.

Analiza

Symbol lambda (λ) to jedenasta litera alfabetu greckiego, a zatem jest on tak stary jak cała cywilizacja zachodnioeuropejska. Rachunek lambda (*http://pl.wikipedia.org/wiki/Rachunek_lambda*) jest równie stary jak sama informatyka. W tym kontekście wyrażenia lambda są *niewielkimi fragmentami obliczeń, do których można się odwoływać*. Są one funkcjami, które można traktować jako dane. W tym sensie są one bardzo podobne do anonimowych funkcji wewnętrznych, choć chyba lepszym rozwiązaniem byłoby wyobrażenie ich sobie jako *anonimowych metod*. Są one zasadniczo stosowane jako zamienniki klas wewnętrznych w kodzie wykorzystującym *interfejsy funkcyjne* — czyli interfejsy definiujące tylko jedną metodę („funkcję"). Doskonałym przykładem takiego interfejsu funkcyjnego jest `ActionListener`, powszechnie stosowany w kodzie obsługi interfejsu użytkownika. Interfejs ten definiuje tylko jedną metodę:

```
public void actionPerformed(ActionEvent);
```

Przykłady wykorzystania wyrażeń lambda w obsłudze interfejsu użytkownika można znaleźć w rozdziale 14. Poniżej, aby rozbudzić zainteresowanie Czytelnika, zamieściłem jeden z nich:

```
quitButton.addActionListener(e -> System.exit(0));
```

Jednak obecnie już nie wszyscy tworzą aplikacje z graficznym interfejsem użytkownika, dlatego zacznę od przykładu, który nie jest z nimi w żaden sposób związany. Załóżmy, że dysponujemy zbiorem obiektów deskryptorów aparatów fotograficznych, które *zostały już wczytane z bazy danych i zapisane w pamięci*, a teraz chcemy napisać interfejs programistyczny ogólnego przeznaczenia pozwalający na ich przeszukiwanie, którego moglibyśmy używać w pozostałych miejscach naszej aplikacji.

Pierwszym pomysłem mogłoby być stworzenie następującego interfejsu:

```
public interface CameraInfo {
    public List<Camera> findByMake();
    public List<Camera> findByModel();
    ...
}
```

Jednak być może Czytelnik już zauważył problem wiążący się z takim rozwiązaniem. Otóż wraz ze zwiększaniem się stopnia złożoności naszej aplikacji konieczne byłoby także zaimplementowanie metod `findByPrice()`, `findByMakeAndModel()`, `findByYearIntroduced()` i tak dalej.

Można by sobie wyobrazić metodę „znajdź na podstawie przykładu", do której byłby przekazywany obiekt `Camera`, a metoda odnajdywałaby inne obiekty, używając do porównania wszystkich pól argumentu o wartościach różnych od `null`. Ale w jaki sposób należałoby zaimplementować wyszukiwanie wszystkich aparatów z wymiennymi obiektywami o cenie poniżej 1500 złotych?[1]

[1] Gdybyśmy kiedyś musieli wykonywać tego typu operacje na informacjach przechowywanych w bazie danych, wykorzystując przy tym Java Persistence API (patrz receptura 18.1), to warto się zainteresować projektem Apache DeltaSpike (*http://deltaspike.apache.org/*), który pozwala na definiowanie interfejsów zawierających metody o nazwach takich jak `findCameraByInterchangeableTrueAndPriceLessThen(double price)` i *jest w stanie sam je zaimplementować*. W serwisie GitHub dostępne są wzorce projektów korzystających z CDI oraz DeltaSpike: Java SE (*https://github.com/os890/javase-cdi-ds-project-template*) oraz Java Web (*https://github.com/os890/javaweb-cdi-ds-project-template*).

A zatem można sądzić, że lepszym sposobem na wykonanie takiego porównania byłoby zastosowanie „funkcji zwrotnej". Pozwoliłoby ono na stworzenie anonimowej klasy wewnętrznej, która odpowiadałaby za wykonanie odpowiednich poszukiwań. Chcielibyśmy, żeby funkcja zwrotna mogła wyglądać w następujący sposób:

```
public boolean choose(Camera c) {
    return c.isIlc() && c.getPrice() < 500;
}
```

W tym celu musielibyśmy stworzyć interfejs o poniższej postaci[2]:

/functional/CameraAcceptor.java

```
/** Interfejs wybiera (akceptuje) niektóre elementy kolekcji. */
public interface CameraAcceptor {
    boolean choose(Camera c);
}
```

Teraz aplikacja wyszukująca aparaty mogłaby udostępnić metodę:

```
public List<Camera> search(CameraAcceptor acc);
```

którą można by wywołać, używając następującego fragmentu kodu (zakładając, że potrafimy posługiwać się anonimowymi klasami wewnętrznymi):

```
results = searchApp.search(new CameraAcceptor() {
    public boolean choose(Camera c) {
        return c.isIlc() && c.getPrice() < 500;
    }
}
```

Gdyby ktoś nie potrafił posługiwać się anonimowymi klasami wewnętrznymi, musiałby użyć następującego rozwiązania:

```
class MyIlcPriceAcceptor implements CameraAcceptor {
    public boolean choose(Camera c) {
        return c.isIlc() && c.getPrice() < 500;
    }
}

CameraAcceptor myIlcPriceAcceptor = nwq MyIlcPriceAcceptor();
results = searchApp.search(myIlcPriceAcceptor);
```

