

Joshua Bloch

Java

EFEKTYWNE PROGRAMOWANIE



Wydanie III

Helion 

Tytuł oryginału: Effective Java (3rd Edition)

Tłumaczenie: Rafał Jońca

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

ISBN: 978-83-283-4577-5

Authorized translation from the English language edition, entitled: EFFECTIVE JAVA, Third Edition; ISBN 0134685997; by Joshua Bloch; published by Pearson Education, Inc, publishing as Addison-Wesley Professional. Copyright © 2018 by Pearson Education, Inc.

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 Pearson Education, Inc. Polish language edition published by HELION S.A. Copyright © 2018.

Portions copyright © 2001-2008 Oracle and/or its affiliates.

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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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/javap3_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Książkę dedykuję mojej rodzinie — Cindy, Timowi i Mattowi.

Spis treści

Słowo wstępne	9
Przedmowa	11
Podziękowania	15
Rozdział 1. Wprowadzenie	19
Rozdział 2. Tworzenie i usuwanie obiektów	23
Temat 1. Tworzenie statycznych metod fabrycznych zamiast konstruktorów	23
Temat 2. Zastosowanie budowniczego do obsługi wielu parametrów konstruktora	28
Temat 3. Wymuszanie właściwości singleton za pomocą prywatnego konstruktora lub typu enum	36
Temat 4. Wykorzystanie konstruktora prywatnego w celu uniemożliwienia utworzenia obiektu	38
Temat 5. Stosuj wstrzykiwanie zależności zamiast odwoływania się do zasobów na sztywno	39
Temat 6. Unikanie powielania obiektów	41
Temat 7. Usuwanie niepotrzebnych referencji do obiektów	45
Temat 8. Unikanie finalizatorów i oczyszczaczy	48
Temat 9. Preferuj konstrukcję try z zasobami zamiast try-finally	54
Rozdział 3. Metody wspólne dla wszystkich obiektów	57
Temat 10. Zachowanie założeń w trakcie przedefiniowywania metody equals	58
Temat 11. Przedefiniowywanie metody hashCode wraz z equals	70
Temat 12. Przedefiniowywanie metody toString	75

Temat 13. Rozsądne przedefiniowywanie metody clone	78
Temat 14. Implementacja interfejsu Comparable	86
Rozdział 4. Klasy i interfejsy	93
Temat 15. Ograniczanie dostępności klas i ich składników	93
Temat 16. Stosowanie metod akcesorów zamiast pól publicznych w klasach publicznych	98
Temat 17. Zapewnianie niezmienności obiektu	100
Temat 18. Zastępowanie dziedziczenia kompozycją	107
Temat 19. Projektowanie i dokumentowanie klas przeznaczonych do dziedziczenia	113
Temat 20. Stosowanie interfejsów zamiast klas abstrakcyjnych	119
Temat 21. Projektowanie interfejsów na długie lata	124
Temat 22. Wykorzystanie interfejsów jedynie do definiowania typów	127
Temat 23. Zastępowanie oznaczania klas hierarchią	129
Temat 24. Zalety stosowania statycznych klas składowych	132
Temat 25. Ograniczenie pliku źródłowego do pojedynczej klasy głównego poziomu	135
Rozdział 5. Typy ogólne	139
Temat 26. Nie korzystaj z typów surowych	139
Temat 27. Eliminowanie ostrzeżeń o braku kontroli	144
Temat 28. Korzystanie z list zamiast tablic	147
Temat 29. Stosowanie typów ogólnych	151
Temat 30. Stosowanie metod ogólnych	156
Temat 31. Zastosowanie związanych szablonów do zwiększania elastyczności API	159
Temat 32. Ostrożne łączenie typów ogólnych i parametrów varargs	166
Temat 33. Wykorzystanie heterogenicznych kontenerów bezpiecznych dla typów	171
Rozdział 6. Typy wyliczeniowe i adnotacje	177
Temat 34. Użycie typów wyliczeniowych zamiast stałych int	177
Temat 35. Użycie pól instancyjnych zamiast kolejności	188
Temat 36. Użycie EnumSet zamiast pól bitowych	189
Temat 37. Użycie EnumMap zamiast indeksowania kolejnością ...	191
Temat 38. Emulowanie rozszerzalnych typów wyliczeniowych za pomocą interfejsów	196
Temat 39. Korzystanie z adnotacji zamiast wzorców nazw	200
Temat 40. Spójne użycie adnotacji Override	207
Temat 41. Użycie interfejsów znacznikowych do definiowania typów	210

Rozdział 7. Lambdy i strumienie	213
Temat 42. Stosuj lambdy zamiast klas anonimowych	213
Temat 43. Wybieraj referencje do metod zamiast lambd	217
Temat 44. Korzystaj ze standardowych interfejsów funkcyjnych ...	219
Temat 45. Rozważnie korzystaj ze strumieni	223
Temat 46. Stosuj w strumieniach funkcje bez efektów ubocznych	231
Temat 47. Zwracaj kolekcje zamiast strumieni	236
Temat 48. Ostrożnie korzystaj ze strumieni zrównoleglonych	242
Rozdział 8. Metody	247
Temat 49. Sprawdzanie poprawności parametrów	247
Temat 50. Defensywne kopiowanie	250
Temat 51. Projektowanie sygnatur metod	255
Temat 52. Rozsądne korzystanie z przeciążania	257
Temat 53. Rozsądne korzystanie z metod varargs	263
Temat 54. Zwracanie pustych tablic lub kolekcji zamiast wartości null	265
Temat 55. Rozsądne zwracanie obiektów opcjonalnych	267
Temat 56. Tworzenie komentarzy dokumentujących dla udostępnianych elementów API	272
Rozdział 9. Programowanie	281
Temat 57. Ograniczanie zasięgu zmiennych lokalnych	281
Temat 58. Stosowanie pętli for-each zamiast tradycyjnych pętli for	284
Temat 59. Poznanie i wykorzystywanie bibliotek	287
Temat 60. Unikanie typów float i double, gdy potrzebne są dokładne wyniki	290
Temat 61. Stosowanie typów prostych zamiast opakowanych typów prostych	292
Temat 62. Unikanie typu String, gdy istnieją bardziej odpowiednie typy	296
Temat 63. Problemy z wydajnością przy łączeniu ciągów znaków	298
Temat 64. Odwoływanie się do obiektów poprzez interfejsy	299
Temat 65. Stosowanie interfejsów zamiast refleksyjności	301
Temat 66. Rozważne wykorzystywanie metod natywnych	304
Temat 67. Unikanie przesadnej optymalizacji	306
Temat 68. Wykorzystanie ogólnie przyjętych konwencji nazewnictwa	309

Rozdział 10. Wyjątki	313
Temat 69. Wykorzystanie wyjątków w sytuacjach nadzwyczajnych	313
Temat 70. Stosowanie wyjątków przechwytywanych i wyjątków czasu wykonania	316
Temat 71. Unikanie niepotrzebnych wyjątków przechwytywanych	318
Temat 72. Wykorzystanie wyjątków standardowych	320
Temat 73. Zgłaszanie wyjątków właściwych dla abstrakcji	323
Temat 74. Dokumentowanie wyjątków zgłaszanych przez metodę	325
Temat 75. Udostępnianie danych o błędzie	326
Temat 76. Zachowanie atomowości w przypadku błędu	328
Temat 77. Nie ignoruj wyjątków	330
Rozdział 11. Współbieżność	333
Temat 78. Synchronizacja dostępu do wspólnych modyfikowalnych danych	333
Temat 79. Unikanie nadmierowej synchronizacji	338
Temat 80. Stosowanie wykonawców, zadań i strumieni zamiast wątków	344
Temat 81. Stosowanie narzędzi współbieżności zamiast wait i notify	346
Temat 82. Dokumentowanie bezpieczeństwa dla wątków	352
Temat 83. Rozsądne korzystanie z późnej inicjalizacji	355
Temat 84. Nie polegaj na harmonogramie wątków	358
Rozdział 12. Serializacja	361
Temat 85. Stosuj rozwiązania alternatywne wobec serializacji Javy	361
Temat 86. Rozsądne implementowanie interfejsu Serializable	365
Temat 87. Wykorzystanie własnej postaci serializowanej	368
Temat 88. Defensywne tworzenie metody readObject	375
Temat 89. Stosowanie typów wyliczeniowych zamiast readResolve do kontroli obiektów	381
Temat 90. Użycie pośrednika serializacji zamiast serializowanych obiektów	385
Dodatek A Tematy odpowiadające drugiemu wydaniu	389
Dodatek B Zasoby	393
Skorowidz	399

Słowo wstępne

Jeżeli kolega powie Ci: „Małżonka moja tej nocy tworzy niezwykle jedzenie w domu. Czy się dołączysz?”, prawdopodobnie pomyślisz o dwóch sprawach — po pierwsze, jesteś zaproszony na kolację, a po drugie, Twój kolega nie opanował zbyt dobrze języka polskiego.

Jeżeli kiedykolwiek uczyłeś się języka obcego i próbowałeś z niego korzystać poza zajęciami w szkole czy na kursie, wiesz, że powinieneś opanować trzy rzeczy — sposób budowania zdań (gramatyka), słowa, jakich musisz używać (słownictwo), oraz zwroty określające często wykonywane czynności. Zbyt często nauka skupia się jedynie na pierwszych dwóch, co powoduje, że gdy będziesz próbował korzystać z poznanego języka, tubylcy będą ukrywać śmiech, próbując zrozumieć, o co Ci właściwie chodzi.

Podobnie jest z językami programowania. Musisz poznać rdzeń języka — czy jest algorytmiczny, funkcyjny, czy obiektowy. Musisz znać słownictwo — jakie struktury danych, operacje i udogodnienia są oferowane przez biblioteki standardowe. Musisz również poznać często stosowane i efektywne sposoby tworzenia kodu. Książki na temat języków programowania często opisują jedynie pierwsze dwa zagadnienia. Bywa też, że praktyczne wykorzystanie języka omówione jest jedynie powierzchownie. Być może dzieje się tak dlatego, że pierwsze dwa zagadnienia są najprostsze do opisania. Gramatyka i składnia są charakterystyczne dla samego języka, natomiast sposoby jego stosowania są związane ze środowiskiem, w którym język jest wykorzystywany, a bardzo często autorzy nie korzystają z opisywanego języka na co dzień.

Język Java jest językiem obiektowym z dziedziczeniem jednobazowym, który wewnątrz każdej metody korzysta z imperatywnego (zorientowanego na instrukcje) stylu kodowania. Biblioteki obsługują operacje graficzne, sieciowe, programowanie rozproszone oraz zabezpieczenia. W jaki sposób najlepiej wykorzystać w praktyce oferowane funkcje?

Jest to kolejny problem. Programy w przeciwieństwie do zdań oraz większości książek i czasopism mają tendencję do zmieniania się z biegiem czasu. Zwykle nie wystarczy napisanie kodu, który prawidłowo działa i może być łatwo zrozumiany przez innych programistów — należy zbudować program w taki sposób, aby można było go łatwo modyfikować. Pewne zadanie T może być zrealizowane na dziesięć sposobów. Z tych dziesięciu siedem sposobów będzie niezręcznych, nieefektywnych lub skomplikowanych. Który z pozostałych trzech będzie najbardziej podobny do kodu potrzebnego do rozwiązania zadania T w przyszłorocznej wersji programu?

Istnieje kilka książek, z których możesz nauczyć się gramatyki języka Java, na przykład *The Java Programming Language*, której autorami są Arnold, Gosling i Holmes lub *The Java Language Specification*, którego autorami są Gosling, Joy, niżej podpisany oraz Bracha. Istnieje również wiele książek poświęconych bibliotekom dostarczonym wraz z językiem Java.

Książka ta jest poświęcona trzeciemu zagadnieniu — efektywnemu wykorzystaniu języka Java. Joshua Bloch od lat korzysta z tego języka oraz tworzy go w firmie Sun Microsystems. Analizował kod programów pisanych przez wiele osób, w tym również programy mojego autorstwa. W książce tej jest zawartych wiele wartościowych porad oraz metod organizowania kodu w taki sposób, aby inni ludzie mogli go łatwo zrozumieć, dzięki czemu przyszłe modyfikacje i usprawnienia będą łatwiejsze. Być może Twoje programy staną się bardziej eleganckie i poprawne.

Guy L. Steele Jr.
Burlington, Massachusetts
Kwiecień 2001

Przedmowa

Przedmowa do wydania trzeciego

W 1997 roku, gdy Java była jeszcze stosunkowo nowym językiem, James Gosling (jego główny twórca), nazwał go „językiem niebieskich kołnierzyków”, który był „dosyć prosty” [Gosling97]. Mniej więcej w tym samym czasie Bjarne Stroustrup (twórca języka C++) opisywał swoje dzieło jako „język wieloparadygmatowy”, który „celowo różni się od języków zaprojektowanych do obsługi tylko jednego stylu pisania programów” [Stroustrup95]. Stroustrup ostrzegał:

Względna prostota języka Java — podobnie jak większości nowych języków — jest po części tylko iluzją, a po części wynikiem jej niekompletności. Wraz z upływem czasu Java znacząco się rozrośnie zarówno pod względem rozmiarów, jak i złożoności. Podwoi lub nawet potroi swoje rozmiary i powstaną zależne od implementacji rozszerzenia lub biblioteki.

Dziś, czyli dwadzieścia lat później, możemy z czystym sumieniem powiedzieć, że rację miał zarówno Gosling, jak i Stroustrup. Java jest obecnie duża i złożona, z wieloma abstrakcjami dotyczącymi różnorodnych kwestii — od wykonywania równoległego, przez iteracje, po sposób reprezentacji dat i czasu.

Nadal lubię Javę, choć mój entuzjazm nieco osłabł wraz z rozwojem platformy. Z powodu zwiększonego rozmiaru i złożoności potrzeba uaktualnienia najlepszych praktyk stawała się coraz istotniejsza. W trzecim wydaniu książki staram się uzupełnić tę lukę. Mam nadzieję, że to wydanie sprosta potrzebom nowych Czytelników, a jednocześnie nie zmieni swego ducha z dwóch poprzednich wydań.

Małe jest piękne, ale proste nie jest łatwe.

San Jose, Kalifornia
Listopad 2017

PS. Byłoby niedbalstwem, gdybym nie wspomniał o tym, że obecnie znaczną część mojego czasu zajmują najlepsze praktyki związane z całą branżą. Od momentu jej powstania w latach 50. XX wieku bez przeszkód implementowaliśmy ponownie API wymyślane przez innych. Ta praktyka stała się podstawą oszałamiającego sukcesu technologii komputerowej. Walczę o zachowanie tej swobody [CompSci17] i zachęcam, byś do mnie dołączył. Aby nasza profesja pozostała zdrowa i dynamicznie się rozwijała, musimy zachować prawo do własnych implementacji API wymyślonego przez innych.

Przedmowa do wydania drugiego

Wiele zdarzyło się w platformie Java od czasu napisania pierwszego wydania tej książki w roku 2001 i był najwyższy czas na wydanie drugie. Najważniejszymi zmianami w Java 5 było dodanie typów ogólnych, wyliczeniowych, adnotacji, autoboxingu oraz pętli for-each. Drugim co do ważności dodatkiem była nowa biblioteka współbieżności, `java.util.concurrent`, również dodana w Java 5. Gilad Bracha i ja mieliśmy szczęście przewodzić zespołowi, który zaprojektował nowe funkcje języka. Miałem również przyjemność być członkiem zespołu, który zaprojektował i napisał bibliotekę współbieżności i którego szefem był Doug Lea.

Inną dużą zmianą było powszechne przyjęcie nowoczesnych środowisk zintegrowanych (IDE), takich jak Eclipse, IntelliJ IDEA oraz NetBeans, jak również statycznych narzędzi analitycznych, takich jak FindBugs. Choć nie byłem zaangażowany w ich tworzenie, bardzo wiele z nich skorzystałem i nauczyłem się, jak wpływają one na tworzenie oprogramowania.

W roku 2004 zmieniłem pracę z firmy Sun na Google, ale przez kolejne cztery lata kontynuowałem rozwój platformy Java, pomagając w tworzeniu API współbieżności i kolekcji, w czym pomogły mi biura Google oraz Java Community Process. Miałem również przyjemność korzystania z platformy Java do tworzenia bibliotek wykorzystywanych w Google. Teraz wiem, jak to jest być użytkownikiem.

Tak jak w roku 2001, gdy napisałem pierwsze wydanie, moim podstawowym celem było dzielenie się doświadczeniem z Czytelnikami, aby mogli powtarzać moje sukcesy, unikając moich błędów. Nowy materiał nadal zawiera przykłady praktycznych zastosowań z bibliotek platformy Java.

Pierwsze wydanie odniosło sukces przekraczający moje najśmielsze oczekiwania, więc zrobiłem wszystko, aby pozostać w tym samym duchu, dodając nowy materiał potrzebny do uaktualnienia książki. Nieuniknione było, że książka musi urosnąć z pięćdziesięciu siedmiu do siedemdziesięciu ośmiu tematów. Nie tylko dodałem dwadzieścia trzy tematy, ale dokładnie przejrzałem cały wcześniejszy materiał i usunąłem kilka tematów, których dni już minęły. W dodatku przedstawiłem, jak materiał z tej książki odnosi się do materiału z pierwszego wydania.

W przedmowie do pierwszego wydania napisałem, że język programowania Java i jego biblioteki są ściśle związane z jakością i wydajnością, jak również radością z programowania. Zmiany w wersjach 5. i 6. spowodowały, że dobre rzeczy stały się jeszcze lepsze. Platforma jest obecnie znacznie większa niż w roku 2001 i bardziej skomplikowana, jednak gdy nauczysz się wzorców i idiomów dotyczących nowych funkcji, Twoje programy będą lepsze, a życie łatwiejsze. Mam nadzieję, że w tym wydaniu można zauważyć mój stały entuzjazm i że książka ta pomoże w korzystaniu z nowej platformy oraz w efektywniejszym stosowaniu nowych funkcji.

San Jose, Kalifornia
Kwiecień 2008

Przedmowa do wydania pierwszego

W roku 1996 roku spakowałem walizki i przeprowadziłem się na zachód, aby pracować dla firmy JavaSoft. Przez pierwszych pięć lat pracowałem jako projektant bibliotek języka Java. Zaprojektowałem, napisałem i zmieniałem wiele bibliotek, a przy innych pomagałem jako konsultant. Opiekowanie się tymi bibliotekami w czasie rozwijania się platformy Java było niepowtarzalnym doświadczeniem. Bez przesady mogę powiedzieć, że miałem przyjemność pracować z jednymi z najlepszych programistów naszego pokolenia. Przez ten czas nauczyłem się wiele na temat Javy — jak działa i jak najlepiej wykorzystać sam język i biblioteki.

Książka ta jest próbą podzielenia się z Tobą moim doświadczeniem, dzięki czemu będziesz mógł powtórzyć moje sukcesy, nie popełniając moich błędów. Układ książki zapożyczyłem z publikacji *Effective C++* Scotta Meyersa, składającej się z pięćdziesięciu tematów. Każdy z nich opisuje jedną określoną zasadę, która ulepsza Twoje programy i projekty. Uważam, że układ ten jest niezwykle efektywny; mam nadzieję, że i Ty podzielisz to zdanie.

W wielu przypadkach pozwoliłem sobie zilustrować tematy przykładami z bibliotek platformy Java. Gdy opisuję coś, co powinno być napisane lepiej, staram się zamieścić kod, który sam napisałem, choć czasami wykorzystuję kod kolegów. Jeżeli mimo moich najlepszych chęci kogoś przez to obraziłem, serdecznie przepraszam. Negatywne przykłady są przedstawiane nie w celu obwiniania kogokolwiek, ale w celu wykorzystania doświadczenia innych programistów.

Choć książka ta nie jest adresowana jedynie do twórców komponentów przeznaczonych do powtórnego użytku, nieuchronnie jest skażona moim doświadczeniem przy pisaniu takich komponentów przez ostatnie dwadzieścia lat. W sposób naturalny patrzę na wszystkie problemy przez pryzmat udostępnianego **API** (ang. *Application Programming Interface*) i zachęcam do myślenia w ten sam sposób. Nawet jeżeli nie tworzysz tego rodzaju komponentów, przedstawione zasady poprawiają jakość pisanych programów. Ponadto często tworzymy komponenty do powtórnego wykorzystania, nie zdając sobie z tego sprawy. Jeżeli napiszesz coś użytecznego, dzielisz się tym ze swoimi współpracownikami z pokoju — szybko będziesz miał kilku użytkowników. W takim momencie nie masz już możliwości dowolnej modyfikacji API i będziesz sobie wdzięczny za dokładne jego zaprojektowanie.

Moje podejście do projektowania API może wydawać się nieco nienaturalne dla entuzjastów nowych, oszczędnych metodologii tworzenia oprogramowania, takich jak *Extreme Programming* (programowanie ekstremalne). Metodologie te skupiają się na pisaniu możliwie prostych, działających programów. Jeżeli korzystasz z tych metodologii, możesz odkryć, że skupianie się na projekcie API przydatne jest w procesie **refactoringu**. Podstawowym celem refactoringu jest poprawienie struktury systemu i unikanie duplikowania kodu. Stosowanie tych zasad jest nie-możliwe bez wykorzystania dobrze zaprojektowanego API komponentów systemu.

Żaden język nie jest doskonały, choć niektóre są rzeczywiście świetne. Uważam, że język Java i jego biblioteki ogromnie sprzyjają jakości, efektywności i przyjemności programowania. Mam nadzieję, że udało mi się zawrzeć w tej książce mój entuzjazm i pomoże Ci ona w efektywniejszym wykorzystaniu języka.

Cupertino, Kalifornia
Kwiecień 2001

Podziękowania

Podziękowania do trzeciego wydania

Dziękuję Czytelnikom dwóch pierwszych wydań tej książki za tak entuzjastyczne opinie, za podzielenie się swoimi pomysłami oraz pokazanie mi, jak pozytywny wpływ miała ona na ich pracę. Dziękuję wielu profesorom, którzy wykorzystali tę książkę w czasie zajęć, oraz wielu zespołom programistów, którzy po nią sięgnęli.

Dziękuję całemu zespołowi w Addison-Wesley i Pearson za uprzejmość, profesjonalizm, cierpliwość oraz elegancję pomimo napiętych terminów. Greg Doench pozostał niezastąpiony — to doskonały redaktor i perfekcyjny dżentelmen. Obawiam się, że w wyniku tego projektu na jego głowie pojawiły się oznaki siwizny, za co bardzo przepraszam. Kierownik produktu Julie Nahil oraz redaktorka Dana Wilson zapewniały wszystko, czego potrzebowałem — były aktywne, szybkie, zorganizowane i przyjacielskie. Moja korektorka Kim Wimpsett była drobiazgowa i miała niezwykle wyczucie smaku.

Kolejny raz zostałem obdarzony najlepszym zespołem recenzentów, jaki można sobie wyobrazić, i chciałbym podziękować każdemu z nich. W skład podstawowego zespołu, który recenzował prawie każdy rozdział, wchodziły następujące osoby: Cindy Bloch, Brian Kernighan, Kevin Bourrillion, Joe Bowbeer, William Chargin, Joe Darcy, Brian Goetz, Tim Halloran, Stuart Marks, Tim Peierls i Yoshiki Shibata. Pozostałymi recenzentami byli: Marcus Biel, Dan Bloch, Beth Bottos, Martin Buchholz, Michael Diamond, Charlie Garrod, Tom Hawtin, Doug Lea, Aleksey Shipilëv, Lou Wasserman i Peter Weinberger. Recenzenci znów przedstawili wiele sugestii, które pozwoliły na znaczne ulepszenie tej książki i uchroniły mnie przed wieloma sytuacjami, w których czułbym się zakłopotany.

Szczególnie chciałbym podziękować Williamowi Charginowi, Dougowi Lea'owi oraz Timowi Peierlowski, z którym konsultowałem wiele pomysłów wykorzystanych w książce. William, Doug oraz Tim niezwykle hojnie obdarowali mnie swoim czasem i wiedzą.

Na koniec chciałbym podziękować mojej żonie, Cindy Bloch, za zachęcanie mnie do pisania, za czytanie każdego rozdziału w surowej postaci, za przygotowanie skorowidzu, za pomoc w czasie pracy nad książką, która przecież nie jest małym projektem, a także za wsparcie w trakcie pisania.

Podziękowania do drugiego wydania

Dziękuję Czytelnikom pierwszego wydania tej książki za tak entuzjastyczne opinie, za podzielenie się swoimi pomysłami oraz pokazanie mi, jak pozytywny wpływ miała ona na ich pracę. Dziękuję wielu profesorom, którzy wykorzystali tę książkę w czasie zajęć, oraz wielu zespołom programistów, którzy po nią sięgnęli.

Dziękuję całemu zespołowi w Addison-Wesley za uprzejmość, profesjonalizm, cierpliwość oraz elegancję pomimo napiętych terminów. Na największe brawa zasługuje Greg Doench — doskonały redaktor i perfekcyjny dżentelmen. Moja kierowniczka produktu, Julie Nahil, była ucieleśnieniem wszystkiego, czym powinien być kierownik produktu — była aktywna, szybka, zorganizowana i przyjacielska. Moja korektorka, Barbara Wood, była drobiazgowa i miała niezwykle wyczucie smaku.

Kolejny raz zostałem obdarzony najlepszym zespołem recenzentów, jaki można sobie wyobrazić, i chciałbym podziękować każdemu z nich. W skład podstawowego zespołu, który recenzował każdy rozdział, wchodziły następujące osoby: Lexi Baugher, Cindy Bloch, Beth Bottos, Joe Bowbeer, Brian Goetz, Tim Halloran, Brian Kernighan, Rob Konigsberg, Tim Peierls, Bill Pugh, Yoshiki Shibata, Peter Stout, Peter Weinberger oraz Frank Yellin. Pozostałymi recenzentami byli: Pablo Bellver, Dan Bloch, Dan Bornstein, Kevin Bourrillion, Martin Buchholz, Joe Darcy, Neal Gafter, Laurence Gonsalves, Aaron Greenhouse, Barry Hayes, Peter Jones, Angelika Langer, Doug Lea, Bob Lee, Jeremy Manson, Tom May, Mike McCloskey, Andriy Tereshchenko oraz Paul Tyma. Kolejny raz recenzenci przedstawili wiele sugestii, które pozwoliły na znaczne ulepszenie tej książki i zaoszczędziły mi wiele zmartwień. Jednak muszę powtórzyć, że w pełni odpowiadam za wszystkie usterki.

Szczególnie chciałbym podziękować Dougowi Lea oraz Timowi Peierlowski, z którym konsultowałem wiele pomysłów wykorzystanych w książce. Doug oraz Tim niezwykle hojnie obdarowali mnie swoim czasem i wiedzą.

Dziękuję mojej kierowniczce w Google, Prabha Krishna, za jej stałe wsparcie i zainteresowanie.

Na koniec chciałbym podziękować mojej żonie, Cindy Bloch, za zachęcanie mnie do pisania, za czytanie każdego rozdziału w surowej postaci, za pomoc przy FrameMakerze, za sporządzenie indeksu i wspieranie mnie w czasie pracy nad książką.

Podziękowania do pierwszego wydania

Dziękuję Patrickowi Chanowi za zasugerowanie mi napisania tej książki i podrzucenie tego pomysłu Lisie Friendly, reaktorowi zarządzającemu serii; Tomowi Lindholmowi, redaktorowi merytorycznemu serii, i Mike'owi Hendricksonowi, redaktorowi prowadzącemu w wydawnictwie Addison-Wesley Professional. Dziękuję Lisie, Timowi i Mike'owi za zachęcenie mnie do zajęcia się tym projektem i za ich nieskończoną cierpliwość i wiarę, że w końcu napiszę tę książkę.

Na moją wdzięczność zasłużył również James Gosling i jego zespół — za stworzenie wspaniałego produktu, o którym mogłem napisać; dziękuję również wielu inżynierom platformy Java, którzy kontynuowali dzieło Jamesa. W szczególności dziękuję kolegom z grupy Java Platform Tools and Libraries za ich wnikliwość, poparcie i pomoc. Zespół stanowią: Andrew Bennet, Joe Darcy, Nepal Gafter, Iris Garcia, Konstantin Kladko, Ian Little, Mike McCloskey i Mark Reinhold. Wcześniej w zespole byli jeszcze Zhenghua Li, Bill Maddox i Naveen Sanjeeva.

Dziękuję mojemu kierownikowi Andrew Bennetowi i dyrektorowi Larry'emu Abrahamsowi za ich entuzjazm i wsparcie tego projektu. Dziękuję Richowi Greenowi, wiceprezesowi do spraw inżynierii w Java Software, za udostępnienie środowiska, w którym inżynierowie mogą swobodnie wymyślać i udostępniać swoje pomysły.

Błogosławieństwem był najlepszy zespół korektorów, jaki mogłem sobie wyobrazić; każdemu z nich muszę szczególnie podziękować. Byli nimi: Andrew Bennett, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Gilad Bracha, Mary Campione, Joe Darcy, Dawid Eckhardt, Joe Fiali, Lisa Friendly, James Gosling, Peter Haggard, David Holmes, Brian Kerningham, Konstantin Kladko, Dough Lea, Zhenghua Li, Tim Lindholm, Mike McCloskey, Tim Peierls, Mark Reinhold, Ken Russel, Bill Shannon, Peter Stout, Phil Wadler oraz dwie anonimowe osoby. Zaproponowali mnóstwo poprawek, które ulepszyły tę książkę i zaoszczędziły mi wielu kłopotów. Wszystkie pozostałe błędy są jedynie moją winą.

W dyskusjach, które zaowocowały poprawieniem jakości tej książki, brało udział wielu moich kolegów z firmy Sun, jak również spoza niej. Ben Gomes, Steffen Grarup, Peter Kessler, Richard Roda, Jon Rose i David Stoutamire przekazali mi wiele użytecznych uwag. Na specjalne podziękowania zasłużył Doug Lea, który sprawdzał wiele pomysłów zawartych w tej książce. Doug wspaniałomyślnie dzielił się swoim czasem i wiedzą.

Dziękuję Julii Dinicoli, Jacqui Doucette, Mike'owi Hendricksonowi, Heather Olszyk, Tracy Russ i całemu zespołowi Addison-Wesley za wsparcie i profesjonalizm. Nawet w przypadku bardzo krótkich terminów byli zawsze przyjaźni i uprzejmi.

Dziękuję Guyowi Steele'owi za napisanie słowa wstępnego. Jestem zaszczycony, że współpracował ze mną przy tym projekcie.

Na koniec dziękuję mojej żonie, Cindy Bloch, za wsparcie i pierwsze czytanie każdego tematu, za pomoc przy obsłudze FrameMakera, za sporządzenie indeksu i pomoc w czasie pisania.

Wprowadzenie

Książka ta została napisana, aby nauczyć Cię najefektywniejszego wykorzystywania języka programowania Java oraz jego podstawowych bibliotek: `java.lang`, `java.util` i `java.io`, a także istotnych podpakietów, takich jak `java.util.concurrent` i `java.util.function`. Od czasu do czasu wspomina się również o innych bibliotekach.

Książka składa się z dziewięćdziesięciu podrozdziałów, z których każdy opisuje jedną zasadę. Zasady opisują mechanizmy uznane przez większość doświadczonych programistów za przydatne. Podrozdziały te są zebrane w jedenastu rozdziałach, z których każdy opisuje jeden aspekt projektowania oprogramowania. Książka nie musi być przeczytana od deski do deski, każdy z podrozdziałów jest odrębną całością. Omawiane tematy są ze sobą mocno powiązane, więc na podstawie tej książki możesz zaprojektować własny kurs.

Od czasu opublikowania poprzedniego wydania do Javy zostało dodanych wiele nowych funkcji. Większość tematów zamieszczonych w tej książce w pewien sposób je uwzględnia. Tabela 1.1 zawiera listę tematów, w których znajduje się główny opis poszczególnych funkcji.

Tabela 1.1. Zestawienie nowych tematów książki

Funkcja	Temat	Wydanie
Lambdy	Tematy 42. – 44.	Java 8
Strumienie	Tematy 42. – 44.	Java 8
Opcjonalność	Temat 55.	Java 8
Metody domyślne w interfejsach	Temat 21.	Java 8
Konstrukcja try z zasobami	Temat 9.	Java 7
@SafeVarargs	Temat 32.	Java 7
Moduły	Temat 15.	Java 9

Większość podrozdziałów jest ilustrowanych przykładami programów. Bardzo ważną cechą tej książki jest zamieszczenie przykładów kodu, ilustrujących wiele wzorców projektowych i idiomów. Tam, gdzie jest to przydatne, umieszczone są odсылacze do literatury, np. [Gamma95].

Wielokrotnie w tekście książki i w zamieszczonych fragmentach kodu znajdują się przykłady niezalecanych technik. Przykłady takie, czasami nazywane **antywzorcami**, są oznaczone komentarzem `// NIE RÓB TAK!`. W takich przypadkach w tekście znajduje się objaśnienie, dlaczego technika ta nie jest zalecana, oraz sugestia alternatywnego podejścia.

Książka ta nie jest przeznaczona dla początkujących. Zakładamy, że znasz już dosyć dobrze język Java. Jeżeli nie, przeczytaj jedną z wielu świetnych książek wprowadzających, np. *Java Precisely* [Sestoft16] autorstwa Petera Sestofta. Ponieważ książka jest zaprojektowana dla wszystkich, którzy znają język Java, powinna być przydatna nawet dla zaawansowanych programistów.

Większość zasad opisanych w tej książce oparta jest na kilku fundamentalnych regułach. Najważniejsza jest jasność i prostota rozwiązania. Użytkownik komponentu nigdy nie powinien być zaskakiwany jego działaniem. Komponenty powinny być możliwie małe, ale nie zbyt małe (w książce tej **komponentem** nazywamy dowolny fragment oprogramowania, który może być ponownie wykorzystany — od pojedynczej metody do skomplikowanego systemu, składającego się z wielu pakietów). Kod powinien być ponownie wykorzystywany, a nie kopiowany. Poszczególne komponenty powinny być od siebie możliwie niezależne. Błędy powinny być wykrywane na możliwie najwcześniejszym etapie; idealnym przypadkiem jest ich wykrywanie podczas kompilacji.

Stosowanie tych zasad nie zabiera 100% czasu poświęconego na programowanie. W większości przypadków reguły te stosowane są w najlepszych technikach programowania. Nie powinieneś jednak ślepo się ich trzymać, ponieważ czasami ich naruszenie będzie najlepszym rozwiązaniem. Nauczenie się sztuki programowania, podobnie jak większości innych dyscyplin, składa się przede wszystkim z poznania zasad, a następnie nauczania się sposobów ich łamania.

W książce tej nie poświęciliśmy zbyt dużo miejsca wydajności. Traktuje ona o pisaniu jasnych, prawidłowych, elastycznych i łatwych w konserwacji programów. Jeżeli będziesz tworzył takie programy, zwykle łatwo osiągniesz wymaganą wydajność (temat 67.). Przy okazji poruszania niektórych tematów omawiane były zagadnienia wydajności, w kilku z nich zamieszczone zostały wyniki szybkości działania. Wyniki te, rozpoczynające się od zdania „Na moim komputerze”, powinny być traktowane najwyżej jako przybliżenie.

Komputerem tym jest starzejący się, domowej roboty czterordzeniowy Intel Core i7-4770K z zegarem 2,6 GHz, 16 GB RAM typu DDR3-1866 CL9 i systemem

Windows 7 Professional SP1 (wersja 64-bitowa), na którym działa OpenJDK w wydaniu 9.0.0.15 (wydanie Zulu oferowane przez Azul).

Omawiając własności języka programowania Java oraz dostarczanych z nim bibliotek, czasami odwołujemy się do konkretnych wersji. Do identyfikacji tych wersji korzystamy w tej książce z „obiegowego numeru wersji”, a nie z oficjalnej nazwy wersji. W tabeli 1.2 zamieściliśmy oficjalne nazwy kolejnych wersji środowiska i odpowiadające im nazwy obiegowe.

Tabela 1.2. *Wersje platform Java*

Oficjalna nazwa wersji	Obiegowa nazwa wersji
JDK 1.0.x	Java 1.0
JDK 1.1.x	Java 1.1
Java 2 Platform, Standard Edition, v1.2	Java 2
Java 2 Platform, Standard Edition, v1.3	Java 3
Java 2 Platform, Standard Edition, v1.4	Java 4
Java 2 Platform, Standard Edition, v5.0	Java 5
Java Platform, Standard Edition 6	Java 6
Java Platform, Standard Edition 7	Java 7
Java Platform, Standard Edition 8	Java 8
Java Platform, Standard Edition 9	Java 9

Przykłady są zwykle kompletne, ale nad ich kompletność przedkładamy czytelność. Są w nich wykorzystane klasy zawarte w pakietach `java.util` i `java.io`. Aby można było skompilować te przykłady, być może konieczne będzie dodanie jednej lub kilku deklaracji importu lub innego dodatkowego kodu. Pod adresem <ftp://ftp.helion.pl/przyklady/javap3.zip> dostępne są również kompletne i rozszerzone wersje zamieszczonych w książce przykładów. Po pobraniu można je od razu skompilować i uruchomić.

W książce tej wykorzystujemy terminy techniczne zgodnie z ich definicjami zawartymi w *The Java Language Specification, Java SE 8 Edition* [JLS]. Kilka terminów wymaga jednak specjalnego wyjaśnienia. Język pozwala na stosowanie czterech rodzajów typów danych: **interfejsów** (włącznie z **adnotacjami**), **klas** (włącznie z **wyliczeniami**), **tablic** i **wartości prostych**. Pierwsze trzy są również znane pod nazwą **typy referencyjne**. Instancje klas i tablic są **obiektami**, wartości proste nimi nie są. **Składniki** klasy to **pola**, **metody**, **klasy składowe** i **interfejsy składowe**. **Sygnatura** metody to jej nazwa i typy parametrów formalnych — sygnatura **nie** zawiera zwracanego typu.

Kilka terminów jest w tej książce wykorzystywanych nieco inaczej niż w *The Java Language Specification*. W przeciwieństwie do *The Java Language Specification* w książce tej używamy zamiennie terminu **dziedziczenie** i tworzenie **klas podrzędnych**. Zamiast wykorzystywania terminu dziedziczenie dla interfejsów w książce tej piszemy, że klasa **implementuje** interfejs lub jeden interfejs **rozszerza** inny. Aby opisać poziom dostępu stosowany w przypadku braku jego definicji przez programistę, korzystamy z opisowego terminu **prywatny w ramach pakietu**, zamiast technicznie poprawnego **dostęp domyślny** [JLS 6.6.1].

Korzystamy również z kilku terminów, które nie zostały wspomniane w specyfikacji języka Java. Termin **eksportowane** API lub po prostu API określa klasy, interfejsy, konstruktory i składniki, za pomocą których programista może korzystać z klas, interfejsów lub pakietów (termin API jest skrótem od *application programming interface*; jest on wykorzystywany zamiast słowa **interface**, aby uniknąć pomyłek z konstrukcją języka o tej samej nazwie). Programista, piszący aplikację wykorzystującą API, jest określany jako **użytkownik** API. Klasa wykorzystująca API jest nazywana **klientem** API.

Klasy, interfejsy, konstruktory i składniki są nazywane **elementami** API. Eksportowane API składa się z elementów API dostępnych spoza pakietu, w którym API to jest zdefiniowane. Są to elementy API, do których mają dostęp klienci i co do których autor deklaruje oficjalne wsparcie. Nieprzypadkowo są to elementy, dla których program Javadoc, korzystający z domyślnych ustawień, generuje dokumentację. Można powiedzieć, że eksportowane API pakietu składa się z publicznych i zabezpieczonych składników oraz konstruktorów każdej publicznej klasy i interfejsu znajdujących się w pakiecie.

W Javie 9 do platformy dołączył **system modułów**. Jeśli biblioteka korzysta z systemu modułów, jej eksportowane API stanowi połączenie eksportowanego API wszystkich pakietów wskazanych jako eksportowane w deklaracji modułu biblioteki.

Tworzenie i usuwanie obiektów

W rozdziale tym skupimy się na tworzeniu i usuwaniu obiektów — kiedy i w jaki sposób należy tworzyć obiekty, kiedy i w jaki sposób należy unikać ich tworzenia, jak można zapewnić ich usunięcie w odpowiednim momencie oraz w jaki sposób należy wykonywać dodatkowe czynności, które poprzedzają usunięcie obiektu.

Temat 1. Tworzenie statycznych metod fabrycznych zamiast konstruktorów

Najczęściej spotykanym sposobem na udostępnienie obiektu danej klasy jest utworzenie publicznego konstruktora. Istnieje również inna, mniej znana technika, która powinna być szerzej stosowana przez programistów. Klasa może zawierać publiczną **statyczną metodę fabryczną**, która jest po prostu zwykłą metodą statyczną, zwracającą obiekt tej klasy. Poniżej przedstawiamy prosty przykład dla klasy `Boolean` (jest to **klasa osłona** dla typu prostego `boolean`). Jest to statyczna metoda fabryczna, która zamienia prostą wartość typu `boolean` na obiekt typu `Boolean`:

```
public static Boolean valueOf(boolean b) {  
    return (b ? Boolean.TRUE : Boolean.FALSE);  
}
```

Trzeba zauważyć, że statyczne metody fabryczne nie są tym samym co wzorzec **metody fabrycznej** z książki *Design Patterns* [Gamma95]. Statyczne metody fabryczne nie mają bezpośredniego odpowiednika w *Design Patterns*.

Klasa może oferować swoim klientom statyczne metody fabryczne zarówno zamiast konstruktorów, jak i obok nich. Zastosowanie statycznych metod fabrycznych zamiast publicznych konstruktorów ma wady i zalety.

Pierwszą zaletą statycznych metod fabrycznych jest posiadanie przez nie nazw — w przeciwieństwie do konstruktorów. Jeżeli parametry przekazywane do konstruktora nie spełniają żadnej innej roli poza określeniem rodzaju zwracanego obiektu, zastosowanie statycznej metody fabrycznej o odpowiedniej nazwie ułatwi używanie klasy i spowoduje, że kod, w którym zostanie ona zastosowana, będzie łatwiejszy do analizy. Na przykład zamiast konstruktora `BigInteger(int, int, Random)`, który zwraca obiekt typu `BigInteger` reprezentujący liczbę, która prawdopodobnie jest liczbą pierwszą, być może lepiej zastosować statyczną metodą fabryczną o nazwie `BigInteger.probablePrime` (ta statyczna metoda fabryczna została ostatecznie dodana w Javie 4).

Klasa może posiadać tylko jeden konstruktor o odpowiedniej sygnaturze. Programiści obchodzą to ograniczenie poprzez tworzenie dodatkowego konstruktora, którego lista parametrów różni się jedynie kolejnością typów parametrów. Nie jest to dobra technika. Użytkownicy tej klasy prawdopodobnie nie będą pamiętali, który konstruktor służy do czego, i prawdopodobnie użyją tego niewłaściwego. Czytający kod, który wykorzystuje te konstruktory, nie zrozumieją go bez sięgania do dokumentacji klasy.

Ponieważ statyczne metody fabryczne posiadają nazwy, nie występuje tu niedogodność spotykana w przypadku konstruktorów i określona sygnatura typów może się powtarzać. W przypadkach, gdy klasa wymaga zastosowania kilku konstruktorów o tej samej sygnaturze typów, powinieneś pomyśleć o zastąpieniu części konstruktorów statycznymi metodami fabrycznymi, których nazwy będą jasno przedstawiały różnice w działaniu.

Drugą zaletą statycznych metod fabrycznych jest to, że — w przeciwieństwie do konstruktorów — nie jest wymagane utworzenie nowego obiektu podczas ich wywołania. Pozwala to tworzyć klasy niezmiennie (temat 17.), korzystające z wstępnie tworzonych obiektów lub tak zarządzające procesem tworzenia obiektów, aby unikać tworzenia niepotrzebnie dublujących się obiektów. Metoda `Boolean.valueOf(boolean)` unika tworzenia niepotrzebnie powielających się obiektów — **nigdy** nie tworzy nowego obiektu. Technika ta jest zbliżona do wzorca *Flyweight* [Gamma95]. Jeżeli obiekty są często tworzone, dzięki tej technice można znacznie poprawić wydajność aplikacji, szczególnie, jeżeli tworzenie tych obiektów wymaga dużych nakładów czasowych.

Zdolność statycznych metod fabrycznych do zwracania tego samego obiektu w czasie kolejnych wywołań może być również wykorzystana do ścisłej kontroli nad istniejącymi w danej chwili obiektami. Klasy, które tak robią, nazywa się **kontrolowanymi przez instancje**. Istnieją dwa powody stosowania takiej kontroli. Po pierwsze,

pozwala to na zagwarantowanie, że klasa jest klasą singleton (temat 3.) lub nie można jej konkretyzować (temat 4.). Po drugie, pozwala klasie niezmiennej (temat 17.) upewnić się, że nie istnieją dwa identyczne obiekty: `a.equals(b)` jest spełnione wtedy i tylko wtedy, gdy `a==b`. To podstawa wzorca *Flyweight* [Gamma95]. Optymalizację taką stosują konstrukcje enum (temat 34.).

Trzecią zaletą statycznych metod fabrycznych jest możliwość zwracania typu, który jest podtypem zdefiniowanego, zwracanego typu. Pozwala to na dowolne wybieranie typu zwracanego obiektu.

Jednym z zastosowań tej techniki jest możliwość zwracania obiektów, których klasy nie są klasami publicznymi. Ukrycie implementacji klas pozwala na tworzenie niewielkich API. Technika ta jest stosowana w **bibliotekach korzystających z interfejsów** (temat 20.), w których statyczne metody fabryczne korzystają z typów zdefiniowanych w interfejsie.

Przed Javą 8 interfejsy nie mogły zawierać metod statycznych, więc powstała konwencja, zgodnie z którą statyczne metody fabryczne dla interfejsu o nazwie `Type` były umieszczane w **klasie nietworzącej obiektów** (temat 4.) o nazwie `Types`. Na przykład biblioteka `Collections` posiada czterdzieści pięć wygodnych implementacji interfejsów kolekcji, pozwalających na stosowanie kolekcji niemodyfikowalnych, kolekcji synchronizowanych itp. Większość z tych implementacji jest udostępniana poprzez statyczne metody fabryczne, zawarte w jednej nieinstancyjnej klasie (`java.util.Collections`). Obiekty zwracane przez tę klasę są obiektami klas niepublicznych.

API biblioteki `Collections` jest dużo mniejsze, niż mogłoby być, gdyby dla wygody implementacji zostało wyeksportowanych wszystkich 45 klas publicznych, po jednej na każdą implementację. Nie została jednak zmniejszona „masa” biblioteki, a raczej jej „masa koncepcyjna”. Użytkownicy wiedzą, że zwracane obiekty są zgodne z odpowiednim interfejsem, więc nie jest potrzebna dodatkowa dokumentacja klasy. Ponadto użycie statycznych metod fabrycznych wymusza na użytkowniku odwoływanie się do zwracanych obiektów poprzez interfejs, a nie przez klasę implementującą, co jest zalecaną praktyką (temat 64.).

Od Javy 8 zniesiono ograniczenie, że interfejs nie może zawierać metod statycznych, więc obecnie nie ma już zbyt wielu przekonujących powodów, aby dołączać do interfejsu dodatkową klasę, której obiektów nie można tworzyć. Wiele publicznych składowych statycznych, które dawniej znalazłyby się w takiej klasie, obecnie można umieścić w samym interfejsie. Pamiętajmy jednak, że nadal cała „masa” biblioteki kryjąca się za kodem w metodach statycznych musi znaleźć się w osobnych klasach prywatnych dla pakietu. Wynika to z faktu, że Java 8 wymaga, aby wszystkie składowe statyczne interfejsu były publiczne. Java 9 rozluźniła wymogi i dopuszcza prywatne metody statyczne, ale pola statyczne i klasy składowe nadal muszą być publiczne.

