

O'REILLY®

Wydanie IV

Java

Najlepsze
rozwiązania zadań
programistycznych

Receptury



Helion 

Ian F. Darwin

Tytuł oryginału: Java Cookbook: Problems and Solutions for Java Developers, 4th Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-7086-9

© 2021 Helion SA

Authorized Polish translation of the English edition of Java Cookbook, 4th Edition ISBN 9781492072584 © 2020 RejmiNet Group, Inc.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jarec4>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	13
1. Rozpoczynanie pracy: kompilacja i uruchamianie	27
1.0. Wprowadzenie	27
1.1. Kompilacja i uruchamianie programów napisanych w Javie — JDK	28
1.2. Kompilacja i uruchamianie: stosowanie GraalVM dla lepszej wydajności	30
1.3. Kompilacja, uruchamianie i testowanie programów przy użyciu IDE	32
1.4. Poznawanie Javy przy wykorzystaniu JShell	36
1.5. Efektywne wykorzystanie zmiennej środowiskowej CLASSPATH	39
1.6. Pobieranie przykładów dołączonych do tej książki i korzystanie z nich	41
1.7. Automatyzacja zależności, kompilacji, testowania i wdrażania przy użyciu programu Apache Maven	49
1.8. Automatyzacja zależności, kompilacji, testowania i wdrażania przy użyciu programu Gradle	53
1.9. Komunikaty o odrzuconych metodach	55
1.10. Zapewnianie poprawności kodu za pomocą testów jednostkowych: JUnit	57
1.11. Zarządzanie kodem z wykorzystaniem ciągłej integracji	60
1.12. Uzyskiwanie czytelnych komunikatów o wyjątkach	65
1.13. Poszukiwanie przykładowych kodów źródłowych	66
1.14. Poszukiwanie gotowych bibliotek Javy	67
2. Interakcja ze środowiskiem	71
2.0. Wprowadzenie	71
2.1. Pobieranie wartości zmiennych środowiskowych	71
2.2. Pobieranie informacji z właściwości systemowych	72
2.3. Stosowanie kodu zależnego od używanej wersji JDK lub systemu operacyjnego	74
2.4. Stosowanie rozszerzających interfejsów programistycznych lub innych API	79
2.5. Stosowanie systemu modułów Javy	80

3. Łącuchy znaków i przetwarzanie tekstów	85
3.0. Wprowadzenie	85
3.1. Odczytywanie fragmentów łańcucha	88
3.2. Łączenie łańcuchów znaków przy użyciu klasy StringBuilder	93
3.3. Przetwarzanie łańcucha znaków po jednej literze	95
3.4. Wyrównywanie łańcuchów znaków	97
3.5. Konwersja pomiędzy znakami Unicode a łańcuchami znaków	100
3.6. Odwracanie kolejności słów lub znaków w łańcuchu	103
3.7. Rozwijanie i kompresja znaków tabulacji	104
3.8. Kontrola wielkości liter	108
3.9. Wprowadzanie znaków niedrukowalnych	109
3.10. Usuwanie odstępów z końca łańcucha	111
3.11. Tworzenie komunikatów z użyciem zasobów wielojęzycznych	113
3.12. Stosowanie konkretnych ustawień lokalnych	116
3.13. Tworzenie wiązki zasobów	117
3.14. Program — proste narzędzie do formatowania tekstów	118
3.15. Program — fonetyczne porównywanie nazwisk	121
4. Dopasowywanie wzorców przy użyciu wyrażeń regularnych	125
4.0. Wprowadzenie	125
4.1. Składnia wyrażeń regularnych	127
4.2. Wykorzystanie wyrażeń regularnych w języku Java — sprawdzanie występowania wzorca	134
4.3. Odnajdywanie tekstu pasującego do wzorca	138
4.4. Zastępowanie określonego tekstu	140
4.5. Wyświetlanie wszystkich wystąpień wzorca	143
4.6. Wyświetlanie wierszy zawierających fragment pasujący do wzorca	145
4.7. Kontrola wielkości znaków w metodach match() i subst()	147
4.8. Dopasowywanie znaków z akcentami lub znaków złożonych	148
4.9. Odnajdywanie znaków nowego wiersza	149
4.10. Program — analiza dziennika serwera Apache	151
4.11. Program — pełna wersja programu grep	153
5. Liczby	159
5.0. Wprowadzenie	159
5.1. Sprawdzanie, czy łańcuch znaków stanowi poprawną liczbę	162
5.2. Konwertowanie liczb na obiekty i na odwrot	163
5.3. Pobieranie ułamka z liczby całkowitej bez konwertowania go na postać zmiennoprzecinkową	164
5.4. Stosowanie liczb zmiennoprzecinkowych	166
5.5. Formatowanie liczb	170