To naprawdę masa kodu do napisania — i to tylko po to, by przekazać jedną metodę do mechanizmu wyszukującego. O wyposażenie języka Java w wyrażenia lambda bądź w domknięcia (ang. *closure*) postulowano na wiele (i to dosłownie) lat, zanim eksperci doszli do porozumienia, jak należy to zrobić. A efekt jest niewiarygodnie prosty. Jednym ze sposobów wyobrażenia sobie wyrażeń lambda w Javie jest potraktowanie ich jako *metod implementujących interfejs funkcyjny*. Z wykorzystaniem wyrażeń lambda powyższy kod można zapisać w następującej postaci:

```
results = searchApp.search(c -> c.isIlc() && c.getPrice() < 500);
```

[2] Czytelnikom, którzy nie interesują się aparatami fotograficznymi, wyjaśniam, że określenie „aparat z wymiennym obiektywem" obejmuje dwie kategorie aparatów, które obecnie, w 2014 roku, można znaleźć w sklepach: tradycyjne lustrzanki cyfrowe (ang. *Digital Single Lens Reflection*, DSLR) oraz nową kategorię „aparatów kompaktowych", takich jak Nikon 1, Sony ILCE (znany wcześniej pod nazwą NEX) oraz Canon EOS-M, które są mniejsze i lżejsze od starszych aparatów DSLR.

Zapis ze strzałką (->) oznacza kod, który należy wykonać. Jeśli jest to proste wyrażenie, takie jak w powyższym przykładzie, to można je zapisać w przedstawionej postaci. Jeśli jednak w kodzie występuje instrukcja warunkowa lub inne instrukcje, to podobnie jak w zwyczajnym kodzie pisanym w Javie, trzeba będzie zastosować blok kodu:

```
results = searchApp.search(c -> {
    if (c.isIlc() && c.getPrice() < 500)
        return true;
    else
        return false;
});
```

Pierwsze `c` umieszczone w nawiasach odpowiada parametrowi `Camera c` jawnie zaimplementowanej metody `choose()`: typ można pominąć, ponieważ kompilator go zna! Jeśli metoda ma więcej niż jeden argument, to należy je zapisać w nawiasach. Załóżmy, że dysponujemy metodą porównującą, która wymaga przekazania dwóch obiektów `Camera` i zwraca wartość liczbową (życzę powodzenia komuś, kto spróbowałby doprowadzić do porozumienia dwóch fotografików odnośnie do sposobu działania *takiego* algorytmu!):

```
double goodness = searchApp.compare((c1, c2) -> {
    // tu byłby zapisany nasz magiczny kod
});
```

Można sądzić, że taki sposób zapisu wyrażeń lambda zapewnia ogromne możliwości — i faktycznie tak jest! Będzie można znaleźć bardzo wiele przykładów takich rozwiązań, jak tylko Java 8 zyska popularność.

Do tej pory dla każdej metody, która miała być stosowana w formie wyrażeń lambda, konieczne było napisanie odpowiedniego interfejsu. W następnej recepturze przedstawię kilka predefiniowanych interfejsów, których można użyć, by jeszcze bardziej uprościć (czyli także skrócić) swój kod.

No i koniecznie należy pamiętać, że dostępnych już jest całkiem dużo takich „funkcyjnych" interfejsów, jak na przykład interfejs `ActionListener` stosowany w aplikacjach z graficznym interfejsem użytkownika. Co ciekawe, zintegrowane środowisko programistyczne IntelliJ (patrz receptura 1.3) jest w stanie automatycznie rozpoznawać definicje klas wewnętrznych, które można by zastąpić wyrażeniami lambda, i w przypadku korzystania z opcji „zwijania kodu" (ang. *code folding*, możliwości pozwalającej na reprezentację definicji całej metody w jednym wierszu kodu) zastępuje taką klasę wewnętrzną wyrażeniem lambda! Rysunki 9.1 oraz 9.2 pokazują kod w jego początkowej postaci oraz po zwinięciu.

9.2. Stosowanie predefiniowanych interfejsów lambda zamiast własnych

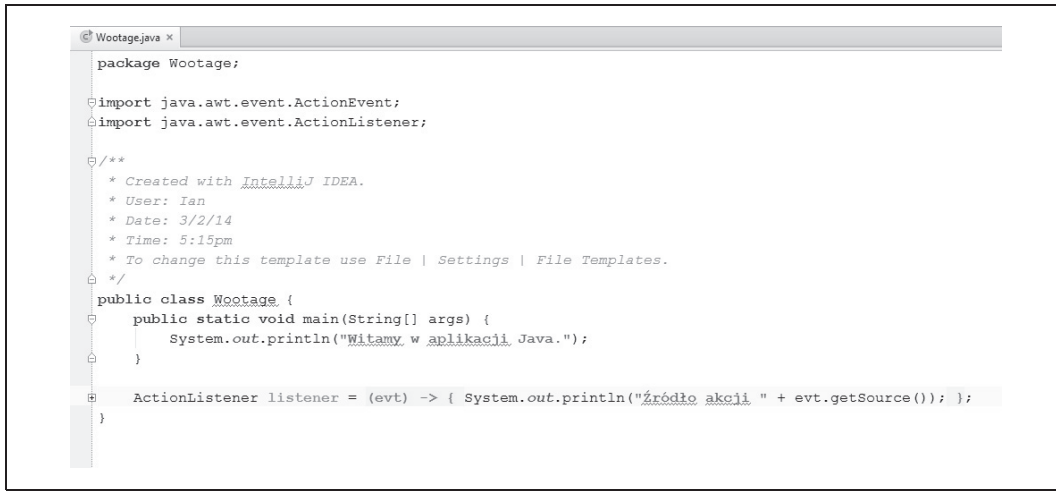
Problem 8

Chcemy stosować wyrażenia lambda, używając przy tym nie własnych, lecz predefiniowanych interfejsów.