Czwartą zaletą statycznych metod fabrycznych jest to, że w zależności od tego, jakie wartości parametrów są przekazywane do metody, ulegać zmianie może klasa zwracanego obiektu. Można zwracać obiekty dowolnej klasy pochodnej do klasy zadeklarowanej jako zwracana. Klasa zwracanego obiektu może również różnić się od jej odpowiednika w kolejnych wersjach pakietu.

Klasa EnumSet (temat 36.) nie ma publicznych konstruktorów, lecz jedynie statyczne metody fabryczne. W implementacji OpenJDK zwracają one dwie implementacje, zależne od rozmiaru bazowego typu wyliczeniowego — jeżeli ma on co najwyżej sześćdziesiąt cztery elementy, czyli większość stosowanych typów wyliczeniowych, statyczna metoda fabryczna zwraca obiekt RegularEnumSet, który bazuje na jednej liczbie typu long; jeżeli typ wyliczeniowy zawiera ponad sześćdziesiąt cztery elementy, metoda fabryczna zwraca obiekt JumboEnumSet, bazujący na tablicy liczb long.

Istnienie tych dwóch klas implementacji jest niewidoczne dla klienta. Jeżeli klasa RegularEnumSet nie dawałaby zysku wydajnościowego dla małych typów wyliczeniowych, mogłaby być wyeliminowana w kolejnych wersjach bez żadnych efektów ubocznych. Podobnie w kolejnych wersjach może pojawić się trzecia lub czwarta implementacja EnumSet, jeżeli poprawiłoby to wydajność aplikacji. Klienci nie muszą się zajmować rodzajem klasy otrzymanej z metody fabrycznej — wystarczy, że „wiedzą”, że jest klasą dziedziczącą po EnumSet.

Piątą zaletą statycznych metod fabrycznych jest to, że klasa obiektu zwracanego przez statyczną metodę fabryczną może jeszcze nie istnieć, kiedy pisana jest klasa zawierająca metodę statyczną. Tak duża elastyczność statycznych metod fabrycznych jest wykorzystywana w **bibliotekach dostawcy usług**, jak na przykład *Java Database Connectivity API* (JDBC). Biblioteka dostawcy usług jest systemem, w którym dostawcy tworzą implementacje interfejsu API, dostępnego dla użytkowników biblioteki, izolując ich od implementacji.

Biblioteka dostawcy usług zawiera trzy podstawowe komponenty — **interfejs usługi**, implementowany przez dostawcę; **API rejestracji dostawcy**, będące systemem pozwalającym na rejestrowanie implementacji w systemie, co daje klientom dostęp do tych implementacji; oraz **API dostępu do usług**, które klienci wykorzystują do uzyskania instancji usługi. API dostępu do usług zwykle pozwala, ale nie wymaga od klienta, aby zdefiniował pewne kryteria wyboru dostawcy. W przypadku braku takiej specyfikacji API zwraca instancję domyślnej implementacji. API dostępu do usług jest „elastyczną statyczną metodą fabryczną”, która tworzy podstawę biblioteki dostawcy usług.

Opcjonalnym, czwartym składnikiem biblioteki dostawcy usług jest **interfejs dostawcy usług**, który jest implementowany przez dostawcę w celu utworzenia instancji w implementacji usługi. W przypadku braku interfejsu dostawcy usługi implementacje są rejestrowane z użyciem nazwy klasy i tworzone z wykorzystaniem

refleksji (temat 65.). W przypadku JDBC Connection stanowi część interfejsu usługi, DriverManager.registerDriver jest API rejestracji dostawcy, DriverManager.getConnection jest API dostępu do usługi, natomiast Driver jest interfejsem dostawcy usługi.

Istnieje wiele odmian wzorca biblioteki dostawcy usługi. Na przykład API dostawcy usługi może zwracać bogatszy interfejs niż ten wymagany przez dostawcę, korzystając w tym celu z wzorca *Bridge* [Gamma95]. Systemy wstrzykiwania zależności (temat 5.) można traktować jako rozbudowanych dostawców usług. Od Javy 6 platforma zawiera własną bibliotekę dostawcy usług do ogólnego stosowania, java.util.ServiceLoader, więc nie trzeba, a często nawet nie należy pisać własnej implementacji (temat 59.).

Najważniejszą wadą stosowania statycznych metod fabrycznych jest to, że klasy nieposiadające publicznego lub zabezpieczonego konstruktora nie mogą być dziedziczone. Nie można na przykład odziedziczyć po żadnej z klas implementacji w bibliotece Collections. Powoduje to zachęcanie programistów do korzystania z kompozycji zamiast z dziedziczenia (temat 18.), a także jest wymagane w celu użycia typów niezmiennych (temat 17.).

Drugą wadą stosowania statycznych metod fabrycznych jest to, że nie można ich od razu odróżnić od innych metod statycznych. Nie są one umieszczone w dokumentacji razem z konstruktorami. Dlatego może być trudno zorientować się na podstawie dokumentacji klasy, w jaki sposób utworzyć obiekt klasy, korzystając ze statycznej metody fabrycznej zamiast z konstruktora. Być może w przyszłości narzędzie Javadoc pozwoli na wskazanie takich metod w dokumentacji jako szczególnej odmiany. Niedogodność ta może być zmniejszona dzięki stosowaniu standardowych konwencji nazewnictwa. Konwencje te ciągle się rozwijają, więc poniższa lista to zbiór nazw statycznych metod fabrycznych, które stały się już powszechne.

- **from** — **metoda konwersji typu**, która przyjmuje jeden parametr i zwraca obiekt, który posiada tę samą wartość co parametr:
- **of** — **metoda agregująca**, która przyjmuje wiele parametrów i zwraca instancję odpowiedniego typu zawierającą wszystkie parametry:

```
Date d = Date.from(instant);
```

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf** — nieco bardziej rozwlekła alternatywa dla from i of, np.:
- **instance** lub **instanceOf** — zwraca obiekt opisany przez parametr (jeśli istnieje), ale nie można powiedzieć, że ma tę samą wartość:

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** lub **newInstance** — podobna do `instance` lub `getInstance`, poza tym że gwarantuje, iż każda zwracana instancja jest inna od pozostałych:

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **getType** — podobna do `getInstance`, ale używana, gdy metoda fabryczna znajduje się w osobnej klasie. *Type* wskazuje na typ obiektu zwracanego przez metodę fabryczną:

```
FileStore fs = Files.getFileStore(path);
```

- **newType** — podobna do `newInstance`, ale używana, gdy metoda fabryczna znajduje się w osobnej klasie. *Type* wskazuje na typ obiektu zwracanego przez metodę fabryczną:

```
BufferedReader br = Files.newBufferedReader(path);
```

- **type** — zwięzła alternatywa dla `getType` lub `newType`:

```
List<Complaint> litany = Collections.list(legacyLitany);
```

Podsumujmy — statyczne metody fabryczne oraz publiczne konstruktory mają swoje zastosowania i warto poznać ich zalety. Zastosowanie statycznych metod fabrycznych jest często zalecane, więc należy unikać pokusy stosowania refleksji w konstruktorach publicznych przed wcześniejszym rozważeniem zastosowania statycznych metod fabrycznych.

Temat 2. Zastosowanie budowniczego do obsługi wielu parametrów konstruktora

Styczne metody fabryczne oraz konstruktory mają wspólne ograniczenie — słabo obsługują dużą liczbę parametrów opcjonalnych. Rozważmy przypadek klasy reprezentującej etykiety z informacją o składnikach odżywczych, które znajdują się na opakowaniach żywności. Etykieta ta zawiera niewiele pól wymaganych — wielkość porcji, ilość porcji na opakowanie oraz liczbę kalorii na porcję — jak również ponad dwadzieścia pól opcjonalnych — tłuszcze razem, tłuszcze nasycone, tłuszcze trans, cholesterol, sód i tak dalej. Większość produktów ma wartości niezerowe dla tylko kilku pól opcjonalnych.

Jakiego rodzaju konstruktor lub statyczną metodę fabryczną należy napisać dla takiej klasy? Zazwyczaj programiści wykorzystują wzorzec **konstruktora teleskopowego**, w którym tworzy się konstruktor z parametrami wymaganymi, kolejny z dodatkowo jednym parametrem opcjonalnym, trzeci z dwoma parametrami

opcjonalnymi i tak dalej, kończąc na konstruktorze z wszystkimi parametrami opcjonalnymi. Jak to działa w praktyce? Dla uproszczenia użyjemy tylko czterech pól opcjonalnych:

// Wzorzec konstruktora teleskopowego - niezbyt dobrze się skaluje!

```
public class NutritionFacts {
    private final int servingSize; // (mL) wymagane
    private final int servings; // (na opakowanie) wymagane
    private final int calories; // opcjonalne
    private final int fat; // (g) opcjonalne
    private final int sodium; // (mg) opcjonalne
    private final int carbohydrate; // (g) opcjonalne

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium,
                          int carbohydrate) {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

Gdy chcemy utworzyć obiekt, możemy skorzystać z konstruktora o najkrótszej liście parametrów, zawierającej wszystkie używane parametry:

```
NutritionFacts cocaCola =
    new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Zwykle wywołanie takiego konstruktora będzie wymagało podania wielu parametrów, których nie chcemy podawać, ale jesteśmy zmuszeni i tak je podać. W tym przypadku przekazaliśmy wartość 0 dla parametru *fat*. Przy „tylko” sześciu parametrach może to nie wyglądać źle, ale szybko wymknie się spod kontroli wraz ze wzrostem liczby parametrów.

W skrócie — **wzorzec konstruktora teleskopowego działa, ale trudno jest pisać kod klienta w przypadku korzystania z wielu parametrów, a dodatkowo kod jest nieczytelny**. Aby zorientować się, co znaczą te parametry, Czytelnik musi uważnie je policzyć. Długie sekwencje parametrów tego samego typu mogą powodować powstanie subtelnych błędów. Jeżeli klient przypadkowo zamieni kolejność dwóch parametrów, kompilator tego nie wykryje, ale program będzie nieprawidłowo działał (temat 51.).

Drugim sposobem radzenia sobie z wieloma parametrami konstruktora jest wzorzec **JavaBeans**, w którym wykorzystuje się konstruktor bezparametrowy do tworzenia obiektu, a następnie wywołuje się metody ustawiające wartości każdego z wymaganych i opcjonalnych parametrów:

```
// Wzorzec JavaBeans - pozwala na niespójność, wymusza niezmienność
public class NutritionFacts {
    // parametry inicjowane wartościami domyślnymi (jeżeli istnieją)
    private int servingSize = -1; // wymagane, brak wartości domyślnej
    private int servings = -1; // wymagane, brak wartości domyślnej
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }

    // metody ustawiające
    public void setServingSize(int val) { servingSize = val; }
    public void setServings(int val) { servings = val; }
    public void setCalories(int val) { calories = val; }
    public void setFat(int val) { fat = val; }
    public void setSodium(int val) { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

Wzorzec ten nie ma wad wzorca konstruktora teleskopowego. Jest prosty, tworzenie instancji jest nieco zbyt rozwlekłe, ale wynikowy kod jest czytelny:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Niestety, wzorzec JavaBeans ma inne poważne wady. Ponieważ tworzenie obiektu jest podzielone na wiele wywołań, **w czasie tworzenia obiektu z wykorzystaniem JavaBeans obiekt znajduje się przejściowo w stanie niespójnym**. Klasa nie ma możliwości wymuszania spójności przez sprawdzenie poprawności parametrów konstruktora. Próba użycia obiektu w stanie niespójnym może powodować awarie, które ujawniają się w kodzie odległym od kodu powodującego błąd, przez co są trudne do debugowania. Związaną z tym niedogodnością jest to, że **wzorzec JavaBeans uniemożliwia wykonanie klasy niezmiennej** (temat 17.) i wymaga dodatkowego nakładu pracy na zapewnienie bezpieczeństwa dla wątków.

Możliwe jest ograniczenie tych wad przez ręczne „zamrożenie” obiektu po zakończeniu tworzenia go i uniemożliwienie korzystania z niego w stanie zamrożonym, ale taka metoda jest niewygodna i w praktyce rzadko wykorzystywana. Co więcej, może powodować powstanie błędów wykonania, ponieważ kompilator nie może zapewnić, że programista wywoła metodę zamrażającą obiekt przed jego użyciem.

Na szczęście istnieje trzecia alternatywa, łącząca bezpieczeństwo wzorca konstruktora teleskopowego z czytelnością wzorca JavaBeans. Jest to odmiana wzorca *Builder* [Gamma95]. Zamiast bezpośrednio generować wymagany obiekt, klient wywołuje konstruktor (lub statyczną metodę fabryczną) z wszystkimi wymaganymi parametrami i otrzymuje **obiekt budowniczego**. Następnie klient wywołuje metody podobne do ustawiających wartości, aby ustawić każdy z wymaganych parametrów opcjonalnych. Na koniec klient wywołuje bezparametrową metodę *build* w celu wygenerowania obiektu, który jest niezmienny. Budowniczy jest statycznym składnikiem klasy (temat 24.) zwracającym budowaną klasę. W praktyce działa to następująco:

// Wzorzec Builder

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        //wymagane parametry
        private final int servingSize;
        private final int servings;

        //parametry opcjonalne - inicjalizacja wartościami domyślnymi
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;
```

```
public Builder(int servingSize, int servings) {
    this.servingSize = servingSize;
    this.servings = servings;
}

public Builder calories(int val)
    { calories = val; return this; }
public Builder fat(int val)
    { fat = val; return this; }
public Builder carbohydrate(int val)
    { carbohydrate = val; return this; }
public Builder sodium(int val)
    { sodium = val; return this; }

public NutritionFacts build() {
    return new NutritionFacts(this);
}

private NutritionFacts(Builder builder) {
    servingSize = builder.servingSize;
    servings = builder.servings;
    calories = builder.calories;
    fat = builder.fat;
    sodium = builder.sodium;
    carbohydrate = builder.carbohydrate;
}
}
```

Należy zwrócić uwagę, że klasa `NutritionFacts` jest niezmienna i wszystkie wartości domyślne parametrów znajdują się w jednej lokalizacji. Metody ustawiające budowniczego zwracają samego budowniczego, dzięki czemu wywołania mogą być łączone w łańcuch, tworząc bardzo **przyjazne** API. Kod kliencki wygląda następująco:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
    .calories(100).sodium(35).carbohydrate(27).build();
```

Taki kod klienta jest łatwy do napisania, a co ważniejsze, do czytania. **Wzorzec Builder symuluje nazwane parametry opcjonalne** dostępne w językach Python i Scala.

Aby nie zaciemniać przykładu, zrezygnowaliśmy z testów poprawności parametrów. Aby wykrywać nieodpowiednie wartości, sprawdzaj parametry w konstruktorze i metodach budowniczego. Podobnie jak w przypadku konstruktora, metoda `build` może kontrolować niezmienniki dotyczące kilku parametrów. Niezwykle ważne jest, aby były one kontrolowane po skopiowaniu parametrów z budowniczego do obiektu, dzięki czemu mogą być kontrolowane na polach obiektu, a nie polach budowniczego (temat 50.). Jeżeli jakikolwiek niezmiennik lub parametr nie zostanie spełniony, kod powinien zgłosić wyjątek `IllegalStateException` (temat 72.). Metody szczegółów wyjątku powinny wskazywać niezmiennik, który nie został zachowany (temat 75.).

Wzorzec Builder dobrze dopasowuje się do hierarchii klas. Wystarczy użyć równoległej hierarchii budowniczych, gdzie każdy jest zagnieżdżony w odpowiedniej klasie. Klasy abstrakcyjne mają abstrakcyjnych budowniczych, a klasy konkretne konkretnych budowniczych. Rozważmy na przykład klasę abstrakcyjną stanowiącą początek hierarchii związanej z różnymi rodzajami pizzy:

```
// Wzorzec Builder dla hierarchii klas
public abstract class Pizza {
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping) {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }
    }

    abstract Pizza build();

    // podklasy muszą przesłonić tę metodę, aby zwrócić "this"
    protected abstract T self();
}

Pizza(Builder<?> builder) {
    toppings = builder.toppings.clone(); // patrz temat 50.
}
}
```

Zauważ, że `Pizza.Builder` jest **typem uogólnionym z rekurencyjnym parametrem typu** (temat 30.). Rozwiązanie to, w połączeniu z abstrakcyjną metodą `self`, umożliwia tworzenie łańcuchów metod działających poprawnie także w podklasach bez potrzeby dokonywania rzutowania. To obejście ograniczenia języka Java, któremu brakuje typu „własnego”, nazywamy **symulowaniem typu „własnego”**.

Oto dwie konkretne podklasy klasy `Pizza` — pierwsza reprezentuje standardową pizzę w stylu nowojorskim, a druga w stylu calzone. Pierwsza wymaga podania rozmiaru pizzy, a druga pozwala określić, czy sos ma się znajdować wewnątrz, czy na zewnątrz.

```
public class NyPizza extends Pizza {
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> {
        private final Size size;

        public Builder(Size size) {
            this.size = Objects.requireNonNull(size);
        }
    }
}
```

```

        @Override public NyPizza build() {
            return new NyPizza(this);
        }

        @Override protected Builder self() { return this; }
    }

    private NyPizza(Builder builder) {
        super(builder);
        size = builder.size;
    }
}

public class Calzone extends Pizza {
    private final boolean sauceInside;

    public static class Builder extends Pizza.Builder<Builder> {
        private boolean sauceInside = false; // domyślnie

        public Builder sauceInside() {
            sauceInside = true;
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }

        @Override protected Builder self() { return this; }
    }

    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}

```

Zwróć uwagę, że metoda `build` w każdym budowniczym podklasie została zadeklarowana tak, że zwraca właściwą podklasę — metoda `build` z `NyPizza.Builder` zwraca `NyPizza`, a metoda `build` z `Calzone.Builder` zwraca `Calzone`. Przedstawiona technika, w której to metoda podklasy została zadeklarowana tak, że zwraca podtyp typu zadeklarowanego do zwracania w klasie nadrzędnej, nosi nazwę kowariancji zwracanych typów (ang. *covariant return typing*). Dzięki temu klient może korzystać z budowniczego bez dodatkowego rzutowania.

Kod kliencki dla tych „hierarchicznych budowniczych” jest w zasadzie taki sam jak dla prostego budowniczego klasy `NutritionFacts`. Przedstawiony poniżej kod kliencki zakłada użycie importu statycznego dla stałych enum w celu poprawy czytelności:

```
NyPizza pizza = new NyPizza.Builder(SMALL)
    .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
    .addTopping(HAM).sauceInside().build();
```

Niewielką przewagą budowniczego nad konstruktorami jest możliwość korzystania z wielu parametrów typu `varargs`. Ponieważ budowniczy korzysta z osobnych metod do ustawienia każdego z parametrów, można użyć dowolnej liczby parametrów `varargs`, po jednym na każdą metodę ustawiającą. Z drugiej strony budowniczy może agregować parametry przekazywane w wielu wywołaniach metod do jednego pola, co ilustruje metoda `addTopping` z wcześniejszego przykładu.

Wzorzec *Builder* jest elastyczny. Jeden obiekt budowniczego może być wykorzystywany do budowy wielu obiektów. Parametry budowniczego mogą być modyfikowane pomiędzy operacjami tworzenia obiektów, dzięki czemu mogą mieć one różną charakterystykę. Budowniczy może automatycznie wypełniać niektóre pola, takie jak numer seryjny, automatycznie zwiększający się przy każdym utworzeniu obiektu.

Wzorzec *Builder* ma oczywiście wady. Aby utworzyć obiekt, należy wcześniej utworzyć budowniczego. Choć koszt tworzenia budowniczego w praktyce będzie prawdopodobnie niezauważalny, to jednak może stanowić problem w niektórych przypadkach krytycznych wydajnościowo. Dodatkowo wzorzec *Builder* jest bardziej obszerny niż wzorzec konstruktora teleskopowego, więc powinien być używany tylko w przypadkach, gdy operujemy na większej liczbie parametrów, załóżmy, czterech lub więcej. Należy jednak pamiętać, że w przyszłości może zająć konieczność dodania większej liczby parametrów. Jeżeli zaczniemy od konstruktora lub statycznej metody fabrycznej i dodamy budowniczego, gdy klasa rozrośnie się do punktu, w którym liczba parametrów przestanie dać się wygodnie obsługiwać, to niepotrzebne konstruktory lub statyczne metody fabryczne będą przeszkadzać. Z tego powodu lepiej jest rozpoczynać od razu od wzorca *Builder*.

Podsumujmy. **Wzorzec *Builder* jest dobrym wyborem przy projektowaniu klas, których konstruktory lub statyczne metody fabryczne mają więcej niż kilka parametrów, szczególnie gdy większość z tych parametrów jest opcjonalna.** Kod klienta jest znacznie łatwiejszy do czytania i pisania przy użyciu budowniczego niż przy tradycyjnym wzorcu konstruktora teleskopowego, a dodatkowo budowniczy jest znacznie bezpieczniejszy niż `JavaBeans`.

Temat 3. Wymuszanie właściwości singleton za pomocą prywatnego konstruktora lub typu enum

Singleton to klasa posiadająca tylko jedną instancję [Gamma95]. Klasy typu singleton najczęściej reprezentują obiekty bezstanowe, takie jak funkcja (temat 24.), lub składniki systemu, które ze swojej natury występują pojedynczo. **Wymuszenie klasy jako singleton może utrudniać testowanie jej klientów**, ponieważ niemożliwe jest podstawienie imitacji implementacji singletonu, o ile nie implementuje on interfejsu służącego jako jego typ.

Istnieją dwa typowe podejścia do tworzenia obiektów singleton. Oba wykorzystują prywatny konstruktor, natomiast klienty do otrzymania referencji do tego obiektu korzystają z publicznego statycznego składnika klasy. W pierwszej metodzie publiczny statyczny składnik klasy jest polem oznaczonym jako `final`:

```
// Klasa typu singleton z polem z atrybutem final
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

    public void leaveTheBuilding() { ... }
}
```

Prywatny konstruktor jest wywoływany jedynie raz w celu inicjalizacji publicznego statycznego pola `Elvis.INSTANCE`. Brak publicznego i zabezpieczonego konstruktora **gwarantuje** utworzenie tylko jednego obiektu. Po zainicjowaniu klasy `Elvis` będzie istnieć tylko jeden egzemplarz tej klasy — nie mniej, nie więcej. Żadna operacja wykonywana przez klienta nie może tego zmienić, z jednym wyjątkiem — uprawniony klient może wywołać konstruktor prywatny z użyciem refleksji (temat 65.) przy pomocy metody `AccessibleObject.setAccessible`. Jeżeli chcemy obronić się przed takim atakiem, należy zmodyfikować konstruktor, aby zgłaszał wyjątek, jeżeli będzie on próbował utworzyć drugą instancję.

Druga metoda tworzenia singletonów korzysta z publicznej statycznej metody fabrycznej:

```
// Klasa typu singleton ze statyczną metodą fabryczną
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }

    public void leaveTheBuilding() { ... }
}
```

Wywołania statycznej metody `Elvis.getInstance` zwracają referencję do tego samego obiektu; nie są tworzone kolejne instancje klasy `Elvis` (z tym samym zastrzeżeniem, które zostało wspomniane powyżej).

Najważniejszą zaletą pierwszej metody jest fakt, że deklaracja składników klasy od razu wskazuje, że klasa ta będzie klasą typu singleton — publiczne statyczne pole jest opatrzone atrybutem `final`, więc pole to zawsze będzie zawierać referencję do tego samego obiektu. Dodatkową zaletą jest jej prostota.

Najważniejszą zaletą drugiej metody jest możliwość zmiany typu klasy. Jeżeli uznasz, że klasa nie musi być typu singleton, możesz ją zmodyfikować bez potrzeby zmiany definicji API. Statyczna metoda fabryczna dla klas singleton zwraca jedną instancję tej klasy, ale można ją łatwo zmienić, aby na przykład zwracała obiekt unikalny dla każdego wątku wywołującego metodę. Druga zaleta, związana z tworzeniem **uogólnionej metody fabrycznej**, jest przedstawiona w temacie 30. Trzecia zaleta użycia metody fabrycznej polega na tym, że można użyć referencji do metody jako dostawcy, czyli `Elvis::instance` jest `Supplier<Elvis>`. Często żadna z tych zalet nie ma znaczenia, a metoda z polem `final` jest prostsza.

Aby umożliwić **serializację** klasy typu singleton (rozdział 12.), nie wystarczy jedynie dodać frazę `implements Serializable` do jej deklaracji. Aby zagwarantować utrzymanie unikalności klasy typu singleton, należy napisać metodę `readResolve` (temat 89.) i zadeklarować wszystkie pola instancji jako `transient`. W przeciwnym razie każda deserializacja i serializacja obiektu spowoduje utworzenie nowego egzemplarza klasy, co z kolei sprawi, że klasa nie będzie już klasą singleton. Aby zabezpieczyć się przed taką sytuacją, dodamy do naszej klasy `Elvis` następującą metodę `readResolve`:

```
// Metoda readResolve zapewnia zachowanie właściwości singleton
private Object readResolve() throws ObjectStreamException {
    /*
     * Zwraca jedyny obiekt Elvis i pozwala procesowi zbierania nieużytków
     * na zajęcie się dublującymi się obiektami klasy Elvis.
     */
    return INSTANCE;
}
```

Istnieje też trzeci sposób tworzenia singletonów. Wystarczy utworzyć typ wyliczeniowy z jednym elementem:

```
// Singleton wyliczeniowy - zalecane podejście
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding() { ... }
}
```

Podejście to funkcjonalnie odpowiada metodzie z polem publicznym, ale jest bardziej spójne, zapewnia od razu obsługę serializacji oraz daje zabezpieczenie przeciw tworzeniu wielu obiektów, nawet w przypadku złożonych ataków złożonej serializacji lub refleksji. Choć podejście to wydaje się nieco nienaturalne, **typ wyliczeniowy (enum) z jednym elementem jest najlepszym sposobem na implementację singletonu**. Pamiętaj, że podejścia tego nie można zastosować, jeśli singleton musi rozszerzać klasę nadrzędną inną niż Enum (choć można użyć typu wyliczeniowego do implementacji interfejsów).

Temat 4. Wykorzystanie konstruktora prywatnego w celu uniemożliwienia utworzenia obiektu

Czasami występuje potrzeba napisania klasy, która służy jedynie do zgrupowania metod oraz pól statycznych. Klasy te mają bardzo złą reputację, ponieważ niektórzy programiści wykorzystują je do pisania programów proceduralnych za pomocą języka obiektowego, ale istnieją dla nich odpowiednie zastosowania. Są one stosowane do grupowania związanych ze sobą metod, działających na wartościach prostych, lub tablic, jak na przykład `java.lang.Math` i `java.util.Arrays`. Są również stosowane do grupowania statycznych metod, włączając w to fabryki (temat 1.) w klasach implementujących odpowiedni interfejs, jak na przykład `java.util.Collections`. (Od Javy 8 można również umieścić takie metody w interfejsie, jeśli mamy kontrolę nad jego implementacją). Klasy te służą także do grupowania metod w klasach typu `final`, bo nie można ich umieścić w podklasach.

Nie powinno się tworzyć obiektów tych **klas narzędziowych**. Obiekt taki będzie bezużyteczny. Jeżeli jawnie nie utworzymy konstruktora, kompilator utworzy bezparametrowy **konstruktor domyślny**. Użytkownik nie może odróżnić takiego konstruktora od konstruktora utworzonego przez projektanta klasy. Dosyć często można spotkać w różnych API klasy, z których przypadkowo można utworzyć obiekty.

Blokowanie możliwości tworzenia obiektów takich klas przez oznaczanie ich jako abstrakcyjne nie jest dobrym rozwiązaniem. Można utworzyć klasę dziedziczącą po niej i utworzyć obiekt podklasy. Dodatkowo wprowadzamy w błąd użytkowników, sugerując, że klasa ta jest przeznaczona do dziedziczenia (temat 19.). Istnieje prosta metoda, blokująca możliwość tworzenia obiektów danej klasy. Domyślny konstruktor jest generowany jedynie wtedy, gdy klasa nie posiada jawnie zdefiniowanego konstruktora, więc **możemy zablokować możliwość tworzenia obiektów przez zdefiniowanie jednego prywatnego konstruktora**:

```
// Klasa użytkowa bez możliwości tworzenia obiektu
public class UtilityClass {
    // zablokowanie domyślnego konstruktora
    private UtilityClass() {
        throw new AssertionError();
    }

    ... // pozostała część pominięta
}
```

Ponieważ zdefiniowany przez nas konstruktor jest prywatny, jest on niedostępny spoza klasy. Dodanie zgłoszenia wyjątku `AssertionError` nie jest wymagane, ale daje pewność, że konstruktor nie zostanie wywołany przypadkowo z wnętrza klasy. Gwarantuje to, że nie będzie możliwości utworzenia obiektu tej klasy niezależnie od okoliczności. Metoda ta jest raczej mało intuicyjna, ponieważ konstruktor jest zdefiniowany specjalnie po to, aby nie można było go wywołać. Dlatego dobrze jest jasno podać w komentarzu przeznaczenie tego konstruktora.

Efektem ubocznym zastosowania takiego rozwiązania jest zablokowanie możliwości dziedziczenia z tej klasy. Wszystkie konstruktory muszą wywoływać, jawnie bądź niejawnie, dostępne konstruktory klas bazowych, a w tym przypadku taki konstruktor nie istnieje.

Temat 5. Stosuj wstrzykiwanie zależności zamiast odwoływania się do zasobów na sztywno

Wiele klas zależy od jednego zasobu lub większej liczby zasobów. Przykładowo mechanizm sprawdzania poprawności ortograficznej zależy od słownika. Nierzadko widzi się tego rodzaju klasy zaimplementowane jako statyczne klasy użytkowe (temat 4.):

```
// Niewłaściwe użycie klasy użytkowej - nieelastyczne i niemożliwe do przetestowania!
public class SpellChecker {
    private static final Lexicon dictionary = ...;

    private SpellChecker() {} // niemożność utworzenia instancji

    public static boolean isValid(String word) { ... }
    public static List<String> suggestions(String typo) { ... }
}
```

Często spotyka się kod, w którym zastosowano inne podejście, czyli omawiane wcześniej singletony (temat 3.):

// Niewłaściwe użycie singletonu - nieelastyczne i niemożliwe do przetestowania!

```
public class SpellChecker {
    private final Lexicon dictionary = ...;

    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);

    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Przedstawione podejścia nie są satysfakcjonujące, bo zakłada, że istnieje tylko jeden jedyny słownik godny użycia. W praktyce każdy język posiada własny słownik. Co więcej, istnieją słowniki specjalne, wykorzystywane przez różne branże. W trakcie testów klasy zapewne chcielibyśmy użyć innego słownika. Myślenie, że jeden słownik wystarczy na cały czas życia klasy, to mrzonka.

Moglibyśmy spowodować, aby klasa `SpellChecker` obsługiwała wiele słowników, czyniąc pole `dictionary` nieostatecznym i dodając metodę do zmiany języka w istniejącym mechanizmie korekty ortograficznej, ale byłoby to dziwne, narażone na błędy i niestosowne w systemie współbieżnym. **Statyczne klasy użytkowe i singletony nie nadają się do stosowania w klasach, których zachowanie jest parametryzowane pewnymi zewnętrznymi zasobami.**

Potrzebujemy tak naprawdę możliwości obsługi wielu instancji klasy (w tym przykładzie klasy `SpellChecker`), z których każda korzysta z zasobów wskazanych przez klienta (w tym przykładzie będzie to odpowiedni słownik). Prosty wzorzec, który spełnia te założenia, polega na **przekazaniu zasobu w konstruktorze w momencie tworzenia nowej instancji**. To jedna z postaci **wstrzykiwania zależności**: słownik jest **zależnością** mechanizmu sprawdzania poprawności ortograficznej i jest **wstrzykiwany** do słownika w momencie jego tworzenia.

// Wstrzykiwanie zależności zapewnia elastyczność i łatwość testowania

```
public class SpellChecker {
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Wzorzec wstrzykiwania zależności jest tak prosty, że wielu programistów używa go od lat, choć nigdy nie poznało jego nazwy. Choć przykład sprawdzania poprawności ortograficznej stosuje tylko jeden zasób (słownik), wstrzykiwanie zależności działa poprawnie dla dowolnej liczby zasobów i dowolnych grafów zależności.

Zachowuje niezmiennosc (temat 17.), więc wielu klientów może współdzielić zależne obiekty (o ile oczywiście wymagają tych samych zasobów bazowych). Wstrzykiwanie zależności można z powodzeniem stosować dla konstruktorów, statycznych metod fabrycznych (temat 1.) lub budowniczych (temat 2.).

Przydatną odmianą wzorca jest przekazywanie do konstruktora **fabryki** zasobów. Fabryka to obiekt, który może być wywoływany wielokrotnie w celu tworzenia instancji określonego typu. Takie fabryki stosują wzorzec **metody fabrycznej** (*Factory Method*) [Gamma95]. Interfejs `Supplier<T>`, wprowadzony w Javie 8, w sposób wręcz idealny nadaje się do reprezentowania fabryk. Metody przyjmujące `Supplier<T>` jako dane wejściowe powinny zazwyczaj ograniczać parametr typu fabryki za pomocą **ograniczonego typu wieloznacznego** (temat 31.), aby pozwolić klientowi na przekazanie fabryki, która tworzy dowolny podtyp wskazanego typu. Oto przykład metody, która tworzy mozaikę, wykorzystując fabrykę dostarczoną przez klienta odpowiedzialną za generowanie poszczególnych kafelków:

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

Choć wstrzykiwanie zależności znacząco poprawia elastyczność i testowalność, dosyć szybko potrafi zrobić spory bałagan w dużym projekcie, który zazwyczaj ma tysiące zależności. Ten bałagan na szczęście łatwo posprzątać — wystarczy biblioteka wstrzykiwania zależności, taka jak Dagger [Dagger], Guice [Guice] lub Spring [Spring]. Sposób użycia tych bibliotek wykracza poza ramy tej książki, ale zapamiętaj, że API zaprojektowane do ręcznego wstrzykiwania zależności bardzo łatwo dostosować do potrzeb wymienionych bibliotek.

Podsumujmy — nie używaj singletonów lub statycznych klas użytkowych do implementacji klas, które zależą od jednego lub więcej zasobów wpływających na zachowanie klasy i same tego zasobu nie tworzą. Zamiast tego przekaz zasób (lub tworzącą go fabrykę) do konstruktora (lub statycznej metody fabrycznej czy też budowniczego). Rozwiązanie to, nazywane wstrzykiwaniem zależności, znacząco wpłynie na elastyczność, a także łatwość wielokrotnego użycia i testowania.

Temat 6. Unikanie powielania obiektów

Prawidłową techniką jest powtórne wykorzystanie istniejącego obiektu zamiast tworzenia nowego za każdym razem, gdy obiekt taki jest potrzebny. Powtórne użycie jest szybsze i prezentuje lepszy styl programowania. Jeżeli obiekt jest niezmienny, zawsze może być powtórnie wykorzystywany (temat 17.).

Skrajnym przypadkiem stylu, w jakim nie należy pisać kodu, jest następujące wyrażenie:

```
String s = New String("bikini"); //NIE RÓB TAK!
```

Wyrażenie to tworzy nowy obiekt typu `String` za każdym wywołaniem, choć żaden z tych nowych obiektów nie jest potrzebny. Argument konstruktora `String("bikini")` jest sam w sobie obiektem klasy `String`, funkcjonalnie identycznym z obiektem utworzonym przez konstruktor. Jeżeli wyrażenie to umieszczone zostanie w pętli lub często wywoływanej metodzie, może niepotrzebnie zostać utworzonych wiele obiektów.

Lepiej zastosować następującą wersję powyższego wyrażenia:

```
String s = "bikini";
```

Zamiast tworzyć za każdym razem nowy obiekt, korzystamy teraz z jednego obiektu typu `String`. Dodatkowo gwarantujemy, że obiekt ten zostanie powtórnie wykorzystany przez każdy inny program, uruchomiony na tej samej maszynie wirtualnej, jeżeli będzie zawierał ten sam ciąg znaków [JLS, 3.10.5].

Możesz również uniknąć powielania obiektów, korzystając ze **statycznych metod fabrycznych** (temat 1.) zamiast z konstruktorów w przypadku klas niezmiennych dostarczających oba rozwiązania. Na przykład zastosowanie statycznej metody fabrycznej `Boolean.valueOf(String)` jest niemal zawsze lepsze niż zastosowanie konstruktora `Boolean(String)`. Konstruktor **zawsze** tworzy nowy obiekt w przeciwieństwie do statycznej metody fabrycznej, która nie musi tego robić i najczęściej tego nie robi. Oprócz niezmiennych obiektów możesz również powtórnie wykorzystywać zwykłe obiekty, o których wiesz, że nie należy ich modyfikować.

Niektóre tworzenia obiektów są bardziej kosztowne niż inne. Jeśli często potrzebujesz takiego „kosztownego obiektu”, warto otworzyć pamięć podręczną i starać się go wykorzystać wielokrotnie. Niestety, nie zawsze w oczywisty sposób widać, że tworzy się tego rodzaju obiekt. Przypuśćmy, że chcemy napisać metodę sprawdzającą, czy przekazany tekst jest liczbą rzymską. Oto w jaki sposób moglibyśmy wykonać zadanie za pomocą wyrażenia regularnego:

```
// Wydajność tego rozwiązania można znacząco zwiększyć!
static boolean isRomanNumeral(String s) {
    return s.matches("(^(?=.)M*(C[MD]|D?C{0,3})"
        + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

Problemem tej implementacji jest korzystanie z metody `String.matches`. **Choć metoda ta to najprostszy sposób sprawdzenia, czy tekst pasuje do wyrażenia regularnego, nie nadaje się do wielokrotnego stosowania we fragmentach wymagających dużej wydajności.** Problem polega na tym, że tworzy wewnętrznie obiekt `Pattern` dla wyrażenia regularnego, wykorzystuje go tylko raz, a następnie pozostawia do usunięcia przez mechanizm zbierania nieużytków. Tworzenie instancji klasy `Pattern` jest kosztowne, bo wymaga kompilacji wyrażenia regularnego do automatu skończonego.

Aby poprawić wydajność rozwiązania, jawnie skompilujmy wyrażenie regularne do instancji `Pattern` (która jest niezmienna) jako część inicjalizacji klasy, a następnie wykorzystujemy tę wersję w każdym wywołaniu metody `isRomanNumeral`:

```
// Wielokrotne użycie kosztownego obiektu poprawia wydajność
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
        "(?=.)M*(C[MD]|D?C{0,3})"
        + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s) {
        return ROMAN.matcher(s).matches();
    }
}
```

Ulepszona wersja metody `isRomanNumeral` zapewnia znaczący wzrost wydajności, jeśli jest wywoływana wielokrotnie. Na moim komputerze pierwsza wersja potrzebowała 1,1 μ s dla ośmioznakowego tekstu, natomiast druga 0,17 μ s, czyli działa około 6,5 razy szybciej. Poprawiła się nie tylko szybkość działania, ale również czytelność kodu. Utworzenie statycznego i ostatecznego pola dla wcześniej niewidocznego obiektu `Pattern` pozwoliło nadać mu nazwę, co samo wyrażenie regularne uczyniło czytelniejszym.

Jeżeli klasa zawierająca ulepszoną wersję metody `isRomanNumeral` nie zostałaby wywołana ani razu, inicjalizacja pola `ROMAN` byłaby niepotrzebna. Możliwe jest wyeliminowanie niepotrzebnej inicjalizacji przez **późne inicjowanie** pola (temat 83.) podczas pierwszego wywołania metody `isRomanNumeral`, jednak **nie** jest to zalecane. Późna inicjalizacja komplikuje program, a zysk wydajności jest najczęściej niewielki (temat 67.).

Gdy obiekt jest niezmienny, oczywistym jest, że można go wykorzystać wielokrotnie. Istnieją jednak sytuacje, w których jest to mniej oczywiste. Rozważmy przypadek **adapterów** [Gamma95], nazywanych również **widokami**. Adapter to obiekt przydzielony do obiektu bazowego, który udostępnia alternatywny interfejs dla tego bazowego obiektu. Ponieważ adapter jest ściśle związany ze swoim obiektem bazowym, nie ma potrzeby tworzenia więcej niż jednego obiektu adaptera dla jednego obiektu bazowego.

Na przykład metoda `keySet` interfejsu `Map` zwraca widok `Set` obiektu `Map`, zawierający wszystkie klucze mapowania. Może się wydawać, że każde wywołanie `keySet` tworzy nowy obiekt `Set`, ale w rzeczywistości metoda `keySet`, wywołana dla podanego obiektu `Map`, zwraca zawsze ten sam obiekt `Set`. Choć zwracany obiekt `Set` najczęściej może być modyfikowany, to wszystkie zwracane obiekty są funkcjonalnie identyczne. Gdy jeden ze zwracanych obiektów zostanie zmieniony, zmieniają się wszystkie, ponieważ są one związane z tym samym obiektem `Map`. Choć tworzenie wielu instancji obiektów `keySet` nie jest szkodliwe, nie jest też konieczne i nie ma żadnych zalet.