5.6. Konwersje pomiędzy różnymi systemami liczbowymi — dwójkowym, ósemkowym, dziesiętnym i szesnastkowym	174
5.7. Operacje na grupie liczb całkowitych	175
5.8. Formatowanie z zachowaniem odpowiedniej postaci liczby mnogiej	177
5.9. Generowanie liczb losowych	179
5.10. Mnożenie macierzy	182
5.11. Operacje na liczbach zespolonych	184
5.12. Obsługa liczb o bardzo dużych wartościach	187
5.13. Program TempConverter	189
5.14. Program — generowanie liczbowych palindromów	191
6. Daty i godziny	195
6.0. Wprowadzenie	195
6.1. Określanie bieżącej daty	198
6.2. Wyświetlanie daty i czasu w zadanym formacie	200
6.3. Konwersja liczb określających datę i czas oraz liczbę sekund	202
6.4. Analiza łańcuchów znaków i ich zamiana na daty	203
6.5. Obliczanie różnicy pomiędzy dwiema datami	204
6.6. Dodawanie i odejmowanie dat	206
6.7. Obsługa zdarzeń cyklicznych	207
6.8. Obliczanie dat z uwzględnieniem stref czasowych	209
6.9. Stosowanie starych klas Date i Calendar	210
7. Strukturalizacja danych w języku Java	213
7.0. Wprowadzenie	213
7.1. Strukturalizacja danych przy użyciu tablic	214
7.2. Modyfikacja wielkości tablic	216
7.3. Framework kolekcji	217
7.4. Klasa podobna do tablicy, lecz bardziej dynamiczna	219
7.5. Stosowanie typów ogólnych we własnych klasach	222
7.6. Jak przeglądać zawartość kolekcji? Wyliczenie dostępnych sposobów	226
7.7. Unikanie powtórzeń dzięki zastosowaniu zbioru	229
7.8. Strukturalizacja danych z wykorzystaniem list połączonych	230
7.9. Odzworowywanie z wykorzystaniem klas Hashtable oraz HashMap	235
7.10. Zapisywanie łańcuchów znaków w obiektach Properties i Preferences	237
7.11. Sortowanie kolekcji	242
7.12. Unikanie konieczności sortowania danych	246
7.13. Odnajdywanie obiektu w kolekcji	248
7.14. Zamiana kolekcji na tablicę	251
7.15. Zapewnianie możliwości przeglądania danych przy użyciu iteratora	252
7.16. Używanie stosu obiektów	254

7.17. Struktury wielowymiarowe	257
7.18. Upraszczenie obiektów danych przy wykorzystaniu bibliotek Lombok lub Record	259
7.19. Program — porównanie szybkości działania	262
8. Techniki obiektowe	265
8.0. Wprowadzenie	265
8.1. Metody klasy Object — formatowanie obiektów przy użyciu metody toString() i porównywanie ich metodą equals()	268
8.2. Wykorzystanie klas wewnętrznych	275
8.3. Tworzenie metod zwrotnych z wykorzystaniem interfejsów	278
8.4. Polimorfizm i metody abstrakcyjne	282
8.5. Wartości wyliczeniowe bezpieczne dla typów	283
8.6. Unikanie wyjątków NPE z użyciem klasy Optional	287
8.7. Wymuszanie użycia wzorca Singleton	289
8.8. Zgłaszanie własnych wyjątków	291
8.9. Wstrzykiwanie zależności	293
8.10. Program Plotter	296
9. Techniki programowania funkcyjnego: interfejsy funkcyjne, strumienie i kolekcje równoległe	301
9.0. Wprowadzenie	301
9.1. Stosowanie wyrażeń lambda lub domknięć zamiast klas wewnętrznych	304
9.2. Stosowanie predefiniowanych interfejsów lambda zamiast własnych	307
9.3. Upraszczenie przetwarzania z wykorzystaniem interfejsu Stream	311
9.4. Upraszczenie strumieni przy użyciu kolektorów	312
9.5. Poprawianie przepustowości dzięki wykorzystaniu strumieni i kolekcji równoległych	315
9.6. Używanie istniejącego kodu w sposób funkcyjny dzięki wykorzystaniu odwołań do metod	317
9.7. Wstawianie istniejącego kodu metod	321
10. Wejście i wyjście: odczyt, zapis i sztuczki z katalogami	323
10.0. Wprowadzenie	323
10.1. Klasy InputStream/OutputStream oraz Reader/Writer	325
10.2. Odczytywanie pliku tekstowego	327
10.3. Odczytywanie informacji ze standardowego strumienia wejściowego bądź z konsoli/terminala	329
10.4. Wyświetlanie tekstów przy użyciu klasy Formatter i metody printf	334
10.5. Analiza zawartości pliku przy użyciu klasy StringTokenizer	338
10.6. Analiza danych wejściowych przy użyciu klasy Scanner	342
10.7. Analiza danych wejściowych o strukturze gramatycznej	345
10.8. Kopiowanie pliku	349