Rozwiązanie

Należy skorzystać z interfejsów funkcyjnych języka Java 8 zdefiniowanych w interfejsie `java.util.function`.

Rysunek 9.1. Kod w środowisku IntelliJ w pełnej postaci



Rysunek 9.2. Ten sam kod po zwinięciu

Analiza

W recepturze 9.1 stosowaliśmy metodę `acceptCamera()` zdefiniowaną w interfejsie `CameraAcceptor`. Metody o podobnym charakterze i działaniu są stosowane dosyć często, dlatego też pakiet `java.util.function` zawiera interfejs `Predicate<T>`, którego możemy użyć zamiast interfejsu `CameraAcceptor`. Interfejs ten definiuje tylko jedną metodę — `bool test(T t)`:

```
interface Predicate<T> {
    boolean test(T t);
}
```

Pakiet ten zawiera około 50 najczęściej stosowanych interfejsów funkcyjnych, takich jak `IntUnaryOperator`, który pobiera jeden argument typu `int` i zwraca wartość tego samego typu, lub `LongPredicate`, który pobiera wartość typu `long` i zwraca wynik typu `boolean` — i tak dalej.

Jak zawsze w przypadku stosowania typów ogólnych, aby skorzystać z interfejsu `Predicate`, należy użyć `Camera` (oczywiście w naszym przypadku) jako parametru typu, co da nam typ `Predicate<Camera>` przedstawiony w poniższym przykładzie (choć nie musimy go jawnie umieszczać w kodzie):

```
interface Predicate<Camera> {
    boolean test(Camera c);
}
```

A zatem naszą aplikację wyszukującą możemy aktualnie zmienić tak, by udostępniała następującą metodę:

```
public List<Camera> search(Predicate p);
```

Na szczęście z *punktu widzenia metod anonimowych implementowanych przez wyrażenia lambda ma ona taką samą sygnaturę* co nasz interfejs `CameraAcceptor`, dzięki czemu pozostałe elementy naszego kodu ulegają zmianie! A zatem poniższa instrukcja wciąż stanowi poprawne wywołanie metody `search()`:

```
results = searchApp.search(c -> c.isIlc() && c.getPrice() < 500);
```

Poniżej została przedstawiona implementacja metody `search()`:

/functional/CameraSearchPredicate.java

```
    public List<Camera> search(Predicate<Camera> tester) {
        List<Camera> results = new ArrayList<>();
        privateListOfCameras.forEach(c -> {
            if (tester.test(c))
                results.add(c);
        });
        return results;
    }
```

Załóżmy, że na każdym elemencie listy musi zostać wykonana tylko jedna operacja, a następnie cała lista zostanie usunięta. Po chwili zastanowienia dojdziemy do wniosku, że takiej listy wcale nie trzeba zwracać, a jedynie określić poszczególne elementy spełniające zadane warunki.

9.3. Upraszczanie przetwarzania z wykorzystaniem interfejsu Stream

Problem ⑧

Chcemy przetworzyć dane z wykorzystaniem mechanizmu przypominającego potoki.

Rozwiązanie

Należy użyć interfejsu `Stream` oraz jego metod.

Analiza

Strumienie (ang. *Streams*) są nowym mechanizmem wprowadzonym w języku Java 8, pozwalającym kolekcjom na przesyłanie swojej zawartości kolejno, element po elemencie, przez mechanizm przypominający potoki, gdzie mogą być przetwarzane, i to w sposób (w różnym stopniu) równoległy. Można wyróżnić trzy grupy metod związanych z wykorzystaniem strumieni:

- metody wytwarzające strumienie (patrz receptura 7.3);
- metody przekazujące, które operują na strumieniu i zwracają odwołanie do niego, pozwalając na tworzenie sekwencji wywołań; należą do nich takie metody jak: `distinct()`, `filter()`, `limit()`, `map()`, `peek()`, `sorted()`, `unsorted()` itd.;
- metody kończące działanie strumieni, które stanowią zakończenie wykonywanych na nich operacji; należą do nich takie metody jak: `count()`, `findFirst()`, `max()`, `min()`, `reduce()` czy też `sum()`.

Listing 9.1 przedstawia listę obiektów `Hero` reprezentujących superbohaterów w różnym wieku. Użyjemy metod interfejsu `Stream`, by wybrać tylko tych spośród nich, którzy są dorośli, i zsumować ich wiek, a następnie w podobny sposób posortujemy ich imiona alfabetycznie.