Istnieje inny sposób tworzenia niepotrzebnych obiektów. Jest on nazywany *autoboxing* i pozwala programiście na mieszanie typów prostych i opakowanych typów prostych — pakowanie i odpakowywanie jest wykonywane automatycznie w razie potrzeby. *Autoboxing* **zaciera, ale nie usuwa różnicy pomiędzy typami prostymi i opakowanymi typami prostymi**. Istnieją subtelne różnice semantyczne i nie tak subtelne różnice w wydajności (temat 61.). Weźmy pod uwagę poniższy program, który oblicza sumę wszystkich dodatnich liczb typu `int`. W tym celu program korzysta z operacji na liczbach `long`, ponieważ nie da się umieścić sumy wszystkich liczb typu `int` w zmiennej o tym samym typie:

```
// Okropnie powolny kod! Czy zauważasz tworzenie obiektów?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

Program zwraca prawidłowe wyniki, ale jest **znacznie** powolniejszy, niż mógłby być, a to z powodu jednoznakowego błędu typograficznego. Zmienna `sum` jest zadeklarowana jako `Long` zamiast `long`, co oznacza, że program tworzy około 2^{31} niepotrzebnych obiektów `Long` (po jednym na każde dodanie wartości `long i` do `Long sum`). Zmiana deklaracji zmiennej `sum` z `Long` na `long` powoduje, na moim komputerze, zmniejszenie czasu działania z 6,3 sekundy do 0,59 sekundy. Lekcja jest jasna — **należy używać typów prostych zamiast opakowanych typów prostych i uważać na niezamierzone operacje *autoboxingu***.

Nie należy jednak zakładać, że tworzenie obiektów jest kosztowne i powinno się go unikać. Z drugiej strony tworzenie i zwracanie małych obiektów, których konstruktory wykonują niewiele operacji, jest mało kosztowne, szczególnie w implementacjach nowoczesnych JVM. Tworzenie dodatkowych obiektów w celu zwiększenia klarowności, prostoty lub możliwości programu jest zazwyczaj dobre.

Z kolei unikanie tworzenia obiektów przez utrzymywanie własnych **pul obiektów** jest złym pomysłem, chyba że obiekty w puli są niezwykle ciężkie. Klasycznym przykładem obiektów, które **uzasadniają** zastosowanie puli obiektów, są połączenia z bazą danych. Koszt zestawienia połączenia jest na tyle wysoki, że sensowne jest ponowne używanie tych obiektów. Zazwyczaj jednak utrzymywanie własnej puli obiektów powoduje zabałaganienie kodu, zwiększa zużycie pamięci i źle wpływa na wydajność. Nowoczesne implementacje JVM mają wysoko zoptymalizowane procedury zbierania nieużytków, które łatwo prześcigają pule lekkich obiektów.

Przeciwieństwem tego tematu jest temat 50., dotyczący **kopiowania defensywnego**. W tym temacie mówimy: „Nie twórz nowych obiektów, jeżeli można ponownie wykorzystać istniejące”, natomiast w temacie 50.: „Nie wykorzystuj ponownie

istniejącego obiektu, jeżeli powinieneś utworzyć nowy”. Należy zwrócić uwagę, że koszt ponownego użycia obiektu przy wywołaniu kopiowania defensywnego jest znacznie większy niż koszt niepotrzebnego tworzenia powielonego obiektu. Brak wykonania kopii defensywnej, jeżeli jest to wymagane, może prowadzić do ukrytych błędów i dziur w systemie bezpieczeństwa; niepotrzebne tworzenie obiektów może co najwyżej wpływać na styl i wydajność.

Temat 7. Usuwanie niepotrzebnych referencji do obiektów

Jeżeli korzystałeś wcześniej z języka, w którym zarządzanie pamięcią było wykonywane przez programistę, na przykład C lub C++, to korzystanie z języka posiadającego mechanizm zbierania nieużytków powoduje uproszczenie procesu programowania. Po zakończeniu pracy z obiektem pamięć zajmowana przez niepotrzebne obiekty jest automatycznie odzyskiwana. Może to szybko doprowadzić do sytuacji, w której zaczniesz uważać, że nie musisz zajmować się zarządzaniem pamięcią, a nie jest to prawda.

Popatrzmy na prostą implementację stosu:

```
// Czy możesz zlokalizować „wyciek pamięci”?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size==0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Zapewnienie miejsca na co najmniej jeden element, przy dwukrotnym
     * zwiększeniu rozmiaru tablicy za każdym razem, gdy musi być ona powiększona.
     */
}
```

```
private void ensureCapacity() {  
    if (elements.length == size) {  
        elements = Arrays.copyOf(elements, 2 * size + 1);  
    }  
}
```

Na pierwszy rzut oka nie ma w tym fragmencie programu nic nieprawidłowego (ale zwróć uwagę na temat 29., w którym jest opisana wersja z typami ogólnymi). Możesz go dokładnie przetestować i każdy przypadek testowy zostanie prawidłowo obsługany. Jednak możemy odkryć tu problem, zwany popularnie „wyciekami pamięci”, objawiający się zmniejszeniem wydajności z powodu zwiększonej aktywności procesu zbierania nieużytków lub zwiększonego zużycia pamięci. W ekstremalnych przypadkach wycieki pamięci doprowadzają do intensywnego korzystania z dysku podczas stronicowania pamięci lub nawet zatrzymania programu z błędem `OutOfMemoryError`, choć zdarza się to dosyć rzadko.

Gdzie występuje wyciek pamięci? Jeżeli stos rośnie, a następnie zmniejsza się, obiekty znajdujące się na stosie nie są niszczone, nawet jeżeli program korzystający ze stosu nie przechowuje na nim żadnych referencji. Dzieje się tak, ponieważ stos przechowuje **niepotrzebne referencje** do obiektów. Niepotrzebne referencje to takie, które nigdy nie zostaną już wykorzystane. W tym przypadku wszystkie referencje poza „aktywną częścią” elementów tablicy są niepotrzebne. Aktywna część to elementy o indeksach mniejszych niż wartość zmiennej `size`.

Wycieki pamięci (prawidłowo nazywane **nieświadomym przetrzymywaniem obiektów**) w językach obsługujących proces usuwania nieużytków mają ogromny wpływ na aplikację. Jeżeli referencja do obiektu jest nieświadomie przetrzymywana, nie tylko ten obiekt nie będzie podlegał procesowi usuwania nieużytków, ale również wszystkie obiekty, do których odwołuje się ten obiekt itd. Nawet jeżeli nieświadomie przetrzymywanych jest kilka referencji, może to spowodować przetrzymywanie wielu obiektów i doprowadzić tym samym do dużego spadku wydajności.

Poprawienie tego błędu jest dosyć proste — wystarczy przypisać wartość `null` do referencji, które przestają być wykorzystywane. W przypadku naszej klasy `Stack` referencja staje się niepotrzebna, gdy obiekt zostaje zdjęty ze stosu. Prawidłowa wersja metody `pop` wygląda następująco:

```
public Object pop() {  
    if (size==0)  
        throw new EmptyStackException();  
    Object result = elements[--size];  
    elements[size] = null; //usunięcie niepotrzebnej referencji  
    return result;  
}
```

Dodatkową zaletą przypisywania wartości `null` do niepotrzebnych referencji jest to, że jeżeli przez pomyłkę będziemy próbować się do nich odwołać, program zostanie zatrzymany i wygeneruje wyjątek `NullPointerException`, zamiast wykonania operacji w sposób nieprawidłowy. Szybkie wykrywanie błędów w kodzie jest bardzo przydatne.

Gdy programiści spotkają się pierwszy raz z przedstawionym tu problemem, często przypisują wartość `null` do wszystkich referencji, gdy program przestaje z nich korzystać. Nie jest to ani niezbędne, ani korzystne, ponieważ zaciemnia to niepotrzebnie kod i może zmniejszyć wydajność aplikacji. **Przypisywanie wartości `null` do referencji powinno być raczej wyjątkiem, a nie normą.** Najlepszą metodą eliminowania niepotrzebnych referencji jest powtórne wykorzystywanie zawierających je zmiennych lub ich wyjście z zasięgu. Najbardziej naturalną metodą zagwarantowania tego jest definiowanie zmiennych w możliwie wąskim zakresie (temat 57.).

Kiedy należy przypisywać wartość `null` do referencji? Jakie aspekty działania klasy `Stack` pozwalają nam przypuszczać, że zajdzie zjawisko wycieku pamięci? Najprościej rzecz ujmując, klasa `Stack` **samodzielnie zarządza pamięcią. Pula magazynowa** składa się z elementów tablicy `elements` (dokładniej z referencji do obiektów, a nie samych obiektów). Elementy znajdujące się w aktywnej części tablicy (według przedstawionej wcześniej definicji) są **przydzielone**, natomiast pozostała część tablicy jest **wolna**. Proces zbierania nieużytków „nie wie” jednak o tym, dla niego wszystkie referencje zapisane w tablicy są identyczne. Tylko programista wie, że nieaktywna część tablicy jest nieważna. Dlatego na programiście ciąży obowiązek poinformowania procesu zbierania nieużytków o tym fakcie przez „ręczne” przypisanie wartości `null` do elementu tablicy, który przechodzi do nieaktywnej części tablicy.

Można przyjąć, że **we wszystkich przypadkach, gdy klasa samodzielnie zarządza pamięcią, programista powinien uważać na wycieki pamięci.** Gdy przechowywany element jest zwalniany, do referencji do tego obiektu powinny być przypisywane wartości `null`.

Innym częstym źródłem wycieków pamięci są mechanizmy buforowania lub pamięci podręczne. Po umieszczeniu referencji do obiektu w buforze często zdarza się, że obiekt został w nim umieszczony i pozostaje tam długo po tym, jak przestał być wykorzystywany. Istnieją dwa możliwe rozwiązania tego problemu. Jeżeli stworzyłeś mechanizm bufora w taki sposób, że jego elementy są istotne tak długo, jak długo istnieją referencje do ich kluczy poza buforem, możesz zastosować do implementacji bufora klasę `WeakHashMap`. Dzięki temu elementy będą usuwane, gdy staną się nieważne. Pamiętaj, że klasa `WeakHashMap` ma zastosowanie tylko wtedy, gdy oczekiwany czas życia obiektu w buforze określają zewnętrzne referencje do klucza, a nie do wartości.

Częściej spotyka się sytuację, w której przedział czasu, w jakim elementy znajdujące się w buforze są istotne, jest niezdefiniowany, natomiast pozycje znajdujące się w buforze są z upływem czasu coraz mniej użyteczne. W takim przypadku bufor powinien być okresowo czyszczony z nieużywanych elementów. Czyszczenie takie może być realizowane przez dodatkowy wątek (na przykład korzystający z `ScheduledThreadPoolExecutor`) lub jako efekt uboczny podczas dodawania nowych elementów do bufora. Klasa `java.util.LinkedHashMap` ułatwia zrealizowanie takiego mechanizmu dzięki metodzie `removeEldestEntry`. W celu uzyskania bardziej skomplikowanych buforów należy bezpośrednio użyć `java.lang.ref`.

Trzecim częstym źródłem wycieków pamięci są nasłuchy i inne metody wywołania zwrotnego. Jeżeli implementujemy API, w którym klient rejestruje metody wywołania zwrotnego, ale ich jawnie nie wyrejestrowuje, mogą się one nakładać, o ile nie wykonamy odpowiednich akcji. Najlepszym sposobem na zapewnienie szybkiego niszczenia procedur wywołania zwrotnego jest przechowywanie tylko **słabych referencji** do nich, na przykład przechowywanie ich jako kluczy w `WeakHashMap`.

Wycieki pamięci nie objawiają się tak, jak inne oczywiste błędy; mogą występować w systemie przez bardzo długi czas. Są one zwykle wykrywane podczas bardzo szczegółowej inspekcji kodu lub za pomocą programu wspomagającego uruchamianie, zwanego **profilerem stogu**. Dlatego ważne jest nauczenie się przewidywania problemów tego typu przed ich wystąpieniem.

Temat 8. Unikanie finalizatorów i oczyszczaczy

Finalizatory są nieprzewidywalne, często niebezpieczne i zwykle niepotrzebne. Ich zastosowanie może powodować nieprawidłową pracę programu, problemy z przenośnością i obniżenie wydajności. Finalizatory mają kilka zastosowań, które opiszemy w dalszej części rozdziału, ale jako zasadę powinniśmy przyjąć ich unikanie. Wraz z pojawieniem się Javy 9 finalizatory są wycofywane z użycia, ale nadal korzystają z nich biblioteki Javy. Java 9 zastępuje finalizatory **oczyszczaczami**. Choć są mniej niebezpieczne od finalizatorów, to nadal są **nieprzewidywalne, powolne i zwykle niepotrzebne**.

Programiści C++ powinni być ostrzeżeni, aby nie traktować finalizatorów lub oczyszczaczy jako odpowiedników destruktory. W języku C++ destruktory są normalnym sposobem odzyskiwania zasobów związanych z obiektem — są to niezbędne dopełnienia konstruktorów. W języku Java pamięć związaną z obiektami zarządza proces zbierania nieużytków. Pamięć jest odzyskiwana bez potrzeby wykonywania jakiegokolwiek akcji ze strony programisty, gdy obiekt przestaje być

dostępny. Destruktory w C++ są używane również do odzyskiwania zasobów innych niż pamięć. W języku Java do tego celu zwykle stosuje się bloki try-finally lub try z zasobami (temat 9.).

Nie ma żadnej gwarancji, że finalizator lub oczyszczacz zostanie wykonany natychmiast [JLS, 12.6]. Może upłynąć dosyć długi czas od momentu wyjścia obiektu z zasięgu do wykonania finalizatora lub oczyszczacza. Oznacza to, że **przez finalizator lub oczyszczacz nie powinny być wykonywane żadne operacje krytyczne czasowo**. Na przykład poważnym błędem jest poleganie na finalizatorach lub oczyszczaczach przy zamykaniu plików, ponieważ deskryptory plików są ograniczonym zasobem. Jeżeli zostanie pozostawionych zbyt wiele otwartych plików, ponieważ JVM będzie opóźniać wykonywanie finalizatorów lub oczyszczaczy, program może zostać zatrzymany, gdyż nie będzie mógł utworzyć kolejnego pliku.

Opóźnienie wykonywania finalizatorów lub oczyszczaczy zależy przede wszystkim od algorytmu odzyskiwania nieużytków, który różni się znacznie w różnych implementacjach JVM. Zachowanie programu polegającego na szybkości wykonywania finalizatorów może się różnić na różnych JVM. Może się zdarzyć, że program będzie działał świetnie na Twojej JVM, a na maszynie wirtualnej stosowanej przez Twojego największego klienta będzie generował serie błędów.

Opóźnienie procesu finalizacji nie jest tylko teoretycznym problemem. Utworzenie finalizatora dla klasy może, w rzadkich przypadkach, znacznie opóźnić usuwanie obiektów. Kolega analizował długo działającą aplikację graficzną, która tajemniczo zatrzymywała się z błędem `OutOfMemoryError`. Analiza wykazała, że w momencie wystąpienia błędu aplikacja posiadała tysiące obiektów graficznych, czekających w kolejce finalizatora na ich zakończenie i usunięcie. Niestety, wątek finalizatora miał niższy priorytet niż inny wątek programu, więc obiekty były tworzone o wiele szybciej, niż mogły być usuwane. Specyfikacja języka Java nie określa, w którym wątku będą wykonywane finalizatory, więc nie ma uniwersalnej metody zabezpieczenia się przed takimi problemami — z wyjątkiem powstrzymania się od korzystania z finalizatorów. Oczyszczacze są nieco lepsze od finalizatorów, bo twórcy klasy mają kontrolę nad własnymi wątkami oczyszczającymi, ale oczyszczacze nadal działają w tle pod kontrolą mechanizmu odzyskiwania nieużytków, więc nadal nie ma gwarancji szybkiego oczyszczenia.

Specyfikacja nie tylko nie gwarantuje szybkiego wykonywania finalizatorów lub oczyszczaczy; nie gwarantuje tego wcale. Możliwe jest, że program zakończy się bez ich wykonania na niektórych obiektach, które przestały być dostępne. W konsekwencji **nigdy nie powinieneś polegać na finalizatorach lub oczyszczaczach w celu zmiany ważnego stanu aplikacji**. Poleganie na finalizatorach na przykład do zwalniania trwałych blokad zasobów współdzielonych, powiedzmy baz danych, jest dobrym sposobem na zablokowanie całego systemu rozproszonego.

Nie daj się zwieść metodom `System.gc` i `System.runFinalization`. Mogą one zwiększyć częstotliwość wykonywania finalizatorów lub oczyszczaczy, ale tego nie gwarantują. Dwoma metodami, które kiedyś twierdziły, że mogą zagwarantować finalizację, były `System.runFinalizersOnExit` i jej bliźniaczka `Runtime.runFinalizersOnExit`. Metody te są jednak pełne wad i ich używanie nie jest zalecane [`ThreadStop`].

Jeżeli nadal nie jesteś przekonany, że należy unikać stosowania finalizatorów, warto wiedzieć o jeszcze jednej wadzie. Jeżeli w czasie finalizacji wystąpi nieobsłużony wyjątek, jest on ignorowany, a finalizacja obiektu zostaje przerwana [JLS, 12.6]. Nieobsłużony wyjątek może spowodować pozostawienie uszkodzonego obiektu. Jeżeli inny wątek będzie próbował skorzystać z tego uszkodzonego obiektu, jego działanie będzie nieprzewidywalne. Normalnie, jeżeli wystąpi nieobsłużony wyjątek, wątek zostanie zatrzymany i wyświetlana będzie zawartość stosu. Jeżeli jednak wyjątek taki wystąpi w finalizatorze, nie zostanie nawet wypisane ostrzeżenie. Oczyszczacze nie są narażone na tę wadę, bo to biblioteka je wykorzystująca ma pełną kontrolę nad wątkiem oczyszczania.

Użycie finalizatorów lub oczyszczaczy powoduje znaczny spadek wydajności.

Na moim komputerze czas tworzenia prostego obiektu `AutoCloseable`, zamknięcia go za pomocą konstrukcji `try` z zasobami i usunięcia przez mechanizm odzyskiwania nieużytków wynosi około 12 ns. Dodanie finalizatora zwiększa ten czas do 550 ns. Inaczej mówiąc, tworzenie i usuwanie obiektów z finalizatorami jest około 50 razy wolniejsze. Wynika to z faktu, iż finalizatory hamują wydajne odzyskiwanie nieużytków. Oczyszczacze są podobne w kwestii szybkości do finalizatorów, jeśli używa się ich do wyczyszczenia wszystkich instancji klasy (około 500 ns na instancję na moim komputerze), ale są znacznie szybsze, jeśli korzysta się z nich tylko jako „siatka zabezpieczająca” w sposób opisany poniżej. W takich okolicznościach tworzenie, czyszczenie i niszczenie obiektu zajmują na moim komputerze około 66 ns, co oznacza, że koszt ubezpieczenia wynosi pięciokrotność (a nie pięćdziesięciokrotność), **jeśli** nie skorzystamy z usuwania.

Finalizatory mają poważny problem z bezpieczeństwem — otwierają klasę na ataki finalizacji. Sam pomysł przeprowadzenia takiego ataku jest prosty — jeśli wyjątek zostanie zgłoszony z poziomu konstruktora lub jego odpowiednika dotyczącego serializacji, czyli metod `readObject` i `readResolve` (rozdział 12.), finalizator złośliwej podklasy może działać na częściowo skonstruowanym obiekcie, który powinien „umrzeć przy narodzinach”. Finalizator może zapisać referencję do obiektu do pola statycznego, co zapobiegnie jego usunięciu. Po zapisaniu takiego uszkodzonego obiektu można na nim wywoływać dowolne metody, choć sam obiekt nigdy nie powinien istnieć. **Zgłoszenie wyjątku z poziomu konstruktora powinno spowodować, że obiekt nigdy nie powstanie — z powodu istnienia finalizatorów tak się nie dzieje.** Takie ataki mogą mieć bardzo poważne konsekwencje. Klasy ostateczne są odporne na ataki finalizacji, bo nikt nie może napisać dla niej złośliwej podklasy. **Aby uchronić klasy nieostateczne przed atakiem finalizacji, napisz ostateczną metodę `finalize`, która nic nie robi.**

Czego powinniśmy używać zamiast finalizatorów w przypadku klas, które korzystają z zasobów wymagających zwolnienia — takich jak pliki lub wątki? Po prostu należy spowodować, aby **klasa implementowała `AutoCloseable`**, a od klientów klasy należy żądać wywołania metody `close`, gdy obiekt przestaje być potrzebny, najczęściej z poziomu konstrukcji try z zasobami, aby zapewnić czyszczenie także w przypadku wyjątku (temat 9.). Należy zaznaczyć, że obiekt musi zapamiętywać fakt wykonania tej metody w prywatnym polu obiektu. Inne metody powinny sprawdzać zawartość tego pola i, jeżeli zostaną wykonane po metodzie zwalnianej zasoby, zgłaszać wyjątek `IllegalStateException`.

W jakich zastosowaniach sprawdzają się więc finalizatory i oczyszczacze? Istnieją dwa uzasadnione przypadki. Jeden z nich to użycie finalizatora jako „siatki zabezpieczającej” na wypadek, gdyby właściciel obiektu zapomniał wywołać metodę `close`. Choć nie ma gwarancji, że finalizator lub oczyszczacz zostanie wywołany natychmiast (lub w ogóle), to lepiej jest zwolnić zasoby krytyczne późno niż wcale, oczywiście w tych przypadkach (miejmy nadzieję, rzadkich), gdy klient nie wywoła odpowiedniej metody kończącej. Jeżeli rozważamy napisanie takich finalizatorów zabezpieczających, należy poważnie się zastanowić, czy dodatkowe zabezpieczenie jest warte dodatkowych kosztów. Niektóre klasy biblioteki Javy, np. `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor` i `java.sql.Connection`, implementują „siatki zabezpieczające” w postaci finalizatorów.

Drugim dopuszczalnym zastosowaniem finalizatorów są obiekty z zewnętrznymi **obiettami macierzystymi**. Obiekt taki odwołuje się bezpośrednio do składników systemu operacyjnego poprzez metody macierzyste (ang. *native*). Ponieważ obiekt zewnętrzny nie jest normalnym obiektem języka Java, proces zbierania nieużytków „nic o nim nie wie”. Nie może więc w takim przypadku prawidłowo przeprowadzić procesu usuwania obiektu zewnętrznego w trakcie usuwania głównego obiektu. Do wykonania tej czynności może posłużyć finalizator lub oczyszczacz, pod warunkiem że zewnętrzny obiekt macierzysty nie blokuje żadnych zasobów krytycznych. Jeżeli obiekt zewnętrzny korzysta z zasobów, które muszą być zwalniane możliwie szybko, klasa powinna zawierać zdefiniowaną metodę `close`, identycznie jak zostało to opisane powyżej.

Oczyszczacze są nieco bardziej podstępne w użyciu. Poniżej znajduje się prosta klasa `Room`, ilustrująca ich wykorzystanie. Założmy, że pokoje muszą zostać oczyszczone przed ich ponownym użyciem. Klasa `Room` implementuje `AutoCloseable`; fakt, że automatyczna siatka bezpieczeństwa używa oczyszczacza, to jedynie szczegół implementacyjny. W odróżnieniu od finalizatorów oczyszczacze nie zaśmiecają publicznego API klasy:

```
// Klasa implementująca AutoCloseable używa oczyszczacza jako siatki bezpieczeństwa
public class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();
```

```

// Zasoby wymagające oczyszczenia. Nie można odnosić się do Room!
private static class State implements Runnable {
    int numJunkPiles; // liczba stert śmieci

    State(int numJunkPiles) {
        this.numJunkPiles = numJunkPiles;
    }

    // wywoływane przez metodę close lub oczyszczacz
    @Override public void run() {
        System.out.println("Czyszczenie pokoju");
        numJunkPiles = 0;
    }
}

// stan pokoju współdzielony z cleanable
private final State state;

// Nasz cleanable. Czyści pokój, gdy ten ma zostać usunięty przez
// odzyskiwanie nieużytków
private final Cleaner.Cleanable cleanable;

public Room(int numJunkPiles) {
    state = new State(numJunkPiles);
    cleanable = cleaner.register(this, state);
}

@Override public void close() {
    cleanable.clean();
}
}

```

Zagnieżdżona, statyczna klasa `State` przechowuje zasoby, które są wymagane przez obiekt `cleaner` do wyczyszczenia pokoju. W tym przypadku jest to tylko pole `numJunkPiles` obrazujące nieporządek w pokoju. W bardziej realistycznym przykładzie byłoby to ostateczne pole `long` zawierające wskaźnik na obiekt macierzysty. Klasa `State` implementuje `Runnable`, a jej metoda `run` zostaje wywołana co najwyżej raz przez obiekt `Cleanable`, który otrzymujemy po rejestracji instancji `State` u oczyszczacza wewnątrz kodu konstruktora `Room`. Wywołanie metody `run` może zostać spowodowane przez jedną z dwóch sytuacji: zazwyczaj będzie to wywołanie metody `close` klasy `Room`, która wywoła metodę `clean` z `Cleanable`. Jeśli klient z jakiegoś powodu nie wywoła metody `close`, zanim instancja klasy `Room` będzie miała zostać przekazana do odzyskania nieużytku, oczyszczacz wywoła (taką mamy nadzieję) metodę `run` ze `State`.

Niezwykle ważnym aspektem jest brak odwołania w instancji `State` do obiektu `Room`. Gdyby takie odwołanie się pojawiło, powstałoby zapętlenie odwołań, co uniemożliwiłoby wskazanie instancji `Room` do automatycznego odzyskania pamięci (czyli również automatycznego oczyszczenia). Z tego powodu `State` musi być

statyczną klasą zagnieżdżoną, bo wersja niestatyczna zawierałaby referencję do zawierającej ją instancji (temat 24.). Warto w tym miejscu powstrzymać się od stosowania `lambd`, bo one również mogą łatwo domknąć się na referencjach do obiektów, w których się znajdują.

Jak wcześniej wspomnieliśmy, `oczyszczacz` w klasie `Room` służy tylko jako siatka bezpieczeństwa. Jeśli klient zawsze otacza wszystkie utworzenia obiektów `Room` w konstrukcjach try z zasobami, automatyczne czyszczenie nigdy nie będzie potrzebne. Oto przykład klienta używającego klasy we właściwy sposób:

```
public class Adult {
    public static void main(String[] args) {
        try (Room myRoom = new Room(7)) {
            System.out.println("Do widzenia");
        }
    }
}
```

Jak można oczekiwać, uruchomienie programu `Adult` wyświetli tekst `Do widzenia`, a za nim tekst `Czyszczenie pokoju`. Co się jednak stanie w przypadku programu zachowującego się nieodpowiednio, czyli niezawierającego czyszczenia pokoju?

```
public class Teenager {
    public static void main(String[] args) {
        new Room(99);
        System.out.println("Nara");
    }
}
```

Oczekiwalibyśmy, że po tekście `Nara` pojawi się tekst `Czyszczenie pokoju`, ale na moim komputerze tekst ten nigdy się nie pojawia — proces po prostu kończy swoje działanie. To nieprzewidywalność, o której wspominaliśmy wcześniej. Specyfikacja klasy `Cleaner` mówi: „Zachowanie `oczyszczaczy` w momencie wywołania polecenia `System.exit` zależy od implementacji. Nie są dawane żadne gwarancje co do wykonania akcji `oczyszczania`”. Choć specyfikacja o tym nie wspomina, podobne ograniczenia dotyczą zwykłego dojścia do końca programu. Na moim komputerze dodanie wywołania `System.gc()` do metody `main` klasy `Teenager` wystarczy, aby tekst `Czyszczenie pokoju` pojawił się przed wyłączeniem procesu, ale nie ma żadnej gwarancji, że to samo zachowanie zaobserwujesz na swoim komputerze.

Podsumujmy. Nie należy korzystać z `oczyszczaczy`, a w przypadku Javy 8 lub wcześniejszej — z `finalizatorów`, poza mechanizmami „siatek bezpieczeństwa” lub zwalniania niekrytycznych zasobów systemu operacyjnego. Nawet wtedy pamiętaj jednak o nieokreśloności i wpływie na wydajność.

Temat 9. Preferuj konstrukcję try z zasobami zamiast try-finally

Biblioteki Javy zawierają wiele zasobów, które muszą być zamykane ręcznie przez wywołanie metody `close`. Przykładami mogą tu być klasy `InputStream`, `OutputStream` lub `java.sql.Connection`. Zamykanie zasobów jest często lekceważone przez klientów, co ma swoje konsekwencje wydajnościowe. Choć wiele z tych zasobów korzysta z finalizatorów jako swoistej siatki bezpieczeństwa, finalizatory nie działają najlepiej (temat 8.).

Dawniej konstrukcja `try-finally` była najlepszym sposobem zagwarantowania, że zasób zostanie poprawnie zamknięty, nawet w przypadku zgłoszenia wyjątku lub szybszego wyjścia z metody:

```
// Konstrukcja try-finally - obecnie nie jest to już najlepszy sposób zamykania zasobów!
static String firstLineOfFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

Choć konstrukcja nie wygląda źle, wszystko zaczyna się komplikować, gdy dodamy drugi zasób:

```
// Konstrukcja try-finally wygląda brzydko, gdy jest stosowana z więcej niż
// jednym zasobem!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

Choć niektórym trudno będzie w to uwierzyć, to nawet dobrzy programiści mylą się w takiej sytuacji w większości przypadków. Sam użyłem błędnej wersji na stronie 88 książki *Java Puzzlers* [Bloch05] i przez lata nikt tego nie zauważył. W 2007 roku dwie trzecie użyć metody `close` w bibliotekach Javy było napisanych nieprawidłowo.

Nawet poprawny kod dotyczący zamykania zasobów za pomocą konstrukcji `try-finally`, czyli taki jak przedstawiony powyżej, ma pewne niedociągnięcia. Kod w obu blokach `try` i w bloku `finally` może zgłaszać wyjątki. Na przykład w metodzie `firstLineOfFile` wywołanie `readLine` może zgłosić wyjątek z powodu błędu z urządzeniem fizycznym, a późniejsze wywołanie `close` może się nie udać z tych samych powodów. W takich okolicznościach drugi wyjątek całkowicie zaciera pierwszy. Pierwszy wyjątek nie pojawi się w stosie wywołań wyjątku, co potrafi znacząco utrudnić debugowanie rzeczywistych systemów — najczęściej zależy nam właśnie na znalezieniu pierwszego wyjątku, aby móc zdiagnozować źródło problemu. Choć można napisać kod, który stłumi drugi wyjątek na rzecz pierwszego, prawie nikt tego nie robi, bo jest to kod bardzo rozwlekły.

Wszystkie wymienione problemy udało się rozwiązać jak za dotknięciem czarodziejskiej różdżki, gdy wraz z Javą 7 pojawiła się konstrukcja `try` z zasobami [JLS, 14.20.3]. Aby móc z niej skorzystać, zasób musi implementować interfejs `AutoCloseable`, który składa się z jednej metody `close` zwracającej `void`. Obecnie wiele klas i interfejsów bibliotek Javy i bibliotek zewnętrznych implementuje lub rozszerza wspomniany interfejs. Jeśli sam piszesz klasę reprezentującą zasób wymagający zamknięcia, powinna ona implementować interfejs `AutoCloseable`.

Oto jak wyglądałby nasz pierwszy przykład w wersji z konstrukcją `try` z zasobami:

```
// Konstrukcja try z zasobami - najlepszy sposób na zamykanie zasobów!
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
        new FileReader(path))) {
        return br.readLine();
    }
}
```

Oto wygląd drugiego przykładu w wersji z konstrukcją `try` z zasobami:

```
// Konstrukcja try z wieloma zasobami - krótko i konkretnie
static void copy(String src, String dst) throws IOException {
    try (InputStream in = new FileInputStream(src);
        OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Wersje try z zasobami nie tylko są krótsze od swoich oryginałów, ale również zapewniają znacznie lepszą diagnostykę. Rozważmy metodę `firstLineOfFile`. Jeśli wyjątek zostanie zgłoszony zarówno przez wywołanie `readCall`, jak i (niewidzialne) wywołanie `close`, drugi wyjątek zostanie **stłumiony** na rzecz pierwszego. W rzeczywistości wiele wyjątków może zostać stłumionych, aby tylko zachować wyjątek, na którym tak naprawdę nam zależy. Stłumione wyjątki nie są jednak porzucane — pojawią się w prezentowanym stosie wywołań wraz z adnotacją, że zostały stłumione. Mamy do nich również dostęp w sposób programowy poprzez metodę `getSuppressed`, która pojawiła się w interfejsie `Throwable` w Javie 7.

Klauzule `catch` mogą być stosowane dla konstrukcji try z zasobami w taki sam sposób jak dla zwykłych konstrukcji try-finally. Pozwala to na obsługę wyjątków bez uciekania się do dodatkowych zagnieżdżeń. Poniższy przykład modyfikuje wcześniejszą metodę `firstLineOfFile` w taki sposób, aby nie zgłaszała wyjątku, ale zwracała pewną domyślną wartość, jeśli nie udało się otworzyć pliku lub odczytać z niego danych:

```
// Konstrukcja try z zasobami wraz z klauzulą catch
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
        new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

Nie powinno być już żadnych wątpliwości — zawsze używaj konstrukcji try z zasobami zamiast konstrukcji try-finally, jeśli pracujesz z zasobami wymagającymi zamknięcia. Wynikowy kod jest krótszy i bardziej czytelny, a generowane wyjątki bardziej użyteczne. Konstrukcja try z zasobami pozwala pisać poprawny kod używający zasobów, które muszą być zamykane, co w przypadku konstrukcji try-finally w praktyce nie jest możliwe.

Metody wspólne dla wszystkich obiektów

Choć na podstawie klasy `Object` można tworzyć obiekty, została ona zaprojektowana przede wszystkim jako klasa bazowa dla dziedziczenia. Wszystkie jej metody, które nie są oznaczone jako ostateczne (`equals`, `hashCode`, `toString`, `clone` i `finalize`), mają jawnie zdefiniowane **ogólne założenia**, ponieważ są przeznaczone do przesłaniania w klasach pochodnych. W gestii klasy przesłaniającej te metody jest trzymanie się ogólnych założeń, ponieważ ich zignorowanie uniemożliwi innym klasom (np. `HashMap` i `HashSet`), korzystającym z tych założeń, prawidłowe współdziałanie z nią.

W rozdziale tym opiszemy, w jaki sposób i kiedy należy przesłaniać metody klasy `Object`. Nie omawiamy tu metody `finalize`, ponieważ została ona omówiona w temacie 8. Choć metoda `Comparable.compareTo` nie jest metodą klasy `Object`, jest ona tutaj omówiona, ponieważ ma charakter podobny do pozostałych opisanych tu metod.

Temat 10. Zachowanie założeń w trakcie przeddefiniowywania metody equals

Przeddefiniowanie metody equals wydaje się proste, choć można przy tym popełnić kilka błędów, których konsekwencje mogą być znaczne. Najprostszą metodą unikania problemów jest unikanie przesłaniania metody equals. W takiej sytuacji, jeżeli nie przeddefiniujemy metody equals, każdy obiekt jest równy jedynie samemu sobie. Jest to prawidłowe działanie, jeżeli zachodzi jeden z poniższych przypadków.

- **Każdy obiekt klasy jest ze swojej natury unikalny.** Jest to prawidłowe dla klas, które reprezentują jednostki aktywne, a nie wartości, na przykład Thread. Implementacja equals, znajdująca się w klasie Object, działa doskonale dla takich klas.
- **Nie musisz sprawdzać, czy klasa zapewnia test „logicznej równoważności”.** Na przykład klasa java.util.regex.Pattern mogłaby przesłaniać metodę equals w celu sprawdzenia, czy dwa obiekty klasy Pattern będą dotyczyły dokładnie tego samego wyrażenia regularnego, ale projektanci tej klasy nie uważali, że takie działanie byłoby przydatne. W takim przypadku implementacja metody equals, odziedziczonej z klasy Object, jest odpowiednia.
- **Klasa bazowa posiada już przesłoniętą metodę equals i jej działanie jest prawidłowe dla danej klasy.** Na przykład większość implementacji klasy Set dziedziczy implementację equals z klasy AbstractSet. Klasa List dziedziczy z AbstractList, a Map dziedziczy z AbstractMap.
- **Klasa jest prywatna lub jest prywatna w ramach pakietu i jesteś pewien, że metoda equals nie zostanie nigdy wywołana.** Istnieją opinie, że metoda equals powinna być w tym przypadku przeddefiniowana, aby można było zabezpieczyć się przed jej przypadkowym wywołaniem.

```
@Override public boolean equals(Object o) {  
    throw new AssertionError(); // metoda nie jest nigdy wywoływana  
}
```

Kiedy należy przesłonić metodę Object.equals? Gdy klasa może określić **logiczną równoważność**, która różni się od zwykłej identyczności obiektów, a klasa bazowa nie zawiera metody equals, realizującej prawidłowo to porównanie. Zwykle jest to realizowane w **klasach o zdefiniowanych wartościach**, na przykład Integer lub String. Programista, porównujący referencje do obiektów za pomocą metody equals, spodziewa się, że sprawdzi, czy są one swoimi logicznymi odpowiednikami, a nie,

czy odwołują się do tego samego obiektu. Przesłanianie metody `equals` nie tylko pozwala spełnić oczekiwania programisty, ale również pozwala na wykorzystywanie obiektów jako klucze odwzorowania lub elementy zbiorów o przewidywalnym i oczekiwanym zachowaniu.

Jednym z rodzajów klas, które **nie** wymagają przesłaniania metody `equals`, jest klasa korzystająca z kontroli instancji (temat 1.) w celu zapewnienia, że w systemie istnieje co najwyżej jeden obiekt o danej wartości. Do tej kategorii zaliczają się typy wyliczeniowe (temat 34.). W przypadku takich klas równoważność logiczna jest taka sama jak identyfikacja obiektu, więc metoda `equals` klasy `Object` działa tak samo jak logiczna metoda `equals`.

Gdy przeddefiniowujesz metodę `equals`, powinieneś stosować się do ogólnych zasad. Poniżej przedstawiamy założenia, skopiowane ze specyfikacji `Object`.

Metoda `equals` realizuje **relację równoważności**:

- **Jest zwrotna** — dla dowolnej niepustej wartości referencyjnej x , $x.equals(x)$ musi zwracać `true`.
- **Jest symetryczna** — dla dowolnych niepustych wartości referencyjnych x i y , $x.equals(y)$ musi zwracać `true` wtedy i tylko wtedy, gdy $y.equals(x)$ zwraca `true`.
- **Jest przechodnia** — dla dowolnych niepustych wartości referencyjnych x i y oraz z , jeżeli $x.equals(y)$ zwraca `true` i $y.equals(z)$ zwraca `true`, to $x.equals(z)$ musi zwracać `true`.
- **Jest spójna** — dla dowolnych niepustych wartości referencyjnych x i y wielokrotne wywołania $x.equals(y)$ zwracają konsekwentnie `true` lub konsekwentnie `false`, przy założeniu, że nie została zmodyfikowana żadna dana wykorzystywana do porównywania obiektów.
- Dla dowolnych niepustych referencji x różnych od `null` $x.equals(null)$ musi zwracać `false`.

Jeżeli nie jesteś zbyt dobry w matematyce, definicja ta może wydawać Ci się skomplikowana, ale nie możesz jej ignorować! Jeżeli naruszysz tę relację, może się okazać, że program działa nieprawidłowo lub zatrzymuje się z komunikatem o błędzie; bardzo trudno będzie wówczas odszukać źródło problemów. Parafrazując Johna Donne’a, żadna klasa nie jest samotną wyspą. Obiekty jednej klasy są często przekazywane do innej. Wiele klas, w tym wszystkie klasy kontenerowe, zakłada, że obiekty przekazane do nich prawidłowo realizują założenia dla metody `equals`.

Gdy znasz już konsekwencje naruszenia założeń metody `equals`, możemy je szczegółowo opisać. Choć wyglądają one nieprzyjaźnie, nie są w rzeczywistości zbyt skomplikowane. Gdy się je dobrze pozna, dosyć łatwo się do nich dostosować.

Czym jest relacja równoważności? Mówiąc prostym językiem, to operator, który dzieli zbiór elementów na podzbiory, których elementy uważa się za równe sobie. Podzbiory te znane są jako **klasy równoważności**. Aby metoda `equals` była użyteczna, wszystkie elementy w każdej z klas równoważności muszą być z perspektywy użytkownika w pełni wymienne. Przeanalizujemy teraz pięć wymienionych wcześniej założeń.

Zwrotność. Pierwsze założenie oznacza, że obiekt musi być równy samemu sobie. Trudno wyobrazić sobie przypadkowe naruszenie tej zasady. Jeżeli uda Ci się jednak naruszyć to założenie i dodasz taki obiekt do kolekcji, to nieomal na pewno nie uda się wyszukanie tego obiektu za pomocą metody `contains` w danej kolekcji.

Symetria. Drugie założenie mówi, że dwa obiekty muszą zgodnie określać, że są sobie równe. W przeciwieństwie do pierwszego założenia nietrudno można sobie wyobrazić przypadkowe naruszenie tej zasady. Przyjrzyj się następującej definicji klasy, która implementuje obsługę tekstów niezależnych od wielkości znaków. Wielkość znaków jest zachowywana przez `toString`, ale ignorowana przy porównaniach `equals`:

```
// Błąd - naruszona symetria!
public final class CaseInsensitiveString {
    private String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Błąd - naruszona symetria!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString)o).s);
        if (o instanceof String) // Współpraca jednokierunkowa!
            return s.equalsIgnoreCase((String)o);
        return false;
    }

    ... // pozostała część pominięta
}
```

Dobrze przemyślana metoda `equals` naiwnie próbuje operować na zwykłych ciągach. Załóżmy, że mamy jeden ciąg bez kontroli wielkości liter i drugi standardowy:

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polski");
String s = "polski";
```

Tak jak się tego spodziewaliśmy, `cis.equals(s)` zwraca `true`. Problem związany jest z tym, że metoda `equals` z klasy `CaseInsensitiveString` może operować na zwykłych ciągach, ale metoda `equals` z klasy `String` jest nieświadoma istnienia klasy `CaseInsensitiveString`. Dlatego `s.equals(cis)` zwraca `false`, co jest naruszeniem symetrii. Załóżmy, że umieścimy nasz ciąg w kolekcji:

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);
```

Co zwróci teraz metoda `list.contains(s)`? Nie można jednoznacznie odpowiedzieć. W bieżącej implementacji OpenJDK metoda ta zwróci `false`, ale jest to efekt uboczny w tej implementacji. W innych implementacjach metoda ta może zwrócić `true` lub zgłosić wyjątek. **Gdy naruszysz założenia metody `equals`, nie możesz być pewien, w jaki sposób inne obiekty będą współpracować z Twoim obiektem.**

Aby wyeliminować ten problem, należy usunąć z metody `equals` operacje na obiektach `String`. Po wykonaniu tej operacji można przebudować metodę na pojedynczą instrukcję `return`:

```
@Override public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
}
```

Przechodniość. Trzecie założenie mówi, że jeżeli pierwszy obiekt jest równy drugiemu, a drugi trzeciemu, to pierwszy obiekt musi być równy trzeciemu obiektowi. I tym razem nietrudno wymyślić sytuację, w której zasada ta zostanie naruszona. Rozważmy przykład programisty tworzącego podklasę, która dodaje nowe **ustawienie** do istniejących w klasie bazowej. Informacja dodawana w podklasie wpływa na wyniki porównania za pomocą metody `equals`. Rozpocznijmy od prostej, niezmiennej klasy, reprezentującej punkt:

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }

    ... // pozostała część pominięta
}
```

Teraz rozszerzymy przedstawioną klasę, dodając informację o kolorze punktu:

```
public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // pozostała część pominięta
}
```

Jak powinna wyglądać metoda equals? Jeżeli pominiemy ją całkowicie, jej implementacja będzie odziedziczona po klasie Point i podczas porównania nie będzie brana pod uwagę informacja o kolorze. Choć nie narusza to zasad dla metody equals, jest jednak nie do zaakceptowania. Załóżmy, że napiszemy metodę equals, która zwraca true tylko wtedy, gdy jej argumentem jest inny kolorowy punkt o tym samym położeniu i kolorze.

```
// Błąd - naruszona symetria!
@Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint)o).color == color;
}
```

Problem w tym, że korzystając z tej metody, możemy otrzymać różne wyniki, porównując zwykły punkt z punktem kolorowym i odwrotnie. Poprzednie porównanie ignorowało kolor, natomiast bieżąca implementacja zawsze będzie zwracała false, ponieważ nieprawidłowy jest typ argumentu. Aby się o tym przekonać, utwórzmy jeden punkt zwykły i jeden punkt kolorowy:

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

Dla tych danych p.equals(cp) zwraca true, natomiast cp.equals(p) zwraca false. Możesz próbować poprawić ten błąd, ignorując kolor w metodzie ColorPoint.equals w przypadku porównywania różnych typów:

```
// Błąd - naruszona przechodność!
@Override public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // jeżeli o jest zwykłym punktem, przeprowadź porównanie bez
    // uwzględniania koloru
    if (!(o instanceof ColorPoint))
        return o.equals(this);
}
```

```

        // jeżeli o jest punktem kolorowym, przeprowadź pełne porównanie
        return super.equals(o) && ((ColorPoint)o).color == color;
    }

```

Podejście to zapewnia symetrię, ale kosztem przechodniości:

```

ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);

```

Tym razem `p1.equals(p2)` i `p2.equals(p3)` zwracają `true`, natomiast `p1.equals(p3)` zwraca `false`, co narusza zasadę przechodniości. Pierwsze dwa porównania nie uwzględniają koloru, natomiast trzecie go uwzględnia.

Podejście to może doprowadzić do nieskończonej rekurencji — przypuśćmy, że istnieją dwie podklasy klasy `Point`, np. `ColorPoint` i `SmellPoint`, a każda z nich zawiera tego rodzaju metodę `equals`. W takiej sytuacji wywołanie `myColorPoint.equals(mySmellPoint)` spowoduje zgłoszenie wyjątku `StackOverflowError`.

Jakie jest więc prawidłowe rozwiązanie? Jest to fundamentalny problem relacji równoważności w językach obiektowych. **Nie ma łatwego sposobu na rozszerzanie definicji klasy o nowe składniki z zachowaniem zasad zdefiniowanych dla metody `equals`, chyba że rezygnujemy z zalet abstrakcji obiektowej.**

Można spotkać się z opinią, że można rozszerzać klasę konkretyzowaną o nowe komponenty, zachowując założenia metody `equals` przez wykorzystanie testu `getClass` w miejsce `instanceof` używanego w metodzie `equals`:

```

// Błąd - narusza zasadę podstawiania Liskov
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}

```

Przynosi to efekt wyłącznie przy porównywaniu obiektów mających te same klasy implementujące. Choć nie wydaje się to niebezpieczne, konsekwencje mogą być nie do zaakceptowania: instancja podklasy klasy `Point` nadal jest typu `Point` i jako taka musi funkcjonować, ale w tym podejściu tak się nie dzieje! Załóżmy, że chcemy napisać metodę sprawdzającą, czy punkt całkowity znajduje się na okręgu jednostkowym. Przykładowa implementacja jest przedstawiona poniżej:

```

// Inicjalizacja UnitCircle. Dodanie wszystkich obiektów Point okręgu jednostkowego
private static final Set<Point> unitCircle = Set.of(
    new Point( 1, 0), new Point( 0, 1),
    new Point(-1, 0), new Point( 0, -1));

```

```
public static boolean onUnitCircle(Point p) {
    return unitCircle.contains(p);
}
```

Choć być może nie jest to najszybszy sposób implementacji tej funkcji, to jednak działa prawidłowo. Załóżmy jednak, że chcemy rozszerzyć klasę `Point` w taki sposób, że nie będą dodane do niej komponenty, na przykład aby konstruktor śledził liczbę utworzonych obiektów:

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
        new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public int numberCreated() { return counter.get(); }
}
```

Zasada podstawiania Liskov mówi, że każda ważna właściwość typu powinna być również przechowywana w typach pochodnych, dzięki czemu każda metoda napisana dla typu powinna równie dobrze działać na typach pochodnych [Liskov87]. To bardziej formalna wersja wcześniejszego stwierdzenia, że podklasa `Point` (np. `CounterPoint`) to nadal klasa `Point`, więc musi zachowywać się jak takowa. Załóżmy jednak, że prześlemy obiekt `CounterPoint` do metody `onUnitCircle`. Jeżeli klasa `Point` będzie korzystała z metody `equals` bazującej na `getClass`, to metoda `onUnitCircle` zwróci `false`, niezależnie od wartości `x` oraz `y` obiektu `CounterPoint`. Dzieje się tak, ponieważ kolekcje takie jak `HashSet`, wykorzystywane w metodzie `onUnitCircle`, korzystają z `equals` do sprawdzania zawartości, a nie ma obiektu `CounterPoint` równego jakimkolwiek obiektowi `Point`. Jeżeli jednak skorzystamy z odpowiedniej metody `equals` korzystającej z `instanceof` na obiekcie `Point`, ta sama metoda `onUnitCircle` będzie działała doskonale z obiektami `CounterPoint`.

Ponieważ nie istnieje satysfakcjonujący sposób na rozszerzenie klasy konkretyzowanej w celu dodania komponentu, można zastosować obejście. Należy wykorzystać poradę z tematu 18., „Zastępowanie dziedziczenia kompozycją”. Zamiast dziedziczyć `ColorPoint` po `Point`, można utworzyć pole prywatne `Point` i publiczną metodę `view` (temat 6.), która zwraca punkt na tej samej pozycji, co punkt kolorowy:

```
// Dodanie komponentu bez naruszania zasad metody equals
public class ColorPoint {
    private final Point point;
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }
}
```



```

    /**
     * Zwraca widok typu Point na bieżący kolorowy punkt.
     */
    public Point asPoint() {
        return point;
    }

    @Override public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }

    ... // pozostała część pominięta
}

```

Istnieje kilka klas w bibliotekach języka Java, które rozszerzają klasy bazowe o dodatkowe atrybuty. Na przykład klasa `java.sql.Timestamp` dziedziczy po klasie `java.util.Date`; dodaje pole `nanoseconds`. Implementacja metody `equals` dla klasy `Timestamp` narusza zasadę przechodniości i może powodować nieprawidłowe działanie, jeżeli obiekty `Timestamp` i `Date` zostaną użyte w tej samej kolekcji lub związane w inny sposób z pozostałymi obiektami aplikacji. Klasa `Timestamp` jest opatrzona stosownym komentarzem, ostrzegającym programistę przed mieszaniem tych obiektów. Jednak klasa `Timestamp` jest przykładem anomalii i nie powinno się na niej wzorować.

Należy zwrócić uwagę, że bez naruszania założeń dla metody `equals` **można** dodawać atrybuty do podklas dziedziczących z klas **abstrakcyjnych**. Jest to ważne dla hierarchii klas przedstawianych w temacie 23. Na przykład możesz mieć klasę abstrakcyjną `Shape` bez dodatkowych atrybutów, w podklasie `Circle` dodać pole `radius`, a w podklasie `Rectangle` pola `length` i `width`. Przedstawione tutaj naruszenie założeń nie wystąpi, ponieważ nie ma możliwości utworzenia obiektu klasy bazowej.

Spójność. Czwarte wymaganie dla metody `equals` określa, że jeżeli dwa obiekty są sobie równe, muszą zachowywać tę równość aż do zmodyfikowania jednego (lub obu) z obiektów. Oczywiście obiekty zmienne mogą być równe innym obiektom, natomiast obiekty niezmiennicze nie mogą. W czasie tworzenia klasy należy dokładnie przeanalizować, czy obiekty będą zmienne, czy niezmiennicze (temat 17.). Jeżeli uznasz, że będą to obiekty niezmiennicze, należy tak napisać kod, aby metoda `equals` wymuszała równość takich samych obiektów i nierówność różnych przez cały czas istnienia obiektu.

Niezależnie od tego, czy klasa jest niezmienna, czy nie, **metoda `equals` nie powinna korzystać z zawodnych zasobów**. Niezmiennie trudno jest spełnić wymagania spójności, jeżeli naruszymy tę zasadę. Na przykład metoda `equals` z `java.net.URL`

korzysta z porównania adresów IP hosta skojarzonego z danym URL. Przekształcenie nazwy hosta na adres IP może wymagać dostępu do sieci i nie jest gwarantowane, że otrzymane wyniki będą za każdym razem takie same. Może to spowodować, że metoda `equals` klasy URL naruszy zasady i będzie powodować problemy w działaniu aplikacji. Niestety, działanie to nie może zostać zmienione z powodu wymagania zachowania zgodności. Aby uniknąć tego rodzaju problemów, metoda `equals` powinna wykonywać realistyczne obliczenia na obiektach znajdujących się w pamięci.

Różność od wartości null. Ostatnie wymaganie nie ma formalnej nazwy. Oznacza ono, że wszystkie obiekty muszą być różne od wartości `null`. Trudno jest sobie wyobrazić przypadkowe zwrócenie wartości `true` w odpowiedzi na wywołanie `o.equals(null)`, ale nietrudno przy takiej okazji przypadkowo wygenerować wyjątek `NullPointerException`. Ogólne zasady zabraniają takiego działania. Wiele klas posiada metody `equals`, zawierające zabezpieczenie przed porównaniem z wartością `null`.

```
@Override public boolean equals(Object o) {  
    if (o==null)  
        return false;  
    ...  
}
```

Test taki nie jest konieczny. Aby sprawdzić równość z argumentem, metoda `equals` musi rzutować argument na właściwy typ, aby mogły być wywołane odpowiednie metody lub można było się odwołać do pól obiektu. Przed operacją rzutowania metoda musi skorzystać z operatora `instanceof` do sprawdzenia, czy argument jest właściwego typu.

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof MyType))  
        return false;  
    MyType mt = (MyType) o;  
    ...  
}
```

Jeżeli kontrola taka zostanie pominięta i do metody `equals` zostanie przekazany argument niewłaściwego typu, zgłosi ona wyjątek `ClassCastException`, co narusza założenia dla tej metody. Jeżeli jednak pierwszym argumentem operatora `instanceof` będzie wartość `null`, zwróci on wartość `false`, niezależnie od typu drugiego argumentu [JLS, 15.19.2]. Dzięki temu, że kontrola typów zwróci `false`, jeżeli przekazaną wartością będzie `null`, nie musimy tworzyć dodatkowego testu.

Łącząc razem te zasady, stworzyliśmy przedstawioną poniżej receptę na utworzenie metody `equals` wysokiej jakości.