10.9. Zmiana skojarzeń standardowych strumieni	350
10.10. Powielanie strumienia podczas realizacji operacji zapisu; zmiana skojarzeń standardowych strumieni	351
10.11. Odczyt i zapis danych zakodowanych w innym zbiorze znaków	354
10.12. Te kłopotliwe znaki końca wiersza	356
10.13. Kod operujący na plikach w sposób zależny od systemu operacyjnego	356
10.14. Odczytywanie i zapisywanie danych binarnych	358
10.15. Odczytywanie i zapisywanie danych w archiwach JAR oraz ZIP	358
10.16. Odnajdywanie plików w sposób niezależny od systemu operacyjnego przy użyciu metod <code>getResource()</code> i <code>getResourceAsStream()</code>	362
10.17. Pobieranie informacji o pliku	364
10.18. Tworzenie nowego pliku lub katalogu	371
10.19. Zmiana nazwy pliku lub jego atrybutów	372
10.20. Usuwanie plików	375
10.21. Tworzenie plików tymczasowych	377
10.22. Tworzenie listy zawartości katalogu	379
10.23. Pobieranie katalogów głównych	380
10.24. Stosowanie usługi <code>WatchService</code> do uzyskiwania informacji o zmianach pliku	382
10.25. Zapisywanie danych użytkownika na dysku	384
10.26. Program <code>Find</code> — poruszanie się po drzewie katalogów	388
11. Nauka o danych i język R	393
11.1. Uczenie maszynowe z użyciem Javy	394
11.2. Stosowanie danych w Apache Spark	395
11.3. Interaktywne stosowanie R	398
11.4. Porównanie i wybór implementacji R	400
11.5. Używanie R w aplikacjach Javy: Renjin	401
11.6. Stosowanie kodu Javy w sesji R	403
11.7. Stosowanie FastR — implementacji R korzystającej z GraalVM	405
11.8. Stosowanie R w aplikacjach internetowych	406
12. Klienci sieciowe	409
12.0. Wprowadzenie	409
12.1. Internetowy klient HTTP/REST	412
12.2. Nawiązywanie połączenia z serwerem z użyciem gniazd	414
12.3. Odnajdywanie adresów sieciowych i zwracanie informacji o nich	415
12.4. Obsługa błędów sieciowych	417
12.5. Odczyt i zapis danych tekstowych	418
12.6. Odczyt i zapis danych binarnych	420
12.7. Datagramy UDP	423

12.8. URI, URL czy może URN?	426
12.9. Program — klient TFTP wykorzystujący protokół UDP	427
12.10. Program — klient pogawędek internetowych używający gniazd	432
12.11. Program — sprawdzanie odnośników HTTP	436
13. Programy Javy działające na serwerze	439
13.0. Wprowadzenie	439
13.1. Tworzenie serwera	440
13.2. Znajdowanie interfejsów sieciowych	443
13.3. Zwracanie odpowiedzi (łańcucha znaków bądź danych binarnych)	444
13.4. Zwracanie informacji o obiektach przez połączenie sieciowe	448
13.5. Obsługa wielu klientów	449
13.6. Serwer obsługujący protokół HTTP	454
13.7. Zabezpieczanie serwera WWW przy użyciu SSL i JSSE	456
13.8. Tworzenie usługi typu RESTful z użyciem JAX-RS	459
13.9. Rejestracja operacji sieciowych	462
13.10. Konfiguracja i stosowanie SLF4J	463
13.11. Rejestracja przez sieć przy użyciu Log4j	466
13.12. Rejestracja przez sieć przy użyciu pakietu java.util.logging	471
14. Przetwarzanie danych w formacie JSON	475
14.0. Wprowadzenie	475
14.1. Bezpośrednie generowanie danych w formacie JSON	477
14.2. Analiza i zapisywanie danych JSON przy użyciu pakietu Jackson	478
14.3. Analiza i zapis danych w formacie JSON przy użyciu pakietu org.json	480
14.4. Analiza i zapis danych w formacie JSON przy użyciu biblioteki JSON-B	481
14.5. Znajdowanie elementów danych w formacie JSON przy użyciu wskaźników	483
15. Pakiety i ich tworzenie	487
15.0. Wprowadzenie	487
15.1. Tworzenie pakietu	488
15.2. Tworzenie dokumentacji klas przy użyciu programu Javadoc	490
15.3. Więcej niż Javadoc — adnotacje i metadane	494
15.4. Tworzenie klasy w taki sposób, by była komponentem JavaBeans	496
15.5. Umieszczanie pakietów w plikach JAR	500
15.6. Wykonywanie programu spakowanego w pliku JAR	501
15.7. Pakowanie komponentów warstwy internetowej do pliku WAR	503
15.8. Tworzenie mniejszego pakietu dystrybucyjnego przy użyciu programu jlink	504
15.9. Stosowanie JPMS do tworzenia modułów	506