W obu przypadkach rozpoczniemy od użycia generatora strumienia (metody `Arrays.stream()`), wykonamy na strumieniu kilka operacji, w tym operację odwzorowania (nie należy jej mylić z klasą `java.util.Map`!), która powoduje przesłanie do potoku innej wartości, a na samym końcu wywołamy operację kończącą. Operacje odwzorowywania oraz filtrowania niemal zawsze są kontrolowane przez wyrażenia lambda (stosowanie klas wewnętrznych w przypadku korzystania z tego stylu programowania byłoby zbyt męczące!).

Listing 9.1. /functional/SimpleStreamDemo.java

```
static Hero[] heroes = {
    new Hero("Grelber", 21),
    new Hero("Roderick", 12),
    new Hero("Franciszek", 35),
    new Hero("Superman", 65),
    new Hero("Jumbletron", 22),
    new Hero("Maverick", 1),
    new Hero("Palladyn", 50),
    new Hero("Atena", 50) };

public static void main(String[] args) {

    long adultYearsExperience = Arrays.stream(heroes)
            .filter(b -> b.age >= 18)
            .mapToInt(b -> b.age).sum();
    System.out.println("Jesteśmy w dobrych rękach! Dorośli " +
            "superbohaterowie mają w sumie " + adultYearsExperience +
            " lata doświadczeń.");

    List<Object> sorted = Arrays.stream(heroes)
            .sorted((h1, h2) -> h1.name.compareTo(h2.name))
            .map(h -> h.name)
            .collect(Collectors.toList());
    System.out.println("Superbohaterowie posortowani według imion: " +
            sorted);
}
```

Spróbujmy teraz wykonać powyższy program, aby przekonać się, czy działa prawidłowo:

```
Jesteśmy w dobrych rękach! Dorośli superbohaterowie mają w sumie 243 lata doświadczeń.
Superbohaterowie posortowani według imion: [Atena, Franciszek, Grelber, Jumbletron, Maverick,
Palladyn, Roderick, Superman]
```

Kompletną listę dostępnych operacji można znaleźć w dokumentacji interfejsu `java.util.stream.Stream`.

9.4. Poprawianie przepustowości dzięki wykorzystaniu strumieni i kolekcji równoległych

Problem ⑧

Chcemy połączyć możliwości interfejsu `Stream` z przetwarzaniem współbieżnym, a przy tym wciąż móc korzystać z interfejsu programistycznego do obsługi kolekcji, który nie jest bezpieczny pod względem wielowątkowym.

Rozwiązanie

Należy użyć strumieni równoległych.

Analiza

Standardowe typy kolekcji, takie jak większość implementacji interfejsów `List`, `Set` oraz `Map`, nie zapewniają możliwości bezpiecznej wielowątkowej aktualizacji zawartości; jeśli w jednym wątku spróbujemy dodać jakiś obiekt do kolekcji lub go z niej usunąć, a jednocześnie inny wątek będzie się odwoływał do obiektów przechowywanych w tej samej kolekcji, to może to doprowadzić do awarii programu. Natomiast nic nie stoi na przeszkodzie, by w większej liczbie wątków jednocześnie odczytywać zawartość tej samej kolekcji. Zagadnienia związane z wielowątkowością opisałem w rozdziale 22.

Szkielet kolekcji udostępnia „klasy synchronizowane", które zapewniają możliwość automatycznej synchronizacji wątków zyskiwaną kosztem wprowadzenia rywalizacji pomiędzy wątkami ograniczającej możliwości działania współbieżnego. Aby zapewnić możliwość efektywnego wykonywania operacji, należy skorzystać ze *strumieni współbieżnych*, które pozwalają na bezpieczne stosowanie standardowych typów kolekcji, o ile tylko *podczas przetwarzania kolekcji ich zawartość nie jest modyfikowana*.

Aby użyć takiego równoległego strumienia, wystarczy o niego poprosić. W tym celu zamiast metody `stream()`, z której skorzystaliśmy w recepturze 9.3, należy wywołać metodę `parallelStream()`.

W ramach przykładu załóżmy, że nasz interes z aparatami cyfrowymi świetnie się rozwija i musimy *naprawdę szybko* wyszukiwać aparaty na podstawie typu i zakresu cen (przy okazji używając krótszego i prostszego kodu niż wcześniej):

/functional/CameraSearchParallelStream.java

```
public static void main(String[] args) {
    for (Object camera : privateListOfCameras.parallelStream().  ❶
            filter(c -> c.isIlc() && c.getPrice() < 500).        ❷
            toArray()) {                                         ❸
        System.out.println(camera);                              ❹
    }
}
```

❶ Tworzymy strumień równoległy na podstawie listy (`List`) obiektów `Camera`. Zawartość kolekcji zwróconej przez strumień zostanie następnie pobrana w pętli „foreach".

❷ Filtrujemy aparaty na podstawie ceny, używając przy tym tego samego wyrażenia lambda typu `Predicate`, który stosowaliśmy już w recepturze 9.1.

❸ Kończymy działanie strumienia, konwertując go na tablicę.

❹ Wewnątrz pętli „foreach" wyświetlamy kolejno poszczególne aparaty zwrócone przez strumień.

Powyższy kod będzie działał niezawodnie wyłącznie w przypadku, jeśli żaden wątek nie spróbuje zmodyfikować zawartości danych w trakcie ich przeszukiwania. Informacje o tym, jak to zapewnić, korzystając z mechanizmu blokowania wątków, można znaleźć w rozdziale 22.