- 1. Skorzystaj z operatora == w celu sprawdzenia, czy argument jest referencją do tego samego obiektu.** Jeżeli tak jest, zwróci on wartość `true`. Służy to tylko poprawie wydajności, ale nie ma nic gorszego niż kosztowne porównanie.
- 2. Użyj operatora instanceof do sprawdzenia, czy argument jest odpowiedniego typu.** Jeżeli nie, operator zwróci `false`. Najczęściej prawidłowym typem jest klasa, w której zdefiniowana jest ta metoda. Czasami klasa implementuje jakiś interfejs. Jeżeli `equals` zapewnia prawidłowe porównanie klas implementujących ten interfejs, należy skorzystać z nazwy interfejsu. Interfejsy kolekcji `Set`, `List`, `Map` i `Map.Entry` spełniają ten warunek.
- 3. Rzutuj argument na właściwy typ.** Ponieważ rzutowanie to poprzedzone było kontrolą typów za pomocą `instanceof`, gwarantuje to prawidłowe wykonanie operacji.
- 4. Dla każdego „ważnego” pola w klasie sprawdź, czy wartość tego pola odpowiada wartości w obiekcie przekazanym jako argument.** Jeżeli wszystkie te porównania zakończą się sukcesem, należy zwrócić `true`. Jeżeli typ sprawdzany w kroku 2. jest interfejsem, musisz odwoływać się do ważnych pól poprzez metody interfejsu. Jeżeli typ ten jest klasą, w zależności od dostępności pól możesz odwoływać się do nich bezpośrednio.

Dla typów prostych, różnych od `float` i `double`, do porównania należy skorzystać z operatora `==`; dla pól zawierających referencje do obiektów należy rekurencyjnie wywołać ich metody `equals`. Dla pola typu `float` użyj metody `Float.compare` ↪ `(float, float)`, natomiast dla pola typu `double` użyj metody `Double.compare` ↪ `(double, double)`. Takie specjalne traktowanie liczby typu `float` i `double` jest spowodowane istnieniem wartości `Float.NaN`, `-0.0f` i analogicznych stałych dla liczb `double` (szczegóły znajdziesz w JLS 15.21.1 lub dokumentacji metody `Float.equals`). Choć można by porównać pola `float` i `double` za pomocą metod statycznych `Float.equals` i `Double.equals`, wiązałoby się to z narzutem systemu *autoboxing*, czyli zmniejszałoby wydajność rozwiązania. W przypadku pól będących tablicami należy zastosować te wskazówki dla każdego elementu. Jeżeli każdy element w tablicy jest ważny, można użyć metod `Arrays.equals`.

Niektóre pola przeznaczone na referencje obiektów mogą mieć wartości `null`. Aby zapobiec możliwości zgłoszenia wyjątku `NullPointerException` przy porównywaniu takich pól, należy skorzystać z metody statycznej `Objects.equals(Object, Object)`.

W przypadku niektórych klas, na przykład przedstawionej wcześniej `CaseInsensitive` ↪ `String`, porównania wartości pól są bardziej skomplikowane i nie polegają tylko na zastosowaniu operatora równości. W tym przypadku powinno to być oczywiste,

na podstawie specyfikacji klasy. Jeżeli zajdzie taki przypadek, można zastosować utworzenie **postaci kanonicznej** każdego obiektu, dzięki czemu metoda `equals` będzie mogła wykonać proste porównanie postaci kanonicznej zamiast kosztownego porównania nie wprost. Technika ta jest najlepsza dla klas niezmiennych (temat 17.), ponieważ postać kanoniczna musi być uaktualniana w przypadku zmiany stanu obiektu.

Wydajność metody `equals` może się zmieniać w zależności od kolejności porównywania pól. Dla uzyskania lepszej wydajności powinieneś na początku porównywać pola, które najczęściej się różnią lub są mniej kosztowne przy porównaniu, a najlepiej gdy oba te przypadki zachodzą razem. Nie powinieneś porównywać pól, które nie wpływają na logiczny stan obiektu, na przykład pól blokad wykorzystywanych do synchronizacji. Nie musisz porównywać **pól nadmiarowych**, których wartości mogą być wyliczone na podstawie pól znaczących. Jednak porównując takie pola, możesz zwiększyć wydajność metody `equals`. Jeżeli istnieje pole, które zawiera wartość będącą opisem całego stanu obiektu, porównanie takich pól pozwoli oszczędzić porównywania wartości pozostałych pól obiektu. Załóżmy, że mamy klasę `Polygon` reprezentującą wielokąt i buforujemy obszar. Jeżeli dwa wieloboki mają nierówne obszary, nie warto zajmować się porównywaniem ich krawędzi i wierzchołków.

Gdy zakończysz pracę nad metodą `equals`, sprawdź, czy jest symetryczna, przechodnia i spójna. Oczywiście nie tylko zastanawiaj się nad tym, ale również napisz sprawdzający to test jednostkowy. Możesz go pominąć, jeśli użyłeś do wygenerowania metody `equals` mechanizmu `AutoValue` (patrz dalsza część podrozdziału). Jeżeli warunki te nie są spełnione, sprawdź, dlaczego się tak stało, i odpowiednio zmodyfikuj kod metody. Oczywiście, nasza metoda `equals` musi spełniać pozostałe dwie właściwości (zwrotność i różność od `null`), ale zwykle są one automatycznie spełnione.

Przykładem metody `equals` skonstruowanej zgodnie z przedstawionym planem jest wersja zawarta w prostej klasie `PhoneNumber` przedstawionej poniżej:

```
// Klasa z typową metodą equals
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "nr kierunkowy");
        this.prefix = rangeCheck(prefix, 999, "prefiks");
        this.lineNum = rangeCheck(lineNum, 9999, "numer linii");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }
}
```

```

@Override public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof PhoneNumber))
        return false;
    PhoneNumber pn = (PhoneNumber)o;
    return pn.lineNum == lineNum && pn.prefix == prefix
        && pn.areaCode == areaCode;
}
... // pozostała część pominięta
}

```

Na koniec przedstawimy jeszcze kilka uwag.

- **Zawsze należy przeddefiniować metodę hashCode w przypadku przeddefiniowania metody equals** (temat 11.).
- **Nie próbuj być zbyt sprytny.** Jeżeli tylko sprawdzisz równość pól, łatwo spełnić założenia metody equals. Jeżeli będziesz zbyt agresywnie sprawdzał równoważność, łatwo możesz popaść w tarapaty. Nie powinno się stosować żadnych form utożsamiania. Na przykład klasa File nie powinna próbować porównywać łączy symbolicznych odwołujących się do tego samego pliku. I faktycznie tego nie czyni
- **Nie korzystaj z typu innego niż Object w deklaracji metody equals.** Dostyc często można spotkać programistów, którzy napisali metodę equals, wyglądającą tak, jak metoda przedstawiona poniżej, i tracą wiele godzin na wykrycie przyczyny jej nieprawidłowego działania.

```

// Błąd - typem parametru musi być Object!
public boolean equals(MyClass o) {
    ...
}

```

Problem tkwi w tym, że metoda ta nie **przesłania** metody Object.equals, której argumentem jest obiekt typu Object, ale ją **przeciąża** (temat 52.). Dodawanie metod equals ze ścisłym sprawdzaniem typów, dostarczanych dodatkowo oprócz normalnej postaci, jest nie do zaakceptowania, bo może doprowadzić adnotację @Override w podklasach do dawania fałszywych oznak poprawności i sugerowania pełnego bezpieczeństwa.

Spójne stosowanie adnotacji @Override, w sposób przedstawiony w tym temacie, zabezpiecza przed popełnieniem takiej pomyłki (temat 40.). Metoda equals nie skompile się i komunikat o błędzie od razu wskaże błąd.

```

// Błąd nadal istnieje, ale przynajmniej kodu nie uda się skompilować
@Override public boolean equals(MyClass o) {
    ...
}

```

Pisanie i testowanie metod `equals` (i `hashCode`) jest żmudne, a wynikowy kod jest czasem naprawdę kosmiczny. Doskonałą alternatywą dla samodzielnego pisania i testowania tych metod jest dostępny na zasadach *open source* framework `AutoValue` stworzony przez firmę Google, który automatycznie generuje odpowiednie metody, a jest wyzwalany pojedynczą adnotacją klasy. W większości sytuacji metody generowane przez `AutoValue` są identyczne z tymi, które napisalibyśmy samodzielnie.

Zintegrowane środowiska programistyczne (IDE) również oferują mechanizmy generowania metod `equals` i `hashCode`, ale wynikowy kod jest bardziej rozwlekły i mniej czytelny od wersji oferowanej przez `AutoValue`, a co gorsza, nie śledzi automatycznie zmian w klasie, więc wymaga pisania testów. Z drugiej strony, jeśli zamierzamy samodzielnie napisać obie metody, lepiej powierzyć to zadanie IDE, bo nie popełniają one, w odróżnieniu od ludzi, bezmyślnych pomyłek w trakcie implementacji.

Podsumowując — nie przesłaniaj metody `equals`, chyba że musisz. W większości przypadków implementacja odziedziczona po `Object` robi dokładnie to, czego oczekujesz. Jeśli jednak przesłaniasz `equals`, pamiętaj o porównaniu wszystkich znaczących pól klasy w sposób zachowujący wszystkie pięć zasad kontraktu metody `equals`.

Temat 11. Przedefiniowywanie metody `hashCode` wraz z `equals`

Jeżeli w klasie została przedefiniowana metoda `equals`, musisz również przeddefiniować `hashCode`. Jeśli tego nie zrobisz, naruszasz ogólne zasady dla metody `hashCode` i uniemożliwiasz prawidłowe współdziałanie klasy z wszystkimi kolekcjami korzystającymi z kodowania mieszającego, na przykład `HashMap` i `HashSet`. Poniżej przytaczamy założenia ze specyfikacji klasy `Object`.

- Jeżeli metoda `hashCode` zostanie kilkakrotnie wywołana na rzecz tego samego obiektu, musi ona konsekwentnie zwracać tę samą liczbę, pod warunkiem że nie zostaną zmodyfikowane żadne dane wykorzystywane do porównywania obiektów w metodzie `equals`. Liczba ta nie musi być identyczna w przypadku ponownego uruchomienia tej samej aplikacji.
- Jeżeli metoda `equals(Object)` wykaże, że dwa obiekty są jednakowe, to metoda `hashCode` dla obu tych obiektów musi zwracać tę samą wartość.
- **Nie** jest wymagane, aby dwa różne obiekty, według porównania za pomocą metody `equals(Object)`, zwracały różne wartości metody `hashCode`. Jednak programista powinien sobie zdawać sprawę, że zwracanie różnych wartości dla różnych obiektów zwiększa szybkość operacji na tablicach mieszających.

Jeśli nie zostanie zdefiniowana metoda hashCode, naruszone zostanie drugie z kluczowych wymagań — równe obiekty muszą mieć te same kody mieszające. Dwa różne obiekty mogą być logicznie równe według metody equals, ale dla metody hashCode z klasy Object są to nadal dwa różne obiekty bez cech wspólnych. Dlatego metoda hashCode zwraca dwie pozornie losowe liczby. W założeniach natomiast powinna zwracać identyczne wartości.

Załóżmy, że będziesz chciał zastosować instancję klasy PhoneNumber z poprzedniego tematu w kontenerze HashMap:

```
Map<PhoneNumber, String> m = new HashMap<>();  
m.put(new PhoneNumber(707, 867, 5309), "Justyna");
```

Mógłbyś się spodziewać, że metoda m.get(new PhoneNumber(707, 867, 5309)) zwróci "Justyna", ale w rzeczywistości otrzymamy null. Zwróć uwagę, że zaangażowane są tutaj dwa obiekty PhoneNumber — jeden zostaje wstawiony do HashMap, a drugi — identyczny — jest używany do (próby) odczytania z kolekcji. Brak zdefiniowanej metody hashCode powoduje, że dwa identyczne obiekty mają różne wartości kluczy mieszających, co narusza założenia metody hashCode. Dlatego metoda get szuka numeru telefonu w innym kubelku tablicy mieszającej niż ten, do którego został on zapisany przez metodę put. Nawet jeżeli dwa obiekty trafią do tego samego kubelka, metoda get niemal na pewno zwróci null, ponieważ HashMap posiada optymalizację buforującą kody mieszające skojarzone z każdą pozycją i jeżeli kody mieszające nie są identyczne, nie jest sprawdzana równość obiektów.

Poprawienie tego błędu jest proste; wystarczy napisać odpowiednią metodę hashCode w klasie PhoneNumber. Jak powinna wyglądać metoda hashCode? Można bardzo łatwo napisać prawidłową, ale niezalecaną postać tej metody.

```
// Najgorsza dopuszczalna postać implementacji hashCode - nigdy nie używać!  
@Override public int hashCode() { return 42; }
```

Jest ona dopuszczalna, ponieważ dla jednakowych obiektów zwraca ona tę samą wartość klucza mieszającego. Nie jest ona zalecana, ponieważ **każdy** obiekt ma tę samą wartość klucza mieszającego. Przez to wszystkie obiekty są przypisywane do tego samego kubelka i tablica mieszająca degeneruje się do zwykłej listy. Programy, które powinny mieć liniową złożoność obliczeniową, osiągają kwadratową złożoność. W przypadku dużych tablic mieszających utrata wydajności jest ogromna.

Dobra funkcja mieszająca dla różnych obiektów będzie zwracała różne wartości. Jest to określone w trzecim założeniu dla funkcji hashCode. Idealna funkcja mieszająca powinna równomiernie rozmieszczać dowolny zbiór różnych obiektów w zakresie wszystkich możliwych wartości. Zapewnienie takiego działania jest niezmiernie trudne. Na szczęście dość łatwo można osiągnąć satysfakcjonujące przybliżenie ideału. Przyjrzyjmy się prostemu schematowi działania.

1. Zadeklaruj zmienną typu `int`, na przykład `result`, i ustaw ją na skrót `c` dla pierwszego znaczącego pola obiektu zgodnie z punktem 2.a. (Przypomnijmy, że zgodnie z poprzednim tematem pole znaczące to takie, które wpływa na wynik porównywania).
2. Dla każdego pozostałego znaczącego pola `f` w obiekcie wykonaj następujące czynności:
 - a) Oblicz wartość kodu mieszającego `c` typu `int` dla pola:
 - i. Jeżeli pole jest typu prostego, oblicz `Typ.hashCode(f)`, gdzie `Typ` to klasa osłonowa dla typu `f`.
 - ii. Jeżeli pole jest referencją do obiektu i metoda `equals` porównuje wartość tego pola poprzez rekurencyjne wywołanie metody `equals`, należy również rekurencyjnie wywołać metodę `hashCode` dla tego pola. Jeżeli potrzebne jest bardziej skomplikowane porównanie, wylicz „reprezentację kanoniczną” dla tego pola i wywołaj `hashCode` dla tej reprezentacji. Jeżeli wartością pola jest `null`, zwróć 0 (może to być inna stała, ale tradycyjnie zwracamy 0).
 - iii. Jeżeli pole jest tablicą, potraktuj każdy element jak osobne pole i policz wartość klucza mieszającego dla każdego ważnego elementu, korzystając z przedstawionych zasad. Wyniki należy połączyć w sposób opisany w punkcie 2.b. Jeśli żaden element tablicy nie jest znaczący, użyj stałej, ale najlepiej innej niż 0. Jeżeli każdy element tablicy jest znaczący, użyj metody `Array.hashCode`.
 - b) Połącz wszystkie wartości kodów mieszających `c` z punktu 2.a w następujący sposób:
$$\text{result} = 31 * \text{result} + c;$$
3. Zwróć wartość `result`.

Po napisaniu całej metody `hashCode` sprawdź, czy równoważne sobie obiekty mają te same wartości kluczy mieszających. Napisz test jednostkowy, sprawdzający, czy masz dobrą intuicję (jeśli używasz `AutoValue` do wygenerowania metod `equals` i `hashCode`, możesz pominąć pisanie testu). Jeżeli identyczne instancje mają różne klucze mieszające, odszukaj błąd i popraw go.

Można dopuścić usunięcie **nadmiarowych pól** z obliczania klucza mieszającego. Inaczej mówiąc, można bezpiecznie pominąć pole, którego wartość można wyliczyć na podstawie wartości innych pól. **Wymagane** jest pomijanie pól, które nie są brane pod uwagę przy porównywaniu obiektów. Branie takiego pola pod uwagę przy wyliczaniu wartości klucza mieszającego może spowodować naruszenie drugiego założenia dla metody `hashCode`.

Mnożenie wykonywane w punkcie 2.b powoduje, że wartość klucza mieszającego zależy od kolejności pól, co w przypadku wystąpienia klasy z wieloma podobnymi polami polepsza jakość funkcji mieszającej. Jeżeli na przykład nie byłoby wykonane takie mnożenie, to w przypadku pól typu String wszystkie anagramy miałyby te same wartości funkcji mieszającej. Mnożnik 31 został wybrany dlatego, że liczba 31 jest liczbą nieparzystą pierwszą. Jeżeli byłaby to liczba parzysta i podczas mnożenia wystąpiłoby przepełnienie, informacja mogłaby zostać utracona, ponieważ mnożenie przez dwa jest równoznaczne przesuwaniu. Korzyści ze stosowania liczb pierwszych nie są zbyt oczywiste, ale tradycyjnie do tego celu wykorzystuje się liczby pierwsze. Przyjemną własnością 31 jest możliwość zastąpienia mnożenia przesunięciem i odejmowaniem, co pozwala zwiększyć wydajność: $31 * i == (i \ll 5) - i$. Nowoczesne maszyny wirtualne wykonują automatycznie takie optymalizacje.

Wykorzystajmy ten schemat w klasie `PhoneNumber`:

```
// Typowa metoda hashCode
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

Ponieważ metoda ta zwraca wynik prostych, deterministycznych obliczeń, których danymi wejściowymi są trzy pola z obiektu `PhoneNumber`, to jednakowe numery telefonu powinny mieć te same kody mieszające. Metoda ta jest rozsądną implementacją metody `hashCode` dla klasy `PhoneNumber` porównywalną z metodą dostępną w bibliotece platformy Java. Jest prosta, odpowiednio szybka i całkiem nieźle rozsiewa różne numery telefonu do różnych kubeków.

Choć przedstawiony tu przepis zapewnia dosyć dobre klucze mieszające, nie są one idealne. Są porównywalne z wartościami oferowanymi przez funkcje mieszające dla typów dostępnych w standardowych bibliotekach Javy, więc sprawdzą się w przypadku większości użytkowników. Jeśli faktycznie chcesz najlepszej funkcji mieszającej minimalizującej kolizje, skorzystaj z `com.google.common.hash.Hashing` [Guava].

Klasa `Objects` posiada metodę statyczną, która przyjmuje dowolną liczbę obiektów i zwraca dla nich klucz mieszający. Metoda ta, nazywana `hash`, pozwala tworzyć jednowierszowe metody `hashCode` o jakości porównywalnej z zaproponowanym tu przepisem. Niestety rozwiązanie to działa znacznie wolniej, ponieważ wymaga tworzenia tablic w celu przekazania zmiennej liczby argumentów, a także automatycznego otaczania, jeśli argumentami są typy proste. To rozwiązanie można polecić tylko w sytuacji, gdy wydajność nie jest krytyczna. Oto funkcja mieszająca dla klasy `PhoneNumber` napisana z użyciem tej techniki:

```
// Jednowierszowa metoda hashCode - średnia wydajność
@Override public int hashCode() {
    return Objects.hash(lineNum, prefix, areaCode);
}
```

Jeżeli klasa jest niezmienna, a koszt obliczenia kodu mieszającego jest znaczny, możesz rozważyć buforowanie kodu mieszającego obiektu zamiast obliczania go za każdym razem, gdy jest potrzebny. Jeżeli uważasz, że większość obiektów tego typu będzie wykorzystywana jako klucze mieszające, powinieneś obliczyć wartość kodu mieszającego podczas tworzenia obiektu. W przeciwnym wypadku możesz zastosować **opóźnioną inicjalizację** w czasie pierwszego wywołania metody hashCode (temat 83.). Nasza klasa `PhoneNumber` co prawda nie wymaga takiego traktowania, ale pokażemy, w jaki sposób należy to zrobić. Pamiętaj, że wartość początkowa pola hashCode (w tym przypadku 0) nie powinna być kluczem mieszającym dla typowej instancji klasy.

```
// Późno inicjowana, buforowana metoda hashCode
private int hashCode; // automatycznie inicjalizowana wartością 0

@Override public int hashCode() {
    int result = hashCode;
    if (result == 0) {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }
    return result;
}
```

Nie daj się skusić na podnoszenie wydajności poprzez wyłączenie znaczących fragmentów obiektu z obliczania klucza mieszającego. Choć może to spowodować zwiększenie szybkości działania funkcji mieszającej, jakość jej może się zmniejszyć do tego stopnia, że korzystanie z tablic mieszających będzie bardzo powolne. W praktyce może się okazać, że funkcja mieszająca będzie wywołana dla dużej liczby obiektów różniących się w pominiętym obszarze. W takim przypadku funkcja mieszająca będzie powodowała mapowanie wszystkich tych obiektów na niewielką liczbę kodów mieszających i doprowadzi do tego, że wydajność kontenerów korzystających z kodowania mieszającego osiągnie wydajność kwadratową.

Nie jest to tylko problem teoretyczny. Funkcja mieszająca dla typu **String** w wersjach środowiska Java wcześniejszych niż 2. brała pod uwagę szesnaście znaków ciągu rozłożonych równomiernie dla tekstu z początkiem w pierwszym znaku. Dla dużych kolekcji zawierających nazwy hierarchiczne, na przykład adresy URL, działanie tej funkcji mieszającej miało takie właśnie, patologiczne skutki.

Nie zapewnij szczegółowej specyfikacji sposobu wyliczania wartości zwracanej przez metodę hashCode, aby klienci nie zaczęli wykorzystywać istniejących zależności; w ten sposób zapewniasz sobie szansę na zmianę sposobu wyliczania wartości w przyszłości. Wiele klas z bibliotek środowiska Java, takie jak String, Integer i Date, zwraca jako wynik funkcji hashCode wartość, jaką posiada dany obiekt. Nie jest to najlepsze rozwiązanie, ale błąd, z którym pozostaje nam żyć, ponieważ ogranicza możliwość ulepszenia funkcji mieszającej w przyszłych wersjach. Jeżeli nie określisz szczegółów działania funkcji mieszającej, a następnie znajdziesz w niej błąd, możesz go bezpiecznie poprawić w kolejnych wersjach, nie obawiając się, że inny obiekt korzysta z konkretnych wartości zwracanych przez funkcję mieszającą.

Podsumowując, **musisz** przesłonić metodę hashCode za każdym razem, gdy przesłaniasz metodę equals, bo w przeciwnym razie program nie będzie działał prawidłowo. Metoda hashCode musi stosować się do ogólnego kontraktu wskazanego w klasie Object oraz w miarę dobry sposób rozprawdzać nierówne sobie obiekty w zakresie dopuszczalnych wartości klucza mieszającego. Nie jest to trudne do uzyskania, choć nieco męczące, jeśli zastosuje się przepis podany kilka stron wcześniej. Jak wskazano w poprzednim temacie, biblioteka AutoValue zapewnia dobrą alternatywę dla ręcznego pisania metod equals i hashCode. Większość IDE również oferuje podobną funkcjonalność.

Temat 12. Przeddefiniowywanie metody toString

Choć klasa Object zawiera implementację metody toString, to jednak zwracany ciąg nie jest czytelny dla użytkownika programu. Składa się on z nazwy klasy, znaku @ oraz kodu mieszającego, zapisanego w postaci liczby szesnastkowej, na przykład "PhoneNumber@163b91". Założenia dla metody toString określają, że ciąg ten musi być „zwięzły, zawierający niezbędne dane w postaci czytelnej dla człowieka”. Można się spierać, czy ciąg "PhoneNumber@163b91" jest zwięzły i czytelny dla człowieka, ale na pewno jest mniej czytelny niż "707-867-5309". Kolejne założenie dla metody toString mówi, że „zaleca się przeddefiniowanie tej metody we wszystkich klasach pochodnych”. To naprawdę dobra rada!

Choć nie jest to tak ważne, jak dotrzymanie założeń dla metod equals i hashCode (temat 10. i 11.), **utworzenie dobrej implementacji toString powoduje, że klasa jest przyjemniejsza w użyciu i czyni ją wygodniejszą w debugowaniu**. Metoda toString jest wywoływana automatycznie, gdy obiekt jest przekazywany do metod println i printf, stosuje się operator złączenia ciągów (+) lub metodę assert, a także jest wyświetlana przez debugger. Nawet jeśli sami nigdy nie skorzystamy

z wywołania metody `toString`, zapewne zrobią to inni. Na przykład komponent posiadający referencję do naszego obiektu może przekazywać jego tekstową implementację do logów związanych z komunikatami o błędach. Jeśli nie przesłonimy metody `toString`, komunikat będzie bezużyteczny.

Jeżeli napiszesz dobrą metodę `toString`, łatwo będzie można generować przydatne komunikaty diagnostyczne, na przykład:

```
System.out.println("Nie udało się połączyć z  
↳ numerem: " + phoneNumber);
```

Można również generować takie komunikaty diagnostyczne bez zdefiniowania `toString`, ale nie będą one zbyt czytelne. Zalety dobrej metody `toString` są dobrze widoczne w przypadku obiektów zawierających referencje do obiektów, na przykład kolekcji. Co chciałbyś zobaczyć, wyświetlając odwzorowanie: "{Justyna=PhoneNumber@163b91}" czy "{Justyna=707-867-5309}"?

Zalecane jest, aby metoda `toString` zwracała wszystkie ważne informacje zawarte w obiekcie, o ile to wykonalne z praktycznego punktu widzenia. Tak właśnie zrobiliśmy w naszym przykładzie z numerem telefonu. Jest to niepraktyczne, jeżeli obiekt jest duży lub zawiera dane, które nie mają odpowiedniej reprezentacji znakowej. W takich przypadkach metoda `toString` powinna zwracać podsumowanie, na przykład "Książka telefoniczna Manhattanu (1487536 pozycji)" lub "Thread[main, 5, main]". Byłoby idealnie, gdyby ten ciąg był zrozumiały (przykład Thread oczywiście nie spełnia tego warunku). Szczególnie dokuczliwa jest sytuacja, gdy nie zawrzemy wszystkich istotnych informacji o obiekcie w jego tekstowej reprezentacji i otrzymamy raport z nieudanego testu w stylu:

```
Assertion failure: expected {abc, 123}, but was {abc, 123}.
```

Ważną decyzją, jaką musisz podjąć przed zaimplementowaniem metody `toString`, jest określenie formatu zwracanego ciągu i zapisanie tego formatu w dokumentacji. Zaleca się wykonanie tej czynności dla **klas posiadających wartość**, na przykład numerów telefonów lub macierzy. Zaletą określenia formatu jest to, że służy on jako standardowy, jednoznaczny i czytelny dla człowieka opis obiektu. Reprezentacja ta może być wykorzystywana w czytelnych dla człowieka dokumentach, na przykład CSV. Jeżeli określisz format, dobrze jest utworzyć odpowiedni konstruktor lub statyczną metodę fabryczną, dzięki czemu programiści będą mogli łatwo przechodzić pomiędzy obiektem i jego reprezentacją w postaci ciągu znaków. Podejście to jest zastosowane w wielu klasach bibliotek języka Java, w tym `BigInteger` i `BigDecimal`, oraz w większości klas osłonowych dla typów prostych.

Jeżeli określimy format ciągu zwracanego przez metodę `toString`, to zakładając, że klasa jest powszechnie wykorzystywana, nie możemy go dowolnie zmieniać. Programiści korzystający z tej klasy zdążyli już np. napisać kod analizujący reprezentację obiektu, tworzący ją oraz wbudowujący go w inne dane. Jeżeli zmienisz

w kolejnej wersji format ciągu, kod, który napisali, przestanie działać. Jeżeli nie określisz formatu, zabezpieczysz sobie możliwość dodania danych lub ulepszenia formatu w przyszłych wersjach.

Niezależnie od tego, czy definiujesz format, czy nie, powinieneś jasno udokumentować swoje zamiary. Jeżeli określasz format, powinieneś zrobić to dokładnie. Poniżej przedstawiamy metodę toString dla klasy PhoneNumber, przedstawionej w poprzednim podrozdziale:

```
/**
 * Zwraca ciąg reprezentujący ten numer telefonu.
 * Ciąg składa się z dwunastu znaków w formacie
 * "XXX-YYY-ZZZZ", gdzie XXX jest numerem kierunkowym, YYY prefiksem,
 * a ZZZZ numerem linii (każda litera reprezentuje jedną cyfrę).
 *
 * Jeżeli dowolna z tych trzech części jest zbyt mała,
 * aby wypełnić swoje pole, jest ono uzupełniane z lewej strony zerami.
 * Na przykład: jeżeli numerem linii jest 123, ostatnie
 * cztery znaki ciągu będą miały postać "0123".
 */
@Override public String toString() {
    return String.format("%03d-%03d-%04d",
                        areaCode, prefix, lineNumber);
}
```

Jeżeli nie chcesz określać formatu, dokumentacja powinna zawierać na przykład taki komentarz.

```
/**
 * Zwraca krótki opis tego eliksiru. Szczegóły reprezentacji nie są określone
 * i mogą się zmienić. Typowym przykładem jest:
 *
 * "[Eliksir #9: typ=miłość, zapach=terpentyna, wygląd=atrament]"
 */
@Override public String toString() { ... }
```

Po przeczytaniu takiego komentarza żaden programista, który tworzy kod korzystający ze szczegółów formatu, nie będzie mógł mieć do Ciebie pretensji w przypadku zmiany formatu.

Niezależnie od tego, czy zdefiniujesz format, czy nie, **zawsze należy umożliwić programowy dostęp do wszystkich danych, zawartych w ciągu zwracanym przez metodę toString.** Na przykład klasa PhoneNumber powinna zawierać metody umożliwiające dostęp do numeru kierunkowego, prefiksu i numeru linii. Jeżeli tego nie zrobisz, **wymuszasz** na programiście, który potrzebuje tej danej, analizę ciągu zwracanego przez toString. Oprócz obniżania szybkości działania aplikacji i niepotrzebnej pracy programisty proces taki jest wrażliwy na błędy, co skutkuje powstaniem systemu, który przestanie działać, gdy zmienisz format ciągu. Jeżeli nie zostaną zdefiniowane metody dostępowe, ciąg ten staje się de facto częścią API, nawet jeżeli zaznaczyłeś, że format może się zmienić.

Nie ma sensu pisanie metody `toString` w statycznej klasie użytkowej (temat 4.). Nie należy też jej dodawać do większości typów wyliczeniowych (temat 34.), bo w tym przypadku Java zapewnia bardzo dobrą implementację. Warto jednak napisać metodę `toString` w klasie abstrakcyjnej, gdy podklasy będą współdzieliły część reprezentacji tekstowej. Na przykład metody `toString` większości implementacji kolekcji są dziedziczone po abstrakcyjnych klasach kolekcji.

Biblioteka `AutoValue` firmy Google (omówiona w temacie 10.), automatycznie wygeneruje dla Ciebie metodę `toString`. Podobnie może to zrobić większość IDE. Rozwiązania te sprawdzają się, gdy trzeba wyświetlić zawartość każdego pola, ale nie są na tyle wyrafinowane, aby uchwycić znaczenie klasy. Nie byłoby więc odpowiednim posunięciem automatyczne generowanie metody `toString` dla klasy `PhoneNumber`, ponieważ istnieje pewien ustalony sposób reprezentacji numerów telefonów. Z drugiej strony automat byłby odpowiedni dla klasy eliksirów. Niezależnie od okoliczności każda automatycznie wygenerowana metoda `toString` jest lepsza od wersji dziedziczonej po klasie `Object`, która **nic** nie mówi o wartości obiektu.

Podsumujmy — przesłonił implementację metody `toString` z `Object` w każdej tworzonej klasie, chyba że zrobiła to już klasa nadrzędna. W ten sposób klasa stanie się bardziej przyjazna w użyciu i prostsza w debugowaniu. Metoda `toString` powinna zwrócić zwięzły i użyteczny opis obiektu w formacie możliwie najbardziej przyjaznym dla oka.

Temat 13. Rozsądne przededefiniowywanie metody `clone`

Interfejs `Cloneable` został zaprojektowany jako **interfejs mieszany** (temat 20.) dla obiektów, które chcą zaznaczyć, że pozwalają na klonowanie. Niestety, nie spełnia on dobrze tego zadania. Pierwszą wadą jest brak metody `clone`, a metoda `Object.clone` jest zabezpieczona. Nie można, bez uciekania się do **refleksji** (temat 65.), wywoływać metody `clone`, mimo iż klasa implementuje interfejs `Cloneable`. Nawet wtedy wywołanie zwrotne może się nie powieść, ponieważ nie ma gwarancji, że obiekt posiada dostępną metodę `clone`. Pomimo tych błędów udogodnienie to jest wykorzystywane na tyle często, że dobrze jest dokładnie poznać ten mechanizm. W tym podrozdziale opiszemy, w jaki sposób pisać dobre metody `clone`, kiedy można to zrobić, i krótko omówimy alternatywy.

Jak **działa** interfejs `Cloneable` (bo przecież nie zawiera metod)? Określa on działanie zabezpieczonej metody `clone` klasy `Object`. Jeżeli klasa implementuje interfejs `Cloneable`, metoda `clone` zwraca kopię obiektu (pole po polu), w przeciwnym wypadku zgłasza wyjątek `CloneNotSupportedException`. Jest to bardzo nietypowe

wykorzystanie interfejsu i nie powinno być emulowane. Normalnie, implementując interfejs, określamy operacje, jakie można wykonać na tej klasie. Jednak w przypadku `Cloneable` zmienia się sposób działania zabezpieczonej metody klasy bazowej.

Choć specyfikacja o tym nie wspomina, w **praktyce oczekuje się, że klasa implementująca `Cloneable` zapewni w pełni funkcjonalną, publiczną metodę `clone`**. Aby implementacja interfejsu `Cloneable` miała jakiegokolwiek efekty, klasa oraz wszystkie jej klasy pochodne muszą zapewniać dość skomplikowany, niemożliwy do wymuszenia i w dużej części nieudokumentowany protokół. Działa tu **mechanizm pozajęzykowy** — obiekty tworzone są bez wywołania konstruktora.

Założenia dla metody `clone` są mało wymagające. Poniżej przedstawimy te założenia, skopiowane ze specyfikacji klasy `Object`:

Tworzy i zwraca kopię tego obiektu. Dokładne znaczenie „kopii” może być uzależnione od klasy obiektu. Generalne założenie jest takie, aby dla każdego obiektu `x` wyrażenie:

```
x.clone() != x
```

zwracało wartość `true`, a wyrażenie:

```
x.clone().getClass() == x.getClass()
```

również zwracało `true`, choć nie jest to absolutnie wymagane.

Zwykle wyrażenie:

```
x.clone().equals(x)
```

zwraca wartość `true`, ale również nie jest to wymagane.

Zgodnie z konwencją obiekt zwracany przez tę metodę powinien zostać pobrany poprzez wywołanie `super.clone`. Jeśli klasa i wszystkie jej klasy nadrzędne (poza `Object`) stosują się do tej konwencji, dojdzie do sytuacji:

```
x.clone().getClass() == x.getClass()
```

Zgodnie z konwencją zwrócony obiekt powinien być niezależny od swojego oryginału. Aby osiągnąć tę niezależność, konieczna może być modyfikacja jednego lub kilku pól obiektu zwróconego przez `super.clone`.

Ten mechanizm jest bardzo podobny do łańcucha wywołań konstruktorów, ale nie jest wymuszany — jeśli metoda `clone` klasy zwróci instancję, która nie została uzyskana wywołaniem `super.clone`, ale wywołaniem konstruktora, kompilator nie zgłosi błędu. Spowoduje to jednak nieprawidłowe działanie metody `clone` w jej podklasie, jeśli ta zdecyduje się na użycie metody `super.clone` w trakcie klonowania. Jeśli klasa przesłaniająca metodę `clone` jest ostateczna, wspomnianą konwencję można bezpiecznie zignorować, bo nie będą istniały żadne podklasy. Jeśli jednak klasa ostateczna posiada metodę `clone`, która nie wywołuje `super.clone`, nie ma powodu, by implementowała interfejs `Cloneable`, bo i tak nie korzysta z zachowania metody `clone` z `Object`.

Założmy, że chcesz zaimplementować interfejs `Cloneable` w klasie, której klasa bazowa zawiera odpowiednią metodę `clone`. W zależności od natury klasy obiekt zwrócony przez `super.clone()` może być bliski temu, który chcemy ewentualnie zwrócić lub nie. Obiekt ten jest, z punktu widzenia klasy bazowej, w pełni funkcjonalną kopią oryginalnego obiektu. Pola zadeklarowane w Twojej klasie (o ile istnieją) mają wartości identyczne jak w powielanym obiekcie. Jeżeli wszystkie pola mają wartości proste lub referencje do niezmiennych obiektów, zwracany obiekt ma już oczekiwaną postać i nie wymaga dalszej obróbki. Takim przypadkiem jest na przykład przedstawiana w temacie 11. klasa `PhoneNumber`. Pamiętaj jednak, że **klasy niezmiennie nigdy nie powinny zapewniać metody `clone`**, bo jedynie zachęcałoby to do zbędnego kopiowania. Mając to na uwadze, w klasie `PhoneNumber` wystarczy dopisać publiczną metodę `clone`:

```
// Metoda clone dla klasy bez odnośników do zmiennego stanu
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber)super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError(); // nie może się zdarzyć
    }
}
```

Oczywiście, aby wszystko zadziało prawidłowo, deklaracja klasy `PhoneNumber` musi teraz informować o implementacji interfejsu `Cloneable`. Choć metoda `clone` z `Object` zwraca `Object`, dodana metoda `clone` zwraca `PhoneNumber`, a nie `Object`. Jest to działanie prawidłowe i zalecane, ponieważ Java obsługuje **kowariancyjne typy zwracane**, będące częścią mechanizmu typów ogólnych. Inaczej mówiąc, obecnie prawidłowe jest zwracanie przez metodę przesłaniającą typu będącego klasą pochodną po typie zwracanym przez metodę przesłanianą. Pozwala to metodzie przesłaniającej na dostarczenie większej ilości informacji o typie zwracanego obiektu i eliminuje potrzebę rzutowania przez klienta. Ponieważ `Object.clone` zwraca `Object`, w `PhoneNumber.clone` musimy rzutować wynik `super.clone()` przed jego zwróceniem. Na szczęście mamy tu gwarancję sukcesu tej operacji.

Wywołanie `super.clone` znajduje się w bloku try-catch. Wynika to z faktu, iż `Object` deklaruje, że metoda `clone` może zgłosić wyjątek `CloneNotSupportedException` ➔ `Exception`, który jest **wyjątkiem przechwytywanym**. Ponieważ `PhoneNumber` implementuje `Cloneable`, wiemy, że wywołanie `super.clone` się powiedzie. Konieczność tego dodatkowego, bezcelowego kodu weryfikującego sugeruje, że wyjątek `CloneNotSupportedException` powinien tak naprawdę być wyjątkiem czasu wykonywania (temat 71.).

Jeżeli nasz obiekt zawiera pola odwołujące się do obiektów zmiennych, skorzystanie z takiej postaci metody `clone` może mieć katastrofalne skutki. Spójrzmy na przykład klasy `Stack`, przedstawianej w temacie 7.:


```

public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // usunięcie niepotrzebnych referencji
        return result;
    }

    // sprawdzenie, czy jest miejsce na kolejny element
    private void ensureCapacity() {
        if (elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}

```

Założmy, że chcemy umożliwić klonowanie tej klasy. Jeżeli metoda `clone` jedynie zwróciłaby obiekt uzyskany przez wywołanie `super.clone`, wynikowy stos miałby właściwą wartość w polu `size`, ale pole `elements` odwoływałoby się do tej samej tablicy, co oryginalny obiekt `Stack`. Modyfikacja oryginalnego obiektu zniszczy wszystkie kopie i odwrotnie. Szybko się przekonasz, że aplikacja będzie zwracała niewłaściwe wyniki lub zgłaszała wyjątki `NullPointerException`.

Sytuacja taka nie będzie miała miejsca, jeżeli zostanie wywołany konstruktor klasy `Stack`. **W efekcie metoda `clone` będzie działała jak kolejny konstruktor. Musisz zapewnić, że nie uszkodzi oryginalnego obiektu i właściwie utworzy kopie obiektu.** Aby metoda `clone` dla klasy `Stack` działała poprawnie, musi ona skopiować dane zapisane we wnętrzu stosu. Najprostszą metodą zrealizowania takiego działania jest rekurencyjne wywołanie metody `clone` dla tablicy `elements`.

```

// Metoda clone dla klasy zawierającej referencje do stanu zmiennego
@Override public Object clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    }
}

```

```

    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}

```

Należy zwrócić uwagę, że nie musimy rzutować wyniku `elements.clone()` na `Object[]`. Wywołanie `clone` na tablicy zwraca tablicę, której typ w momencie kompilacji jest taki sam jak klonowana tablica. To preferowany sposób duplikowania tablicy. W zasadzie tablice to główny użytkownik całego mechanizmu klonowania.

Rozwiązanie takie nie będzie działać, jeżeli pole `elements` zostanie zadeklarowane jako `final`, ponieważ metoda `clone` nie będzie mogła przypisać do niego nowej wartości. Jest to fundamentalny problem — podobnie jak serializacja, **także architektura metody `clone` jest niezgodna z normalnym zastosowaniem pól `final`, odwołujących się do zmiennych obiektów**, poza przypadkami, gdy zmienne obiekty mogą być bezpiecznie współdzielone pomiędzy obiektem i jego klonem. Aby umożliwić klonowanie obiektów, być może będzie trzeba usunąć modyfikatory `final` przy niektórych polach.

Nie zawsze wystarczy rekurencyjnie wywołać metodę `clone`. Załóżmy, że piszemy metodę `clone` dla tablicy mieszającej, która wewnętrznie składa się z tablicy kubełków, które zawierają referencję do pierwszego elementu listy par klucz-wartość lub `null`, jeżeli kubełek jest pusty. Dla poprawy wydajności klasa implementuje samodzielnie listę jednokierunkową, nie korzystając z klasy `java.util.LinkedList`.

```

public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    ... // pozostała część pominięta
}

```

Założmy, że rekurencyjnie będziemy klonować kolejne kubełki z tablicy, podobnie jak to zrobiliśmy dla klasy `Stack`:

```

// Błąd - wynik jest wspólny dla obiektów
@Override public Object clone() {
    try {

```

```

        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}

```

Mimo że sklonowany obiekt posiada własną tablicę kubeków, tablica ta zawiera referencje do tych samych list, co oryginał. Może to łatwo doprowadzić do nieprzewidywalnego zachowania się aplikacji, zarówno w powielonym, jak i w oryginalnym obiekcie. Aby poprawić ten błąd, należy skopiować po kolei listy, znajdujące się we wszystkich kubkach. Poprawiony kod wygląda następująco:

```

// Rekurencyjna metoda clone dla klasy ze złożonym stanem zmiennym
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }

        // Rekurencyjne kopiowanie listy, na której początek wskazuje obiekt Entry
        Entry deepCopy() {
            return new Entry(key, value, next == null ? null :
next.deepCopy());
        }
    }

    @Override public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }

    ... // pozostała część pominięta
}

```

Prywatna klasa `HashTable.Entry` została dodana w celu zapewnienia „dogłębnego kopiowania”. Metoda `clone` klasy `HashTable` tworzy nową tablicę kubeków o odpowiednim rozmiarze i kopiuje listy znajdujące się w niepustych kubkach. Metoda „dogłębnego kopiowania” obiektu `Entry` jest rekurencyjnie wywoływana w celu skopiowania całej listy. Technika ta działa świetnie, jeżeli listy skojarzone z kubkami nie są zbyt długie, ponieważ wymaga ona jednej ramki stosu dla każdego elementu listy. Jeżeli lista jest długa, można spowodować przepełnienie się stosu. Aby ustrzec się przed tym, można zastąpić rekurencyjną metodę `deepCopy` wersją iteracyjną.

```
// Iteracyjne kopiowanie listy, na której początek wskazuje obiekt Entry
Entry deepCopy() {
    Entry result = new Entry(key, value, next);
    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);
    return result;
}
```

Innym podejściem do klonowania złożonych obiektów jest wywołanie metody `super.clone`, przypisanie wszystkim polom wynikowego obiektu początkowych wartości, a następnie wywołanie metod wysokiego poziomu, pozwalających odtworzyć stan obiektu. W przypadku naszej klasy `HashTable` polu `buckets` powinna zostać przypisana nowa tablica kubeków, a następnie — dla każdego mapowania klucz-wartość, zawartego w klonowanej tablicy mieszającej — powinna zostać wywołana metoda `put(key, value)` (nie została ona przez nas pokazana). Podejście takie pozwala zwykle napisać niewielkie, eleganckie metody `clone`, które jednak nie działają tak szybko, jak metody bezpośrednio operujące na wewnętrznych strukturach obiektów. Choć rozwiązanie to jest czyste, stanowi antytezę całej architektury `Cloneable`, ponieważ ślepo nadpisuje pole po polu istniejące już kopie wykonane przez rdzeń architektury.

Podobnie jak konstruktor, metoda `clone` nie powinna wywoływać żadnych metod, które nie są zadeklarowane jako `final` (temat 17.). Jeżeli `clone` wywoła przeddefiniowaną metodę, zostanie ona wywołana, zanim klasa pochodna, w której jest ona zdefiniowana, będzie w stanie przywrócić stan sklonowanego obiektu, co najprawdopodobniej spowoduje uszkodzenie kopii i oryginalnego obiektu. Dlatego wspomniana przez nas metoda `put(key, value)` powinna być zadeklarowana jako `private` lub `final` (jeżeli będzie prywatna, będzie prawdopodobnie metodą pomocniczą dla niefinalnej metody publicznej).

Metoda `clone` klasy `Object` jest zadeklarowana w ten sposób, że zgłasza wyjątek `CloneNotSupportedException`, ale przeddefiniowanie metody `clone` może zmienić tę deklarację. **Publiczna metoda `clone` powinna pominąć klauzulę `throws`**, ponieważ metody niezgłaszające wyjątków są łatwiejsze w użyciu (temat 71.).

Masz dwie drogi, jeśli tworzona klasa została przeznaczona do odziedziczenia (temat 19.), ale niezależnie od tego, którą z nich wybierzesz, klasa **nie** powinna implementować Cloneable. Możesz wybrać powielenie zachowania klasy Object, czyli zapewnić w pełni działającą metodę clone w wersji chronionej wraz z deklaracją zgłaszania wyjątku CloneNotSupportedException. Dzięki temu klasy pochodne będą implementować Cloneable lub nie, podobnie jak w przypadku bezpośredniego dziedziczenia po Object. Alternatywą jest **brak** implementacji działającej metody clone i zapobiegnięcie jej implementacji w podklasach przez zastosowanie poniższej wersji:

```
// Metoda method dla rozszerzalnej klasy, która nie obsługuje Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

Warto wspomnieć o jeszcze jednym szczególe. Jeżeli zdecydujemy się wykonać implementację Cloneable bezpieczną dla wątków, należy pamiętać o prawidłowej synchronizacji metody clone, podobnie jak każdej innej metody (temat 66.). Metoda clone klasy Object nie jest synchronizowana, więc nawet jeżeli jest ona wystarczająca, może zająć konieczność napisania synchronizowanej metody clone, która wywołuje super.clone().

Podsumujmy. Wszystkie klasy implementujące interfejs Cloneable powinny przeddefiniować metodę clone jako metodę publiczną zwracającą typ samej klasy. Ta publiczna metoda powinna najpierw wywołać super.clone, a następnie poprawić wszystkie pola, które tego wymagają. Najczęściej wiąże się to z kopiowaniem wszystkich zmiennych obiektów, które posiadają wewnętrzną złożoną strukturę, i zastąpieniem referencji do tych obiektów referencjami do ich kopii. Kopie takie można często wykonać rekurencyjnie, wywołując metody clone, choć nie zawsze jest to najlepsze rozwiązanie. Jeśli klasa zawiera tylko pola proste lub referencje do obiektów niezmiennych, oznacza to najczęściej, że nie trzeba poprawiać żadnych pól. Istnieją wyjątki od tej reguły. Na przykład pole reprezentujące numer seryjny, inny unikatowy identyfikator lub pole zawierające czas utworzenia obiektu wymaga modyfikacji, nawet jeżeli jest to wartość prosta lub obiekt niezmienny.