16. Stosowanie wątków w Javie	509
16.0. Wprowadzenie	509
16.1. Uruchamianie kodu w innym wątku	511
16.2. Animacja — wyświetlanie poruszających się obrazów	515
16.3. Zatrzymywanie działania wątku	519
16.4. Spotkania i ograniczenia czasowe	522
16.5. Synchronizacja wątków przy użyciu słowa kluczowego synchronized	523
16.6. Upraszczanie synchronizacji przy użyciu blokad	529
16.7. Upraszczanie programu producent-konsument przy użyciu interfejsu Queue	533
16.8. Optymalizacja działania równoległego przy użyciu Fork/Join	535
16.9. Planowanie zadań: operacje w przyszłości, zapis danych w tle w programach edycyjnych	539
17. Introspekcja lub „klasa o nazwie Class”	543
17.0. Wprowadzenie	543
17.1. Pobieranie deskryptora klasy	544
17.2. Określanie oraz stosowanie metod i pól	545
17.3. Uzyskiwanie dostępu do prywatnych pól i metod za pomocą introspekcji	549
17.4. Dynamiczne ładowanie i instalowanie klas	550
17.5. Tworzenie nowej klasy od podstaw przy użyciu obiektu ClassLoader	552
17.6. Tworzenie nowej klasy od podstaw z użyciem kompilatora Javy	554
17.7. Określanie efektywności działania	557
17.8. Wyświetlanie informacji o klasie	561
17.9. Wyświetlanie klas należących do pakietu	563
17.10. Stosowanie i definiowanie adnotacji	565
17.11. Zastosowanie adnotacji do odnajdywania klas pełniących rolę wtyczek	570
17.12. Program CrossRef	573
18. Wykorzystywanie Javy wraz z innymi językami programowania	577
18.0. Wprowadzenie	577
18.1. Uruchamianie zewnętrznego programu	578
18.2. Wykonywanie programu i przechwytywanie jego wyników	582
18.3. Wywoływanie kodu napisanego w innych językach przy użyciu javax.script	586
18.4. Łączenie różnych języków z użyciem GraalVM	588
18.5. Łączenie języków Java i Perl	590
18.6. Dołączanie kodu rodzimego	593
18.7. Wywoływanie kodu Javy z kodu rodzimego	599
Posłowie	603
A Java kiedyś i obecnie	605

Techniki programowania funkcyjnego: interfejsy funkcyjne, strumienie i kolekcje równoległe

9.0. Wprowadzenie

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 Haskell¹, Idris, OCaml, Erlang, Julia oraz rodzina języków LISP. Jednak wymagałoby to odejścia z ekosystemu języka Java. Można by się ewentualnie zastanowić nad wykorzystaniem

¹ Haskell został użyty do napisania dość kompletnej kopii serwisu Twitter mieszczącej się w kilkuset wierszach kodu; patrz <https://github.com/Gabriel439/simple-twitter>.

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. No i oczywiście istnieje jeszcze język Kotlin (<https://kotlinlang.org>), ostatni podobny do Javy język korzystający z JVM.

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 niezmiennicze.
- 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 jakiegokolwiek szczególnej składni, gdyż funkcje są danymi.

Jednym z rozwiązań wprowadzonych w języku Java mających 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. 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 wyrażen lambda.

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

public class RunnableLambda {

    public static void main(String[] args) {
        threadPool.submit(() -> System.out.println("Witam w wątku"));
    }
}
```

Niezmiennie dane to coś, co teoretycznie jest proste: przykładem może być klasa mająca jedynie akcesory odczytujące (metody *get*). Klasa *String* należąca do standardowej biblioteki Javy jest niezmienna: jej metody, takie jak *substring()* bądź *toUpperCase()*, nie zmieniają początkowego łańcucha, lecz tworzą nowe obiekty *String* zawierające żądane modyfikacje. Pomimo to, łańcuchy są użyteczne i powszechnie stosowane. Także typy wyliczeniowe (*enum*) w niejawnym sposób są niezmiennie. Istnieje także propozycja dodania w wersjach Java 14 lub 15 nowego obiektu przypominającego klasę, o nazwie *record*, który byłby w niejawnym sposób niezmienny — zakłada się, że w jego przypadku kompilator będzie generować metody *get* (oraz konstruktor i trzy podstawowe metody klasy *Object*), lecz nie metody *set*.

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 wartość 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.

Jeśli Czytelnik zna potoki i filtry systemu Unix, to ta równoważność będzie miała dla niego sens, jeśli natomiast Czytelnik ich nie zna, to na razie może pominąć kilka kolejnych akapitów. Poniżej przedstawiłem przykład polecenia systemu Unix:

```
cat lines.txt | sort | uniq | wc -l
```

A oto odpowiednik tego polecenia korzystający ze strumieni Javy:

```
jshell> long numberLines =  
    new BufferedReader(  
        new FileReader("lines.txt")).lines().sorted().distinct().count();  
numberLines ==> 5
```

Na listingu 9.1 ten sam kod został zapisany w nieco bardziej idiomatyczny sposób. Jednak obie jego wersje dają ten sam wynik. W przypadku niewielkich zbiorów danych wejściowych potoki systemu Unix działają szybciej; jednak kod napisany w Javie powinien być szybszy, gdy danych będzie dużo, zwłaszcza jeśli zostaną wykorzystane możliwości przetwarzania równoległego.

Listing 9.1. javasrc/main/src/main/java/functional/UnixPipesFiltersReplacement.java

```
long numberLines = Files.lines(Path.of("lines.txt"))  
    .sorted()  
    .distinct()  
    .count();  
System.out.printf("Plik lines.txt zawiera " + numberLines + " unikalnych wierszy.");
```

Z klasami Stream powiązany jest interfejs *Splititerator*, 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* napisanej przez Neala Forda i wydanej przez wydawnictwo O'Reilly.

Dostępna jest także książka Richarda Warburtona *Java 8 Lambdas*, która została w całości poświęcona wyrażeniom lambda oraz związanym z nimi narzędziom.

9.1. Stosowanie wyrażeń lambda lub domknięć zamiast klas wewnętrznych

Problem

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);
```

Stosowanie wyrażen lambda jest obecnie preferowanym sposobem implementacji obsługi graficznego interfejsu użytkownika aplikacji. Poniżej zamieściłem przykład takiego rozwiązania:

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

Dziś już nie wszyscy tworzą aplikacje z graficznym interfejsem użytkownika z wykorzystaniem biblioteki Swing, dlatego zacznę od przykładu, który nie jest z nimi w żaden sposób związany. Założmy, 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?²

² Gdybyśmy kiedyś musieli wykonywać tego typu operacje na informacjach przechowywanych w bazie danych, wykorzystując przy tym Java Persistence API, to warto się zainteresować projektami Spring Data (<https://spring.io/projects/spring-data>) lub Apache DeltaSpike (<http://deltaspike.apache.org/>), który pozwala na definiowanie interfejsów zawierających metody o nazwach takich jak `findCameraByInterchangeableTrueAndPriceLessThan(double price)` i jest w stanie sam je zaimplementować.

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() < 1500;
}
```

W tym celu musielibyśmy stworzyć interfejs o poniższej postaci³:

javasrc/main/src/main/java/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:

```
results = searchApp.search(new CameraAcceptor() {
    public boolean choose(Camera c) {
        return c.isIlc() && c.getPrice() < 1500;
    }
});
```

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() < 1500;
    }
}

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 wyszukiwającego. 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 wyrażen lambda w Javie jest potraktowanie ich jako *metod implementujących interfejs funkcyjny*. Z wykorzystaniem wyrażen lambda powyższy kod można zapisać w następującej postaci:

```
results = searchApp.search(c -> c.isIlc() && c.getPrice() < 1500);
```

³ Czytelnikom, którzy nie interesują się aparatami fotograficznymi, wyjaśniam, że określenie „aparat z wymiennym obiektywem” obejmuje dwie kategorie aparatów, które można znaleźć w sklepach: tradycyjne lustrzanki cyfrowe (ang. *Digital Single Lens Reflection*, DSLR) oraz nową kategorię „aparatów kompaktowych”, takich jak Nikon 1 oraz aparaty serii Z, Sony ILCE (znany wcześniej pod nazwą NEX) oraz Canon EOS-M, które są mniejsze i lżejsze od starszych aparatów DSLR.

Zapis ze strzałką ($\rightarrow$) 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() < 1500)
        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. Założmy, 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

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

Rozwiązanie

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

```

Wootage.java x
package Wootage;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Created with IntelliJ IDEA.
 * User: Ian
 * Date: 3/2/14
 * Time: 5:15pm
 * To change this template use File | Settings | File Templates.
 */
public class Wootage {
    public static void main(String[] args) {
        System.out.println("Witamy w aplikacji Java.");
    }

    ActionListener listener = new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            System.out.println("Źródło akcji " + evt.getSource());
        }
    };
}

```

Rysunek 9.1. Kod w środowisku IntelliJ w pełnej postaci

```

Wootage.java x
package Wootage;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Created with IntelliJ IDEA.
 * User: Ian
 * Date: 3/2/14
 * Time: 5:15pm
 * To change this template use File | Settings | File Templates.
 */
public class Wootage {
    public static void main(String[] args) {
        System.out.println("Witamy w aplikacji Java.");
    }

    ActionListener listener = (evt) -> { System.out.println("Źródło akcji " + evt.getSource()); };
}

```

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()`:

javasrc/main/src/main/java/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;
}
```

Żałujemy, 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.