9.5. Tworzenie własnych interfejsów funkcyjnych

Problem ⑧

Chcemy napisać interfejs funkcyjny, którego moglibyśmy używać do tworzenia wyrażeń lambda.

Rozwiązanie

Należy stworzyć interfejs definiujący jedną metodę abstrakcyjną i opcjonalnie dodać do niego adnotację `@FunctionalInterface`.

Analiza

Jak już wspominałem wcześniej, interfejs funkcyjny to interfejs deklarujący jedną metodę abstrakcyjną. Do niektórych powszechnie znanych interfejsów funkcyjnych należą: `java.lang.Runnable`, `java.util.Observer` oraz `java.awt.event.ActionListener`. Z kolei przykładami interfejsów „niefunkcyjnych" są: `java.util.Observable` oraz `java.awt.event.WindowListener`, gdyż każdy z nich definiuje więcej niż jedną metodę.

Tworzenie własnych interfejsów funkcyjnych nie jest trudne. Zanim jednak się za to zabierzemy, należy pamiętać, że bardzo wiele takich interfejsów istnieje w JDK! Zgodnie z informacjami podanymi w recepturze 9.2 warto sprawdzić dokumentację interfejsu `java.util.function`, który udostępnia wiele predefiniowanych interfejsów funkcyjnych ogólnego przeznaczenia, w tym zastosowany już wcześniej interfejs `Predicate`.

My jednak mimo wszystko chcemy zdefiniować własny interfejs funkcyjny. Poniżej przedstawiłem prosty przykład takiego interfejsu:

/functional/ProcessIntsUsingFunctional.java

```
interface MyFunctionalInterface {
    int compute(int x);
}
```

Można by go użyć w poniższym programie do przetworzenia tablicy liczb całkowitych:

/functional/ProcessIntsUsingFunctional.java

```
    static int[] integers = {1, 2, 3};

    public static void main(String[] args) {
        int total = 0;
        for (int i : integers)
            total += process(i, x ->  x * x + 1);
        System.out.println("Suma wynosi " + total);
    }

    private static int process(int i, MyFunctionalInterface o) {
        return o.compute(i);
    }
```

Gdyby interfejs zawierający metodę `compute()` nie był interfejsem funkcyjnym — gdyby deklarował więcej niż jedną metodę — to nie można by go użyć w taki sposób.

Aby mieć możliwość zagwarantowania, że dany interfejs jest interfejsem funkcyjnym i takim pozostanie, została stworzona adnotacja `@FunctionalInterface`, która jest stosowana tak samo jak adnotacja `@Override` (obie zostały zdefiniowane w pakiecie `java.lang`). Jest to adnotacja opcjonalna, która jest używana w celu zapewnienia, że dany interfejs będzie spełniał wymogi narzucane interfejsom funkcyjnym. Można by ją dodać do naszego interfejsu z poprzedniego przykładu w następujący sposób:

```
@FunctionalInterface
interface MyFunctionalInterface {
    int compute(int x);
}
```

Gdyby później ktoś pracujący nad kodem dodał do niego kolejną metodę, na przykład:

```
int recompute(int x);
```

to interfejs przestałby spełniać warunki interfejsu funkcyjnego, jednak kompilator lub zintegrowane środowisko programistyczne wykryłoby to bezpośrednio po zapisaniu pliku lub jego kompilacji, pozwalając programiście zaoszczędzić czas konieczny na określenie, dlaczego wyrażenia lambda przestały działać. Poniżej przedstawiłem komunikat, który w takim przypadku wygenerowałby kompilator `javac`:

```
C:\javasrc>javac -d build src/lang/MyFunctionalInterface.java
src\lang\MyFunctionalInterface.java:3: error: Unexpected @FunctionalInterface
annotation
@FunctionalInterface
^
  MyFunctionalInterface is not a functional interface
    multiple non-overriding abstract methods found in
    interface MyFunctionalInterface

1 error

C:\javasrc>
```

Oczywiście, czasem może się zdarzyć, że nasz interfejs naprawdę będzie musiał mieć więcej niż jedną metodę. W takich przypadkach złudzenie (lub efekt) funkcyjności interfejsu można zachować poprzez wskazanie metody „domyślnej" — poprzedzenie jej słowem kluczowym `default`. Drugiej metody takiego interfejsu wciąż będzie można używać w wyrażeniach lambda.

```
public interface ThisIsStillFunctional {
  default int compute(int ix) { return ix * ix + 1 };
  int anotherMethod(int y);
}
```

W interfejsach funkcyjnych tylko domyślne metody mogą zawierać instrukcje, a w każdym takim interfejsie może istnieć jedna metoda, w której definicji nie ma słowa kluczowego `default`.

Poza tym przedstawiony wcześniej interfejs `MyFunctionalInterface` można z powodzeniem zastąpić domyślnym interfejsem `java.util.IntUnaryOperator`, zmieniając także nazwę metody z `compute()` na `applyAsInt()`. W przykładach dołączonych do książki w katalogu */functional* dostępna jest wersja programu korzystająca z tego interfejsu — `ProcessIntsIntUnaryOperator`.

Patrz także

Domyślnych metod definiowanych w interfejsach można także używać do wstawiania kodu do innych klas, co opisałem w recepturze 9.7.