Czy tak skomplikowana metoda jest naprawdę potrzebna? Rzadko. Jeżeli rozszerzasz klasę implementującą Cloneable, nie masz wyjścia i musisz utworzyć odpowiednią metodę clone. W przeciwnym wypadku najprawdopodobniej lepiej zastosować inne mechanizmy kopiowania obiektów. **Doskonałym sposobem na kopiowanie obiektów jest udostępnienie konstruktora kopiującego lub fabryki kopiującej.** Konstruktor kopiujący jest to konstruktor, który ma jeden argument, którego typem jest klasa zawierająca ten konstruktor, na przykład:

```
// Konstruktor kopiujący
public Yum( Yum yum);
```

Innym wariantem tej samej metody jest udostępnienie statycznej metody fabrycznej (temat 1.), wywoływanej zamiast konstruktora:

```
// Fabryka kopiująca  
public static Yum newInstance(Yum yum);
```

Zastosowanie konstruktora kopiującego lub statycznej metody fabrycznej jest o wiele bardziej korzystne niż stosowanie `Cloneable` i `clone`. Konstruktor kopiujący i statyczna metoda fabryczna nie korzystają z ryzykownego, pozajęzykowego mechanizmu tworzenia obiektów, nie wymagają stosowania się do słabo udokumentowanych konwencji, nie są w konflikcie z właściwie wykorzystywanymi polami typu `final`, nie wymagają od programisty przechwytywania niepotrzebnych wyjątków oraz udostępniają klientom ustalone typy.

Dodatkowo konstruktor kopiujący lub statyczna metoda fabryczna może mieć argument typu właściwego dla interfejsu implementowanego przez klasę. Na przykład dla kolekcji ogólnego zastosowania utworzony jest konstruktor kopiujący, którego argument jest typu `Collection` lub `Map`. Konstruktor kopiujący, korzystający z interfejsu, oraz metoda fabryczna, precyzyjniej nazywane **konstruktorem konwertującym** lub **konwertującą metodą fabryczną**, pozwalają klientowi wybrać implementację dla kopii i nie wymuszają przyjęcia implementacji z oryginalnego obiektu. Załóżmy, że mamy obiekt `HashSet s` i chcesz utworzyć jego kopię jako `TreeSet`. Metoda `clone` nie dopuszcza takiej operacji, ale łatwo można to zrealizować za pomocą konstruktora kopiującego: `new TreeSet(s)`.

Gdy poznaliśmy już wszystkie problemy związane ze stosowaniem interfejsu `Cloneable`, można bezpiecznie powiedzieć, że nowe interfejsy nie powinny go rozszerzać, a nowe klasy zaprojektowane jako klasy bazowe nie powinny go implementować. Choć klasy oznaczone `final` implementujące `Cloneable` nie są tak szkodliwe, stosowanie w nich tego mechanizmu należy traktować tylko jako optymalizację wykorzystywaną w rzadkich, uzasadnionych przypadkach (temat 67.). Jako zasadę należy przyjąć, że funkcjonalność kopiowania powinna być oferowana za pomocą konstruktorów i fabryk. Istotnym wyjątkiem są tu tablice, które najlepiej kopiować właśnie za pomocą metody `clone`.

Temat 14. Implementacja interfejsu `Comparable`

W przeciwieństwie do innych metod, omówionych w tym rozdziale, `compareTo` nie jest zadeklarowana w klasie `Object`, jest za to jedyną metodą interfejsu `Comparable`. Jest ona podobnego typu co metoda `equals`, poza tym że oprócz przeprowadzaniem prostego testu równości pozwala również określić porządek. Dodatkowo jest

metodą uogólnioną. Implementując interfejs Comparable, klasa wskazuje, że jej obiekty mają **naturalny porządek**. Sortowanie tablicy obiektów implementujących Comparable może być przeprowadzone za pomocą jednego wywołania:

```
Arrays.sort(a);
```

Bardzo łatwo można przeszukiwać i wyszukiwać graniczne wartości i tworzyć automatycznie sortowane kolekcje obiektów klas implementujących Comparable. Poniższy program korzysta z faktu, że String implementuje Comparable i drukuje alfabetyczną listę argumentów wiersza poleceń z usuniętymi powtórzeniami.

```
public class WordList {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        Collections.addAll(s, args);
        System.out.println(s);
    }
}
```

Implementując Comparable, pozwalasz na zastosowanie klasy w wielu standardowych algorytmach i zbiorach implementacji, korzystających z tego właśnie interfejsu. Małym nakładem sił możesz osiągnąć bardzo dobre wyniki. Praktycznie wszystkie klasy z bibliotek platformy Java, które posiadają wartość, implementują interfejs Comparable. Dotyczy to również wszystkich typów wyliczeniowych (temat 34.). Jeżeli piszesz klasę posiadającą wartość, z oczywistym naturalnym porządkiem, na przykład porządkiem alfabetycznym, numerycznym lub chronologicznym, powinieneś poważnie rozważyć implementację interfejsu Comparable.

```
public interface Comparable<T> {
    int compareTo(T, t);
}
```

Ogólne założenia dla metody compareTo są podobne do założeń metody equals.

Metoda compareTo określa porządek bieżącego obiektu w porównaniu z podanym obiektem. Zwraca całkowitą liczbę ujemną, jeżeli obiekt jest mniejszy, zero, gdy jest równy i dodatnią liczbę całkowitą, gdy obiekt jest większy niż podany obiekt. Zgłasza wyjątek `ClassCastException`, jeżeli typ podanego obiektu nie pozwala na porównanie z bieżącym obiektem.

W poniższym opisie wyrażenie `sgn(expression)` oznacza funkcję „znak”, która przyjmuje wartości -1 , 0 i 1 , gdy wartość `expression` jest odpowiednio ujemna, równa zero i dodatnia.

- Programista musi zapewnić, że `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` dla każdej wartości `x` i `y` (implikując to, że `x.compareTo(y)` zgłasza wyjątek jedynie wtedy, gdy `y.compareTo(x)` również zgłasza wyjątek).

- Programista musi zapewnić, że relacja jest przechodnia — jeżeli $(x.compareTo(y) > 0 \& y.compareTo(z) > 0)$, to $x.compareTo(z) > 0$.
- Programista musi zapewnić, że jeżeli $x.compareTo(y) == 0$, to dla każdego z $sgn(x.compareTo(z)) == sgn(y.compareTo(z))$.
- Zalecane jest, choć nie jest to wymagane, że $(x.compareTo(y) == 0) == \neg(x.equals(y))$. Jeżeli klasa implementuje interfejs `Comparable` i nie spełnia tego warunku, powinna wyraźnie wskazać ten fakt. Zalecany komentarz to: „Uwaga, klasa ta posiada naturalny porządek, który jest niespójny z operacją porównania”.

Nie daj się przestraszyć matematycznemu zapisowi tych założeń. Podobnie jak w przypadku założeń dla metody `equals` (temat 10.), założenia dla metody `compareTo` nie są tak skomplikowane, jak mogą wyglądać. Dowolny, rozsądny porządek obiektów klasy spełni założenia dla metody `compareTo`. Metoda ta, w przeciwieństwie do `equals`, nie musi działać na obiektach różnych klas. Dopuszczalne jest zgłaszanie wyjątku `ClassCastException`, jeżeli referencje do porównywanych obiektów wskazują na różne obiekty różnych klas. Najczęściej jest to najlepsze, co można zrobić w takim przypadku. Założenia **dopuszczają** porównywanie obiektów różnych klas, do czego dochodzi, gdy metodę wymusza implementowany interfejs.

Tak jak klasa naruszająca założenia dla metody `hashCode` może spowodować nieprawidłowe działanie innych klas korzystających z kodowania mieszającego, tak samo naruszenie założeń `compareTo` spowoduje nieprawidłowe działanie innych klas, korzystających z porównywania obiektów. Klasami, które korzystają z porównywania, są sortowane kolekcje, `TreeSet` i `TreeMap`, oraz klasy użytkowe `Collections` i `Arrays`, które zawierają przeszukiwanie i sortowanie.

Zajmijmy się zapewnieniem założeń dla metody `compareTo`. Pierwsze założenie mówi, że jeżeli odwrócisz kierunek porównania obiektów, można będzie się spodziewać następujących skutków: jeżeli pierwszy obiekt jest mniejszy niż drugi, to drugi musi być mniejszy niż pierwszy; jeżeli pierwszy obiekt jest równy drugiemu, to drugi musi być równy pierwszemu; jeżeli pierwszy obiekt jest większy niż drugi, to drugi jest mniejszy niż pierwszy. Drugie założenie jest następujące: jeżeli pierwszy obiekt jest większy niż drugi, a drugi jest większy niż trzeci, to pierwszy obiekt musi być większy niż trzeci. Ostatnie założenie mówi, że jeżeli dwa obiekty są równe, muszą dawać te same wyniki podczas porównywania z dowolnym innym obiektem.

Jedną z konsekwencji tych trzech założeń jest to, że testy stosowane w metodzie `compareTo` muszą spełniać wymagania dla metody `equals` — zwrotność, symetryczność i przechodniość. Dlatego występuje ten sam problem — nie ma prostego sposobu na rozszerzenie klasy umożliwiającej tworzenie obiektów o nowe atrybuty przy równoczesnym zachowaniu założenia metody `compareTo`, chyba że chcesz

odpuścić zalety abstrakcji obiektowej (temat 10.). Należy zastosować te same sposoby rozwiązania problemu. Jeżeli chcesz dodać do klasy implementującej Comparable ważne atrybuty, nie można jej rozszerzać. Zamiast tego należy utworzyć niezwiązaną klasę, zawierającą pole pierwszej klasy. Następnie należy utworzyć metodę zwracającą to pole. Pozwala to napisać w drugiej klasie dowolną implementację metody `compareTo`, co umożliwi klientowi dostęp do obiektu pierwszej klasy przy wykorzystaniu obiektu drugiej klasy.

Ostatnie założenie metody `compareTo`, które jest właściwie silną sugestią, a nie prawdziwym wymaganiem, proponuje, aby test równości wykonywany za pomocą metody `compareTo` dawał takie same wyniki, jak test za pomocą metody `equals`. Jeżeli zalecenie to zostanie spełnione, porządek ustalany dzięki metodzie `compareTo` będzie **spójny z porównywaniem** za pomocą metody `equals`. Jeżeli zalecenie to nie jest dotrzymane, porządkowanie jest **niespójne z porównaniem**. Klasa, w której porządkowanie jest niespójne z porównywaniem, działa, ale sortowane kolekcje zawierające obiekty tej klasy nie spełniają założeń dla odpowiednich interfejsów kolekcji (`Collection`, `Set` lub `Map`). Dzieje się tak, ponieważ ogólne założenia dla tych interfejsów są zdefiniowane na podstawie metody `equals`, ale korzystają z porównywania obiektów za pomocą metody `compareTo`. Jeżeli się tak zdarzy, nie jest to jeszcze katastrofa, ale powinieneś o tym wiedzieć.

Rozważmy na przykład klasę `BigDecimal`, w której metoda `compareTo` jest niespójna z `equals`. Jeżeli utworzymy `HashSet` i dodamy `new BigDecimal("1.0")` i `new BigDecimal("1.00")`, zbiór będzie zawierał dwa elementy, ponieważ obiekty te porównywane za pomocą metody `equals` są różne. Jeżeli zamiast `HashSet` skorzystasz z kolekcji `TreeSet`, zbiór będzie zawierał tylko jeden element, ponieważ przedstawione obiekty `BigDecimal`, porównane za pomocą metody `compareTo`, są równe (więcej informacji w dokumentacji klasy `BigDecimal`).

Tworzenie metody `compareTo` jest podobne do tworzenia metody `equals`, istnieje jednak kilka kluczowych różnic. Ponieważ interfejs `Comparable` jest parametryzowany, nie musisz sprawdzać typu argumentu przed rzutowaniem. Jeżeli argument nie ma odpowiedniego typu, wywołanie spowoduje błąd kompilacji. Jeżeli argumentem jest `null`, metoda `compareTo` powinna zgłosić wyjątek `NullPointerException` → `Exception`. Tak się stanie, jeśli tylko spróbujesz użyć dowolnego z pól.

Same porównania pól mają charakter sprawdzania kolejności, a nie sprawdzania równości. Referencje do obiektów możemy porównać rekurencyjnie, wywołując metody `compareTo`. Jeżeli pole nie implementuje `Comparable` lub należy zastosować niestandardowy porządek, możesz jawnie wywołać `Comparator`. Można napisać własny lub wykorzystać istniejący, tak jak zostało to zrealizowane w metodzie `compareTo` dla klasy `CaseInsensitiveString`, opisanej w temacie 10.:

```
// Jednopolowy Comparable związany z polem referencji do obiektu
public final class CaseInsensitiveString
    implements Comparable<CaseInsensitiveString> {
```

```

public int compareTo(CaseInsensitiveString cis) {
    return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
}
... // pozostała część pominięta
}

```

Należy zwrócić uwagę, że klasa `CaseInsensitiveString` implementuje `Comparable<CaseInsensitiveString>`. Oznacza to, że referencje `CaseInsensitiveString` mogą być porównywane wyłącznie z innymi referencjami `CaseInsensitiveString`. Jest to normalne w przypadku deklaracji implementacji `Comparable` przez klasę.

Wcześniejsze wydania książki zalecały, aby metody `compareTo` wykorzystywały operatory relacji `<` i `>` dla typów prostych dotyczących liczb całkowitych, a w odniesieniu do pól zmiennoprzecinkowych korzystały z metod statycznych `Double.compare` lub `Float.compare`. W Javie 7 statyczne metody `compare` dodano do wszystkich klas osłonowych typów prostych. **Zastosowanie operatorów relacji `<` i `>` w metodach `compareTo` jest zbyt rozwlekłe i podatne na błędy, więc nie jest już zalecane.**

Jeżeli klasa zawiera kilka znaczących pól, krytyczna jest kolejność ich porównywania. Musisz rozpocząć od najważniejszego pola i przechodzić do coraz mniej ważnych. Jeżeli wynikiem porównania jest wartość różna od zera (zero oznacza równość), porównanie jest zakończone i można zwrócić wynik. Jeżeli ważniejsze pola są równe, należy przejść do porównywania mniej ważnych pól. Jeżeli wszystkie pola są równe, należy zwrócić zero. Technika ta zostanie zademonstrowana na podstawie metody `compareTo` dla klasy `PhoneNumber`, przytoczonej w temacie 11.:

```

// Wielopolowe Comparable dotyczące pól typów prostych
public int compareTo(PhoneNumber pn) {
    int result = Short.compare(areaCode, pn.areaCode);
    if (result == 0) {
        result = Short.compare(prefix, pn.prefix);
    }
    if (result == 0)
        result = Short.compare(lineNum, pn.lineNum);
    }
    return result;
}

```

W Javie 8 interfejs `Comparator` został wzbogacony zestawem **metod konstruujących komparator**, co pozwala na bardzo płynną konstrukcję komparatorów. Komparatory te mogą posłużyć do implementacji metody `compareTo` wymaganej przez interfejs `Comparable`. Wielu programistów lubi zwięzłość tego podejścia, choć ma ono swój koszt — sortowanie tablic instancji `PhoneNumber` jest na moim komputerze o jakieś 10% wolniejsze. Wybierając ten sposób tworzenia komparatora, rozważ zastosowanie importów statycznych, aby móc się odnosić do metod konstruujących komparator za pomocą ich prostych nazw, co zapewni większą czytelność kodu. Oto jak wyglądałaby metoda `compareTo` dla `PhoneNumber` po zastosowaniu tego podejścia:

```
// Metoda compareTo korzystająca z metod konstruujących komparator
private static final Comparator<PhoneNumber> COMPARATOR =
    comparingInt((PhoneNumber pn) -> pn.areaCode)
        .thenComparingInt(pn -> pn.prefix)
        .thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) {
    return COMPARATOR.compare(this, pn);
}
```

Przedstawiona implementacja buduje komparator w momencie inicjalizacji klasy przy użyciu dwóch metod konstruujących komparator. Pierwszą z tych metod jest `comparingInt`. To metoda statyczna przyjmująca **funkcję wydobywającą klucz**, co pozwala zamienić referencję do obiektu na klucz typu `int`. Metoda zwraca komparator, który porządkuje instancje zgodnie z wartościami wskazanego klucza. W tym konkretnym przykładzie metoda `comparingInt` przyjmuje **lambda**, która wydobywa numer kierunkowy z `PhoneNumber` i zwraca `Comparator<PhoneNumber>` porządkujący numery telefonu na podstawie numerów kierunkowych. Zauważ, że lambda jawnie wskazuje typ parametru wejściowego (`PhoneNumber pn`). Okazuje się, że w tym przypadku mechanizm zgadywania typów stosowany przez Javę zawodzi, więc musimy mu pomóc, aby kod udało się skompilować.

Jeśli dwa numery telefonu mają taki sam numer kierunkowy, musimy dokonać dodatkowego porównania. Odpowiada za to następna z metod konstruujących komparator — `thenComparingInt`. To metoda instancyjna obiektu typu `Comparator`, która przyjmuje funkcję wydobywającą klucz typu `int`, a następnie zwraca komparator, który najpierw stosuje oryginalne porównanie, a w przypadku takich samych wartości używa wydobytego klucza. Możemy w ten sposób wykonać cały łańcuch wywołań `thenComparingInt`, tworząc **porządek leksykograficzny**. W przykładzie metodę wykorzystujemy dwukrotnie — najpierw do układania numerów na podstawie prefiksu, a następnie na podstawie numeru linii. Zwróć uwagę, że tym razem **nie** musimy w funkcji przekazywanej jako parametr metody `thenComparingInt` wskazywać typu. W tym przypadku mechanizm Javy dotyczący zgadywania typów działa prawidłowo.

Klasa `Comparator` ma w zanadrzu cały zestaw metod konstruujących. Istnieją metody analogiczne do `comparingInt` i `thenComparingInt` dla typów prostych `long` i `double`. Wersję dla typu `int` można stosować również dla typów węższych takich jak `short` (zrobiliśmy tak w przykładzie z `PhoneNumber`). Wersji `double` można użyć dla typu `float`. W ten sposób pokrywamy obsługą wszystkie numeryczne typy proste.

Istnieją również metody konstruujące komparator dla typów referencyjnych. Metoda statyczna o nazwie `comparing` ma dwa przeciążenia. Pierwsza przyjmuje funkcję wydobywającą klucz i stosuje naturalny porządek dla wskazanego klucza.

Druga przyjmuje zarówno funkcję wydobywającą klucz, jak i komparator, aby na jego podstawie porządkować klucze. Istnieją trzy przeciążenia metody instancyjnej o nazwie `thenComparing`. Pierwsza wersja przyjmuje tylko komparator i używa go do odpowiedniego porządkowania. Druga przyjmuje tylko funkcję wydobywającą klucz i stosuje naturalny porządek dla wskazanego klucza, jeśli pierwsze porównanie było identyczne. Ostatnia przyjmuje zarówno funkcję wydobywającą klucz, jak i komparator, aby na jego podstawie porządkować klucze w drugiej kolejności.

Czasami spotyka się metody `compareTo` lub `compare`, które wykorzystują fakt, iż różnica między dwoma wartościami jest ujemna, jeśli pierwsza z nich jest mniejsza od drugiej, wynosi 0, jeśli są identyczne, i jest dodatnia, gdy pierwsza wartość jest większa od drugiej. Oto przykład:

```
// NIEDZIAŁAJĄCY komparator bazujący na różnicy w wartościach - łamie
// przechodniość!
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return o1.hashCode() - o2.hashCode();
    }
};
```

Nie korzystaj z tej techniki. Jest narażona na przepełnienia liczb całkowitych i ogólne artefakty arytmetyki liczb zmiennoprzecinkowych w trybie IEEE 754 [JLS 15.20.1, 15.21.1]. Wynikowe metody raczej nie będą znacząco szybsze od tych napisanych z użyciem technik wskazanych w tym temacie. Użyj statycznej metody `compare`:

```
// Komparator bazujący na metodzie statycznej compare
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};
```

lub metody konstruującej komparator:

```
// Komparator bazujący na metodzie konstruującej komparator
static Comparator<Object> hashCodeOrder =
    Comparator.comparingInt(o -> o.hashCode());
```

Podsumowując, za każdym razem, gdy implementujesz klasę zawierającą wartości i pozwalającą na ich sensowne uporządkowanie, powinieneś dodać do klasy implementację interfejsu `Comparable`, aby poszczególne instancje można było łatwo sortować, przeszukiwać i używać w kolekcjach wykorzystujących sortowanie. Gdy porównujesz wartości pól w implementacjach metod `compareTo`, unikaj użycia operatorów `<` i `>`. Zamiast nich stosuj dla typów prostych metodę statyczną `compare` dostępną w ich klasach osłonowych lub używaj metod konstruujących komparatory udostępnianych przez interfejs `Comparator`.

Klasy i interfejsy

Klasy i interfejsy są podstawą programowania w języku Java. Są one podstawowymi jednostkami abstrakcji. Język Java zawiera wiele elementów, których można użyć przy projektowaniu klas i interfejsów. Rozdział ten zawiera wskazówki, które mają na celu pomóc Ci w jak najlepszym wykorzystaniu tych elementów, tak aby klasy i interfejsy były możliwie użyteczne, solidne i elastyczne.

Temat 15. Ograniczanie dostępności klas i ich składników

Jedną z najważniejszych cech, które odróżniają dobrze zaprojektowany komponent od źle zaprojektowanego, jest stopień, w jakim komponent ukrywa szczegóły implementacji, oddzielając od niej API. Komponent komunikuje się z innymi jedynie poprzez API i nie musi mieć informacji o szczegółach działania wykorzystywanych komponentów. Cecha ta jest zwana **ukrywaniem informacji** lub **hermetyzacją** i jest jedną z najważniejszych zasad projektowania oprogramowania [Parnas72].

Ukrywanie informacji jest ważne z wielu powodów, z których większość jest związana z faktem, że daje to możliwość efektywnej **separacji** komponentów wchodzących w skład systemu. Pozwala to na efektywne tworzenie, testowanie, wykorzystywanie, poznawanie i modyfikowanie poszczególnych komponentów. Przyspiesza to tworzenie systemu, ponieważ komponenty mogą być pisane równolegle. Ułatwia to konserwację systemu, ponieważ każdy komponent może być szybko przeanalizowany, a błąd szybko poprawiony, bez obawy wprowadzenia błędów do innych komponentów. Mimo że ukrywanie informacji nie powoduje samo w sobie polepszenia wydajności komponentu, pozwala jednak na efektywne jej

poprawianie. Po zakończeniu tworzenia systemu należy za pomocą profilera znaleźć komponenty, których ulepszenie pozwoli na uzyskanie jak najlepszej wydajności (temat 67.). Ukrywanie informacji zwiększa możliwość powtórnego wykorzystania kodu, ponieważ poszczególne komponenty nie zależą od innych i często okazują się użyteczne w innym kontekście niż w tym, w którym zostały utworzone. I, na koniec, ukrywanie informacji zmniejsza ryzyko niepowodzenia projektu przy tworzeniu dużych systemów — nawet jeżeli projekt jako całość się nie powiedzie, indywidualne komponenty mogą przynieść korzyści.

Język programowania Java ma wiele udogodnień, wspomagających ukrywanie informacji. Jednym z nich jest **mechanizm kontroli dostępu** [JLS, 6.6], który określa **dostępność** klas, interfejsów i ich składników. Dostępność jednostki jest określana przez miejsce deklaracji oraz przez użycie (lub nie) modyfikatora dostępu (`private`, `protected` i `public`) przy deklaracji tej jednostki. Właściwe użycie tych modyfikatorów jest zasadnicze dla właściwego ukrywania informacji.

Naczelną zasadą jest **tworzenie możliwie mało dostępnych klas lub składników**. Inaczej mówiąc, powinieneś korzystać z możliwie niskiego poziomu dostępu, oczywiście niestojącego w sprzeczności z właściwym działaniem pisanego oprogramowania.

Dla klas i interfejsów najwyższego poziomu (niezagnieżdżonych) istnieją tylko dwa możliwe poziomy dostępu: **prywatny w ramach pakietu** i **publiczny**. Jeżeli zadeklarujesz klasę najwyższego poziomu, korzystając z modyfikatora `public`, będzie ona publiczna, w przeciwnym wypadku będzie ona prywatna w ramach pakietu. Jeżeli można utworzyć klasę jako prywatną w ramach pakietu, należy tak zrobić. Gdy stworzysz klasę prywatną w ramach pakietu, staje się ona częścią implementacji, dzięki czemu można ją dowolnie zmieniać lub usunąć w kolejnych wersjach bez obawy, że programy wykorzystujące ten moduł przestaną działać. Jeżeli utworzysz klasę publiczną, jesteś zobowiązany zapewnić jej istnienie w celu zachowania zgodności z poprzednimi wersjami.

Jeżeli klasa najwyższego poziomu, prywatna w ramach pakietu, jest wykorzystywana jedynie w jednej klasie, możesz pomyśleć o jej zmianie na postać prywatnej klasy zagnieżdżonej (lub interfejsu) w klasie ją wykorzystującej (temat 24.). Zmniejsza to jeszcze bardziej jej dostępność. Nie jest to jednak aż tak istotne, jak zmiana niepotrzebnych klas publicznych na klasy prywatne w ramach pakietu, ponieważ klasy prywatne są częścią implementacji, a nie API.

W przypadku składników (pól, metod, klas i interfejsów zagnieżdżonych) dostępne są cztery możliwe poziomy dostępu, wymienione poniżej w kolejności zwiększającej się dostępności.

- **Prywatny** — składnik taki jest dostępny jedynie z klas najwyższego poziomu, w których został zadeklarowany.

- **Prywatny w ramach pakietu** — składnik jest dostępny z dowolnej klasy w pakiecie, w którym został zadeklarowany. Technicznie jest to **domyślny** poziom dostępu, używany w przypadku, gdy nie zostanie podany żaden modyfikator dostępu (z wyjątkiem składowych interfejsu, które domyślnie są publiczne).
- **Zabezpieczony** — składnik jest dostępny z klas pochodnych do klasy, w której został zadeklarowany (oprócz kilku ograniczeń [JLS, 6.6.2]) i z dowolnej klasy z pakietu, w którym został zadeklarowany.
- **Publiczny** — składnik taki jest dostępny z każdego miejsca.

Po dokładnym zaprojektowaniu publicznego API klasy uwaga powinna się skupić na zmianie innych składników na prywatne. Jedynie jeżeli inna klasa z tego samego pakietu potrzebuje dostępu do określonego składnika, można usunąć modyfikator `private`, co spowoduje, że stanie się on prywatny w ramach pakietu. Jeżeli zauważysz, że dzieje się to często, powinieneś ponownie przemyśleć projekt systemu, aby sprawdzić, czy inna dekompozycja może pomóc w uzyskaniu klas bardziej od siebie niezależnych. Składniki klasy, zarówno prywatne, jak i prywatne w ramach pakietu, są częścią implementacji klasy i normalnie nie wpływają na eksportowane API. Mogą jednak „przeciekać” do API w przypadku, gdy klasa implementuje interfejs `Serializable` (tematy 86. i 87.).

W przypadku składników klas publicznych po zmianie poziomu dostępu z prywatnego w ramach pakietu na zabezpieczony następuje znaczny wzrost ich dostępności. Zabezpieczone składniki klasy stają się częścią eksportowanego API klasy i muszą być utrzymywane w celu zapewnienia zgodności z poprzednimi wersjami (temat 17.). Dostyc rzadko potrzebne jest zastosowanie zabezpieczonych składników.

Istnieje tylko jedna reguła ograniczająca dążenie do zmniejszania dostępności metod. Jeżeli metoda przesłania metodę z klasy bazowej, nie może mieć ona poziomu dostępu niższego od poziomu zdefiniowanego w klasie nadrzędnej [JLS, 8.4.6.3]. Jest to niezbędne w celu zapewnienia zasady, że obiekt klasy pochodnej może być użyty wszędzie tam, gdzie występuje obiekt klasy bazowej (**zasada podstawiania Liskov**, patrz temat 15.). Jeżeli naruszysz tę zasadę, kompilator wygeneruje błąd w czasie kompilacji klasy pochodnej. Specjalnym przypadkiem tej reguły jest implementacja interfejsu przez klasę, w której wszystkie metody umieszczone w interfejsie muszą być w klasie zadeklarowane jako publiczne. Powodem tego jest to, że metody w interfejsie są zadeklarowane przez domniemanie publiczne [JLS 9.1.5].

Aby możliwe było testowanie, można ulec pokusie zmniejszenia ograniczeń dostępu do klasy, interfejsu lub składnika. Jest to prawidłowe do pewnego momentu. Akceptowana jest zmiana prywatnego składnika klasy publicznej na składnik prywatny w ramach pakietu na czas testowania, ale nie jest akceptowane podnoszenie dostępności do wyższego poziomu. Inaczej mówiąc, nieakceptowane jest udostępnianie

klasy, interfejsu lub składnika w eksportowanym API tylko w celu umożliwienia testowania. Na szczęście nie jest to potrzebne, ponieważ testy mogą być uruchamiane w ramach testowanego pakietu, co pozwala uzyskać dostęp do elementów prywatnych w ramach pakietu.

Pola instancyjne rzadko kiedy powinny być publiczne (temat 16.). Jeżeli pole instancyjne nie jest oznaczone jako `final` lub jest to referencja do modyfikowalnego obiektu, to zmiana dostępności takiego pola na `public` nie pozwoli nam ograniczać zapisywanych w nim wartości. Oznacza to, że również oddajemy możliwość wymuszania niezmienników związanych z polem. Uniemożliwia to także wykonywanie jakichkolwiek akcji w czasie modyfikowania pola, więc **klasy z publicznymi polami modyfikowalnymi nie są bezpieczne dla wątków**. Nawet jeżeli pole jest oznaczone jako `final` i odwołuje się do obiektu niezmiennego, to udostępnienie takiego pola jako publicznego daje możliwość przełączenia do nowej wewnętrznej reprezentacji danych, w której pole to nie istnieje.

Ta sama porada odnosi się do pól statycznych — z jednym wyjątkiem. Można udostępniać stałe poprzez publiczne statyczne pola `final`, przy założeniu, że stała stanowi integralną część abstrakcji realizowanej przez klasę. Zgodnie z tą konwencją, pola takie mają nazwy składające się z wielkich liter, których słowa są rozdzielane podkreśleniami (temat 68.). Niezwykle ważne jest, aby pola te zawierały wartości typów prostych lub referencje do obiektów niemodyfikowalnych (temat 17.). Pole `final` zawierające referencję do obiektu modyfikowalnego ma wszystkie wady pola nieoznaczonego jako `final`. Choć referencja nie może być modyfikowana, to można zmodyfikować obiekt, na który ona wskazuje — z katastrofalnymi skutkami.

Zwróć uwagę, że tablice o rozmiarze różnym od zera są zawsze modyfikowalne, więc **niemal zawsze błędem jest deklarowanie publicznego pola typu `final`, będącego tablicą**. Jeżeli klasa posiada takie pole, klienci mogą modyfikować zawartość tablicy. Jest to najczęstszą przyczyną powstania luk w systemie bezpieczeństwa:

// Potencjalne źródło luki w systemie bezpieczeństwa

```
public static final Type[] VALUES = { ... };
```

Należy pamiętać, że niektóre środowiska IDE generują akcesory zwracające referencje do pól tablicy prywatnej, co powoduje powstanie dokładnie takiego problemu. Istnieją dwa sposoby jego rozwiązania. Można zmienić tablicę publiczną na prywatną lub użyć publicznej niezmienniej listy:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Można również zmienić dostęp do tablicy na prywatny i dodać publiczną metodę zwracającą kopię tablicy prywatnej:


```
private static final Thing[] PRIVATE_VALUES = { ... };  
public static final Thing[] values() {  
    return PRIVATE_VALUES.clone();  
}
```

Aby wybrać odpowiednią alternatywę, należy pomyśleć o tym, jak klienci będą przetwarzać wyniki. Jaki zwracany typ będzie najbardziej wygodny? Jak osiągnąć najlepszą wydajność?

W Javie 9 jako część **systemu modułów** wprowadzono dwa dodatkowe, niejawne poziomy dostępu. Moduł to zbiór pakietów, podobnie jak pakiet to zbiór klas. Moduł może jawnie eksportować niektóre pakiety jako część **deklaracji eksportu** stanowiących część **deklaracji modułu** (zgodnie z konwencją znajdują się one w pliku źródłowym o nazwie *module-info.java*). Publiczne i zabezpieczone składowe nieeksportowanych pakietów modułu są niedostępne na zewnątrz modułu. Wewnątrz modułu deklaracje eksportu nie mają wpływu na widoczność. Dzięki systemowi modułów możemy współdzielić klasy między pakietami bez potrzeby czynienia ich widocznymi dla całego świata. Publiczne i zabezpieczone składowe klas publicznych w nieeksportowanych pakietach stanowią bazę dla dwóch niejawnych poziomów dostępu. Są to niejako wewnątrzmodułowe analogie do normalnych poziomów publicznych i zabezpieczonych. Potrzeba stosowania tego rodzaju współdzielenia jest dosyć rzadka i najczęściej łatwo ją wyeliminować, zmieniając przydział klas do poszczególnych modułów.

W odróżnieniu od czterech głównych poziomów dostępu dwa poziomy związane z modułami mają głównie zadanie doradcze. Jeśli plik JAR modułu umieścimy w ścieżce klas aplikacji zamiast w ścieżce modułu, pakiety modułu zaczną działać w sposób, który był standardem przed modułami — wszystkie publiczne i zabezpieczone składowe publicznych klas pakietów będą miały standardową widoczność niezależnie od tego, czy pakiet ten jest eksportowany przez moduł [Reinhold, 1.2]. Jedynym miejscem, gdzie nowe poziomy bezpieczeństwa są ściśle wymuszane, jest JDK, bo nieeksportowane pakiety biblioteki Javy są naprawdę niedostępne poza ich modułami.

Ochrona dostępu wprowadzana przez moduły ma dla programisty nie tylko ograniczone zastosowanie i stanowi jedynie wskazówkę, ale również wymaga pewnego nakładu pracy — by skorzystać z niej, trzeba pogrupować pakiety w moduły, jawnie wskazać zależności w deklaracjach modułów, przekształcić drzewa kodu źródłowego i podjąć dodatkowe działania związane z dostępem do niemodułowych pakietów z poziomu pakietu [Reinhold, 3]. Na razie trudno wyrokować, czy moduły zdobędą popularność poza samym JDK. Wydaje się, że najlepiej ich unikać, chyba że ma się szczególne ważne powody do ich stosowania.

Podsumujmy. Zawsze powinienś możliwie maksymalnie zmniejszać dostępność. Po dokładnym zaprojektowaniu minimalnego API powinienś upewnić się, że

żadna przypadkowa klasa, interfejs lub składnik nie stanie się częścią API. Oprócz publicznych pól statycznych typu `final` klasy publiczne nie powinny posiadać pól publicznych. Upewnij się, że nie można modyfikować obiektów wskazywanych przez publiczne statyczne pola typu `final`.

Temat 16. Stosowanie metod akcesorów zamiast pól publicznych w klasach publicznych

Czasami zdarza się nam napisać klasę zdegenerowaną, która nie posiada żadnej funkcji poza grupowaniem pól instancyjnych:

```
// Klasy zdegenerowane, takie jak ta, nie powinny być publiczne!  
class Point {  
    public double x;  
    public double y;  
}
```

Ponieważ pola danych w takich klasach są bezpośrednio dostępne, klasy te nie oferują żadnych z zalet **hermetyzacji** (temat 15.). Nie można zmienić reprezentacji bez zmiany API, nie można wymuszać niezmienników, nie można wykonywać dodatkowych akcji w momencie dostępu do pola. Doświadczeni programiści obiektowi uważają takie klasy za nieprawidłowe i sugerują, że powinny być zawsze zastępowane klasami z polami prywatnymi i publicznymi **metodami akcesorów** (metody `get`) oraz, w przypadku klas modyfikowalnych, **mutatorów** (metody `set`).

```
// Hermetyzacja danych z użyciem metod akcesorów i mutatorów  
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double getX() { return x; }  
    public double getY() { return y; }  
  
    public void setX(double x) { this.x = x; }  
    public void setY(double y) { this.y = y; }  
}
```

Oczywiście, jest to prawidłowe podejście w przypadku klas publicznych — **jeżeli klasa jest dostępna poza pakietem, należy zapewnić metody akcesorów** w celu zachowania możliwości zmiany wewnętrznej reprezentacji klasy. Jeżeli klasa publiczna udostępnia pola danych, możliwość zmiany reprezentacji jest stracona, ponieważ kod kliencki może z niej korzystać.

Jeżeli jednak klasa jest prywatna w ramach pakietu lub jest prywatną klasą zagnieżdżoną w innej klasie, nie ma przeszkód w udostępnianiu pól danych — zakładając, że abstrakcja realizowana przez klasę jest odpowiednio opisana. Podejście takie pozwala na zmniejszenie komplikacji kodu w stosunku do podejścia z akcesorami, zarówno w definicji klasy, jak i w kodzie klienckim. Choć kod kliencki jest związany z wewnętrzną reprezentacją klasy, to jest ograniczony do pakietu zawierającego klasę. Jeżeli konieczna będzie zmiana reprezentacji, można ją wprowadzić bez konieczności modyfikacji jakiegokolwiek kodu znajdującego się poza pakietem. W przypadku prywatnej klasy zagnieżdżonej zasięg zmian jest ograniczony do zawierającej ją klasy.

Kilka klas z bibliotek platformy Java narusza zasadę mówiącą, że klasy publiczne nie powinny bezpośrednio udostępniać pól. Najważniejszymi przykładami są klasy `Point` oraz `Dimension` z pakietu `java.awt`. Klasy te powinny być traktowane jako ostrzeżenia, a nie przykłady do powtarzania. Zgodnie z opisem w temacie 67. decyzja o udostępnieniu zawartości klasy `Dimension` spowodowała powstanie poważnych problemów z wydajnością, które występują do dnia dzisiejszego.

Choć bezpośrednio udostępnianie pól klasy publicznej nie jest dobrym pomysłem, to jednak jest to mniej dokuczliwe, jeżeli pola są niezmiennie. Nie można zmieniać reprezentacji takiej klasy bez zmiany API klasy i nie można wykonywać dodatkowych akcji przy dostępie do pola, ale można wymuszać niezmienniki. Na przykład przedstawiona poniżej klasa gwarantuje, że każdy obiekt reprezentuje prawidłowy czas:

// Klasa publiczna z udostępnionymi polami niezmiennymi - wątpliwa

```
public final class Time {
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOUR = 60;

    public final int hour;
    public final int minute;

    public Time(int hour, int minute) {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Godzina: " + hour);
        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Minuta: " + minute);
        this.hour = hour;
        this.minute = minute;
    }
    ... // pozostała część pominięta
}
```

Podsumujmy. Klasy publiczne nie powinny nigdy udostępniać pól modyfikowalnych. Mniej szkodliwe, choć nadal wątpliwe jest udostępnianie pól niemodyfikowalnych przez klasy publiczne. Czasami korzystne jest udostępnianie pól przez klasy prywatne dla pakietu lub klasy prywatne zagnieżdżone, niezależnie od tego, czy są one modyfikowalne, czy nie.

Temat 17. Zapewnianie niezmienności obiektu

Klasa niezmienna to klasa, której obiekty nie mogą być modyfikowane. Wszystkie dane zawarte w każdym obiekcie tej klasy są podawane w czasie tworzenia obiektu i pozostają niezmiennie przez cały czas istnienia tego obiektu. Biblioteki środowiska Java zawierają wiele niezmiennych obiektów, na przykład: `String`, klasy otoczkowe dla wartości prostych oraz `BigInteger` i `BigDecimal`. Istnieje wiele powodów dla stosowania obiektów niezmiennych: w porównaniu z klasami modyfikowalnymi klasy niezmiennie są łatwiejsze do projektowania, tworzenia i używania. Powodują powstanie mniejszej liczby błędów i są bezpieczniejsze.

Aby utworzyć klasę niezmienną, należy pamiętać o pięciu regułach:

1. **Nie twórz żadnych metod modyfikujących obiekt** (nazywanych również **mutatorami**).
2. **Upewnij się, że klasa nie może zostać rozszerzona.** Uniemożliwia to modyfikację w klasie pochodnej sposobu działania klasy i dopuszczenia do modyfikacji obiektu. Zablokowanie możliwości przesłaniania jest najczęściej realizowane przez deklarację klasy jako `final`, ale istnieją również inne sposoby, które omówimy później.
3. **Zadeklaruj wszystkie pola jako `final`.** W ten sposób jednoznacznie wyrazisz swoje intencje i będą one wymuszane przez system. Być może będzie konieczne zagwarantowanie prawidłowego działania w przypadku, gdy referencja do nowego obiektu jest przekazywana z jednego wątku do drugiego bez synchronizacji, ponieważ powodzenie takiej operacji może zależeć od bieżącego **modelu pamięci** [JLS, 17.5; Goetz06, 16].
4. **Zadeklaruj wszystkie pola jako `private`.** Uniemożliwia to bezpośrednią modyfikację pól przez klienty. Choć technicznie jest dopuszczalne, aby niezmiennie klasy miały publiczne pola `final`, zawierające wartości proste lub referencje do niezmiennych obiektów, to nie jest to zalecane, ponieważ uniemożliwia to zmianę wewnętrznej reprezentacji danych w kolejnych wersjach (tematy 15. i 16.).

5. Zapewnij wyłączny dostęp do dowolnych modyfikowalnych

komponentów. Jeżeli Twoja klasa posiada jakieś pola, które odwołują się do modyfikowalnych obiektów, zapewnij, że klienci klasy nie będą mogły odczytać referencji do tych obiektów. Nigdy nie inicjalizuj tych pól wartościami dostarczonymi przez klienty ani nie zwracaj referencji obiektu do akcesora. Twórz defensywne kopie (temat 50.) w konstruktorach, akcesorach i metodach `readObject` (temat 88.).

Wiele spośród przykładowych klas z poprzednich rozdziałów było klasami niezmiennymi. Jedną z nich jest klasa `PhoneNumber` z tematu 11. Posiada ona akcesory, ale nie posiada odpowiadających im mutatorów. Poniżej przedstawiamy nieco bardziej złożony przykład:

// Niezmienna klasa liczby zespolonej

```
public final class Complex {
    private final float re;
    private final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    // akcesory bez odpowiadających im mutatorów
    public float realPart() { return re; }
    public float imaginaryPart() { return im; }

    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex minus(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex times(Complex c) {
        return new Complex(re*c.re - im*c.im,
                           re*c.im + im*c.re);
    }

    public Complex dividedBy(Complex c) {
        float tmp = c.re*c.re + c.im*c.im;
        return new Complex((re*c.re + im*c.im) / tmp,
                           (im*c.re - re*c.im) / tmp);
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
```

```

        return false;
    Complex c = (Complex)o;

    //zajrzyj do tematu 10., aby sprawdzić, dlaczego używamy compare zamiast ==
    return Double.compare(re, c.re) == 0 &&
        Double.compare(im, c.im) == 0;
}

@Override public int hashCode() {
    return 31 * Double.hashCode(re) + Double.hashCode(im);
}

@Override public String toString() {
    return "(" + re + " + " + im + "i";
}
}

```

Klasa ta reprezentuje **liczby zespolone** (liczby składające się z części rzeczywistej i urojonej). Oprócz standardowych metod klasy `Object` posiada ona akcesory do pobierania rzeczywistej i urojonej części liczby oraz metody pozwalające na wykonywanie czterech podstawowych operacji matematycznych: dodawania, odejmowania, mnożenia i dzielenia. Zwróć uwagę, że operacje matematyczne tworzą i zwracają nowy obiekt `Complex`, nie modyfikując bieżącego obiektu. Jest to znane pod nazwą podejścia **funkcyjnego**, ponieważ metody te zwracają wynik zastosowania funkcji na operandzie bez jego modyfikacji. Innym podejściem jest częściej spotykane podejście **proceduralne** lub **imperatywne**, w którym metody stosują procedurę na operandzie, powodując zmianę jego stanu. Zwróć uwagę, że nazwy metod są przymkami (np. `plus`), a nie czasownikami (np. `add`, czyli „dodaj”). W ten sposób staramy się podkreślić, że metody te nie modyfikują wartości obiektów, względem których są wywoływane. Klasy `BigInteger` i `BigDecimal` **nie** zastosowały się do tej konwencji, co często doprowadza do ich niewłaściwego użycia.

Jeżeli nie korzystasz z podejścia funkcjonalnego, może się ono wydawać z początku nienaturalne, ale pozwala zachować mającą wiele zalet niezmiennosc obiektu. **Obiekty niezmiennicze są proste.** Obiekt niezmienny może znajdować się w dokładnie jednym stanie — stanie, w którym został utworzony. Jeżeli jesteś pewien, że wszystkie konstruktory tworzą niezmienniki klasy, gwarantowane jest, że niezmienniki te pozostaną prawdziwe przez cały czas bez dodatkowego nakładu pracy ze strony Twojej bądź użytkownika klasy. Obiekty modyfikowalne z drugiej strony posiadają określoną przestrzeń stanów. Jeżeli dokumentacja nie zawiera precyzyjnego opisu przeprowadzanych przez mutatory przejść pomiędzy stanami, używanie tej klasy będzie utrudnione lub nawet niemożliwe.