Tworzenie własnego interfejsu funkcyjnego

Choć JDK zawiera obszerny zbiór interfejsów funkcyjnych, to jednak może się zdarzyć, że staniemy w obliczu konieczności utworzenia własnego. Poniżej przedstawiłem prosty przykład takiego interfejsu:

```
javasrc/main/src/main/java/functional/ProcessIntsUsingFunctional.java
```

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

Adnotacja `@FunctionalInterface` każe kompilatorowi upewnić się, że dany interfejs jest oraz pozostanie funkcyjnym. Jej zastosowanie odpowiada użyciu adnotacji `@Override` (obie zostały zdefiniowane w pakiecie `java.lang`). Stosowanie tej adnotacji zawsze jest opcjonalne.

Wspomnianego interfejsu można by użyć w poniższym programie do przetworzenia tablicy liczb całkowitych:

```
javasrc/main/src/main/java/functional/ProcessIntsUsingFunctional.java
```

```
public class ProcessIntsUsingFunctional {  
    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.

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`.

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

9.3. Upraszczanie przetwarzania z wykorzystaniem interfejsu Stream

Problem

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: `collect()`, `count()`, `findFirst()`, `max()`, `min()`, `reduce()` czy też `sum()`.

Listing 9.2 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 9.2. `javasrc/main/src/main/java/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. Upraszczanie strumieni przy użyciu kolektorów

Problem

Tworzymy strumienie, lecz są one złożone i nieefektywne.

Rozwiązanie

Należy użyć kolektorów.

Analiza

Listing 9.2 zakończył pierwszą połowę tego rozdziału wywołaniem metody `collect()`. Argumentem tej metody jest obiekt typu `Collector`, który opiszę bardziej szczegółowo w tej recepturze. Kolektory, czyli implementacje interfejsu `Collector`, są czymś, co w terminologii programowania funkcyjnego jest określane jako *fold*.⁴ Operacje tego typu są także określane mianem operacji redukcji, akumulacji, agregacji, kompresji lub wstrzykiwania. W programowaniu funkcyjnym *fold* jest operacją końcową; odpowiednikiem złożenia całego ciągu połączonych ze sobą biletów w jeden plik (patrz rysunek 9.3). Łańcuch biletów może reprezentować strumień, operacja *fold* — funkcję, a wynik... jest po prostu wynikiem tej operacji. Bardzo często będzie ona zawierać operację łączącą, taką jak policzenie wszystkich złożonych biletów.

⁴ W języku angielskim słowo „fold” oznacza składać lub związać — *przyp. tłum.*



Rysunek 9.3. Strumień biletów przed złożeniem, podczas składania oraz po złożeniu: operacja kończąca

Warto zwrócić uwagę, że na lewym zdjęciu z rysunku 9.3 nie wiemy jak długi jest strumień, jednak oczekujemy, że wcześniej czy później się on skończy.

W terminologii języka Java *kolektor* (ang. *collector*) jest funkcją kończącą, która analizuje lub podsumowuje zawartość strumienia. Z technicznego punktu *Collector* jest interfejsem, którego implementacja jest określana przez trzy (albo cztery) funkcje, współdziałające ze sobą w celu zgromadzenia elementów w formie kolekcji, mapy, bądź też innego modyfikowalnego kontenera wynikowego, oraz opcjonalnego, końcowego przekształcenia zawartości tego kontenera na wynik. Poniżej opisałem każdą z czterech funkcji kolektora:

- utworzenie nowego kontenera wynikowego (`supplier()`);
- dodanie do kontenera wynikowego nowego elementu danych (`accumulator()`);
- połączenie dwóch kontenerów wynikowych w jeden (`combiner()`);
- wykonanie końcowego przekształcenia na kontenerze (`finisher()`); jest to operacja opcjonalna.

Choć bez trudu można tworzyć własne implementacje interfejsu *Collector*, to jednak często wskazane jest stosowanie jednej z wielu predefiniowanych implementacji dostępnych w klasie *Collectors*. Poniżej przedstawiłem kilka prostych przykładów:

```
int howMany = cameraList.stream().collect(Collectors.counting());
double howMuch = cameraList.filter(desiredFilter)
    .collect(Collectors.summingDouble(Camera::getPrice));
```

Listing 9.3 przedstawia implementację klasycznego algorytmu liczenia częstotliwości występowania słów: odczytuje plik tekstowy, dzieli go na słowa, zlicza występowanie poszczególnych wyrazów, a następnie wyświetla n najczęściej występujących wyrazów posortowanych w kolejności malejącej na podstawie liczby ich wystąpień.

W systemie Unix taką operację można wykonać używając następującego polecenia (przy czym $n = 20$):