9.6. Używanie istniejącego kodu w sposób funkcyjny dzięki wykorzystaniu odwołań do metod

Problem 

Dysponujemy już istniejącym kodem spełniającym warunki interfejsów funkcyjnych i chcielibyśmy używać go bez konieczności dopasowywania nazw metod do tych zdefiniowanych w interfejsie.

Rozwiązanie

Należy zastosować odwołania do funkcji, takie jak `MyClass::myFunc` lub `someObj::someFunc`.

Analiza

Słowo „odwołanie" w języku Java ma równie wiele znaczeń co słowo „sesja". Zastanówmy się:

- Ze zwyczajnych obiektów korzystamy zazwyczaj, używając odwołań.
- Typy referencyjne, takie jak `WeakReference`, mają ściśle określone znaczenie dla mechanizmu odzyskiwania pamięci.
- W języku Java 8 pojawiła się zupełnie nowa możliwość odwoływania się do konkretnych metod.
- Można się nawet odwołać do „metody instancyjnej dowolnego obiektu określonego typu".

Nowa składnia pozwalająca na tworzenie takich odwołań składa się z nazwy obiektu lub klasy, dwóch znaków dwukropka oraz nazwy metody, do której chcemy się odwołać w kontekście obiektu lub klasy (zgodnie ze standardowymi regułami języka Java, stosując nazwę klasy, można się odwoływać do jej metod statycznych, natomiast stosując nazwę zmiennej obiektowej — do jej metod instancyjnych). Aby odwołać się do konstruktora, należy użyć słowa kluczowego `new`, na przykład `MyClass::new`. Takie odwołanie tworzy wyrażenie lambda, które można wywołać, zapisać w zmiennej, której typem będzie interfejs funkcyjny, i tak dalej.

W przykładzie przedstawionym na listingu 9.2 tworzymy odwołanie typu `Runnable`, które zamiast standardowej metody `run` zawiera metodę `walk` mającą ten sam typ wartości wynikowej i argumentów. Warto zwrócić uwagę na zastosowanie `this` jako określenia obiektu podczas tworzenia odwołania. Następnie obiekt `Runnable` przekazujemy w wywołaniu konstruktora `Thread` i uruchamiamy wątek — w efekcie zamiast metody `run` zostanie wywołana metoda `walk`.

Listing 9.2. /functional/ReferencesDemo.java

```
/** "Chodź, nie biegaj" */
public class ReferencesDemo {

    // Zakładamy, że to jest istniejąca metoda, której nazwy nie
    // chcemy zmieniać.
    public void walk() {
        System.out.println("ReferencesDemo.walk(): zastępuje wywołanie metody
                           run.");
    }

    // To jest nasza główna metoda, która wykonuje metodę walk w nowym
    // wątku.
    public void doIt() {
        Runnable r = this::walk;
        new Thread(r).start();
    }

    // Zwyczajna, bardzo prosta metoda main, która wszystko uruchomi.
    public static void main(String[] args) {
        new ReferencesDemo().doIt();
    }
}
```

Oto wyniki wykonania tego programu:

```
ReferencesDemo.walk(): zastępuje wywołanie metody run.
```

Przykład przedstawiony na listingu 9.3 tworzy obiekt `AutoCloseable` przeznaczony do użycia w instrukcji `try` zarządzającej zasobami (patrz punkt „Instrukcja try zarządzająca zasobami" w podrozdziale „Nowości wprowadzone w wersji Java 7" dodatku A). Interfejs `AutoCloseable` zawiera metodę `close()`, jednak w naszym przykładzie metoda ta ma nazwę `cloz()`. Używana w programie zmienna referencyjna typu `AutoCloseable` ma nazwę `autoCloseable` i jest tworzona wewnątrz instrukcji `try`, co oznacza, że jej metoda udająca metodę `close()` zostanie wywołana po zakończeniu realizacji bloku `try`. W tym przypadku znajdujemy się w statycznej metodzie `main()` i dysponujemy zmienną referencyjną `rnd2` zawierającą odwołanie do obiektu naszej klasy, zastosujemy zatem tę zmienną do utworzenia odwołania do metody zgodnej z interfejsem `AutoCloseable`.

Listing 9.3. /functional/ReferencesDemo2.java

```
public class ReferencesDemo2 {
    void cloz() {
        System.out.println("Zamiast wywołania metody close().");
    }

    public static void main(String[] args) throws Exception {
        ReferencesDemo2 rd2 = new ReferencesDemo2();

        // Używamy odwołania do metody w celu przypisania do zmiennej
        // typu AutoCloseable "autoCloseable" odwołania do metody
        // o zgodnej sygnaturze "c" (oczywiście chodzi o metodę close,
        // lecz chcę pokazać, że nazwa metody nie ma w tym
        // przypadku znaczenia).
        try (AutoCloseable autoCloseable = rd2::cloz) {
            System.out.println("Wykonujemy jakieś działania.");
        }
    }
}
```

Oto wyniki wykonania tego programu:

```
Wykonujemy jakieś działania.
Zamiast wywołania metody close().
```