Obiekty niezmiennicze są ze swojej natury bezpieczne dla wątków — nie wymagają synchronizacji. Nie mogą zostać uszkodzone przez wiele wątków, odwołujących się do nich równolegle. Jest to o wiele prostsze podejście do zapewnienia bezpieczeństwa dla wątków. Żaden wątek nie zauważy wpływu pozostałych wątków na obiekt

niezmienny. Dlatego **obiekty niezmiennie mogą być bez problemów współdzielone**. Klasy niezmiennie powinny skorzystać z tego, zachęcając klienty do ponownego wykorzystania istniejących obiektów, o ile jest to możliwe. Łatwą metodą zrealizowania tego jest udostępnienie publicznych statycznych stałych typu final dla często wykorzystywanych wartości. Klasa Complex na przykład może udostępniać następujące stałe:

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I    = new Complex(0, 1);
```

Technikę tę można jeszcze rozszerzyć. Klasa niemodyfikowalna może udostępniać statyczną metodę fabryczną (temat 1.), która buforuje odwołania do często używanych obiektów i nie pozwala na tworzenie nowych obiektów, gdy istnieje odpowiedni obiekt. Czynią tak klasy otoczkowe typów prostych i klasa BigInteger. Użycie tych statycznych metod fabrycznych powoduje, że klienty współdzielą istniejące obiekty zamiast tworzyć nowe. Dzięki temu zmniejsza się użycie pamięci i zmniejsza częstotliwość działania procesu zbierania nieużytków. Zastosowanie statycznych metod fabrycznych w miejsce konstruktorów publicznych przy projektowaniu nowej klasy daje nam możliwość późniejszego dodania buforowania, bez konieczności modyfikowania klientów.

Konsekwencją współdzielenia niezmiennych obiektów jest brak konieczności **defensywnego tworzenia kopii** (temat 50.). Właściwie to nigdy nie musisz tworzyć kopii, ponieważ wszystkie kopie będą zawsze ekwiwalentem oryginalnego obiektu. Dlatego dla klasy niezmiennej nie musisz i nie powinieneś tworzyć metody clone ani **konstruktora kopiującego** (temat 13.). Nie było to dostatecznie jasne w czasie, gdy powstawała platforma języka Java, więc klasa String posiada konstruktor kopiujący, który powinien być rzadko używany (temat 6.).

Możesz udostępniać nie tylko niezmiennie obiekty, ale również ich dane wewnętrzne. Na przykład klasa BigInteger korzysta wewnętrznie z notacji znak-moduł. Znak jest reprezentowany przez liczbę int, natomiast moduł przez tablicę liczb int. Metoda negate tworzy nowy obiekt BigInteger o tym samym module, ale przeciwnym znaku. Nie ma potrzeby kopiowania tablicy — nowy obiekt BigInteger wskazuje na tę samą wewnętrzną tablicę, co oryginał.

Obiekty niezmiennie są świetnym materiałem do budowania innych obiektów, zarówno niezmiennych, jak i modyfikowalnych. Dużo łatwiej jest zarządzać skomplikowanymi obiektami, jeżeli wiesz, że obiekty, z których się składa obiekt główny, nie zmieniają się. Specjalnym przypadkiem tej zasady jest zastosowanie niezmiennych obiektów jako kluczy mapowania i elementów zbioru. Gdy znajdą się już w zbiorze, nie musisz obawiać się zmiany ich wartości; zmiana taka mogłaby spowodować uszkodzenie odwzorowań lub niezmienników zbioru.

Obiekty niezmiennie oferują w zasadzie gratis atomowość działań (temat 76.). Ich stan nigdy nie ulega zmianie, więc nie ma możliwości powstania tymczasowości niezgodności.

Jedyną znaczną wadą obiektów niezmiennych jest to, że wymagają one tworzenia osobnych obiektów dla każdej osobnej wartości. Tworzenie takich obiektów może zabierać dużo czasu, szczególnie jeżeli są duże. Załóżmy na przykład, że mamy liczbę **BigInteger** o milionie bitów i chcemy odwrócić najmniej znaczący bit:

```
BigInteger moby = ...;  
moby = moby.flipBit(0);
```

Metoda `flipBit` tworzy nowy obiekt, składający się również z miliona bitów, który od oryginału różni się tylko jednym bitem. Opisana operacja wymaga sporo pamięci i czasu — proporcjonalnie do wielkości obiektu `BigInteger`. Porównajmy takie działanie z działaniem klasy `java.util.BitSet`. Podobnie do `BigInteger`, klasa `BitSet` reprezentuje sekwencje bitów o dowolnej długości, ale jest to klasa modyfikowalna. Klasa `BitSet` zawiera metodę, pozwalającą zmienić w stałym czasie stan dowolnego bitu zapisanego w obiekcie:

```
BitSet moby = ...;  
moby.flip(0);
```

Problem utraty wydajności jest dużo poważniejszy w przypadku operacji wymagających wykonania wielu kroków — w każdym z nich jest tworzony nowy obiekt i ewentualnie usuwany jest stary. Istnieją dwa sposoby rozwiązania tego problemu. Pierwszym jest wykrycie wspólnych wielokrokowych operacji i realizacja ich jako wyrażeń pierwotnych. Jeżeli wielokrokowa operacja jest realizowana jako wyrażenie pierwotne, niezmienna klasa nie musi tworzyć kolejnych obiektów w każdym kroku. Klasy niezmiennie mogą wewnętrznie działać bardzo sprytnie. Na przykład klasa `BigInteger` posiada prywatną w ramach pakietu, modyfikowalną klasę pomocniczą, która jest wykorzystywana do przyspieszenia wielokrokowych operacji, takich jak podnoszenie modułu do potęgi. Dużo trudniej jest korzystać z modyfikowalnych klas pomocniczych, ale na szczęście nie musimy tego robić. Programiści tworzący klasę `BigInteger` wzięli to trudne zadanie na siebie.

Podejście to jest do zaakceptowania, gdy można precyzyjnie określić, które skomplikowane, wieloczęściowe operacje będą wykonywane przez klienta klasy. Jeżeli nie można tego określić, najlepiej udostępnić **publiczną** zmienną klasę pomocniczą. Najważniejszym przykładem takiego podejścia jest klasa `String`, dla której można skorzystać z modyfikowalnej klasy pomocniczej `StringBuilder` (i jej starszej, niezalecanej obecnie wersji `StringBuffer`).

Wiedząc, w jaki sposób można tworzyć klasy niezmiennie oraz znając wady i zalety niezmienności obiektów, możemy omówić kilka alternatywnych rozwiązań. Przypomnijmy sobie, że aby zagwarantować niezmiennosc, klasa nie może pozwalać na dziedziczenie. Zwykle jest to realizowane przez zadeklarowanie klasy jako `final`,

ale dostępne jest również inne, bardziej elastyczne podejście. Alternatywą dla tworzenia finalnych klas niezmiennych jest zadeklarowanie wszystkich konstruktorów jako prywatnych lub prywatnych w ramach pakietu, a następnie dodanie publicznych statycznych metod fabrycznych, zastępujących konstruktory publiczne (temat 1.). Aby zilustrować tę technikę, zamieszczamy fragment klasy `Complex`, korzystającej z tego podejścia:

```
// Klasa niezmienna ze statycznymi metodami fabrycznymi zamiast konstruktorów
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    ... // pozostała część pominięta
}
```

Jest to najczęściej najlepsza spośród przedstawionych tutaj alternatyw. Jest najbardziej elastyczna, ponieważ pozwala na wykorzystanie wielu klas implementacji prywatnych w ramach pakietu. Dla klientów tej klasy, które z reguły są umieszczone poza pakietem, klasa niezmienna jest niejawnie typu `final`, ponieważ nie można rozszerzać klasy znajdującej się w innym pakiecie oraz nie jest dostępny publiczny lub zabezpieczony konstruktor. Oprócz zapewnienia elastyczności wielu klas implementacji podejście to pozwala poprawiać wydajność klasy poprzez ulepszenie mechanizmu buforowania obiektów zaimplementowanych w statycznych metodach fabrycznych.

W trakcie pisania klas `BigInteger` i `BigDecimal` nie było powszechnie wiadome, że klasy niezmiennie muszą być efektywnie typu `final`, więc ich metody mogą zostać przesłonięte. Niestety, nie może zostać to poprawione, ponieważ należy zachować zgodność z poprzednimi wersjami. Jeżeli piszesz klasę, w której bezpieczeństwo zależy od niezmienności argumentów `BigInteger` lub `BigDecimal`, pochodzących od klientów, należy sprawdzać, czy argumentami są „prawdziwe” obiekty `BigInteger` lub `BigDecimal`, czy obiekty klas pochodnych. Jeżeli są to obiekty klasy pochodnej, należy wykonać defensywne kopiowanie, ponieważ trzeba założyć, że stan takich obiektów może ulec zmianie (temat 50.):

```
public static BigInteger safeInstance(BigInteger val) {
    return val.getClass() == BigInteger.class ?
        val : new BigInteger(val.toByteArray());
}
```

Lista zasad dla klas niezmiennych, przedstawiona na początku tego tematu, zawiera zasadę, że żadna metoda nie może zmieniać obiektu i wszystkie pola muszą być zadeklarowane jako `final`. Zasada ta jest nieco bardziej rygorystyczna niż jest to faktycznie potrzebne i może być nieco złagodzona w celu poprawienia wydajności. Metody nie mogą zmieniać stanu obiektu w sposób **widoczny na zewnątrz**. Jednak wiele niezmiennych klas posiada jedno lub więcej nadmiarowych pól nieoznaczonych jako `final`, w których buforowane są wyniki skomplikowanych obliczeń. Jeżeli takie samo wyrażenie będzie wykorzystywane w przyszłości, zwracana jest zapamiętana wartość, co powoduje zaoszczędzenie czasu potrzebnego na ponowne obliczenie wyrażenia. Technika ta może być wykorzystana, ponieważ obiekt jest niezmienny — jego niezmiennosc gwarantuje osiągnięcie tych samych wyników obliczeń za każdym razem.

Na przykład metoda `hashCode` dla klasy `PhoneNumber` (temat 11.) oblicza kod mieszający za pierwszym wywołaniem metody i zapamiętuje ten kod na wypadek, gdyby był powtórnie potrzebny. Technika ta jest klasycznym przypadkiem **późnej inicjalizacji** (temat 83.) i jest wykorzystywana również przez klasę `String`.

Podczas projektowania sposobu serializacji obiektu należy zachować daleko posuniętą ostrożność. Jeżeli chcesz, aby klasa niezmienna implementowała interfejs `Serializable` i zawierała jedno lub więcej pól odwołujących się do obiektów modyfikowalnych, musisz napisać metodę `readObject` lub `readResolve` albo skorzystać z metod `ObjectOutputStream.writeUnshared` i `ObjectInputStream.readUnshared` nawet wtedy, gdy dopuszczalna jest domyślna obsługa serializacji. W przeciwnym razie napastnik może utworzyć modyfikowalną instancję Twojej niezmiennych klasy. Więcej szczegółów na temat serializacji znajduje się w temacie 88.

Podsumujmy bieżący temat. Nie ulegaj pokusie napisania metody `set` dla każdej metody `get`. **Klasy powinny być niezienne, chyba że istnieją poważne powody, aby były one modyfikowalne.** Klasy niezienne posiadają wiele zalet i ich jedyną wadą są potencjalne problemy z wydajnością w niektórych okolicznościach. Małe obiekty posiadające wartość, takie jak `PhoneNumber` czy `Complex`, zawsze powinny być niezienne (istnieje kilka klas wchodzących w skład biblioteki języka Java, takich jak `java.util.Date` i `java.awt.Point`, które powinny być niezienne, ale nie są). Powinieneś poważnie pomyśleć nad tworzeniem również dużych obiektów posiadających wartość, takich jak `String` i `BigInteger`, w postaci klas niezmiennych. Możesz napisać publiczną modyfikowalną klasę pomocniczą dla podstawowej klasy niezmiennych **tylko** wtedy, gdy potwierdzi się, że jest to niezbędne dla poprawienia wydajności (temat 67.).

Istnieją klasy, w których przypadku niezmiennosc jest niepraktyczna. **Jeżeli klasa nie może być niezmienna, powinieneś ograniczyć możliwość modyfikowania takiego obiektu tak mocno, jak tylko jest to możliwe.** Ograniczenie liczby stanów, jakie mogą występować w obiekcie, ułatwia stosowanie obiektu i ogranicza możliwość wystąpienia błędów. Dlatego każde pole powinno być zadeklarowane jako

final, o ile nie istnieje pilna potrzeba, aby nie było ono finalne. Łącząc prezentowaną tu radę z tą, która pojawiła się w temacie 15., możemy powiedzieć, że **każde pole powinno być oznaczone private final, chyba że istnieją dobre powody, aby tak nie robić.**

Konstruktory powinny tworzyć w pełni zainicjowane obiekty ze wszystkimi ich niezmiennikami. Nie powinien udostępniać publicznej metody zapewniającej inicjalizację, która jest inna niż konstruktor lub statyczna metoda fabryczna, chyba że jest **bardzo ważny** powód istnienia takiej metody. Podobnie nie powinien udostępniać metody zapewniającej „powtórna inicjalizację”, która pozwala na powtórne użycie obiektu w taki sam sposób, jakby był w swoim stanie początkowym. Metoda zapewniająca powtórna inicjalizację nie pozwala na znaczne przyspieszenie działania programu, a jedynie zwiększa stopień skomplikowania systemu.

Przykładem takiej klasy jest `CountDownLatch`. Jest ona modyfikowalna, ale jej przestrzeń stanów jest niewielka. Można utworzyć obiekt, użyć go raz i koniec — gdy licznik osiągnie zero, nie można obiektu użyć ponownie.

Ostatnia uwaga dotyczy klasy `Complex`. Przykład ten został pomyślany jedynie do zilustrowania niezmienności. Nie jest to najwyższej jakości implementacja liczb zespolonych. Korzysta ona ze standardowych wzorów na mnożenie i dzielenie liczb pierwszych, które nie są prawidłowo zaokrąglane i nie obsługują prawidłowo zespolonych liczb NaN i nieskończoności [Kahn91, Smith62, Thomas94].

Temat 18. Zastępowanie dziedziczenia kompozycją

Dziedziczenie to bardzo efektywne narzędzie, ułatwiające powtórne wykorzystanie kodu, ale nie zawsze jest najlepszym sposobem osiągnięcia oczekiwanych wyników. Jeżeli zostanie użyte niewłaściwie, powoduje powstanie kodu wrażliwego na błędy. Bezpiecznie można wykorzystać dziedziczenie w ramach tego samego pakietu, gdzie implementacja klasy bazowej i pochodnej jest tworzona przez tego samego programistę. Bezpiecznie można korzystać z dziedziczenia klas zaprojektowanych do tego właśnie celu i odpowiednio udokumentowanych (temat 19.). Dziedziczenie ze zwykłej klasy zadeklarowanej w innym pakiecie jest ryzykownym posunięciem. Należy pamiętać, że w książce to słowo „dziedziczenie” oznacza **dziedziczenie implementacji** (gdy jedna klasa dziedziczy po drugiej). Problemy tu omawiane nie dotyczą **dziedziczenia interfejsu** (gdy klasa implementuje interfejs lub interfejs dziedziczy po innym interfejsie).

W przeciwieństwie do wywoływania metod dziedziczenie narusza hermetyzację [Synder86]. Inaczej mówiąc, aby klasa pochodna działała prawidłowo, musi ona korzystać ze szczegółów implementacji klasy bazowej. Implementacja klasy bazowej

może się zmieniać i może to powodować uszkodzenie klasy pochodnej. W konsekwencji klasa pochodna musi się zmieniać razem z klasą bazową, chyba że autor klasy bazowej odpowiednio ją zaprojektował i udokumentował, aby można było bez problemu rozszerzać taką klasę.

Aby przedstawić problem, przeanalizujemy program korzystający z klasy `HashSet`. W celu poprawienia wydajności naszego programu chcemy sprawdzić, ile elementów zostało dodane do zbioru od czasu jego utworzenia (nie należy mylić z bieżącą wielkością, która może się zmniejszać w przypadku usunięcia elementów). Aby udostępnić taką funkcję, napiszemy odmianę klasy `HashSet`, która zapamiętuje liczbę odwołań do metody wstawiającej elementy i eksportuje akcesor dla tej wartości. Klasa `HashSet` zawiera dwie metody umożliwiające dodawanie elementów, `add` i `addAll`, zatem musimy przesłonić te metody.

// Błąd - niewłaściwe wykorzystanie dziedziczenia!

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    // liczba prób wstawienia elementów
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

Klasa wygląda prawidłowo, ale nie działa. Załóżmy, że utworzymy obiekt i za pomocą metody `addAll` dodamy do niego trzy elementy. Przy okazji zwróć uwagę, że tworzymy listę za pomocą statycznej metody fabrycznej `List.of`, wprowadzonej w Javie 9; jeśli korzystasz z wcześniejszego wydania Javy, użyj metody `Arrays.asList`.

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Tytus", "Romek", "Atomek"));
```

Możemy oczekiwać, że metoda `getAddCount` zwróci w tym momencie trzy, ale w rzeczywistości zwraca sześć. Gdzie tkwi błąd? Wewnętrznie metoda `addAll` klasy `HashSet` korzysta z metody `add`. W dokumentacji klasy `HashSet` nie ma oczywiście takich szczegółów implementacji. Metoda `addAll` klasy `InstrumentedHashSet` dodaje trzy do pola `addCount` i za pomocą `super.addAll` wywołuje metodę `addAll`, pochodzącą z klasy `HashSet`. Ta z kolei wywołuje dla każdego dodawanego elementu metodę `add`, która jest również zdefiniowana w klasie `InstrumentedHashSet`. W przypadku każdego z tych trzech wywołań wartość `addCount` zostaje zwiększona o jeden, co w sumie daje zwiększenie tej wartości o sześć. Każdy element, dodawany do zbioru za pomocą metody `addAll`, jest liczony podwójnie.

Możemy „naprawić” podklasę, usuwając zdefiniowanie metody `addAll`. Chociaż nasza klasa zacznie działać prawidłowo, będzie jednak to zależało od szczegółów implementacji metody `addAll` z klasy `HashSet`. Takie działanie metody `addAll` nie jest gwarantowane we wszystkich implementacjach platformy Java i, dodatkowo, może się zmieniać w kolejnych wersjach platformy. Dlatego klasa `InstrumentedHashSet` może być zawodna.

Nieco lepsze wyniki można osiągnąć, zdefiniowując metodę `addAll` tak, aby przeglądała ona przekazaną kolekcję i dla każdego jej elementu wywoływała metodę `add`. W ten sposób można zagwarantować poprawne działanie niezależnie od tego, czy metoda `addAll` z klasy `HashSet` korzysta z metody `add`, ponieważ metoda `addAll` z klasy `HashSet` nie będzie wywoływana. Technika ta nie rozwiązuje jednak wszystkich problemów. Ponowna implementacja metod pochodzących z klasy bazowej, które mogą korzystać z zdefiniowanych metod klasy podrzędnej, jest trudna, czasochłonna i może powodować powstanie błędów. Dodatkowo nie zawsze jest to możliwe, ponieważ niektóre metody korzystają z prywatnych pól klasy bazowej, które nie są dostępne w podklasie.

Podobnym problemem, który może powodować niestabilność klasy pochodnej, jest dodanie nowych metod w kolejnej wersji klasy bazowej. Załóżmy, że program w założeniach systemu bezpieczeństwa korzysta z faktu, że elementy wstawiane do kolekcji spełniają pewne kryteria. Może być to zagwarantowane przez odziedziczenie kolekcji i zdefiniowanie wszystkich metod, które pozwalają na wstawienie elementu, dodając kontrolę kryterium przed wstawieniem elementu do kolekcji. Metoda taka będzie działać doskonale do momentu, gdy w kolejnej wersji klasy bazowej pojawi się następna metoda pozwalająca na wstawienie elementu do kolekcji. W takiej sytuacji możliwe jest wstawienie „nielegalnego” elementu, gdy klasa pochodna wywoła nową metodę, nieprzeznaczoną w podklasie. Nie jest to problem czysto teoretyczny. Niezbędne było usunięcie kilku tego typu luk w systemie bezpieczeństwa, gdy klasy `Hashtable` i `Vector` zostały przeniesione do schematu `Collections`.

Oba te problemy wynikają z zdefiniowywania metod. Możesz więc pomyśleć, że bezpiecznie można dziedziczyć po klasie dodając jedynie nowe metody i nie

próbując przeddefiniowywać już istniejących. Choć dziedziczenie tego typu jest o wiele bardziej bezpieczne, nie jest pozbawione pewnego ryzyka. Jeżeli klasa bazowa w kolejnej wersji zostanie uzupełniona nowymi metodami i będą one miały tę samą nazwę i sygnaturę, ale inny zwracany typ, klasa pochodna przestanie się kompilować [JLS, 8.4.8.3]. Jeżeli okaże się, że metoda w klasie pochodnej będzie miała tę samą nazwę i sygnaturę, metoda z klasy bazowej zostanie przeddefiniowana, co może spowodować opisane powyżej problemy. Dodatkowo bardzo wątpliwe jest, aby Twoja metoda spełniała założenia dla metody o tej samej nazwie pochodzącej z klasy bazowej, ponieważ nie były one spisane w czasie, gdy Ty pisałeś swoją klasę pochodną.

Na szczęście istnieje możliwość uniknięcia wszystkich tych problemów. Zamiast dziedziczyć po istniejącej klasie, w nowej klasie można utworzyć pole prywatne, odwołujące się do obiektu istniejącej klasy. Technika taka jest nazywana **kompozycją**, ponieważ istniejąca klasa staje się składnikiem nowej klasy. Każda metoda w nowej klasie może wywoływać odpowiednią metodę klasy składowej i zwracać otrzymane wyniki. Jest to znane pod pojęciem **przekazywania**, a metody nowej klasy nazywane są **metodami przekazującymi**. Wynikowa klasa będzie dużo solidniejsza, ponieważ nie zależy od szczegółów implementacji istniejącej klasy. Nawet dodanie nowych metod do istniejącej klasy nie ma wpływu na nową klasę. Aby zilustrować tę technikę, przedstawimy teraz ulepszoną metodę `InstrumentedHashSet`, korzystającą z kompozycji i przekazywania. Należy zwrócić uwagę, że implementacja jest podzielona na dwie części, samą klasę oraz **klasę przekazującą** wielokrotnego użytku, która zawiera jedynie metody przekazujące:

```
// Klasa osłonna - korzysta z kompozycji zamiast dziedziczenia
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

// Klasa przekazująca wielokrotnego użytku

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()           { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()       { return s.isEmpty(); }
    public int size()              { return s.size(); }
    public Iterator iterator()     { return s.iterator(); }
    public boolean add(E e)        { return s.add(e); }
    public boolean remove(Object o) { return s.remove(o); }
    public boolean containsAll(Collection<?> c)
                                { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c)
                                { return s.addAll(c); }
    public boolean removeAll(Collection<?> c)
                                { return s.removeAll(c); }
    public boolean retainAll(Collection<?> c)
                                { return s.retainAll(c); }
    public Object[] toArray()      { return s.toArray(); }
    public <T>[] toArray(T[] a)    { return s.toArray(a); }
    @Override public boolean equals(Object o)
                                { return s.equals(o); }
    @Override public int hashCode() { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }
}
```

Projekt takiej klasy `InstrumentedSet` jest możliwy dzięki istnieniu interfejsu `Set`, który zawiera wszystkie funkcje klasy `HashSet`. Oprócz tego, że klasa ta jest solidna, jest również niezmiernie elastyczna. Klasa `InstrumentedSet` implementuje interfejs `Set` oraz posiada jeden konstruktor, którego argument jest również typu `Set`. Właściwie to klasa ta przekształca jeden zbiór na inny, dodając funkcję zliczania. W przeciwieństwie do podejścia korzystającego z dziedziczenia, które działa jedynie dla jednej klasy i wymaga osobnego konstruktora dla każdego konstruktora w klasie bazowej, klasa osłonowa może współpracować z dowolną implementacją interfejsu `Set` i będzie działała z dowolnym istniejącym konstruktorem. Dla przykładu:

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

Klasa **`InstrumentedSet`** może być wykorzystywana do tymczasowego wzbogacania obiektu zbioru, który do tej pory był wykorzystywany bez tych dodatkowych funkcji:

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // w tej metodzie należy używać iDogs zamiast dogs
}
```

Klasa `InstrumentedSet` jest nazywana klasą **osłonową**, ponieważ każdy obiekt osłania osobną instancję interfejsu `Set`. Metoda ta jest również znana jako wzorzec **dekorator** (ang. *Decorator*) [Gamma95], ponieważ klasa `InstrumentedSet` „dekoruje” zbiór dodatkowymi funkcjami. Czasami połączenie kompozycji i przekazywania jest nieprawidłowo nazywane **delegacją**. Nie jest to jednak delegacja, chyba że obiekt osłonowy przekazuje samego siebie do osłanianego obiektu [Lieberman86; Gamma95].

Istnieje niewiele wad zastosowania klasy osłonowej. Jedną z niedogodności jest brak możliwości korzystania z **wywołań zwrotnych**, w których obiekt przekazuje referencję do samego siebie do innego obiektu, aby mógł on później wywołać metodę zwrotną. Ponieważ osłaniany obiekt „nie zdaje sobie sprawy” z istnienia obiektu osłonowego, przekazuje referencję do samego siebie (`this`) i wywołanie zwrotne pomija obiekt osłonowy. Jest to znane pod nazwą **problemu SELF** [Lieberman86]. Niektórzy programiści obawiają się wpływu metod przekazujących na wydajność aplikacji oraz dodatkowego zużycia pamięci na obiekty osłonowe. W praktyce wpływ ten jest pomijalny. Napisanie metod przekazujących jest dosyć nudne, ale nuda jest częściowo kompensowana faktem, że należy napisać klasę przekazującą tylko raz dla każdego interfejsu, a klasy przekazujące mogą być dostarczane przez pakiet zawierający interfejs. Przykładem może być biblioteka Guava, która oferuje klasy przekazujące dla wszystkich oferowanych interfejsów kolekcji [Guava].

Dziedziczenie jest właściwe jedynie w przypadkach, gdy klasa pochodna jest **podtypem** klasy bazowej. Inaczej mówiąc, klasa *B* może dziedziczyć po klasie *A* jedynie wtedy, gdy pomiędzy tymi klasami istnieje relacja typu generalizacja-specjalizacja. Jeżeli chcesz, aby klasa *B* dziedziczyła po klasie *A*, powinieneś odpowiedzieć na pytanie: „Czy *B* naprawdę jest tym, co *A*?”. Jeżeli nie możesz jednoznacznie odpowiedzieć na takie pytanie, klasa *B* nie powinna dziedziczyć po klasie *A*. Jeżeli odpowiedź brzmi „nie”, często klasa *B* powinna zawierać prywatny obiekt klasy *A* i udostępniać mniejsze i prostsze API. Klasa *A* nie jest częścią klasy *B*, a po prostu istnieje jako fragment implementacji.

Istnieje wiele przykładów niedotrzymania tej zasady w klasach biblioteki języka Java. Na przykład `Stos` nie jest wektorem, więc klasa `Stack` nie powinna dziedziczyć po klasie `Vector`. Podobnie lista właściwości nie jest tablicą mieszającą, więc klasa `Properties` nie powinna dziedziczyć po `Hashtable`. W obu tych przypadkach kompozycja jest lepszym rozwiązaniem.

Jeżeli skorzystasz z dziedziczenia w przypadku, gdy należy zastosować kompozycję, niepotrzebnie udostępniasz szczegóły implementacji. W wyniku tego API jest ściśle związane z oryginalną implementacją, na stałe ograniczając wydajność klasy. Poważniejszym problemem jest to, że udostępniając szczegóły implementacji pozwalasz na bezpośrednie wykorzystywanie ich przez klienty. Dodatkowo może

to prowadzić do powstania niejasnych nazw. Na przykład, jeżeli `p` oznacza obiekt `Properties`, to `p.getProperty(key)` może dawać inne wyniki od `p.get(key)`. Pierwsza metoda bierze pod uwagę wartości domyślne, natomiast druga, odziedziczona z `Hashtable`, nie korzysta z tych wartości. Klient może również uszkodzić niezmienniki klasy podrzędnej przez bezpośrednią modyfikację klasy bazowej. W przypadku klasy `Properties` projektanci założyli, że kluczami i wartościami mogą być jedynie ciągi znaków, ale odwołując się bezpośrednio do bazowej klasy `Hashtable` można naruszyć to założenie. Gdy tak się stanie, nie będzie można skorzystać z niektórych metod API klasy `Properties` (np.: `load` i `store`). Kiedy problem ten zostanie zauważony, będzie zbyt późno na jego poprawienie, ponieważ użytkownicy będą już korzystali z przechowywania kluczy i wartości niebędących ciągami.

Istnieje jeszcze jeden zestaw pytań, pomagających w podjęciu decyzji wyboru dziedziczenia lub kompozycji. Czy klasa, z której chcemy dziedziczyć, nie posiada żadnych błędów w API? Jeżeli takie błędy się znajdują, czy możesz bezpiecznie je powielić w API Twojej klasy? Dziedziczenie powoduje powielenie wszystkich błędów w API klasy, natomiast kompozycja pozwala na zaprojektowanie nowego API, ukrywającego te błędy.

Podsumujmy. Dziedziczenie jest niezmiernie pożyteczne, ale powoduje problemy z powodu naruszania hermetyzacji. Jest ono odpowiednie jedynie wtedy, gdy istnieje relacja podtypu pomiędzy klasą bazową i podklasą. Nawet wtedy dziedziczenie może prowadzić do niepewności podklasy, jeżeli klasa pochodna jest zdefiniowana w innym pakiecie, a klasa bazowa nie była zaprojektowana z myślą o dziedziczeniu. Aby uniknąć tej niestabilności należy zamiast dziedziczenia korzystać z kompozycji oraz przekazywania — szczególnie, gdy istnieje odpowiedni interfejs do implementacji. Klasy osłowne nie tylko są solidniejsze, są one również efektywniejsze.

Temat 19. Projektowanie i dokumentowanie klas przeznaczonych do dziedziczenia

W temacie 18. ostrzegaliśmy o niebezpieczeństwach dziedziczenia „obcych” klas, które nie zostały odpowiednio zaprojektowane i udokumentowane dla celów dziedziczenia. W jaki sposób należy zaprojektować i udokumentować klasę, aby można było z niej bezpiecznie dziedziczyć?

Klasa musi precyzyjnie dokumentować efekty przesłaniania każdej z metod. Oznacza to, że **klasa musi udokumentować wszystkie przypadki wykorzystania swoich metod, które można przesłaniać**. Dla każdej publicznej lub zabezpieczonej

metody albo konstruktora dokumentacja musi zawierać listę wywoływanych metod, które można przeddefiniowywać, ich kolejność oraz sposób, w jaki każde wywołanie wpływa na kolejne kroki przetwarzania (metody, które można **przeddefiniowywać**, to wszystkie metody publiczne i zabezpieczone). Mówiąc ogólniej, klasa musi udokumentować wszystkie przypadki, w których może wywołać przeddefiniowalną metodę. Wywołania mogą na przykład pochodzić z wątków tła lub statycznych metod inicjujących.

Istnieje konwencja, mówiąca, że metoda wywołująca metodę możliwą do przeddefiniowania zawiera opis tego wywołania na końcu komentarza dokumentującego. Opis stanowi specjalną część dokumentacji o nazwie „Implementation Requirements”, czyli tak zwane wymagania implementacyjne. Obecnie do jej wygenerowania służy znacznik Javadoc `@implSpec`. Opis dotyczy zwłaszcza szczegółów działania metody. Przedstawiamy tu przykład skopiowany ze specyfikacji klasy `java.util.AbstractCollection`:

```
public boolean remove(Object o)
```

Usuwa określony obiekt z tej kolekcji, o ile taki obiekt istnieje (operacja opcjonalna). Bardziej formalnie — usuwa element `e` taki, że `Objects.equals(o, e)`, jeżeli kolekcja zawiera co najmniej jeden element. Zwraca `true`, jeżeli kolekcja zawiera odpowiedni element (lub inaczej, jeżeli kolekcja zostanie zmieniona w tym wywołaniu).

Wymagania implementacyjne. Implementacja ta przegląda kolekcję, szukając określonego elementu. Jeżeli element zostanie znaleziony, jest usuwany z kolekcji za pomocą metody `remove` iteratora. Implementacja zgłasza wyjątek `UnsupportedOperationException`, jeżeli iterator zwrócony przez metodę `iterator` tej kolekcji nie implementuje metody `remove`.

Dokumentacja ta nie pozostawia wątpliwości, że przeddefiniowanie metody `iterator` wpłynie na działanie metody `remove`. Dodatkowo opisuje, w jaki sposób `Iterator` zwrócony przez metodę `iterator` wpływa na działanie metody `remove`. Porównaj to z sytuacją przedstawioną w temacie 18., kiedy programista projektujący dziedziczenie po klasie `HashSet` nie wiedział, że przeddefiniowanie metody `add` wpłynie na działanie metody `addAll`.

Czy opis taki nie stoi w sprzeczności ze stwierdzeniem, że dobra dokumentacja API powinna opisywać, **co** metoda robi, a nie, **jak** działa? Tak! Jest to konsekwencja faktu, że dziedziczenie narusza hermetyzację. Aby udokumentować klasę tak, by mogła być bezpiecznie dziedziczona, należy opisać szczegóły implementacji, które w pozostałych przypadkach mogą pozostać ukryte.

Znacznik `@implSpec` został wprowadzony w Javie 8, a w Javie 9 jest stosowany bardzo intensywnie. Choć znacznik powinien być domyślnie włączony, w Javie 9 nadal narzędzie Javadoc go ignoruje, o ile nie prześlemy w momencie jego wywołania dodatkowego przełącznika `-tag "apiNote:a:API Note:"`.

Projektowanie dla celów dziedziczenia wymaga o wiele więcej niż tylko dokumentacji użycia własnych metod. Aby umożliwić programistom tworzenie efektywnych podklas, **klasa musi zapewniać odpowiednie metody zabezpieczone, dołączone w odpowiednich miejscach metod wykonujących podstawowe zadania**. Rzadziej spotykanym podejściem jest wykorzystywanie do tego celu pól zabezpieczonych. Dla przykładu przedstawimy metodę `removeRange` z klasy `java.util.AbstractList`:

```
protected void removeRange(int fromIndex, int toIndex)
```

Metoda `removeRange` usuwa z tej listy wszystkie elementy o indeksach pomiędzy `fromIndex` (włącznie) a `toIndex`, bez indeksu `toIndex`. Przesuwa pozostałe elementy w lewo (zmniejszając ich indeksy). Wywołanie to zmniejsza listę o $(toIndex - fromIndex)$ elementów. (Jeżeli `toIndex == fromIndex`, operacja nie jest wykonywana).

Metoda ta jest wywoływana przez operację `clear` na liście i jej podlistach. Przesłonięcie tej metody w celu skorzystania z wewnętrznej implementacji listy może znacznie poprawić wydajność operacji `clear` na tej liście i jej podlistach.

Wymagania implementacyjne. Implementacja ta pobiera iterator listy umieszczony przez `fromIndex` i kolejno wywołuje `ListIterator.next` oraz `ListIterator.remove`, aż do usunięcia całego zakresu elementów. **Uwaga! Jeżeli `ListIterator.remove` posiada liniową wydajność czasową, implementacja ta będzie miała „kwadratową” wydajność czasową.**

Parametry:

- `fromIndex` — indeks pierwszego elementu do usunięcia;
- `toIndex` — indeks po ostatnim elemencie przeznaczonym do usunięcia.

Metoda ta nie jest przeznaczona dla końcowych użytkowników implementacji klasy `List`. Istnieje w celu ułatwienia utworzenia w podklasie szybkiej metody `clear`, działającej na podlistach. W razie braku metody `removeRange` klasa pochodna musiałaby korzystać z powolnej implementacji metody `clear` lub od początku przepisać cały mechanizm `subList`, co może nie być takie proste.

W jaki sposób należy określić, które zabezpieczone pola i metody należy udostępnić? Niestety, nie ma uniwersalnej odpowiedzi na to pytanie. Najlepszą metodą jest dokładne przemyślenie zadań klasy, wybór najlepszego rozwiązania, a następnie przetestowanie go za pomocą kilku klas pochodnych. Powinieneś udostępnić możliwie niewiele metod i pól zabezpieczonych, ponieważ każdy z tych elementów reprezentuje szczegół implementacji. Z drugiej strony, nie możesz udostępnić zbyt mało, ponieważ brak odpowiednich metod zabezpieczonych spowoduje, że klasa nie będzie się nadawała do dziedziczenia.

Jedynym sposobem na przetestowanie klasy zaprojektowanej do dziedziczenia, jest napisanie klasy pochodnej. Jeżeli pominiemy ważny składnik zabezpieczony,

próba napisania klasy pochodnej boleśnie ujawni naszą pomyłkę. Z drugiej strony, jeżeli powstało kilka klas pochodnych i żadna z nich nie korzysta z zabezpieczonego składnika, powinniśmy zadeklarować go jako prywatny. Doświadczenie pokazuje, że trzy klasy pochodne są zwykle wystarczające do przetestowania klasy rozszerzalnej. Jedna lub więcej klas pochodnych powinna być napisana przez inną osobę niż autor klasy bazowej.

Jeżeli projektujesz klasę, która będzie powszechnie używana, musisz sobie zdawać sprawę, że będziesz musiał już **na zawsze** utrzymywać udokumentowane przypadki użycia własnych metod. Konieczność ta utrudnia lub wręcz uniemożliwia poprawianie wydajności lub zakresu funkcji w kolejnych wersjach klasy. **Z tego powodu należy przetestować klasę przed jej opublikowaniem — przez napisanie klas dziedziczących po niej.**

Trzeba również pamiętać, że dodatkowa dokumentacja wymagana dla prawidłowej obsługi dziedziczenia zwiększa objętość zwykłej dokumentacji, przeznaczonej dla programistów tworzących jedynie obiekty danej klasy i wywołujących ich metody. W czasie pisania tej książki nie było narzędzi ani konwencji komentowania kodu, pozwalających oddzielić dokumentację API od informacji, interesujących jedynie programistów tworzących podklasy.

Istnieje jeszcze kilka ograniczeń, jakie klasa musi spełniać, aby można było ją dziedziczyć. **Konstruktory nie mogą wywoływać metod, które można przesłaniać**, zarówno bezpośrednio, jak i pośrednio. Jeżeli zasada ta zostanie naruszona, prawdopodobne jest, że program nie będzie działał. Konstruktor klasy nadrzędnej jest wywoływany przed konstruktorem podklasy. Jeżeli zdefiniowana metoda opiera się na wartościach, które są przypisywane w konstruktorze podklasy, nie będzie działała w przewidywany sposób. Aby przedstawić taką sytuację, przyjrzyjmy się niewielkiej klasie naruszającej tę zasadę.

```
public class Super {  
    // Błąd - konstruktor wywołuje metodę, którą można przesłonić  
    public Super() {  
        overrideMe();  
    }  
  
    public void overrideMe() {  
    }  
}
```

Klasa podrzędna przesłania metodę `overrideMe`, która nieprawidłowo jest wywoływana przez konstruktor klasy `Super`.

```
final class Sub extends Super {  
    // puste pole final ustawiane przez konstruktor  
    private final Date date;  
}
```

```
Sub() {  
    date = new Date();  
}  
  
// Przesłanie metody wywołanej przez konstruktor klasy bazowej  
@Override public void overrideMe() {  
    System.out.println(date);  
}  
  
public static void main(String[] args) {  
    Sub s = new Sub();  
    s.overrideMe();  
}  
}
```

Można się spodziewać, że program dwukrotnie wypisze datę, ale za pierwszym razem wypisywane jest **null**, ponieważ **overrideMe** jest wywoływane przez konstruktor **Super**, zanim konstruktor **Sub** zainicjuje pole **date**. Program uchwycił pole typu **final** w dwóch różnych stanach. Należy również zauważyć, że jeżeli **overrideMe** wywoła dowolną metodę na **datach**, w momencie wywołania metody **overrideMe** w konstruktorze **Super** wywołanie to zgłosi wyjątek **NullPointerException**. Jedynym powodem, dla którego program nie zgłasza wyjątku **NullPointerException**, jest to, że metoda **println** zawiera specjalną obsługę argumentów **null**.

Pamiętaj, że bezpieczne **jest** wywoływanie metod prywatnych, metod oznaczonych jako **final** i metod statycznych z poziomu konstruktora, bo żadnej z nich nie można przesłonić.

Interfejsy **Cloneable** i **Serializable** sprawiają dodatkową trudność przy projektowaniu z uwzględnieniem dziedziczenia. Najczęściej nie jest zalecane implementowanie tych interfejsów, ponieważ powodują znaczne obciążenie dla programistów rozszerzających klasę. Istnieją jednak sposoby na to, aby pozwolić podklasie implementować te interfejsy. Sposoby te są opisane w tematach 13. i 86.

Jeżeli zdecydujesz się na implementację interfejsów **Cloneable** lub **Serializable** w klasie będącej podstawą dziedziczenia, powinieneś pamiętać, że metody **clone** i **readObject** działają podobnie do konstruktorów i dlatego trzeba stosować się do podobnego ograniczenia. **Ani metoda **clone**, ani **readObject** nie mogą wywoływać metod, które mogą być przesłaniane, zarówno bezpośrednio, jak i pośrednio.** W przypadku metody **readObject** przeddefiniowana metoda zostanie uruchomiona, zanim zostanie przeprowadzona deserializacja stanu obiektu. W przypadku metody **clone** przeddefiniowana metoda będzie uruchomiona, zanim metoda **clone** z klasy pochodnej będzie mogła zmienić stan duplikowanego obiektu. W obu tych przypadkach może to doprowadzić do wystąpienia błędu w aplikacji. W przypadku metody **clone** błąd może spowodować uszkodzenie zarówno kopii obiektu, jak i oryginału. Może się to zdarzyć, gdy na przykład nadpisana metoda zakłada, że modyfikuje kopię głębokiej struktury obiektu, ale kopia nie została jeszcze wykonana.

Na koniec, jeżeli decydujesz się implementować interfejs `Serializable` w klasie podlegającej dziedziczeniu, a klasa posiada metodę `readResolve` lub `writeReplace`, musisz zadeklarować metody `readResolve` lub `writeReplace` jako zabezpieczone, a nie jako prywatne. Jeżeli metody te będą prywatne, zostaną bez żadnego ostrzeżenia zignorowane w klasach pochodnych. Jest to kolejny przypadek, gdy szczegóły implementacji stają się częścią API klasy dopuszczającej dziedziczenie.

Do tej pory powinno być oczywiste, że **projektowanie klasy przeznaczonej do dziedziczenia powoduje konieczność wprowadzenia w niej znacznych ograniczeń**. Nie są to decyzje, które można łatwo podjąć. Istnieją sytuacje, w których jest to właściwe, jak na przykład klasy abstrakcyjne, w tym **szkieletowe implementacje** interfejsów (temat 20.). Istnieją również sytuacje, w których jest to niewłaściwe, na przykład klasy niezmiennie (temat 17.).

Co w przypadku zwykłych klas użytkowych? Zwykle nie są one zadeklarowane jako `final`, a ich projekt oraz dokumentacja nie uwzględniają specjalnie dziedziczenia. Za każdym razem, gdy w klasie tej jest wprowadzana zamiana, istnieje szansa, że klient rozszerzający tę klasę przestanie działać. Nie jest to problem teoretyczny. Dostyc często zdarza się po modyfikacji klasy nieoznaczonej jako `final` i niezaprojektowanej do dziedziczenia otrzymywanie raportów o błędach, związane właśnie z dziedziczeniem.

Najlepszym rozwiązaniem tego problemu jest zabronienie dziedziczenia w klasach, które nie są do tego celu zaprojektowane ani nie posiadają odpowiedniej dokumentacji. Istnieją dwa sposoby na zablokowanie możliwości dziedziczenia. Najłatwiej jest zadeklarować klasę jako `final`. Drugą metodą jest zadeklarowanie wszystkich konstruktorów jako prywatnych lub prywatnych w ramach pakietu i dodanie publicznych statycznych metod fabrycznych, pełniących funkcje konstruktorów. Metoda ta, umożliwiającą wewnętrzne korzystanie z klas pochodnych, opisana jest w temacie 17. Oba te podejścia są akceptowalne.

Porada ta może być nieco kontrowersyjna, ponieważ wielu programistów przyzwyczało się do dziedziczenia zwykłych klas w celu dodania funkcji, powiadomień, synchronizacji lub ograniczenia zestawu funkcji. Jeżeli klasa implementuje interfejs, który zawiera zasadnicze funkcje klasy, na przykład `Set`, `List` lub `Map`, nie powinieneś mieć żadnych oporów przed zablokowaniem dziedziczenia. **Klasy osłowne**, opisane w temacie 18., są świetną alternatywą dla dziedziczenia i zmiany zestawu funkcji.

Jeżeli klasa nie implementuje standardowego interfejsu, to dla niektórych programistów zablokowanie dziedziczenia może być niewygodne. Jeżeli uważasz, że musisz umożliwić dziedziczenie z tej klasy, jedynym rozsądnym podejściem jest upewnienie się, że klasa nie wywołuje swoich metod, które mogą być przesłonięte, i udokumentowanie tego faktu. Inaczej mówiąc, należy wyeliminować wewnętrzne wywołania

metod, które można przeddefiniowywać. Stosując takie środki ostrożności, można utworzyć klasę, której dziedziczenie jest stosunkowo bezpieczne. Przesłonięcie metody nigdy nie wpłynie na działanie innych metod.

Wyeliminowanie wewnętrznych wywołań w klasie może być zrealizowane czysto mechanicznie, bez zmiany jej działania. Wystarczy przenieść treść przeddefiniowalnej metody do prywatnej „metody pomocniczej” i w każdej z przeddefiniowalnych metod wywołać metodę pomocniczą. Następnie usunąć wewnętrzne wywołania przeddefiniowalnych metod, zastępując je wywołaniami metod pomocniczych.

Podsumujmy. Projektowanie klasy na potrzeby dziedziczenia to naprawdę żmudne zajęcie. Trzeba udokumentować wszystkie wywołania własnych metod, a po ich opisaniu trzeba się trzymać tego rozwiązania przez całe życie klasy. Jeśli tego nie zrobimy, podklasy staną się zależne od szczegółów implementacji klasy nadrzędnej i mogą przestać działać, gdy szczegóły jej implementacji się zmieniają. Aby ułatwić innym tworzenie **wydajnych** podklas, można wyeksportować jedną lub dwie metody chronione. Jeśli nie widzisz oczywistego powodu tworzenia podklas na podstawie projektowanej klasy, najlepiej będzie zablokować możliwość dziedziczenia przez oznaczenie klasy jako `final` lub utworzyć tylko i wyłącznie prywatne konstruktory klasy.

Temat 20. Stosowanie interfejsów zamiast klas abstrakcyjnych

Język Java posiada dwa mechanizmy umożliwiające definiowanie typów pozwalających na tworzenie wielu implementacji — interfejsy i klasy abstrakcyjne. Najbardziej oczywistą różnicą pomiędzy tymi dwoma mechanizmami jest to, że klasy abstrakcyjne mogą zawierać implementację niektórych metod, natomiast interfejsy nie mogą zawierać żadnych implementacji. Dużo ważniejszą różnicą jest to, że implementując typ, definiowany przez klasę abstrakcyjną, klasa musi być podklasą klasy abstrakcyjnej. Dowolna klasa definiująca wszystkie wymagane metody i spełniająca ogólne założenia może implementować interfejs, niezależnie od jej umiejscowienia w hierarchii klas. Ponieważ Java pozwala jedynie na dziedziczenie jednobazowe, ograniczenie dotyczące klas abstrakcyjnych znacznie zmniejsza zakres ich zastosowań jako definicji typów. Implementować interfejs może natomiast dowolna klasa definiująca wszystkie wymagane metody i stosująca się do ogólnego kontraktu; nie ma tu znaczenia położenie klasy w hierarchii klas.

Istniejące klasy mogą być łatwo rozszerzone w celu implementacji nowego interfejsu. Wystarczy dodać wszystkie potrzebne metody, które jeszcze nie zostały zdefiniowane, i umieścić klauzulę `implements` w deklaracji klasy. Na przykład po

wprowadzeniu interfejsów Comparable, Iterable i Autocloseable do biblioteki języka wiele istniejących klas zostało uzupełnionych o ich implementację. Istniejące metody nie mogą najczęściej być rozszerzane o dziedziczenie po nowej klasie abstrakcyjnej. Jeżeli chcesz, aby dwie klasy dziedziczyły po tej samej klasie abstrakcyjnej, musisz umieścić klasę abstrakcyjną w hierarchii klas w takim miejscu, aby jej podklasy były przodkami obu interesujących nas klas. Niestety, spowoduje to ogromne skutki uboczne w hierarchii typów, wymuszając na wszystkich potomkach wspólnego przodka dziedziczenie po nowej klasie abstrakcyjnej, niezależnie od tego, czy jest to potrzebne, czy nie.

Interfejsy są idealne do definiowania klas mieszanych. Klasa mieszana to klasa, która oprócz implementacji „typu pierwotnego” deklaruje implementację dodatkowego zestawu funkcji. Na przykład Comparable jest interfejsem mieszanym, pozwalającym klasie zadeklarować, że obiekt może być porównywany z innymi, wzajemnie porównywalnymi obiektami. Interfejs ten jest nazywany mieszanym, ponieważ pozwala na „domieszanie” dodatkowych funkcji do pierwotnego zestawu funkcji typu. Klasy abstrakcyjne nie mogą być wykorzystywane do definiowania klas mieszanych. Klasy nie mogą posiadać więcej niż jednego przodka i nie ma w hierarchii klas odpowiedniego miejsca, w którym można dodać klasę mieszaną.

Interfejsy pozwalają na tworzenie niehierarchicznych modeli typów. Hierarchia typów jest doskonała do organizacji niektórych typów, ale niektórych z nich nie można umieścić w sztywnej hierarchii. Załóżmy na przykład, że mamy interfejs reprezentujący piosenkarza i inny, reprezentujący autora piosenek.