```
prep $file | sort | uniq -c | sort -nr | head -20
```

W powyższym poleceniu prep jest skryptem, który używa uniksowego narzędzia tr, by podzielić wiersze tekstu na poszczególne wyrazy i zapisać je małymi literami.

Listing 9.3. *javasrc/main/src/main/java/functional/WordFreq.java*

```
package functional;

import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

// tag::main[]
/**
 * Implementacja zliczania liczby wystąpień wyrazów, zapisana w dwóch instrukcjach.
 */
public class WordFreq {
    public static void main(String[] args) throws IOException {

        // 1) Zgromadzenie słów w kolekcji Map<String, Long>, ze zliczeniem i usunięciem powtórzeń.
        Map<String, Long> map = Files.lines(Path.of(args[0]))
            .flatMap(s -> Stream.of(s.split(" ")))
            .collect(Collectors.groupingBy(
                String::toLowerCase, Collectors.counting()));

        // 2) Wyświetlenie 20 pierwszych wyników, posortowanych liczbowo w kolejności malejącej.
        map.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
            .limit(20)
            .map(entry -> String.format("%4d %s", entry.getValue(), entry.getKey()))
            .forEach(System.out::println);
    }
}
```

Ten program składa się z dwóch kroków. Pierwszy z nich polega na utworzeniu mapy słów oraz liczby ich wystąpień. Drugi na posortowaniu ich w odwrotnej kolejności, wybranie pierwszych 20, odpowiednim sformatowaniu i wyświetleniu.

Pierwszy krok korzysta z metody `Files.lines()` przedstawionej w rozdziale 10., by pobrać strumień łańcuchów, który jest następnie dzielony na poszczególne słowa przy użyciu metody `flatMap()` interfejsu `Stream` stosowanej wspólnie z metodą `split()` klasy `String`, dzielącą łańcuch na słowa w miejscach wystąpienia jednej lub kilku spacji. Wyniki tej operacji są gromadzone przy użyciu kolektora w kolekcji typu `HashMap`. Początkowo używałem do tego celu samodzielnie przygotowanego kolektora o następującej postaci:

```
.collect(HashMap::new, (m,s)->m.put(s, m.getOrDefault(s,0)+1), HashMap::putAll);
```

Ta wersja metody `collect()` pobiera trzy argumenty:

- Instancję `Supplier<R>` lub metodę wytwórczą służącą do utworzenia pustego kontenera; w tym przypadku zastosowałem konstruktor klasy `HashMap`.

- Akumulator typu `BiConsumer<R, ? super T>` służący do dodawania każdego elementu do mapy i dodania 1 za każdym razem, gdy trafimy na już istniejące słowo.
- Metody typu `BiConsumer<R, R>` używanej do połączenia wszystkich zebranych wyników.

W razie stosowania strumieni równoległych (patrz receptura 9.5) instancja `Supplier` może być wywoływana wiele razy w celu tworzenia wielu kontenerów, a każdy element zawartości strumienia zostanie obsłużony przez inny akumulator i zapisany do innego kontenera. Ostatnia z przekazanych metod, kończąca przetwarzanie, służy do połączenia tych kontenerów w jeden.

Niemniej jednak Sender Mak zauważył, że łatwiej jest używać predefiniowanego kolektora `groupingBy` dostępnego w klasie `Collectors`, połączonego w sposób przedstawiony poniżej z wywołaniem metod `toLowerCase()` oraz `collect()`:

```
.collect(Collectors.groupingBy(String::toLowerCase, Collectors.counting()));
```

Aby jeszcze bardziej uprościć kod, można by połączyć obie przedstawione wcześniej instrukcje w jedną, wykonując w tym celu następujące czynności:

- Usunięcie wartości wynikowej i przypisanie `Map<String, Long> =`.
- Usunięcie średnika umieszczonego za wywołaniem metody `collect()`.
- Usunięcie `.map()` z wywołania `entrySet()`.

Teraz będziemy mogli stwierdzić, że udało się nam zrobić w Javie coś naprawdę użytecznego przy użyciu jednej instrukcji!

9.5. Poprawianie przepustowości dzięki wykorzystaniu strumieni i kolekcji równoległych

Problem

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 16.

Framework 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 założymy, 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):

javasrc/main/src/main/java/functional/CameraSearchParallelStream.java

```
public static void main(String[] args) {  
    System.out.println("Poszukiwanie wyników przy użyciu pętli for.");  
    for (Object camera : privateListOfCameras.parallelStream().  
        filter(c -> c.isIlc() && c.getPrice() < 500).  
        toArray()) {  
        System.out.println(camera);  
    }  
  