Oczywiście istnieje także możliwość stosowania takich rozwiązań, które wykorzystują własne interfejsy funkcyjne, takie jak ten przedstawiony w recepturze 9.5. Czytelnik na pewno też jest świadomy, a przynajmniej domyśla się, że dowolne odwołanie do obiektu w języku Java można przekazać w wywołaniu metody `System.out.println()`, a w efekcie zostanie wyświetlony jakiś opis obiektu. Oba te zagadnienia zostały przedstawione w przykładzie z listingu 9.4. Zdefiniowaliśmy w nim interfejs funkcyjny o nazwie `FunInterface`, którego metoda wymaga przekazania kilku argumentów (w zasadzie tylko po to, by nie można go pomylić z już istniejącymi interfejsami funkcyjnymi). Metoda nosi nazwę `process`, jednak — jak już wiemy — nazwa ta nie ma większego znaczenia — jej implementacja w programie nosi nazwę `work`. Jest to metoda statyczna, przez co nie możemy stwierdzić, że nasza klasa `ReferecesDemo3` implementuje interfejs `FunInterface` (mimo że nazwy metod są takie same — metoda statyczna nie może bowiem przesłonić odziedziczonej metody instancyjnej). Okazuje się jednak, że możemy utworzyć odwołanie lambda do metody `work`. Następnie przekazujemy to odwołanie w wywołaniu metody `println()`, pokazując tym samym, że jego struktura odpowiada obiektowi języka Java.

Listing 9.4. /functional/ReferencesDemo3.java

```
public class ReferencesDemo3 {

    interface FunInterface {
        void process(int i, String j, char c, double d);
    }

    public static void work(int i, String j, char c, double d){
        System.out.println("Muuu");
    }

    public static void main(String[] args) {
        FunInterface sample = ReferencesDemo3::work;
        System.out.println("Główna metoda obliczeniowa: " + sample);
    }
}
```

Poniżej przedstawiłem wyniki generowane przez ten program:

```
Główna metoda obliczeniowa: functional.ReferencesDemo3$$Lambda$1/918221580@4517d9a3
```

Fragment `Lambda$1` w wyświetlonej nazwie odpowiada identyfikatorom `$1` stosowanym w anonimowych klasach wewnętrznych.

Ostatni rodzaj odwołań do funkcji opisany w dokumentacji Javy, określany jako odwołanie do „metody instancyjnej dowolnego obiektu konkretnego typu", jest chyba najbardziej zawiłą nowością wprowadzoną w języku Java 8. Pozwala ona na zadeklarowanie odwołania do metody instancyjnej, jednak bez określania, o który obiekt chodzi. Oznacza to, że można jej używać z dowolnym obiektem danej klasy! W przykładzie przedstawionym na listingu 9.5 mamy tablicę łańcuchów znaków, którą chcemy posortować. Ponieważ nazwiska podane w tablicy mogą się zaczynać zarówno od małych, jak i od wielkich liter, chcemy je posortować, używając metody `compareToIgnoreCase()` klasy `String`, która nie uwzględnia wielkości liter.

Ponieważ chciałbym pokazać kilka różnych sposobów sortowania, utworzyłem dwa odwołania do tablicy: pierwsze — do oryginalnej, nieposortowanej tablicy, a drugie — do kopii roboczej, którą będziemy odtwarzać, sortować i wyświetlać, używając metody pomocniczej (nie przedstawiałem tu jej kodu, gdyż jest to zwyczajna pętla `for` wyświetlająca łańcuchy znaków z przekazanej tablicy).

Listing 9.5. /functional/ReferecesDemo4.java

```
import java.util.Arrays;
import java.util.Comparator;

public class ReferencesDemo4 {

    static final String[] unsortedNames = {
        "Gosling", "de Raadt", "Torvalds", "Ritchie", "Hopper"
    };

    public static void main(String[] args) {
        String[] names;

        // Sortowanie z wykorzystaniem
        // "metody instancyjnej dowolnego obiektu konkretnego typu"
        names = unsortedNames.clone();
        Arrays.sort(names, String::compareToIgnoreCase);                    ❶
        dump(names);

        // Analogiczne sortowanie z użyciem wyrażenia lambda:
        names = unsortedNames.clone();
        Arrays.sort(names, (str1, str2) -> str1.compareToIgnoreCase(str2)); ❷
        dump(names);

        // Analogiczne sortowanie wykonane w standardowy sposób:
        names = unsortedNames.clone();
        Arrays.sort(names, new Comparator<String>() {                       ❸
            @Override
            public int compare(String str1, String str2) {
                return str1.compareToIgnoreCase(str2);
            }
        });
        dump(names);
```

```
        // Najprostszy sposób sortowania, z użyciem istniejące komparatora.
        names = unsortedNames.clone();
        Arrays.sort(names, String.CASE_INSENSITIVE_ORDER);                    ❹
        dump(names);
    }
```

❶ Używając „metody instancyjnej dowolnego obiektu konkretnego typu", deklarujemy odwołanie do metody `compareToIgnoreCase` dowolnego obiektu `String` użytego w wywołaniu.

❷ Przedstawia analogiczne sortowanie wykonane przy użyciu wyrażenia lambda.

❸ Pokazuje sposób, „którego używali nasi dziadkowie, pisząc programy w Javie".

❹ To przykład bezpośredniego użycia wyeksportowanego komparatora, który pokazuje, że wszystko można zrobić na kilka sposobów.