```
public interface Singer {
    AudioClip sing(Song s);
}

public interface Songwriter {
    Song compose(boolean hit);
}
```

Piosenkarze mogą być równocześnie autorami piosenek. Ponieważ do definiowania tych typów skorzystaliśmy z interfejsów, a nie klas abstrakcyjnych, można utworzyć klasę implementującą zarówno interfejs Singer, jak i Songwriter. Możemy również utworzyć nowy interfejs, dziedziczący po obu interfejsach i dodający nowe metody, właściwe dla takiej kombinacji.

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

Nie zawsze potrzebny jest taki poziom elastyczności, ale w takich przypadkach wykorzystanie interfejsów jest jedyną możliwością. Alternatywą jest utworzenie napęczniałej hierarchii klas, zawierającej wszystkie możliwe kombinacje atrybutów.

Jeżeli w systemie typów znajduje się n atrybutów, istnieje 2^n ich kombinacji. Zjawisko takie jest znane pod nazwą **eksplozji typów**. Napęczniała hierarchia klas może powodować powstanie podobnych klas zawierających wiele metod i różniących się jedynie typem argumentów, ponieważ nie ma w hierarchii klas uniwersalnego typu, obejmującego wszystkie cechy.

Interfejsy pozwalają na bezpieczne i efektywne rozszerzanie zestawu funkcji poprzez **klasy osłonowe**, opisane w temacie 18. Jeżeli do definiowania typów wykorzystasz klasy abstrakcyjne, nie pozostawiasz programiście chcącemu rozszerzyć zestaw funkcji innej drogi, jak tylko skorzystanie z dziedziczenia. W wyniku tego powstała klasa jest mniej wydajna i mniej niezawodna niż klasa osłonowa.

Jeśli istnieje oczywisty sposób implementacji metody interfejsu na podstawie innych metod interfejsu, rozważ wspomnienie użytkowników klasy przez dostarczenie metody domyślnej. Przykładem tej techniki jest metoda `removeIf` prezentowana kilka stron dalej. Jeśli zdecydujesz się na dostarczenie metod domyślnych, pamiętaj o ich udokumentowaniu na potrzeby dziedziczenia za pomocą znacznika `JavaDoc @implSpec` (temat 19.).

Istnieje jednak ograniczona pomoc implementacyjna, którą można zaoferować dzięki metodom domyślnym. Choć wiele interfejsów określa zachowanie metod klasy `Object`, takich jak `equals` i `hashCode`, nie można dla nich zaoferować metod domyślnych. Interfejsy nie mogą zawierać pól instancyjnych lub niepublicznych składowych statycznych (z wyjątkiem prywatnych metod statycznych). Nie można też dodać metod domyślnych do interfejsu, którego się nie kontroluje.

Można połączyć zalety interfejsów i klas abstrakcyjnych, tworząc abstrakcyjne **szkieletowe implementacje klas** dla każdego bardziej skomplikowanego interfejsu, który jest udostępniany programistom. Interfejs nadal definiuje typ i być może dostarcza pewne metody domyślne, a szkieletowa implementacja przejmuję zadanie implementacji pozostałych metod interfejsu, bazując również na metodach domyślnych z interfejsu. Rozszerzenie implementacji szkieletowej uwalnia programistę konkretnej klasy od większości prac związanych z implementacją interfejsu. To wzorzec nazywany **metodą szablonową** (*Template Method*) [Gamma95].

Według standardowej konwencji implementacje szkieletowe noszą nazwy o postaci `AbstractInterface`, gdzie *Interface* jest nazwą implementowanego interfejsu. Na przykład biblioteka `Collections` zawiera szkieletowe implementacje dla wszystkich głównych interfejsów kolekcji: `AbstractCollection`, `AbstractSet`, `AbstractList` i `AbstractMap`. Można oczywiście spierać się, że odpowiednie byłyby nazwy `SkeletalCollection`, `SkeletalSet`, `SkeletalList` oraz `SkeletalMap`, ale konwencja `Abstract` jest obecnie dobrze ugruntowana. Jeżeli szkieletowa implementacja jest dobrze zaprojektowana (czy to jako osobna klasa abstrakcyjna, czy jedynie jako metody domyślne interfejsu), pozwala na **bardzo** szybkie tworzenie własnych implementacji

interfejsów. Dla przykładu przedstawiamy statyczną metodę fabryczną, zawierającą kompletną, działającą implementację interfejsu `List`, realizowaną na podstawie `AbstractList`:

```
// Konkretna implementacja bazująca na implementacji szkieletowej
static List<Integer> intArrayAsList(int[] a) {
    Objects.requireNonNull(a);

    // Użycie w tym przypadku operatora rombu bez typu jest dopuszczalne
    // tylko w Javie 9,
    // więc jeśli korzystasz ze wcześniejszych wydać, użyj <Integer>.
    return new AbstractList<>() {
        @Override public Integer get(int i) {
            return a[i]; // autoboxing (temat 5.)
        }

        @Override public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val; // automatyczny unboxing
            return oldVal; // autoboxing
        }

        @Override public int size() {
            return a.length;
        }
    };
}
```

Gdy zdasz sobie sprawę, że wszystkie implementacje `List` mogą być przez Ciebie wykorzystane, przykład ten staje się imponującym przykładem efektywności implementacji szkieletowych. Przypadkowo, przykład ten jest **adapterem** [Gamma95], który pozwala przeglądać tablicę liczb `int` jako listy obiektów `Integer`. Z powodu ciągłych konwersji pomiędzy wartościami `int` i obiektami `Integer` (boxing i unboxing) wydajność nie jest zbyt dobra. Należy zwrócić uwagę, że implementacja przyjmuje formę **klasy anonimowej** (temat 24.).

Elegancja implementacji szkieletowej polega na tym, że pozwala pomóc w implementacji klasy abstrakcyjnej bez wprowadzania poważnych ograniczeń do klasy abstrakcyjnej, służącej jako definicja typu. W przypadku większości implementacji interfejsów rozszerzenie implementacji szkieletowej jest najbardziej naturalnym działaniem, choć w pełni dobrowolnym. Jeżeli istniejąca klasa nie może rozszerzać szkieletowej implementacji, programista może samodzielnie napisać kompletną implementację interfejsu. Klasa może bez przeszkód korzystać z pomocy oferowanej przez metody domyślne. Dodatkowo implementacja szkieletowa nadal może pomagać przy implementacji. Klasa implementująca interfejs może przekazywać wywołania metod interfejsu do wewnętrznej prywatnej instancji klasy, która dziedziczy po implementacji szkieletowej. Technika ta, znana pod nazwą **symulowania**

dziedziczenia wielobazowego, jest ściśle związana z klasami osłonowymi, omawianymi w temacie 18. Pozwala to na wykorzystanie większości zalet dziedziczenia wielobazowego przy jednoczesnym uniknięciu jego wad.

Tworzenie implementacji szkieletowej jest dosyć proste, a czasami nawet nużące. Najpierw należy przeanalizować interfejs i zdecydować, które metody są metodami pierwotnymi, czyli takimi, za pomocą których można utworzyć pozostałe metody. Te metody pierwotne będą metodami abstrakcyjnymi w naszej implementacji szkieletowej. Następnie zapewniamy w interfejsie metody domyślne dla wszystkich metod, które mogą powstać w całości na bazie wcześniejszych metod pierwotnych. Przypomnijmy, że nie możemy w ten sposób dostarczyć metod domyślnych dla metod z `Object` takich jak `equals` i `hashCode`. Jeśli metody pierwotne i domyślne w pełni opisują interfejs, nie potrzebujemy osobnej klasy implementacji szkieletowej. W przeciwnym razie trzeba napisać klasę zadeklarowaną jako implementacja interfejsu wraz z implementacjami wszystkich pozostałych metod interfejsu. Klasa może zawierać dowolne niepubliczne pola i metody niezbędne do realizacji zadania.

Jako przykład rozważmy interfejs `Map.Entry`. Oczywistymi metodami pierwotnymi są `getKey`, `getValue` i (opcjonalnie) `setValue`. Interfejs określa zachowanie metod `equals` i `hashCode`, a na podstawie tych metod pierwotnych można wskazać oczywistą implementację metody `toString`. Ponieważ nie możemy dostarczyć domyślnych implementacji dla metod klasy `Object`, wszystkie implementacje trafią do klasy implementacji szkieletowej:

```
// Klasa implementacji szkieletowej
public abstract class AbstractMapEntry<K, V>
    implements Map.Entry<K, V> {
    // wpisy w modyfikowalnych odwzorowaniach muszą przedefiniować tę metodę
    @Override public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // implementacja ogólnych założeń dla Map.Entry.equals
    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (! (o instanceof Map.Entry))
            return false;
        Map.Entry<?, ?> arg = (Map.Entry)o;
        return Objects.equals(e.getKey(), arg.getKey())
            && Objects.equals(e.getValue(), arg.getValue());
    }

    // implementacja ogólnych założeń dla Map.Entry.hashCode
    @Override public int hashCode() {
        return Objects.hashCode(getKey())
            ^ Objects.hashCode(getValue());
    }
}
```

```
@Override public String toString() {  
    return getKey() + "=" + getValue();  
}  
}
```

Zwróć uwagę, że tej implementacji szkieletowej nie można zaimplementować w interfejsie `Map.Entry` lub innym interfejsie po nim dziedziczącym, ponieważ metody domyślne nie mogą przesłaniać metod klasy `Object`, takich jak `equals`, `hashCode` lub `toString`.

Ponieważ szkieletowa implementacja jest wykorzystywana jako baza dla klas podrzędnych, powinienś stosować wskazówki dotyczące projektowania i dokumentowania, przedstawione w temacie 19. W celu zmniejszenia objętości komentarze dokumentujące zostały usunięte z przedstawionego kodu, ale **dobra dokumentacja jest w przypadku szkieletowych implementacji absolutnie niezbędna**. Nie ma tu znaczenia, czy przyjmuje ona formę metod domyślnych, czy osobnej klasy abstrakcyjnej.

Odmianą implementacji szkieletowej jest **prosta implementacja**, której przykładem jest `AbstractMap.SimpleEntry`. Prosta implementacja jest podobna do implementacji szkieletowej, ponieważ implementuje ona interfejs i jest zaprojektowana do dziedziczenia, ale różni się tym, że nie jest abstrakcyjna — jest to najprostsza możliwa działająca implementacja. Można użyć jej jako podstawy lub odziedziczyć ją jako warunek przypadku.

Podsumujmy. Interfejsy są zwykle najlepszą metodą definiowania typów umożliwiających tworzenie wielu implementacji. Jeżeli udostępniasz nietrywialny interfejs, powinienś poważnie pomyśleć o utworzeniu odpowiedniej implementacji szkieletowej. Jeśli tylko to możliwe, zapewnij taką implementację szkieletową w postaci metod domyślnych interfejsu, aby każdy implementujący mógł z nich skorzystać. Z drugiej strony ograniczenia związane z interfejsami powodują, że w wielu sytuacjach implementacja szkieletowa przyjmuje formę klasy abstrakcyjnej.

Temat 21. Projektowanie interfejsów na długie lata

Przed Javą 8 nie było możliwe dodanie metod do interfejsu bez uszkodzania jego istniejących implementacji. Po dodaniu nowej metody do interfejsu jego aktualne implementacje nie posiadałyby tej metody, co prowadziłoby do błędu kompilacji. W Javie 8 wprowadzono konstrukcję **metody domyślnej** [JLS 9.4] z myślą o umożliwieniu dodawania nowych metod do już istniejących interfejsów. W rzeczywistości dodawanie nowych metod do istniejących interfejsów jest obarczone ryzykiem.

Deklaracja metody domyślnej zawiera **implementację domyślną**, wykorzystywaną przez wszystkie klasy implementujące interfejs, ale nieimplementujące tej metody domyślnej. Choć wprowadzenie do Javy metod domyślnych pozwala na wzbogacenie interfejsu o nowe metody, nie ma gwarancji, że te metody będą działały prawidłowo we wszystkich dawniejszych implementacjach. Metody domyślne zostają „wstrzyknięte” do istniejących implementacji bez wiedzy i zgody ich implementatorów. Przed Javą 8 implementacje były pisane przy założeniu, że interfejsy **nigdy** nie wzbogacą się o nowe metody.

Wiele nowych metod domyślnych wprowadzono w Javie 8 do głównych interfejsów kolekcji, aby wspomóc użycie lambd (rozdział 7.). Metody domyślne bibliotek Javy to wysokiej jakości implementacje ogólnego stosowania, które w większości sytuacji działają poprawnie. Niestety, **nie zawsze możliwe jest napisanie metody domyślnej zachowującej wszystkie inwarianty dla każdego możliwego rodzaju implementacji**.

Rozważmy na przykład metodę `removeIf`, która została dodana w Javie 8 do interfejsu `Collection`. Metoda usuwa wszystkie elementy, dla których pewna funkcja logiczna (lub **predykat**) zwróci `true`. Domyślna implementacja jest napisana tak, że do poruszania się po kolekcji używa iteratora, wywołuje predykat dla każdego elementu, a następnie używa metody `remove` iteratora, aby usunąć element, jeśli predykat zwrócił dla niego wartość `true`. Prawdopodobnie deklaracja wygląda podobnie do poniższej:

```
// Metoda domyślna dodana do interfejsu Collection w Javie 8
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;
    for (Iterator<E> it = iterator(); it.hasNext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

To najlepsza ogólna implementacja, którą ktokolwiek mógłby napisać dla metody `removeIf`. Niestety, w przypadku niektórych rzeczywistych implementacji `Collection` nie działa prawidłowo. Rozważmy klasę `org.apache.commons.collections4.collection.SynchronizedCollection`. Klasa ta, pochodząca z biblioteki Apache Commons, przypomina klasę zwracaną przez fabrykę statyczną `Collections.synchronizedCollection` z `java.util`. Wersja z Apache pozwala dodatkowo na użycie dostarczonego przez klienta obiektu blokad zamiast kolekcji. Możemy powiedzieć, że jest to klasa osłonowa (temat 18.), której wszystkie metody synchronizują się na podstawie obiektu blokady, zanim delegują pracę do otaczanej kolekcji.

Choć klasa `SynchronizedCollection` z biblioteki Apache nadal jest aktywnie wspierana, to gdy powstawał ten tekst, nadal nie przesłaniała metody `removeIf`. Jeśli klasa zostanie użyta w połączeniu z Javą 8, otrzyma domyślną implementację `removeIf`, która nie realizuje (bo nie **potrafi**) głównego zobowiązania klasy — automatycznej synchronizacji wokół każdego wywołania metody klasy. Domyślna implementacja nie wie nic na temat synchronizacji i nie ma dostępu do pola zawierającego obiekt blokady. Jeśli klient wywoła metodę `removeIf` na instancji `SynchronizedCollection` w przypadku wystąpienia współbieżnej modyfikacji kolekcji przez inny wątek, może dojść do zgłoszenia wyjątku `ConcurrentModificationException` lub innej nieokreślonej sytuacji.

Aby zapobiec takiej sytuacji w implementacjach bibliotek platformy Java, np. takich jak prywatna na poziomie pakietu klasa zwracana przez `Collections.synchronizedCollection`, twórcy JDK muszą przesłonić domyślną implementację `removeIf` i innych podobnych metod, aby zapewnić właściwą synchronizację przed wywołaniem implementacji domyślnej. Wcześniejsze implementacje kolekcji, które nie stanowią części platformy Java, nie miały okazji wprowadzić podobnych zmian przed wydaniem nowej platformy, a część bibliotek do tej pory nie nadrobiła zaległości.

W przypadku wystąpienia metod domyślnych istniejąca implementacja interfejsu może się kompilować bez błędów lub ostrzeżeń, ale nie działać prawidłowo po uruchomieniu. Choć nie jest to sytuacja częsta, sam problem nie daje się wyizolować do pojedynczych incydentów. Wiadomo, że część metod dodanych do interfejsów kolekcji w Javie 8 jest podejrzana i znane są istniejące implementacje, na które nowe metody wpływają negatywnie.

Wykorzystywania metod domyślnych do wprowadzenia nowych metod do istniejących interfejsów powinno się unikać, chyba że zmiana jest naprawdę istotna. Wtedy warto się dobrze zastanowić, czy jakaś istniejąca implementacja interfejsu nie przestanie działać poprawnie w efekcie wprowadzenia metody domyślnej. Z drugiej strony metody domyślne są bardzo użyteczne w zapewnianiu standardowych implementacji metod w momencie tworzenia interfejsu, co znacząco upraszcza późniejszą implementację interfejsu (temat 20.).

Warto w tym miejscu podkreślić, że metody domyślne nie były projektowane do obsługi usuwania metod z interfejsów lub do zmiany sygnatur już istniejących metod. Żadnej z tych zmian nie uda się przeprowadzić bez zepsucia istniejących klientów interfejsu.

Morał z całej historii jest oczywisty. Choć obecnie metody domyślne stanowią część platformy Java, **nadal niezwykle istotne jest projektowanie interfejsów z ogromną dbałością.** Choć metody domyślne pozwalają na dodanie metod do istniejących interfejsów, sięgnięcie po to rozwiązanie obarczone jest ryzykiem. Jeśli interfejs zawiera drobną skazę, może już na zawsze irytować użytkowników. Jeśli interfejs jest zbyt ubogi, może spowodować porażkę dostarczanego API.

Z wyżej wymienionych powodów bardzo ważne jest sprawdzanie każdego nowego interfejsu przed jego wydaniem. Przynajmniej kilku programistów powinno zaimplementować interfejs na kilka różnych sposobów. Jako minimum warto postawić sobie za cel przynajmniej trzy różnorodne implementacje. Równie ważne jest napisanie przynajmniej kilku programów klienckich wykorzystujących interfejs z kilku różnych perspektyw. W ten sposób upewnimy się, że interfejs spełnia wszystkie wymogi i działa poprawnie. Dzięki takiemu rygorowi wychwycimy problemy w interfejsie przed jego wydaniem, co pozwoli na łatwe wprowadzenie poprawek. **Choć czasem niektóre usterki interfejsu można poprawić po jego wydaniu, to należy tego unikać.**

Temat 22. Wykorzystanie interfejsów jedynie do definiowania typów

Gdy klasa implementuje interfejs, interfejs ten służy jako **typ**, który jest wykorzystywany do odwoływania się do obiektów klasy. Klasa ta powinna określić operacje, jakie klient może wykonywać na obiektach. Niewłaściwe jest definiowanie interfejsów przeznaczonych do innych celów.

Jednym z rodzajów interfejsów, które nie spełniają tego założenia, są tak zwane **interfejsy stałych**. Interfejsy takie nie zawierają metod, składają się jedynie ze statycznych pól final, z których każde udostępnia stałą. Klasy korzystające z tych stałych implementują interfejs, co pozwala uniknąć kwalifikowania nazw stałych nazwą klasy. Na przykład:

```
// Interfejs stałych - nie używać!  
public interface PhysicalConstants {  
    // liczba Avogadra (1/mol)  
    static final double AVOGADROS_NUMBER = 6.022_140_857e23;  
  
    // stała Boltzmannna (J/K)  
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;  
  
    // masa elektronu (kg)  
    static final double ELECTRON_MASS = 9.109_383_56e-31;  
}
```

Użycie interfejsu stałych jest niewłaściwym zastosowaniem interfejsów. To, że klasa wewnętrznie korzysta ze stałych, jest częścią implementacji. Implementując interfejs stałych, powodujemy, że szczegóły implementacji przenikają do eksportowanego API klasy. Implementowanie interfejsu stałych nie powoduje żadnych ubocznych skutków dla użytkownika klasy, może go jedynie mylić. Jeżeli w przyszłych wersjach klasa zostanie tak zmodyfikowana, że stałe przestaną być potrzebne, w celu zapewnienia zgodności binarnej nadal trzeba implementować interfejs. Jeżeli klasa implementująca implementuje interfejs stałych, wszystkie jej podklasy będą miały przestrzeń nazw zaśmiecone stałymi, zdefiniowanymi w interfejsie.

W bibliotekach platformy Java istnieje kilka interfejsów stałych, na przykład `java.io.ObjectStreamConstants`. Interfejsy te powinny być traktowane jako anomalie i nie powinny być naśladowane.

Jeżeli chcesz udostępnić stałe, istnieje kilka rozsądnych sposobów. Jeżeli stałe są ściśle związane z klasą lub interfejsem, należy je dodać do klasy lub interfejsu. Wszystkie numeryczne klasy osłonowe w bibliotekach języka Java, takie jak `Integer` czy `Float`, udostępniają stałe `MIN_VALUE` i `MAX_VALUE`. Jeżeli stałe mają być dostępne jako składniki typu wyliczeniowego, najlepiej udostępnić je w postaci klasy obsługującej typ wyliczeniowy (temat 34.). W przeciwnym wypadku należy udostępnić stałe w postaci **klasy użytkowej**, która nie pozwala na utworzenie obiektów (temat 4.). Przedstawiamy poniżej klasę `PhysicalConstants`, która ma działanie identyczne, jak przedstawione w poprzednim przykładzie:

```
// Klasa użytkowa ze stałymi
package com.effectivejava.science;

public class PhysicalConstants {
    private PhysicalConstants() { } // zablokowanie możliwości
                                   // tworzenia obiektów

    public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    public static final double BOLTZMANN_CONST = 1.380_648_52e-23;
    public static final double ELECTRON_MASS   = 9.109_383_56e-31;
}
```

Przy okazji zwróć uwagę na użycie znaku podkreślenia (`_`) w literalach liczbowych. Podkreślenia, które mogą być stosowane od Javy 7, nie zmieniają wartości liczb, ale mogą przyczynić się do ich lepszej czytelności, o ile są stosowane właściwie. Rozważ dodawanie podkreśleń do liczb (zarówno stało- jak i zmiennoprzecinkowych), jeśli zawierają pięć lub więcej cyfr. W przypadku liczb dziesiętnych przyjęło się stosowanie podkreśleń co trzy cyfry, aby w ten sposób wyróżnić poszczególne potęgi tysiąca.

Normalnie klasa użytkowa wymaga od klientów kwalifikowania nazw stałych za pomocą nazwy klasy, na przykład `PhysicalConstants.AVOGADROS_NUMBER`. Jeżeli intensywnie wykorzystujemy stałe eksportowane przez klasę użytkową, można uniknąć potrzeby kwalifikowania stałej nazwą klasy przez skorzystanie ze **statycznych importów**.

```
// Użycie statycznych importów w celu uniknięcia kwalifikowania stałych
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }
}
```



```

...
    // wiele innych zastosowań PhysicalConstants uzasadnia statyczny import
}

```

Podsumujmy. Interfejsy powinny być stosowane jedynie do definiowania typów. Nie powinny być wykorzystywane do eksportowania stałych.

Temat 23. Zastępowanie oznaczania klas hierarchią

Czasami zdarza się nam użyć klasy, której obiekty są dwóch lub więcej rodzajów i rodzaj ten jest rozpoznawany z użyciem pola **znacznika**. Przykładem może być poniższa klasa, reprezentująca okrąg lub prostokąt:

```

// Klasa oznaczana - znacznie gorsza niż hierarchia klas!
class Figure {
    enum Shape { RECTANGLE, CIRCLE };

    // pole znacznika - kształt figury
    final Shape shape;

    // te pola są używane tylko wtedy, gdy kształtem jest RECTANGLE
    double length;
    double width;

    // te pola są używane tylko wtedy, gdy kształtem jest CIRCLE
    double radius;

    // konstruktor dla okręgu
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // konstruktor dla prostokąta
    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }

    double area() {
        switch(shape) {
            case RECTANGLE:
                return length * width;
            case CIRCLE:
                return Math.PI * (radius * radius);
        }
    }
}

```

```

        default:
            throw new AssertionError(shape);
    }
}

```

Takie **klasy oznaczane** mają wiele wad. Są one zatłoczone mechanizmami obsługi oznaczeń, takich jak deklaracje enum, pola znacznikowe i instrukcje switch. Czytelność jest ograniczona, ponieważ w jednej klasie jest upchanych wiele implementacji. Dodatkowo zwiększone jest zużycie pamięci, ponieważ obiekty zawierają niepotrzebne pola, wykorzystywane przez inne odmiany. Pola nie mogą być oznaczone jako final, chyba że konstruktory będą inicjowały niepotrzebne pola, co powoduje powstanie kolejnego niepotrzebnego kodu. Konstruktory muszą ustawiać pole i inicjować prawidłowe pola danych bez żadnego wsparcia ze strony kompilatora — jeżeli zostaną zainicjowane niewłaściwe pola, program ulegnie awarii w czasie działania. Nie można dodawać kolejnych rodzajów do klas oznaczonych, o ile nie ma się dostępu do pliku źródłowego. Jeżeli dodajemy kolejny rodzaj, trzeba pamiętać o dodaniu sekcji case do każdej instrukcji switch, w przeciwnym razie powstanie błąd. Dodatkowo typ danych obiektu nie daje żadnej informacji o jego rodzaju. Krótko mówiąc, **klasy oznaczane są rozwlekłe, podatne na błędy i nieefektywne**.

Na szczęście języki obiektowe, takie jak Java, oferują znacznie lepszą alternatywę dla definiowania jednego typu danych pozwalającego na reprezentowanie obiektów różnych rodzajów — dziedziczenie. **Klasy oznaczane są kiepską imitacją hierarchii klas**.

Aby przekształcić klasę oznaczaną w hierarchię klas, na początek należy zdefiniować klasę abstrakcyjną zawierającą abstrakcyjną metodę dla każdej z metod klasy oznaczanej, której działanie zależy od wartości znacznika. W klasie Figure mamy tylko jedną taką metodę — area. Taka klasa abstrakcyjna staje się korzeniem hierarchii klas. Jeżeli można wyróżnić inne metody, których działanie nie zależy od wartości znacznika, należy umieścić je w tej klasie. Podobnie, jeżeli istnieją pola danych wykorzystywane przez wszystkie rodzaje, należy je również umieścić w tej klasie bazowej. W klasie Figure nie ma metod ani pól niezależnych od rodzaju.

Następnie dla każdej odmiany oryginalnej klasy oznaczanej należy zdefiniować konkretną klasę pochodną. W naszym przykładzie będą to dwie klasy: Circle oraz Rectangle. W każdej z klas pochodnych należy umieścić pola danych wykorzystywane przez ten rodzaj. W naszym przykładzie pole radius jest wykorzystywane w klasie okręgu, a length i width w klasie prostokąta. Dodatkowo w każdej klasie pochodnej należy zdefiniować odpowiednią implementację każdej z metod abstrakcyjnych klasy korzenia. Poniżej przedstawiona jest hierarchia klas odpowiadająca początkowej klasie Figure:

```
// Hierarchia klas zastępująca klasę oznaczaną
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    @Override double area() { return Math.PI * (radius * radius); }
}

class Rectangle extends Figure {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    @Override double area() { return length * width; }
}
```

Taka hierarchia klas koryguje wiele wad przedstawionych wcześniej klas oznaczanych. Kod jest prosty i jasny, ponieważ nie zawiera dodatkowego kodu obsługi znajdującego się w oryginale. Implementacja każdego rodzaju jest umieszczona w osobnej klasie i w żadnej z tych klas nie ma niepotrzebnych pól danych. Wszystkie pola są oznaczone jako `final`. Kompilator zapewnia, że konstruktor każdej z klas inicjuje jej pola danych, a w każdej z tych klas znajduje się implementacja każdej metody abstrakcyjnej zadeklarowanej w klasie bazowej. Eliminuje to możliwość awarii aplikacji spowodowanej brakującą klauzulą `case`. Wielu programistów może niezależnie rozszerzać hierarchię bez konieczności dostępu do kodu źródłowego klasy bazowej. Z każdym rodzajem obiektu skojarzony jest osobny typ danych, co pozwala programistom wskazać rodzaj zmiennej i graniczyć zmienne i parametry wejściowe do określonego rodzaju.

Inną zaletą hierarchii klas jest to, że odzwierciedlają naturalne relacje między typami, co zwiększa elastyczność i pozwala na lepszą kontrolę typów w czasie kompilacji. Załóżmy, że klasa oznaczana z początkowego przykładu pozwala również na obsługę kwadratów. Hierarchia klas powinna odzwierciedlać fakt, że kwadrat jest specjalnym rodzajem prostokąta (pod warunkiem, że obie klasy są niemodyfikowalne).

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

Warto zwrócić uwagę, że pola z powyższej hierarchii są bezpośrednio dostępne i nie ma potrzeby korzystać z metod akcesorów. Zostało to zrealizowane dla uproszczenia i jest nieakceptowalne, jeżeli hierarchia byłaby publiczna (temat 16.).

Podsumujmy. Klasy oznaczane są bardzo rzadko prawidłowym rozwiązaniem. Jeżeli będziemy mieli zamiar napisać klasę z jawnym polem znacznikowym, warto pomyśleć, jak można je wyeliminować przez napisanie hierarchii klas. Jeżeli napotkamy istniejącą klasę z polem znacznika, warto rozważyć jej przepisanie na hierarchię.

Temat 24. Zalety stosowania statycznych klas składowych

Klasa zagnieżdżona to klasa zdefiniowana w innej klasie. Klasa zagnieżdżona powinna być wykorzystywana tylko w klasie, która ją zawiera. Jeżeli klasa zagnieżdżona będzie przydatna również w innym kontekście, powinna być zmieniona na zwykłą klasę. Istnieją cztery rodzaje klas zagnieżdżonych: **statyczne klasy składowe**, **niestatyczne klasy składowe**, **klasy anonimowe** i **klasy lokalne**. Wszystkie, oprócz pierwszej, są nazywane **klasami wewnętrznymi**. Opiszemy teraz, kiedy i dla czego należy stosować poszczególne typy klas.

Statyczne klasy składowe są najprostszym rodzajem klas zagnieżdżonych. Najlepiej myśleć o nich jak o zwykłych klasach zadeklarowanych wewnątrz drugiej klasy, które mają dostęp do wszystkich składowych tej klasy, nawet prywatnych. Statyczna klasa składowa jest statycznym składnikiem, przechowującym wewnętrzną klasę, spełniającym te same zasady dostępu, co inne składniki statyczne. Jeżeli jest zadeklarowana jako prywatna, jest dostępna tylko z klasy, w której jest zdefiniowana itd.

Jednym z częstych zastosowań statycznych klas składowych jest tworzenie publicznych klas pomocniczych, działających tylko w połączeniu ze swoją klasą zewnętrzną. Dla przykładu rozważmy typ wyliczeniowy, opisujący operacje dostępne w kalkulatorze (temat 34.). Klasa `Operation` powinna być publicznym składnikiem statycznym klasy `Calculator`. Klienci klasy `Calculator` mogą odwoływać się do operacji, korzystając z ich nazw, np.: `Calculator.Operation.PLUS` i `Calculator.Operation.MINUS`.

Składniowo jedyną różnicą pomiędzy składnikiem statycznym a niestatycznym jest zastosowanie modyfikatora `static` w deklaracji składnika statycznego. Pomimo podobieństwa składniowego oba rodzaje klas zagnieżdżonych różnią się bardzo od siebie. Każdy obiekt niestatycznego składnika klasy jest niejawnie związany **zawierającym go obiektem** klasy, w którym jest zdefiniowany. W metodach instancyjnych niestatycznej klasy składowej możliwe jest wywoływanie metod zawierającego ją obiektu lub odczytywanie referencji do tego obiektu przy użyciu konstrukcji **kwalifikowanego this** [JLS, 15.8.4]. Jeżeli obiekt klasy zagnieżdżonej może istnieć

w izolacji od klasy nadrzędnej, taka klasa zagnieżdżona **musi** być statyczną klasą zagnieżdżoną. Nie można utworzyć obiektu niestatycznej klasy zagnieżdżonej bez tworzenia obiektu nadrzędnego.

Połączenie pomiędzy obiektem niestatycznej klasy składowej a nadrzędnym obiektem jest tworzone w czasie tworzenia obiektu zagnieżdżonego i nie może być później zmieniane. Normalnie, połączenie to jest tworzone automatycznie przez wywoływanie konstruktora niestatycznej klasy składowej z metody klasy nadrzędnej. Możliwe jest, choć rzadkie, utworzenie połączenia „ręcznie” przy wykorzystaniu wyrażenia `enclosingInstance.new MemberClass(args)`. Jak można się spodziewać, połączenie zajmuje pamięć i zwiększa ilość czasu potrzebnego na utworzenie głównego obiektu.

Częstym zastosowaniem niestatycznych klas składowych jest definiowanie **adaptera** [Gamma95], który pozwala na to, aby klasa zewnętrzna była widziana jako obiekt zupełnie innej, niezwiązanej klasy. Dla przykładu, implementacje interfejsu `Map` do utworzenia swoich **widoków kolekcji** korzystają zwykle z niestatycznej klasy zagnieżdżonej. Widoki te są zwracane przez metody `keySet`, `entrySet` i `values`. Podobnie implementacje interfejsów kolekcji, takich jak `Set` i `List`, korzystają z niestatycznych klas zagnieżdżonych do tworzenia iteratorów:

```
// Typowe zastosowanie niestatycznych klas zagnieżdżonych
public class MySet<E> extends AbstractSet<E> {
    ... // pozostała część klasy pominięta

    public Iterator<E> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

Jeżeli zadeklarujesz klasę zagnieżdżoną, która nie wymaga dostępu do zawierającego ją obiektu, pamiętaj, aby zawsze w jej deklaracji umieścić modyfikator `static`, co powoduje utworzenie statycznej klasy zagnieżdżonej. Jeżeli pominiesz ten modyfikator, każdy obiekt będzie zawierał nadmiarową referencję do zagnieżdżonego obiektu. Utrzymanie tej referencji wymaga czasu i miejsca w pamięci, bez żadnych dodatkowych korzyści (temat 7.). W przypadku niestatycznych klas zagnieżdżonych nie da się również utworzyć obiektu zagnieżdżonego bez obiektu go zawierającego.

Częstym zastosowaniem prywatnych statycznych klas zagnieżdżonych jest reprezentacja składowych obiektu klasy nadrzędnej. Przyjrzyjmy się obiektom typu `Map`, które kojarzą ze sobą klucze i ich wartości. Wiele implementacji `Map` posiada wewnętrzny obiekt `Entry` dla każdego odwzorowania klucz-wartość. Choć każda

pozycja jest skojarzona z odwzorowaniem, metody operujące na pozycjach (`getKey`, `getValue` i `setValue`) nie muszą odwoływać się do odwzorowania. Dlatego rozrzutnością byłoby użycie niestaticznej klasy zagnieżdżonej, reprezentującej pozycje — prywatna statyczna klasa składowa spełnia tu najlepiej swoje zadanie. Jeżeli przypadkowo opuścisz modyfikator `static` w deklaracji pozycji, odwzorowanie będzie działało prawidłowo, ale każda pozycja będzie zawierała nadmiarową referencję do obiektu `Map`, co powoduje spowolnienie działania i dodatkowe zajęcie pamięci.

Równie istotne jest poprawne wybranie statycznej lub niestaticznej klasy składowej, jeśli taka klasa jest publiczną lub zabezpieczoną składową eksportowanej klasy. W takim przypadku klasa zagnieżdżona jest udostępnianym elementem API i nie może być zmieniana pomiędzy wersją statyczną i niestaticzną, ponieważ zmodyfikowana klasa nie będzie binarnie zgodna z poprzednią wersją.

Jak można się spodziewać, klasa anonimowa nie posiada nazwy. Nie jest składnikiem zawierającej ją klasy. Nie jest też deklarowana wraz z innymi składnikami klasy — jest deklarowana i natychmiast tworzona w miejscu użycia. Klas anonimowych można używać we wszystkich miejscach, gdzie wyrażenie takie może zostać zastosowane. Klasy anonimowe zachowują się jak statyczne bądź niestaticzne klasy zagnieżdżone, w zależności od miejsca zastosowania — wykorzystywane w kontekście niestaticznym związane są z zawierającą je klasą. Jednak nawet jeżeli występują w kontekście statycznym, nie mogą posiadać żadnych statycznych składników poza tak zwanymi **zmiennymi stałymi**, czyli ostatecznymi wartościami prostymi lub polami tekstowymi inicjalizowanymi literałami [JLS, 4.12.4].

Istnieje kilka ograniczeń stosowania klas anonimowych. Ponieważ są one jednocześnie deklarowane i tworzone, mogą być używane jedynie wtedy, gdy obiekt jest wykorzystywany tylko w jednym miejscu kodu. Ponieważ klasa anonimowa nie ma nazwy, może być użyta tylko wtedy, gdy nie ma potrzeby późniejszego odwołania się do tak utworzonego obiektu lub korzystania z testów `instanceof`. Klasy anonimowe nie mogą implementować kilku interfejsów lub rozszerzać klasy i implementować innego interfejsu. Nowe metody nie są deklarowane, ponieważ nie istnieje nazwany typ, pozwalający na odwołanie się do tych metod. Klasy anonimowe są stosowane wewnątrz wyrażeń, więc powinny być bardzo krótkie, nie dłuższe niż dwadzieścia wierszy. Dłuższe klasy anonimowe powodują nieczytelność kodu programu.

Przed wprowadzeniem do języka Java `lambd` (rozdział 7.), częstym zastosowaniem klas anonimowych było tworzenie **obiektów funkcyjnych** lub **obiektów procesowych**, ale obecnie preferowanym rozwiązaniem są właśnie `lambdy` (temat 42.). Innym zastosowaniem klas anonimowych jest użycie takiej klasy w statycznej metodzie fabrycznej (patrz metoda `intArrayAsList` w temacie 20.).

Klasy lokalne są najprawdopodobniej najrzadziej wykorzystywanymi klasami zagnieżdżonymi. Klasa lokalna może być zadeklarowana w dowolnym miejscu, w którym może być zadeklarowana zmienna lokalna i podlega tym samym zasadom

określania zasięgu. Klasy lokalne mają podobne cechy do innych klas zagnieżdżonych. Podobnie do klas składowych posiadają nazwy i mogą być wielokrotnie wykorzystywane. Podobnie do klas anonimowych posiadają nadrzędne obiekty, jeżeli są użyte w kontekście niestatycznym. Tak samo, jak klasy anonimowe powinny być krótkie, aby nie zaciemniać kodu.

Podsumujmy. Istnieją cztery rodzaje klas zagnieżdżonych, z których każdy ma swoje zastosowanie. Jeżeli klasa zagnieżdżona musi być widoczna poza jedną metodą lub jest zbyt długa, aby wygodnie mieściła się wewnątrz metody, należy skorzystać z klasy składowej. Jeżeli każdy obiekt klasy składowej wymaga dostępu do zawierającego ją obiektu, klasa ta nie powinna być statyczna — w przeciwnym wypadku należy zastosować klasę statyczną. Zakładając, że klasa istnieje tylko wewnątrz metody, to jeżeli chcesz utworzyć obiekty tylko w jednym miejscu i istnieje już predefiniowany typ charakteryzujący tę klasę, należy zastosować klasę anonimową — w przeciwnym wypadku należy zastosować klasę lokalną.

Temat 25. Ograniczenie pliku źródłowego do pojedynczej klasy głównego poziomu

Choć kompilator Javy dopuszcza definiowanie wielu klas najwyższego poziomu w jednym pliku źródłowym, nie ma zalet związanych z takim podejściem, za to pojawia się kilka ryzyk. Ryzyka te wynikają z faktu, iż zdefiniowanie kilku klas najwyższego poziomu w pliku źródłowym pozwala na uzyskanie wielu definicji klasy. To, która definicja zostanie użyta, zależy od kolejności przekazywania plików źródłowych do kompilatora.

Aby skonkretyzować ten na razie teoretyczny wywód, rozważmy plik źródłowy, który zawiera tylko klasę `Main` odnoszącą się do składowych dwóch klas najwyższego poziomu (`Utensil` i `Dessert`).

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(Utensil.NAME + " z " + Dessert.NAME);  
    }  
}
```

Przypuśćmy teraz, że definiujemy obiekt klasy (`Utensil` i `Dessert`) w jednym pliku źródłowym o nazwie `Utensil.java`:

```
// Dwie klasy zdefiniowane w jednym pliku. Nigdy tak nie rób!  
class Utensil {  
    static final String NAME = "piekarnik";  
}
```

```
class Dessert {  
    static final String NAME = "ciastem";  
}
```

Oczywiście główny program wyświetli tekst piekarnik z ciastem.

Przypuśćmy, że teraz przypadkowo utworzymy **inny** plik źródłowy o nazwie *Dessert.java*, który definiuje te same dwie klasy:

```
// Dwie klasy zdefiniowane w jednym pliku. Nigdy tak nie rób!  
class Utenzil {  
    static final String NAME = "talerz";  
}  
  
class Dessert {  
    static final String NAME = "ciastkiem";  
}
```

Jeśli uda Ci się kompilować program poleceniem `javac Main.java Dessert.java`, kompilacja się nie powiedzie, a kompilator wyświetli komunikat o próbie wielokrotnego zdefiniowania klas *Utenzil* i *Dessert*. Dzieje się tak, ponieważ kompilator najpierw analizuje plik *Main.java*, gdzie widzi odniesienie do *Utenzil* (pojawia się ono przed odniesieniem do *Dessert*), więc poszuka pliku *Utenzil.java*, gdzie znajdzie zarówno klasę *Utenzil*, jak i *Dessert*. Gdy kompilator zacznie analizę wskazanego w wierszu poleceń pliku *Dessert.java*, ponownie znajdzie definicje klas *Utenzil* i *Dessert*, których się nie spodziewał.

Jeśli skompilujesz program za pomocą polecenia `javac Main.java lub javac Main.java Utenzil.java`, efekt będzie taki sam jak przed napisaniem pliku *Dessert.java*, czyli wyświetlenie tekstu piekarnik z ciastem. Jednakże próba kompilacji programu poleceniem `javac Dessert.java Main.java` spowoduje wyświetlenie tekstu talerz z ciastkiem. Oznacza to, że zachowanie programu zależy od kolejności przekazania plików źródłowych do kompilatora, co z oczywistych względów jest nie do zaakceptowania.

Rozwiązanie problemu jest proste — umieszczenie klas najwyższego poziomu (*Utenzil* i *Dessert*) w osobnych plikach źródłowych. Jeśli już musisz umieścić kilka klas najwyższego poziomu w jednym pliku źródłowym, rozważ użycie statycznych klas składowych (temat 24.) jako alternatywę dla rozmieszczania klas w wielu plikach. Jeśli klasy są podrzędne wobec innej klasy, przekształcenie ich w statyczne klasy składowe jest lepsze, bo poprawia czytelność kodu i pozwala zmniejszyć dostępność do klas przez uczynienie ich klasami prywatnymi (temat 15.). Oto w jaki sposób wyglądałby wcześniejszy przykład po przekształceniu na statyczne klasy składowe:

```
// Statyczne klasy składowe zamiast wielu klas najwyższego poziomu  
public class Test {  
    public static void main(String[] args) {  
        System.out.println(Utenzil.NAME + Dessert.NAME);  
    }  
}
```



```
private static class Utensil {  
    static final String NAME = "piekarnik";  
}  
  
private static class Dessert {  
    static final String NAME = "ciastem";  
}  
}
```

Wnioski są oczywiste — **nigdy nie umieszczaj kilku klas najwyższego poziomu lub interfejsów w jednym pliku źródłowym**. Stosowanie tej zasady gwarantuje, że nie pojawi się kilka definicji klasy w momencie kompilacji. To natomiast zagwarantuje, że pliki klas generowane w trakcie kompilacji i zachowanie wynikowego programu przestaną być zależne od kolejności przekazywania plików źródłowych do kompilatora.

Typy ogólne

Od Java 5 zostały dodane typy ogólne. Przed ich wprowadzeniem konieczne było rzutowanie każdego elementu odczytanego z kolekcji. Jeżeli ktoś przypadkowo wstawił obiekt niewłaściwego typu, rzutowanie zwykle prowadziło do powstania błędu wykonania. Przy użyciu typów ogólnych wskazujemy kompilatorowi typ obiektów dozwolonych w każdej kolekcji. Kompilator sam obsługuje rzutowanie i **w czasie kompilacji** informuje nas o przypadkach wstawienia obiektu o niewłaściwym typie. Powoduje to, że programy są zarówno bezpieczniejsze, jak i czytelniejsze, jednak zalety te są okupione dodatkową komplikacją. W tym rozdziale przedstawimy sposoby maksymalizacji zalet i minimalizacji komplikacji.

Temat 26. Nie korzystaj z typów surowych

Na początek kilka terminów. Klasa lub interfejs, którego deklaracja ma jeden lub więcej **parametrów typu**, jest **ogólną** klasą lub interfejsem [JLS, 8.1.2, 9.1.2]. Na przykład interfejs `List` ma jeden parametr typu, `E`, reprezentujący element listy. Technicznie rzecz biorąc, nazwą tego interfejsu jest teraz `List<E>`, ale ludzie często nazywają go krótko `List`. Klasy i interfejsy ogólne są nazywane **typami ogólnymi**.

Każdy typ ogólny definiuje zbiór **typów parametryzowanych**, który składa się z nazwy klasy lub interfejsu, zakończonej w nawiasach trójkątnych listą **rzeczywistych parametrów typów**, odpowiadających formalnym parametrom typów [JLS, 4.4, 4.5]. Na przykład `List<String>` jest typem parametryzowanym reprezentującym listę, której elementy są typu `String` (`String` jest rzeczywistym parametrem typu odpowiadającym formalnemu parametrowi typu `E`).

Na koniec każdy typ ogólny definiuje **typ surowy**, który jest nazwą typu ogólnego bez dodatkowych aktualnych parametrów typu [JLS, 4.8]. Na przykład typem surowym dla `List<E>` jest `List`. Typy surowe działają tak, jakby wszystkie informacje

o typie ogólnym zostały usunięte z deklaracji. Z praktycznego punktu widzenia surowy typ `List` działa w taki sam sposób, jak typ interfejsu `List` przed dodaniem typów ogólnych do platformy. Istnienie tego typu związane jest głównie z koniecznością zachowania zgodności wstecznej z kodem powstałym przed wprowadzeniem typów ogólnych.

Przed wprowadzeniem do Javy typów ogólnych przykładowa deklaracja kolekcji byłaby podobna do poniższej. W Javie 9 nadal jest to forma dopuszczalna, ale należy jej unikać.

// Użycie surowego typu kolekcji - nie rób tak!

// Moja kolekcja znaczków. Zawiera tylko obiekty Stamp.
 private final **Collection** stamps = ... ;

Jeżeli przypadkowo włożymy monetę do naszej kolekcji znaczków, nieprawidłowe wstawienie skompiluje się i uruchomi bez błędów:

// Błędne wstawienie monety do kolekcji znaczków
 stamps.add(new Coin(...)); *// Spowoduje zgłoszenie ostrzeżenia*
// niesprawdzonego wywołania

Nie otrzymamy komunikatu o błędzie do momentu pobrania znaczka z kolekcji:

// Użycie surowego typu iteratora - nie rób tak!
 for (Iterator i = stamps.iterator(); i.hasNext();) {
 Stamp s = (**Stamp**) i.next(); *// zgłasza ClassCastException*
 stamp.cancel();
 }

Jak już kilkakrotnie wspominaliśmy, bardzo opłaca się wykrywanie błędów możliwie szybko po ich popełnieniu, najlepiej na etapie kompilacji. W tym przypadku nie wykryjemy tego błędu do czasu uruchomienia aplikacji, na długo po jego popełnieniu i w kodzie znajdującym się daleko od kodu zawierającego błąd. Gdy zobaczymy komunikat `ClassCastException`, musimy wyszukać w kodzie wywołania, w których do kolekcji znaczków jest dodawana moneta. Kompilator nie pomoże nam, ponieważ „nie rozumie” komentarza *Zawiera tylko obiekty Stamp*.