    System.out.println("Poszukiwanie wyników przy użyciu prostszego, bardziej funkcyjnego  
↳ rozwiązania.");  
    privateListOfCameras.parallelStream().  
        filter(c -> c.isIlc() && c.getPrice() < 500).  
        forEach(System.out::println);  
}
```

- 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.
- ❺ Bardziej zwarty sposób zapisu operacji wyszukiwania.



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 16.

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, jak to określa dokumentacja Javy, „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 9.4 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 9.4. `javasrc/main/src/main/java/functional/ReferencesDemo.java`

```
/** "Chodź, nie biegać" */
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 9.5 tworzy obiekt `AutoCloseable` przeznaczony do użycia w instrukcji `try` zarządzającej zasobami. 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 9.5. `javasrc/main/src/main/java/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, zdefiniowany w sposób opisany w podpunkcie pt. „Tworzenie własnego interfejsu funkcyjnego” w recepturze 9.2. 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 9.6. 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 `ReferenesDemo3` 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 9.6. `javasrc/main/src/main/java/functional/ReferenesDemo3.java`

```
public class ReferenesDemo3 {

    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 = ReferenesDemo3::work;
        System.out.println("Główna metoda obliczeniowa: " + sample);
    }
}
```

Poniżej przedstawiłem wyniki generowane przez ten program:

```
Główna metoda obliczeniowa: functional.ReferenesDemo3$$Lambda$1/918221580@4517d9a3
```

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

Ostatni rodzaj odwołań do funkcji 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 9.7 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 chciałem 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 9.7. `javasrc/main/src/main/java/functional/ReferencesDemo4.java`

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

public class ReferencesDemo4 {

    static final String[] unsortedNames = {
        "Gosling", "de Raadt", "Amdahl", "Turing", "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 istniejącego 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:

```
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
```

9.7. Wstawianie istniejącego kodu metod

Problem

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 1.10). 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 ramki pt. „Klasa pochodna, klasa abstrakcyjna czy interfejs?” w recepturze 8.3. Przeanalizujmy następujący przykład:

```
javasrc/main/src/main/java/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.
- Zapewnieniu możliwości uzupełniania istniejących interfejsów o nowe metody bez konieczności wprowadzania zmian w *starych* implementacjach. Jak w przypadku wielu zmian wprowadzanych w Javie w przeszłości, także tym razem zapewnienie zgodności wstecz było niezwykle ważne.

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.

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Java: najlepsze rozwiązania najważniejszych zadań!

Java jest jednym z najpopularniejszych języków programowania. Równocześnie należy do najbardziej innowacyjnych technologii informatycznych; zawsze stanowiła awangardę. Programiści cenią Javę za dojrzałość, bezpieczeństwo i wszechstronność. Jednak nawet tak znakomity język niekiedy przysparza problemów podczas pracy. Mogą się one okazać całkiem proste do rozwiązania, jeśli tylko wiadomo, jak się do tego zabrać.

Oto zbiór aktualnych i kompletnych receptur instruktażowych, z których każda pomaga w rozwiązaniu konkretnego problemu. Wszystkie zostały starannie przetestowane i wielokrotnie udowodniły swoją przydatność. Każdą poprzedzono krótkim wprowadzeniem i omówieniem zastosowanych mechanizmów działania. Ta książka nie jest klasycznym podręcznikiem programowania, jednak z pewnością przyspieszy praktyczne wykorzystanie możliwości Javy. Wśród receptur znalazł się szeroki zakres zagadnień, od podstawowych operacji na ciągach znaków, poprzez programowanie funkcyjne, po komunikację sieciową, rozwiązania big data i współdziałanie z kodem napisanym w innych językach. W tym wydaniu uwzględniono zmiany wprowadzone w wersjach Javy 12, 13 i 14.

W książce między innymi:

- kompilacja kodu, uruchamianie i debugowanie oraz pakowanie klas Javy
- praca z tekstem, wyrażenia regularne i wzorce
- programowanie obiektowe i funkcyjne oraz programowanie sieciowe
- format JSON i wymiana danych
- wielowątkowość i współbieżność
- big data i Java

Ian F. Darwin jest programistą i konsultantem z wieloletnim doświadczeniem. Javą zajmuje się od chwili jej wprowadzenia, jest założycielem i członkiem grupy Sun/Oracle Java Champions. Prowadzi kursy programowania w Javie oraz administrowania systemami Unix. Mieszka w wiejskiej posiadłości, gdzie wraz z rodziną hoduje kury.

Helion 

 **helion.pl**

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

Sprawdź nasze szkolenia!

SKOLENIA



AKADEMIA IT & BUSINESS

HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej! 



ISBN 978-83-283-7086-9



9 788328 370869