Na wszelki wypadek wykonałem powyższy przykład i uzyskałem następujące wyniki:

```
de Raadt Gosling Hopper Ritchie Torvalds
de Raadt Gosling Hopper Ritchie Torvalds
de Raadt Gosling Hopper Ritchie Torvalds
de Raadt Gosling Hopper Ritchie Torvalds
```

9.7. Wstawianie istniejącego kodu metod

Problem ⑧

Słyszeliśmy o możliwości tworzenia „wstawek", czyli wykorzystywania istniejących metod w innych klasach, i chcielibyśmy z niej skorzystać.

Rozwiązanie

Należy skorzystać z importu statycznego lub zadeklarować jeden bądź więcej interfejsów funkcyjnych z metodami „domyślnymi" zawierającymi kod, którego chcemy użyć, a następnie zaimplementować te metody.

Analiza

Programiści używający innych języków programowania często szydzili z Javy, wyśmiewając brak możliwości tworzenia tak zwanych „wstawek" (ang. *mixin*), czyli wykorzystywania fragmentów kodu pochodzącego z innych typów.

Jedną z możliwości uzyskania takiego efektu jest skorzystanie z „importu statycznego", która jest dostępna w Javie już od dekady. Jest ona powszechnie stosowana w tekstach jednostkowych (patrz receptura 1.13). Rozwiązanie to ma jednak tę wadę, że pozwala na wykorzystywanie wyłącznie metod statycznych, a nie instancyjnych.

Nowszy mechanizm korzysta z interesującego efektu ubocznego, będącego konsekwencją zmian wprowadzonych w języku Java 8, związanych z obsługą wyrażeń lambda: pozwala on na dołączanie do typu kodu zaimplementowanego w zupełnie odrębnych, niezwiązanych z nim typach danych. Czy twórcy Javy w końcu zrezygnowali ze swojego nieustępliwego sprzeciwu wobec wielokrotnego dziedziczenia? Na pierwszy rzut oka mogłoby się tak wydawać, ale spokojnie: nasze możliwości ograniczają się do wykorzystywania metod z wielu

interfejsów, a nie z wielu klas. Gdyby Czytelnik jeszcze nie zorientował się, że w interfejsach można definiować metody (a nie jedynie deklarować je), to powinien zajrzeć do receptury 9.5. Przeanalizujmy następujący przykład:

/lang/MixinsDemo.java

```
interface Bar {
    default String filter(String s) {
        return "Przefiltrowane " + s;
    }
}

interface Foo {
    default String convolve(String s) {
        return "zwinięte " + s;
    }
}

public class MixinsDemo implements Foo, Bar{

    public static void main(String[] args) {
        String input = args.length > 0 ? args[0] : "Witam";
        String output = new MixinsDemo().process(input);
        System.out.println(output);
    }

    private String process(String s) {
        return filter(convolve(s)); // Wywołanie wstawionych metod!
    }
}
```

Poniżej przedstawiłem wyniki wykonania tego programu:

```
C:\javasrc>javac -d build lang/MixinsDemo.java
C:\javasrc>java -cp build lang.MixinsDemo
Przefiltrowane zwinięte Witam

C:\javasrc>
```

No i proszę — obecnie Java już obsługuje wstawki!

Czy to oznacza, że mamy jak szaleni tworzyć interfejsy zawierające implementacje metod? Nie. Trzeba pamiętać, że rozwiązanie to opracowano z myślą o tworzeniu „interfejsów funkcyjnych" wykorzystywanych w wyrażeniach lambda. Stosowane z umiarem faktycznie pozwala na tworzenie wstawek i konstruowanie aplikacji w nieco inny sposób niż przy wykorzystaniu tradycyjnego dziedziczenia, agregacji oraz technik programowania aspektowego. Jednak nadużywanie tej techniki może prowadzić do powstania nieczytelnego kodu, doprowadzać do szaleństwa programistów przyzwyczajonych do starszych wersji Javy i wprowadzać chaos.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Ź

Ż

O'REILLY®

Java. Receptury

Wydanie III

Java to jeden z języków programowania najchętniej wybieranych do tworzenia zaawansowanych systemów informatycznych. Systemy bankowe, aukcyjne oraz inne zaawansowane programy wspomagające codzienną pracę tysięcy ludzi opierają się na Javie i narzędziach z nią związanych. Jeżeli chcesz śmiało wkroczyć w świat tego języka, musisz mieć tę książkę!

Książka należy do cenionej serii „Receptury", która przedstawia różne zagadnienia w postaci krótkich przepisów. Nie inaczej jest w tym przypadku. Sięgnij po nią i zobacz, jak kompilować, uruchamiać i testować tworzony kod. W kolejnych rozdziałach zaznajomisz się z najlepszymi metodami przetwarzania ciągów znaków oraz nauczysz się korzystać z wyrażeń regularnych i wykonywać operacje na liczbach. Ponadto zdobędziesz dogłębną wiedzę na temat urządzeń wejścia-wyjścia, używania systemu plików, sieci oraz drukarek. To obowiązkowa pozycja na półce każdego programisty Javy!

Dzięki tej książce:

- poznasz język programowania Java
- nauczysz się korzystać z systemu plików oraz sieci
- wykorzystasz wyrażenia regularne
- błyskawicznie rozwiążesz typowe problemy

Najlepsze przepisy dla programistów Javy!

ISBN 978-83-246-9570-6