Przy użyciu typów ogólnych można zastąpić komentarz ulepszoną deklaracją kolekcji, w której przekazujemy kompilatorowi informacje ukryte wcześniej w komentarzu:

// Parametryzowany typ kolekcji - bezpieczny dla typów
 private final **Collection**<**Stamp**> stamps = ... ;

Na podstawie takiej deklaracji kompilator „wie”, że kolekcja stamps powinna zawierać tylko obiekty Stamps, i **gwarantuje** on, że będzie to spełnione pod warunkiem, że cały kod skompiluje się bez generowania (lub pomijania, patrz temat 27.) żadnego ostrzeżenia. Gdy zmienna stamps jest zadeklarowana przy użyciu typu

parametryzowanego, błędne operacje wstawiania powodują wygenerowanie komunikatu o błędzie w czasie kompilacji, który **precyzyjnie** informuje o problemie:

```
Test.java:9: error: incompatible types: Coin cannot be
    ↪converted to Stamp
      c.add(new Coin());
            ^
```

Kompilator wprowadza za nas niewidoczne rzutowanie w momencie pobierania elementu z kolekcji i gwarantuje, że zawsze się one udadzą (przy założeniu, że nie zostały wygenerowane lub pominięte żadne ostrzeżenia). Choć przypadkowe wstawienie monety do kolekcji znaczków może wydawać się wydumane, to jednak ilustruje ono rzeczywisty problem. Na przykład łatwo sobie wyobrazić, że ktoś wstawi obiekt `BigInteger` do kolekcji, która zawiera tylko obiekty `BigDecimal`.

Jak wcześniej pokazaliśmy, można korzystać z typów kolekcji i innych typów ogólnych bez dostarczania parametrów typu, ale nie powinno się tak robić. **Jeżeli korzystamy z surowych typów, tracimy całe bezpieczeństwo i łatwość pracy z typami ogólnymi.** Jeżeli nie należy korzystać z typów surowych, to dlaczego projektanci języka pozwolili na to? W celu zachowania zgodności. Platforma Java wkraczała w drugą dekadę istnienia, gdy zostały dodane typy ogólne, a więc była ogromna ilość kodu Java, w którym nie korzystano z typów ogólnych. Niezbędne było, aby cały ten kod był nadal używany i współpracował z nowym kodem, który korzysta z typów ogólnych. Konieczne stało się umożliwienie przekazywania typów parametryzowanych do metod, które były zaprojektowane dla zwykłych typów, i odwrotnie. Wymaganie to, znane jako **zgodność migracji**, wymusiło decyzję o wsparciu dla typów surowych i implementację ogólności poprzez mechanizm **zacierania** (temat 28.).

Choć nie powinno się używać typów surowych, takich jak `List`, można korzystać z typów parametryzowanych pozwalających na wstawianie dowolnych obiektów, takich jak `List<Object>`. Jaka jest różnica pomiędzy typem surowym `List` a parametryzowanym `List<Object>`? Mówiąc krótko, pierwsza wersja wyłącza sprawdzanie typów, natomiast w drugiej jawnie wskazujemy kompilatorowi, że zmienna pozwala na przechowywanie obiektów dowolnego typu. Choć można przekazać `List<String>` do parametru typu `List`, to nie można przekazać go do parametru typu `List<Object>`. Istnieją zasady dziedziczenia dla typów ogólnych, zgodnie z którymi `List<String>` dziedziczy po surowym typie `List`, a nie po parametryzowanym typie `List<Object>` (temat 28.). W konsekwencji tracimy bezpieczeństwo typów w przypadku korzystania z typów takich jak `List`, ale można się przed tym zabezpieczyć, korzystając z typów parametryzowanych takich jak `List<Object>`.

Aby przedstawić ten problem, przeanalizujmy następujący program:

```
// Użycie typów surowych (List) - błąd w czasie wykonania!
public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
```

```

        unsafeAdd(strings, new Integer(42));
        String s = strings.get(0); // rzutowanie wygenerowane przez kompilator
    }

    private static void unsafeAdd(List list, Object o) {
        list.add(o);
    }

```

Program ten kompiluje się, ale ponieważ korzysta z typu surowego List, otrzymujemy ostrzeżenie:

```

Test.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    list.add(o);
        ^

```

Faktycznie, jeżeli uruchomimy ten program, otrzymamy wyjątek `ClassCastException` w momencie próby rzutowania typu `Integer` w wyniku wywołania `strings.get(0)` na `String`. Jest to rzutowanie wygenerowane przez kompilator, więc normalnie gwarantowane jest jego poprawne wykonanie, ale w tym przypadku zignorowaliśmy ostrzeżenie kompilatora i zapłaciliśmy za to.

Jeżeli w deklaracji `unsafeAdd` zastąpimy typ surowy `List` typem parametryzowanym `List<Object>`, program przestanie się kompilować. Otrzymamy następujący komunikat o błędzie:

```

Test.java:5: error: incompatible types: List<String> cannot be
converted to List<Object>
    unsafeAdd(strings, Integer.valueOf(42));
        ^

```

Można ulec pokusie użycia typów surowych dla kolekcji, których typ elementów jest nieznanym i nie ma znaczenia. Załóżmy, że chcemy napisać metodę, która pobiera dwa zbiory i zwraca liczbę wspólnych elementów. Jeżeli dopiero poznajemy typy ogólne, możemy napisać taką metodę w następujący sposób:

```

// Użycie typów surowych dla nieznanymi typów elementów - nie rób tak!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}

```

Metoda ta działa, ale korzysta z typów surowych, co jest niebezpieczne. Java oferuje bezpieczną alternatywę, nazwaną **niezwiązanymi szablonami typów**. Jeżeli chcemy użyć typów ogólnych, a nie wiemy lub nie ma to dla nas znaczenia, jaki jest typ parametru aktualnego, możemy użyć znaku zapytania w miejsce tego typu.

Na przykład niezwiązany szablon typu dla typu ogólnego `Set<E>` to `Set<?>`. Jest to najbardziej ogólny parametryzowany typ `Set`, który może przechowywać **dowolny** zbiór danych. Poniżej przedstawiona jest metoda `numElementsInCommon` wykorzystująca niezwiązane szablony typów:

```
// Niezwiązany szablon typu - bezpieczny dla typów i elastyczny
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

Jaka jest różnica między niezwiązanym typem `Set<?>` a surowym typem `Set?`? Czy ten znak zapytania faktycznie coś zmienia? Nie chcemy się tutaj rozpisywać — po prostu typy szablony są bezpieczne, a surowe nie są. Można umieścić dowolny element w kolekcji typu surowego, łatwo naruszając niezmiennik typu kolekcji (jak jest to przedstawione w metodzie `unsafeAdd` we wcześniejszej części tego tematu); nie można umieścić dowolnego elementu (innego niż `null`) w `Collection<?>`. Próba wykonania takiej operacji spowoduje błąd kompilacji, podobny do poniższego:

```
Wildcard.java:13: error: incompatible types: String cannot be
↳ converted to CAP#1
    c.add("zakazane");
      ^
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?
```

Oczywiście, ten komunikat o błędzie nie jest idealny, ale kompilator wykonuje swoje zadanie ochrony przed naruszeniem niezmiennika typu kolekcji. Nie tylko nie można umieścić dowolnego elementu (innego niż `null`) w `Collection<?>`, ale również nie można nic zakładać na temat typu pobieranych obiektów. Jeżeli te ograniczenia są nieakceptowane, można użyć **metod ogólnych** (temat 30.) lub **związanych szablonów typu** (temat 31.).

Istnieją pewne niewielkie wyjątki od zasady niewykorzystywania typów surowych. **Konieczne jest używanie typów surowych w literałach klas.** Specyfikacja nie pozwala na stosowanie typów sparametryzowanych (choć pozwala na typ tablicowy i typy proste) [JLS, 15.8.2]. Inaczej mówiąc, `List.class`, `String[].class` oraz `int.class` są prawidłowe, natomiast `List<String>.class` i `List<?>.class` nie są.

Drugi wyjątek od reguły jest związany z operatorem `instanceof`. Ponieważ informacje o typach ogólnych są usuwane w czasie działania programu, nie można użyć operatora `instanceof` dla typu parametryzowanego, poza niezwiązanym typem szablonowym. Użycie niezwiązanego szablonu typu w miejsce typu surowego w żaden sposób nie wpływa na działanie operatora `instanceof`. W tym przypadku nawiasy trójkątne i znak zapytania są po prostu szumem informacyjnym. **Preferowany sposób użycia operatora `instanceof` jest następujący:**

```
// Prawidłowe użycie typu surowego - operator instanceof
if (o instanceof Set) { // typ surowy
```

```
Set<?> m = (Set<?>) o; // typ szablonowy
...
}
```

Należy zwrócić uwagę, że po zorientowaniu się, że `o` jest typu `Set`, konieczne jest rzutowanie na typ szablonowy `Set<?>`, a nie typ surowy `Set`. Jest to kontrolowane rzutowanie, więc nie spowoduje ostrzeżenia kompilatora.

Podsumujmy. Stosowanie typów surowych może prowadzić do powstania wyjątków w czasie działania, więc nie należy z nich korzystać w nowym kodzie Są one dostępne wyłącznie w celu zapewnienia zgodności i współpracy ze starym kodem, napisanym przed wprowadzeniem typów ogólnych. Dla przypomnienia, `Set<Object>` jest typem parametryzowanym reprezentującym zbiór zawierający obiekty dowolnego typu, `Set<?>` jest typem szablonowym reprezentującym zbiór, który zawiera tylko obiekty pewnego nieznanego typu, a `Set` jest typem surowym, który wyłącza system typów ogólnych. Pierwsze dwa są bezpieczne, a ostatni nie.

Terminy wprowadzone w tym temacie (i w kilku innych punktach tego rozdziału) są zebrane w poniższej tabeli.

Termin	Przykład	Temat
Typ parametryzowany	<code>List<String></code>	Temat 26.
Aktualny parametr typu	<code>String</code>	Temat 26.
Typ ogólny	<code>List<E></code>	Tematy 26. i 27.
Formalny parametr typu	<code>E</code>	Temat 26.
Niezwiązany szablon typu	<code>List<?></code>	Temat 26.
Typ surowy	<code>List</code>	Temat 26.
Związany parametr typu	<code><E extends Number></code>	Temat 29.
Rekurencyjny typ związany	<code><T extends Comparable<T>></code>	Temat 30.
Związany szablon typu	<code>List<? extends Number></code>	Temat 31.
Metoda ogólna	<code>static <E> List<E> asList(E[] a)</code>	Temat 30.
Token typu	<code>String.class</code>	Temat 33.

Temat 27. Eliminowanie ostrzeżeń o braku kontroli

Gdy programujemy z użyciem typów ogólnych, często widzimy wiele ostrzeżeń kompilatora — ostrzeżenia o niekontrolowanych rzutowaniach, niekontrolowanych wywołaniach metod, niekontrolowanych tworzeniach tablic ogólnych

i niekontrolowanych konwersjach. Im więcej doświadczenia nabywamy w korzystaniu z typów ogólnych, tym mniej otrzymujemy ostrzeżeń, ale nie należy oczekiwać, że nowo napisany kod wykorzystujący typy ogólne skompiluje się bez problemu.

Wiele ostrzeżeń o braku kontroli można łatwo wyeliminować. Załóżmy, że przypadkowo napisaliśmy taką deklarację:

```
Set<Lark> exaltation = new HashSet();
```

Kompilator grzecznie pokaże nam, co zrobiliśmy źle:

```
Venery.java:4: warning: [unchecked] unchecked conversion
    Set<Lark> exaltation = new HashSet();
                        ^
    required: Set<Lark>
    found:    HashSet
```

Należy więc wprowadzić poprawki, dzięki czemu ostrzeżenie zniknie. Pamiętaj, że dzięki wprowadzeniu w Javie 7 **operatora rombu** (<>) nie trzeba po stronie tworzenia obiektu wskazywać parametru typu, lecz jedynie zaznaczyć zamiar jego użycia. W takiej sytuacji kompilator **odgadnie** poprawny parametr typu na podstawie innych informacji (w tym przypadku będzie to typ Lark):

```
Set<Lark> exaltation = new HashSet<Lark>();
```

Niektóre ostrzeżenia będą jednak **znacznie** trudniejsze do wyeliminowania. W tym rozdziale przedstawiamy wiele takich ostrzeżeń. Gdy napotkasz ostrzeżenie wymagające nieco zastanowienia, nie poddawaj się! **Konieczn****ie trzeba wyeliminować każde ostrzeżenie o braku kontroli**. Jeżeli wyeliminujemy wszystkie ostrzeżenia, zapewnimy, że nasz kod będzie bezpieczny dla typów, co bardzo się opłaca. Oznacza to, że nie otrzymamy w czasie działania programu wyjątków `ClassCastException` i spowodujemy, że będziemy bardziej pewni prawidłowego działania programu.

Jeżeli nie możemy wyeliminować ostrzeżenia, ale możemy udowodnić, że kod powodujący wygenerowanie ostrzeżenia jest bezpieczny dla typów, to w takim przypadku (i tylko w takim) możemy wyłączyć ostrzeżenie za pomocą adnotacji `@SuppressWarnings("unchecked")`. Jeżeli wyłączymy ostrzeżenia bez wcześniejszego udowodnienia, że kod jest bezpieczny dla typów, stworzymy fałszywe poczucie bezpieczeństwa. Kod może kompilować się bez generowania ostrzeżeń, ale nadal generuje w czasie działania wyjątki `ClassCastException`. Jeżeli jednak zignorujemy ostrzeżenia o braku kontroli w kodzie, o którym wiemy, że jest bezpieczny (zamiast ich wyłączenia), możemy nie zauważyć, że pojawi się nowe ostrzeżenie informujące o prawdziwym problemie. Nowe ostrzeżenie może zostać zagubione w powodzi fałszywych alarmów, których nie wyłączyliśmy.

Adnotacja `SuppressWarnings` może być używana z dowolną rozdzielczością, od pojedynczych deklaracji zmiennych lokalnych do całej klasy. **Zawsze należy używać adnotacji `SuppressWarnings` w najmniejszym możliwym zakresie**. Zwykle

będzie to deklaracja zmiennej lub bardzo krótka metoda albo konstruktor. Nigdy nie należy używać `SuppressWarnings` dla całej klasy. W ten sposób możemy zamaskować ważne ostrzeżenia.

Jeżeli zauważymy, że adnotacja `SuppressWarnings` jest używana dla metody lub konstruktora o długości większej niż jeden wiersz, warto przenieść go do deklaracji zmiennej lokalnej. Być może zajdzie konieczność zadeklarowania nowej zmiennej lokalnej, ale warto to zrobić. Jako przykład rozważymy metodę `toArray`, pochodzącą z `ArrayList`:

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

Jeżeli skompilujemy **`ArrayList`**, metoda wygeneruje następujące ostrzeżenie:

```
ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                                   ^
required: T[]
found:    Object[]
```

Umieszczanie adnotacji `SuppressWarnings` w instrukcji `return` nie jest dopuszczalne, ponieważ nie jest to deklaracja [JLS, 9.7]. Można mieć zamiar umieszczenia adnotacji w całej metodzie, ale nie należy tego robić. Zamiast tego należy zadeklarować zmienną lokalną przechowującą zwracaną wartość, tak jak w poniższym przykładzie:

```
// Dodanie zmiennej lokalnej, aby ograniczyć zakres @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // To rzutowanie jest prawidłowe, ponieważ tworzona tablica
        // jest tego samego typu co przekazana, czyli T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

Metoda ta kompiluje się bez problemów, a dodatkowo minimalizujemy zakres, w którym są wyłączone ostrzeżenia o braku kontroli.

Za każdym razem, gdy używamy adnotacji `@SuppressWarnings("unchecked")`, należy dodać komentarz wyjaśniający, dlaczego jest to bezpieczne. Pomoże to zrozumieć nasz kod, a co ważniejsze, zmniejszy to niebezpieczeństwo, że ktoś zmieni kod w taki sposób, że operacje nie będą wykonywane w bezpieczny sposób. Jeżeli okaże się, że trudno jest napisać taki komentarz, warto jeszcze raz przemyśleć kod. Być może zorientujesz się, że niekontrolowana operacja wcale nie jest bezpieczna.

Podsumujmy. Ostrzeżenia o braku kontroli są ważne. Nie należy ich ignorować. Każde ostrzeżenie o braku kontroli reprezentuje potencjalny wyjątek `ClassCastException` → `Exception` w czasie działania. Najlepiej wyeliminować wszystkie te ostrzeżenia. Jeżeli nie można tego zrobić, ale można udowodnić, że kod, w którym występuje ostrzeżenie, jest bezpieczny dla typów, należy wyłączyć ostrzeżenie za pomocą `@SuppressWarnings("unchecked")` w najwęższym możliwym zakresie. Należy napisać uzasadnienie tej decyzji w komentarzu.

Temat 28. Korzystanie z list zamiast tablic

Tablice różnią się od typów ogólnych na dwa ważne sposoby. Po pierwsze, tablice są **kowariancyjne**. To strasznie brzmiące słowo oznacza, że jeżeli `Sub` jest typem podrzędnym względem `Super`, to tablica typu `Sub[]` jest typem podrzędnym względem `Super[]`. Typy ogólne z kolei są **niezmiennicze** — dla dwóch różnych typów `Typ1` i `Typ2`, `List<Typ1>` nie jest ani typem podrzędnym, ani nadrzędnym dla `List<Typ2>` [JLS, 4.10; Naftalin07, 2.5]. Można uważać, że powoduje to niedoskonałość typów ogólnych, ale w rzeczywistości to tablice są niedoskonałe. Poniższy fragment kodu jest prawidłowy:

```
// Błąd w czasie wykonania!
Object[] objectArray = new Long[1];
objectArray[0] = "Nie pasuje"; // zgłasza ArrayStoreException
```

a ten nie jest:

```
// Nie kompiluje się!
List<Object> ol = new ArrayList<Long>(); // niezgodne typy
ol.add("Nie pasuje");
```

W obu przypadkach nie można umieścić obiektu `String` w kontenerze `Long`, ale w przypadku tablic dowiadujemy się o naszej pomyłce w czasie wykonania, a w przypadku list — w czasie kompilacji. Oczywiście, preferujemy lokalizowanie błędów w czasie kompilacji.

Drugą ważną różnicą pomiędzy tablicami i typami ogólnymi jest to, że tablice są **uściślone** [JLS, 4.7]. Oznacza to, że tablica zna i wymusza typy elementów w czasie działania programu. Jak wcześniej pokazaliśmy, jeżeli będziemy próbowali wstawić `String` do tablicy elementów `Long`, otrzymamy wyjątek `ArrayStoreException`. Typy

ogólne są z kolei implementowane poprzez **zacieranie** [JLS, 4.6]. Oznacza to, że ograniczenia typów są wymuszane wyłącznie w czasie kompilacji, a w czasie działania informacje o typach elementów są pomijane (**usuwane**). Zacieranie pozwala typom ogólnym na współpracę z bazą starszego kodu, który nie wykorzystuje typów ogólnych (temat 26.) wprowadzonych w Javie 5.

Z powodu tych fundamentalnych różnic tablice i typy ogólne niezbyt dobrze ze sobą współpracują. Na przykład nie można utworzyć tablicy typu ogólnego, typu parametryzowanego lub parametru typu. Żadna z tych instrukcji tworzenia nie jest dopuszczalna: `new List<E>[]`, `new List<String>[]` i `new E[]`. Wszystkie te instrukcje powodują wygenerowanie **błędów tworzenia tablicy ogólnej**.

Dlaczego nie można tworzyć tablic ogólnych? Ponieważ nie jest zachowane bezpieczeństwo typów. Jeżeli byłoby to dopuszczalne, rzutowania wygenerowane przez kompilator w prawidłowym programie powodowałyby powstanie błędów `ClassCastException` ➔ `Exception`. Naruszałoby to podstawową gwarancję zapewnianą przez system typów ogólnych.

Aby lepiej przedstawić ten problem, przeanalizujmy następujący fragment kodu:

```
// Dlaczego tworzenie ogólnej tablicy jest niedopuszczalne - nie kompiluje się!
List<String>[] stringLists = new List<String>[1]; // (1)
List<Integer> intList = Arrays.asList(42);        // (2)
Object[] objects = stringLists;                  // (3)
objects[0] = intList;                             // (4)
String s = stringLists[0].get(0);                 // (5)
```

Żałujemy, że pierwszy wiersz, w którym tworzymy tablicę, jest prawidłowy. W wierszu 2. tworzymy i inicjujemy listę `List<Integer>` zawierającą jeden element. W wierszu 3. umieszczamy tablicę `List<String>` w zmiennej tablicowej typu `Object`, co jest dopuszczalne, ponieważ tablice są kowariancyjne. W wierszu 4. umieszczamy `List<Integer>` w jedynym elemencie tablicy `Object`, co się udaje, ponieważ typy ogólne są implementowane przez zacieranie — w czasie działania typ obiektu `List<Integer>` jest po prostu typem `List`, a typ `List<String>[]` jest `List[]`, więc przypisanie nie powoduje wygenerowania `ArrayStoreException`. Teraz mamy kłopoty. Umieściliśmy obiekt `List<Integer>` w tablicy, która według deklaracji ma przechowywać tylko obiekty `List<String>`. W wierszu 5. pobieramy jedyny element z jedynej listy w tej tablicy. Kompilator automatycznie rzutuje pobrany element na `String`, ale jest to `Integer`, więc otrzymujemy w czasie działania wyjątek `ClassCastException`. Aby temu zapobiec, w wierszu 1. (w którym tworzona jest tablica ogólna) generowany jest błąd kompilacji.

Typy takie jak `E`, `List<E>` oraz `List<String>` są technicznie nazywane jako typy **nieuściślone** [JLS, 4.7]. Intuicyjnie — typy nieuściślone to takie typy, których reprezentacja w czasie wykonania zawiera mniej informacji niż reprezentacja w czasie

kompilacji. Jedynymi typami parametryzowanymi, które są uściślone, są niezwiązane typy szablonowe, takie jak `List<?>` lub `Map<?,?>` (temat 26.). Prawidłowe, choć rzadko przydatne, jest tworzenie tablic niezwiązanych typów szablonowych.

Brak możliwości tworzenia tablic ogólnych może być irytujący. Oznacza to na przykład, że dla typów ogólnych nie jest możliwe zwracanie tablicy elementów jej typu (ale częściowe rozwiązanie jest przedstawione w temacie 33.). Oznacza to, że można otrzymać mylące ostrzeżenia przy korzystaniu z metod `varargs` (temat 53.) w połączeniu z typami ogólnymi. Dzieje się tak, ponieważ za każdym razem, gdy wywołujemy metodę `varargs`, tworzona jest tablica przechowująca parametry. Jeżeli typ elementu w tej tablicy jest nieuściślony, otrzymujemy ostrzeżenie. Niewiele możemy poradzić na te ostrzeżenia poza ich wyłączeniem adnotacją `SafeVarargs` (temat 32.).

Gdy otrzymamy błąd tworzenia tablicy ogólnej, najlepszym rozwiązaniem jest często użycie kolekcji typu `List<E>` zamiast tablicy typu `E[]`. Być może poświęcimy w ten sposób nieco wydajności lub spójności, ale w zamian otrzymamy lepsze bezpieczeństwo typów i możliwość współpracy.

Powiedzmy, że chcemy napisać klasę `Chooser` z konstruktorem, który przyjmuje kolekcję, i jedną metodą, która losowo zwraca element tej kolekcji. W zależności od przekazanej do konstruktora kolekcji mechanizmem wybierania może być rzut kostką, magiczna kula lub źródło danych dla symulacji Monte Carlo. Poniżej przedstawiony jest kod realizujący to zadanie bez użycia typów ogólnych:

```
// Chooser - klasa bardzo potrzebująca typów ogólnych!
public class Chooser {
    private final Object[] choiceArray;

    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
    }

    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

Aby użyć tej klasy, musimy rzutować wartość zwracaną przez metodę `choose` z typu `Object` na pożądaný typ przy każdym jej użyciu. Jeśli pomylimy się z rodzajem rzutowania, błąd pojawi się dopiero w trakcie działania programu. Ponieważ chcemy dostosować się do zaleceń z zadania 29., próbujemy zmienić klasę `Chooser` tak, aby obsługiwała typy ogólne. Zmiany zostały pogrubione:

```
// Chooser - pierwsze podejście do wprowadzenia typów ogólnych;
// kodu nie uda się skompilować!
public class Chooser {
```

```

private final T[] choiceArray;

public Chooser(Collection choices) {
    choiceArray = choices.toArray();
}

// metoda choose nie uległa zmianie
}

```

Przy próbie kompilacji zostanie zgłoszony poniższy komunikat:

```

Chooser.java:9: error: incompatible types: Object[] cannot be
converted to T[]
    choiceArray = choices.toArray();
                        ^
where T is a type-variable:
  T extends Object declared in class Chooser

```

Stwierdzamy, że wprowadzenie zmiany nie będzie problemem, i zaczynamy rzutować tablicę `Object` na tablicę `T`:

```

choiceArray = (T[]) choices.toArray();

```

Udało się nam w ten sposób pozbyć błędu, ale zamiast tego otrzymujemy ostrzeżenie:

```

Chooser.java:9: warning: [unchecked] unchecked cast
    choiceArray = (T[]) choices.toArray();
                        ^
required: T[], found: Object[]
  where T is a type-variable: T extends Object declared in class
  Chooser

```

Kompilator informuje nas, że nie może sprawdzić bezpieczeństwa rzutowania w czasie wykonania, ponieważ „nie wie”, czym jest `E` w czasie działania aplikacji — należy pamiętać, że informacje o typach elementów są usuwane z typów ogólnych w czasie działania. Czy program będzie działał? Tak, okazuje się, że będzie działał, ale kompilator nie może tego udowodnić. Moglibyśmy sprawdzić poprawność programu, umieścić dowód w komentarzu i zablokować pojawianie się ostrzeżenia za pomocą adnotacji, ale lepszym wyjściem będzie eliminacja powodu pojawiania się ostrzeżenia (temat 27.).

Co powinniśmy zrobić, aby pozbyć się ostrzeżenia o niesprawdzalnym rzutowaniu? Wykorzystać listę zamiast tablicy. Poniżej znajduje się wersja klasy `Chooser`, która kompiluje się bez błędów i ostrzeżeń:

```

// Klasa Chooser bazująca na obiektach List, co zapewnia bezpieczeństwo typów
public class Chooser<T> {
    private final List<T> choiceList;

    public Chooser(Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
    }
}

```

```

public T choose() {
    Random rnd = ThreadLocalRandom.current();
    return choiceList.get(rnd.nextInt(choiceList.size()));
}
}

```

Ta wersja jest nieco dłuższa i nieco wolniejsza niż wersja korzystająca z tablic, ale jest uspakajająca, ponieważ wiemy, że w czasie działania nie pojawi się błąd `ClassCastException`.

Podsumujmy. Tablice i typy ogólne mają zupełnie inne reguły. Tablice są kowariancyjne i uściślone; typy ogólne są niezmiennicze i zacierane. W konsekwencji tablice zapewniają bezpieczeństwo typów w czasie działania, ale nie dają bezpieczeństwa typów w czasie kompilacji — dla typów ogólnych sytuacja jest odwrotna. Najczęściej typy ogólne i tablice niezbyt dobrze ze sobą współpracują. Jeżeli zdarzy się nam ich połączenie i otrzymamy komunikat o błędzie lub ostrzeżenie, na początek należy zmienić tablice na listy.

Temat 29. Stosowanie typów ogólnych

Zwykle nie jest zbyt trudno parametryzować deklaracje kolekcji i wykorzystać typy i metody ogólne dostępne w JDK. Tworzenie własnych typów ogólnych jest nieco trudniejsze, ale warto się tego nauczyć.

Jako przykład weźmy prostą implementację stosu z tematu 7.

```

// Kolekcja bazująca na Object - podstawowy kandydat dla użycia typów ogólnych
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // usuwanie niepotrzebnych referencji
        return result;
    }
}

```

```

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

```

Klasa ta nadaje się do **uogólnienia**, czyli rozbudowania w celu wykorzystania zalet typów ogólnych. Możemy ją sparametryzować bez wpływu na klienty oryginalnej, niesparametryzowanej wersji. Jak widać, konieczne jest rzutowanie obiektów zdejmowanych ze stosu i rzutowanie to może się nie powieść w czasie działania programu. Pierwszym krokiem uogólniania klasy jest dodanie do jej deklaracji jednego lub więcej parametrów typu. W tym przypadku jest to tylko jeden parametr reprezentujący typ elementów na stosie, a standardową nazwą dla tego parametru jest *E* (temat 68.).

Następnym krokiem jest wymiana wszystkich zastosowań typu *Object* na odpowiedni parametr typu, a następnie próba kompilacji otrzymanego programu:

// Pierwsza próba uogólnienia klasy Stack - nie kompiluje się!

```

public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size==0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // eliminowanie niepotrzebnych odwołań
        return result;
    }
    ... // metody isEmpty i ensureCapacity bez zmian
}

```

Zwykle otrzymamy co najmniej jeden błąd lub ostrzeżenie i ta klasa nie jest wyjątkiem. Na szczęście klasa ta generuje tylko jeden błąd:


```
Stack.java:8: generic array creation
      elements = new E[DEFAULT_INITIAL_CAPACITY];
                ^
```

Zgodnie z tym, o czym pisaliśmy w temacie 28., nie można utworzyć tablicy typu nieuściśłonego, na przykład `E`. Problem ten pojawia się za każdym razem, gdy piszemy typ ogólny korzystający z tablicy. Istnieją dwa sposoby na rozwiązanie tego problemu. Pierwsze rozwiązanie bezpośrednio eliminuje problem braku możliwości tworzenia tablicy ogólnej — utworzenie tablicy `Object` i rzutowanie jej na typ tabeli ogólnej. Teraz w miejsce błędu kompilator generuje ostrzeżenie. Takie użycie jest legalne, ale nie jest (zazwyczaj) bezpieczne dla typów:

```
Stack.java:8: warning: [unchecked] unchecked cast
found: Object[], required: E[]
      elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                  ^
```

Kompilator może nie być w stanie udowodnić, że program jest bezpieczny dla typów, ale my możemy. Musimy przekonać siebie, że niekontrolowane rzutowanie nie naruszy bezpieczeństwa typów w programie. Interesująca nas tablica (`elements`) jest przechowywana w polu prywatnym i nie jest zwracana do klienta lub przekazywana do innej metody. Elementy przechowywane w tablicy są wyłącznie przekazywane do metody `push`, a ponieważ elementy te są wyłącznie typu `E`, dlatego niekontrolowane rzutowanie nie powoduje problemów.

Po udowodnieniu, że niekontrolowane rzutowanie jest bezpieczne, można wyłączyć ostrzeżenie w możliwie wąskim zakresie (temat 27.). W tym przypadku konstruktor zawiera wyłącznie niekontrolowane tworzenie tablicy, więc można wyłączyć ostrzeżenia w całym konstruktorze. Po dodaniu odpowiedniej adnotacji klasa `Stack` kompiluje się bez problemów i można używać jej bez jawnego rzutowania i bez obawy o `ClassCastException`:

```
// Elementy tablicy zawierają tylko obiekty E pochodzące z push(E).
// Jest to wystarczające dla zachowania bezpieczeństwa typów,
// ale w czasie działania typem tablicy nie będzie E[], lecz Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

Drugim sposobem na wyeliminowanie błędu tworzenia tablicy ogólnej w `Stack` jest zmiana typu pola `elements` z `E[]` na `Object[]`. Jeżeli to zrobimy, otrzymamy inny błąd:

```
Stack.java:19: incompatible types
found: Object, required: E
      E result = elements[--size];
                  ^
```

Możemy zmienić ten błąd na ostrzeżenie przez rzutowanie elementów pobieranych z tablicy od `Object` na `E`:

```
Stack.java:19: warning: [unchecked] unchecked cast
found: Object, required: E
    E result = (E) elements[--size];
                ^
```

Ponieważ `E` jest typem nieuściślonym, nie ma sposobu na sprawdzenie przez kompilator rzutowania w czasie wykonywania programu. I tutaj możemy łatwo udowodnić, że niekontrolowane rzutowanie jest bezpieczne, więc można zablokować ostrzeżenie. Zgodnie z poradą z tematu 27. wyłączamy ostrzeżenia tylko w przypisaniu zawierającym niekontrolowane rzutowanie, a nie w całej metodzie `pop`:

```
// Odpowiednie wyłączenie ostrzeżenia niekontrolowanego rzutowania
public E pop() {
    if (size==0)
        throw new EmptyStackException();

    // push wymaga elementów typu E, więc rzutowanie jest bezpieczne
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // wyeliminowanie niepotrzebnych odwołań
    return result;
}
```

Obie przedstawione tu techniki eliminowania błędu tworzenia tablicy ogólnej mają swoje wady. Pierwsza jest bardziej czytelna: tablica jest deklarowana jako typ `E[]`, co wyraźnie wskazuje na chęć przechowywania elementów tylko tego typu. Jest też bardziej zwięzła. Pierwsza technika wymaga tylko jednego rzutowania (w momencie tworzenia tablicy), natomiast druga wymaga rzutowania przy każdym odczycie danych z tablicy. Dlatego to pierwsze rozwiązanie jest w praktyce wykorzystywane znacznie częściej. Powoduje jednak **skażenie sterty** (temat 32.) — typ tablicy w trakcie wykonywania nie odpowiada typowi z czasu kompilacji (chyba że `E` jest typu `Object`). Niektórym programistom to nie odpowiada i wybierają z tego powodu drugie rozwiązanie, choć w tym przypadku skażenie sterty jest nieszkodliwe.

Poniższy program demonstruje użycie naszej ogólnej klasy `Stack`. Program wyświetla wartości argumentów wiersza wywołania w odwrotnej kolejności, konwertując je na wielkie litery. W wywołaniu metody `toUpperCase` z klasy `String` nie jest potrzebne jawne rzutowanie elementów zdjętych ze stosu i gwarantowane jest poprawne wykonanie automatycznie generowanego rzutowania.

```
// Mały program testujący naszą ogólną klasę Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
```

```
for (String arg : args)
    stack.push(arg);
while (!stack.isEmpty())
    System.out.println(stack.pop().toUpperCase());
}
```

Przedstawiony przykład może wydawać się niezgodny ze wskazówkami z tematu 28., w którym zalecaliśmy użycie list zamiast tablic. Użycie list w typach ogólnych nie zawsze jest możliwe lub pożądane. Java nie obsługuje natywnie list, więc niektóre typy ogólne, takie jak `ArrayList`, **muszą** być zaimplementowane na bazie tablic. Inne typy ogólne, takie jak `HashMap`, są implementowane na bazie tablic w celu osiągnięcia odpowiedniej wydajności.

Większość typów ogólnych, takich jak nasz przykładowy `Stack`, nie ma ograniczeń na parametr typu. Można tworzyć `Stack<Object>`, `Stack<int[]>`, `Stack<List<String>>` lub stos elementów innego typu referencyjnego obiektu. Trzeba pamiętać, że nie można tworzyć obiektu `Stack` dla typów prostych — próba utworzenia obiektu `Stack<int>` lub `Stack<double>` spowoduje błąd kompilacji. Jest to podstawowe ograniczenie systemu typów ogólnych w języku Java. Można ominąć to ograniczenie przez użycie opakowanych typów prostych (temat 61.).

Istnieją typy ogólne, które ograniczają dostępne wartości ich parametrów typu. Jako przykład można przytoczyć `java.util.concurrent.DelayQueue`, której deklaracja wygląda następująco:

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>;
```

Typ parametru listy (`<E extends Delayed>`) wymaga, aby typ parametru aktualnego `E` był typem pochodnym po `java.util.concurrent.Delayed`. Dzięki temu implementacja `DelayQueue` i jej klienci mogą korzystać z możliwości metod `Delayed` w elementach `DelayQueue` bez potrzeby jawnego rzutowania lub ryzyka wystąpienia wyjątku `ClassCastException`. Parametr typu `E` jest nazywany **związanym parametrem typu**. Należy zwrócić uwagę, że relacja typu pochodnego jest tak zdefiniowana, że każdy typ jest typem pochodnym po samym sobie [JLS, 4.10], więc prawidłowe jest tworzenie `DelayQueue<Delayed>`.

Podsumujmy. Typy ogólne są bezpieczniejsze i łatwiejsze w użyciu niż typy wymagające rzutowania w kodzie klienta. Gdy stworzymy nowe typy, należy upewnić się, że mogą one działać bez takiego rzutowania. Często powoduje to powstanie typów ogólnych. Warto uogólniać istniejące typy, o ile pozwala na to czas. Ułatwia to pracę nowym użytkownikom bez konieczności wprowadzania modyfikacji w istniejących klientach (temat 26.).

Temat 30. Stosowanie metod ogólnych

Uogólnianie można przeprowadzać zarówno dla klas, jak i dla metod. Statyczne metody użytkowe są szczególnie dobrymi kandydatami do uogólniania. Wszystkie metody algorytmów w `Collections` (takie jak `binarySearch` czy `sort`) są uogólnione.

Pisanie metod ogólnych jest podobne do pisania typów ogólnych. Weźmy jako przykład metodę zwracającą sumę dwóch zbiorów:

```
// Użycie typów surowych - nieakceptowane! (temat 26.)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

Metoda ta kompiluje się, ale z dwoma ostrzeżeniami:

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
    Set result = new HashSet(s1);
                  ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
    result.addAll(s2);
              ^
```

Aby usunąć te ostrzeżenia i zapewnić bezpieczeństwo typów, należy zmodyfikować deklarację metody, dodając **parametr typu** reprezentujący typ elementu dla obu zbiorów (dwa argumenty oraz zwracana wartość), oraz użyć parametrów typu w metodzie. **Lista parametrów typu, która deklaruje typ parametru, znajduje się pomiędzy modyfikatorami metody a zwracanym typem.** W tym przykładzie listą parametrów typu jest `<E>`, a zwracanym typem jest `Set<E>`. Konwencja nazewnictwa dla parametrów typu jest dla metod ogólnych taka sama jak dla typów ogólnych (tematy 29. i 68.):

```
// Metoda ogólna
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

To wszystko, co należy zrobić — przynajmniej dla prostych metod ogólnych. Teraz metoda kompiluje się bez generowania ostrzeżeń i zapewnia bezpieczeństwo typów, jak również jest prosta w użyciu. Poniżej zamieszczony jest prosty przykład przedstawiający wywołanie naszej metody. Program nie zawiera żadnej operacji rzutowania i kompiluje się bez błędów i ostrzeżeń:

```
// Prosty program korzystający z metod ogólnych
public static void main(String[] args) {
    Set<String> guys = new HashSet<String>({
        Arrays.asList("Tomasz", "Darek", "Jerzy"));
    Set<String> stooges = new HashSet<String>({
        Arrays.asList("Lucek", "Marek", "Czesław"));
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

Gdy uruchomimy program, wyświetli on napis [Marek, Jerzy, Tomasz, Czesław, Lucek, Darek]. Kolejność elementów jest zależna od implementacji.

Ograniczeniem metody `union` jest to, że wszystkie trzy zbiory (zarówno parametry wejściowe, jak i zwracana wartość) muszą być tego samego typu. Aby metoda była bardziej elastyczna, możemy użyć **związanych szablonów typów** (temat 31.).

Czasami tworzymy obiekt, który jest niemodyfikowalny, ale można go stosować dla wielu typów. Ponieważ typy ogólne są implementowane przez zacieranie (temat 28.), można użyć jednego obiektu dla wszystkich wymaganych parametryzacji typu, ale należy napisać statyczną metodę fabryczną do wielokrotnego tworzenia obiektu dla każdej potrzebnej parametryzacji typu. Wzorec ten, nazywany **fabryką singletonów uogólnionych**, jest najczęściej wykorzystywany dla obiektów funkcyjnych (temat 42.), takich jak `Collections.reverseOrder`, ale jest również używany dla kolekcji takich jak `Collections.emptySet`.

Teraz założmy, że chcemy udostępnić funkcję tożsamościową. Biblioteki zapewniają `Function.identity`, więc nie trzeba pisać własnej, choć to ciekawe ćwiczenie. Niepotrzebne jest tworzenie nowej funkcji za każdym razem, gdy jest wymagana, ponieważ jest ona bezstanowa. Jeżeli typy ogólne byłyby uściślone, potrzebna byłaby jedna funkcja dla każdego typu, ale w obecnej implementacji potrzebujemy tylko jednego ogólnego singletonu. Wygląda on w następujący sposób:

```
// Wzorec fabryki singletonów uogólnionych
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

Rzutowanie `IDENTITY_FN` na `(UnaryFunction<T>)` generuje ostrzeżenie o niekontrolowanym rzutowaniu, ponieważ `UnaryFunction<Object>` nie jest `UnaryFunction<T>` dla każdego `T`. Jednak funkcja tożsamościowa jest specjalna — zwraca ona niezmodyfikowany argument, więc wiemy, że bezpiecznie możemy użyć jej jako `UnaryFunction<T>` dla dowolnego `T`. Z tego powodu możemy wyłączyć generowane przez to rzutowanie ostrzeżenie. Gdy adnotacja ta jest dodana, kod kompiluje się bez błędów i ostrzeżeń.

Poniżej zamieszczony jest przykładowy program korzystający z naszego ogólnego singletonu jako `UnaryFunction<String>` oraz `UnaryFunction<Number>`. Jak zwykle nie zawiera on żadnej operacji rzutowania i kompiluje się bez błędów i ostrzeżeń:

```
// Przykładowy program testujący singleton uogólniony
public static void main(String[] args) {
    String[] strings = { "juta", "konopie", "nylon" };
    UnaryFunction<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryFunction<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

Dopuszczalne jest, choć dosyć rzadkie, aby parametr typu był związany z pewnym wyrażeniem zawierającym sam parametr. Często jest to nazywane **rekurencyjnym wiązaniem typów**. Najczęstszym użyciem rekurencyjnego wiązania typów jest wykorzystanie interfejsu `Comparable`, który definiuje naturalny porządek w danym typie (temat 14.). Oto definicja interfejsu:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Parametr typu `T` definiuje typ, z którym powinny być porównywane elementy typu implementującego `Comparable<T>`. W praktyce niemal wszystkie typy mogą być porównywane z elementami własnego typu. Na przykład `String` implementuje `Comparable<String>`, `Integer` implementuje `Comparable<Integer>` i tak dalej.

Istnieje wiele metod, które oczekują kolekcji elementów implementujących `Comparable`, co pozwala na jej sortowanie, wyszukiwanie, wyliczanie minimum lub maksimum i inne podobne operacje. Aby wykonać te operacje, wymagane jest, by każdy element w kolekcji był porównywalny z każdym innym elementem kolekcji, czyli aby elementy kolekcji były **wzajemnie porównywalne**. Poniżej przedstawiony jest sposób wyrażenia takiego ograniczenia:

```
// Zastosowanie rekurencyjnego wiązania typów do wyrażenia wzajemnej
// porównywalności.
public static <E extends Comparable<E>> E max(Collection<E> c) {...}
```

Wiązanie typów `<E extends Comparable<E>>` może być czytane jako „dla każdego typu `E`, który można porównać ze sobą”, co mniej więcej odpowiada notacji wzajemnej porównywalności.

Poniżej zamieszczona jest metoda wykorzystująca powyższą deklarację. Oblicza ona maksymalną wartość kolekcji zgodnie z naturalnym porządkiem elementów i kompiluje się bez błędów i ostrzeżeń.

```
//Zwraca maksymalną wartość z kolekcji, korzysta z rekurencyjnego wiązania typów
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Pusta kolekcja");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return result;
}
```

Zwróć uwagę, że metoda zgłasza wyjątek `IllegalArgumentException`, jeśli kolekcja jest pusta. Lepszym rozwiązaniem byłoby w tym przypadku zwrócenie `Optional<E>` (temat 55.).

Rekurencyjne wiązanie typów może być znacznie bardziej skomplikowane niż to, ale na szczęście nie zdarza się to zbyt często. Jeżeli zrozumiemy ten idiom, jego odmianę ze wzorcem wieloznacznym (temat 31.) oraz idiom symulowanego typu własnego (temat 2.), będziemy w stanie poradzić sobie z wieloma rekurencyjnymi wiązaniami typów, wykorzystywanymi w praktyce.

Podsumujmy. Metody ogólne, podobnie jak typy ogólne, są bezpieczniejsze i łatwiejsze do użycia niż metody wymagające od klientów rzutowania parametrów i zwracanych wartości. Podobnie jak w przypadku typów, powinniśmy upewnić się, że nowe metody mogą być używane bez rzutowania, co często powoduje, że są one zmieniane na ogólne. Dodatkowo, również jak w przypadku typów, warto uogólniać istniejące metody, aby ułatwić życie nowym użytkownikom, bez wprowadzania błędów do istniejącego kodu (temat 26.).

Temat 31. Zastosowanie związanych szablonów do zwiększania elastyczności API

Jak wspomnieliśmy w temacie 28., typy parametryzowane są **niezmiennicze**. Inaczej mówiąc, dla dwóch różnych typów `Typ1` i `Typ2`, `List<Typ1>` nie jest ani typem podrzędnym, ani nadrzędnym dla `List<Typ2>`. Choć nieintuicyjne jest, że `List<String>` nie jest podtypem `List<Object>`, to jednak faktycznie ma to sens. Można wstawić dowolny obiekt do `List<Object>`, ale w `List<String>` można umieszczać tylko napisy. Ponieważ `List<String>` nie może wykonywać wszystkiego, co może robić `List<Object>`, nie może być jego typem podrzędnym (zgodnie z zasadą podstawiania Liskov z tematu 10.).

Czasami potrzebujemy większej elastyczności niż zapewnia typowanie niezmiennicze. Jako przykład weźmy stos z tematu 29. Dla przypomnienia — jego publiczne API wygląda następująco:

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Założymy, że chcemy dodać metodę, która oczekuje sekwencji elementów i kładzie je na stos. Poniżej mamy pierwsze podejście:

```
// Metoda pushAll bez typów wieloznacznych - niedoskonała!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

Metoda ta kompiluje się bez błędów, ale nie jest w pełni satysfakcjonująca. Jeżeli typ elementów `Iterable src` dokładnie odpowiada tym na stosie, działa ona bez zarzutu. Założymy jednak, że mamy `Stack<Number>` i wywołujemy `push(intVal)`, gdzie `intVal` jest typu `Integer`. Program działa, ponieważ `Integer` dziedziczy po `Number`. Tak więc logicznie rzecz biorąc, ten kod również powinien działać:

```
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Jeżeli spróbujemy go skompilować, otrzymamy komunikat o błędzie, ponieważ, jak wcześniej wspomnieliśmy, typy parametryzowane są niezmiennicze:

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
    numberStack.pushAll(integers);
                        ^
```

Na szczęście istnieje rozwiązanie problemu. Język Java zapewnia specjalny rodzaj typu parametryzowanego, nazywany **związanym szablonem typu**, który pozwala nam radzić sobie z takimi sytuacjami. Typem parametru wejściowego metody `pushAll` nie powinien być „`Iterable` typu `E`”, ale „`Iterable` typu pochodnego po `E`”, co przy użyciu związanego szablonu typu jest wyrażane w następujący sposób: `Iterable<? extends E>`. (Użycie słowa kluczowego `extends` jest nieco mylące — jak pamiętamy z tematu 29., **podtyp** jest tak zdefiniowany, że każdy typ jest swoim podtypem, nawet jeżeli nie jest rozszerzany). Zmodyfikujmy teraz `pushAll`, aby skorzystać z tego typu:

```
// Szablon typu dla parametru służącego jako producent E
public void pushAll(Iterable<? extends E> src) {
```



```

        for (E e : src)
            push(e);
    }

```

Po tej zmianie nie tylko Stack kompiluje się bez problemów, ale również kod klienta, który nie kompilował się przy pierwszej deklaracji pushAll. Ponieważ Stack i kod klienta kompiluje się bez problemów, wiemy że jest zachowane bezpieczeństwo typów.

Założmy, że chcemy napisać metodę popAll, współpracującą z pushAll. Metoda popAll zdejmuję wszystkie elementy ze stosu i dodaje je do przekazanej kolekcji. Poniżej mamy pierwsze podejście do napisania metody popAll:

```

// Metoda popAll bez związanego szablonu typu - nieakceptowalne!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}

```

I tym razem kod kompiluje się bezbłędnie i działa świetnie, jeżeli typy elementów kolekcji docelowej odpowiadają dokładnie tym ze stosu. Jednak i w tym przypadku nie jest ona w pełni satysfakcjonująca. Założmy, że mamy stos Stack<Number> i zmienną typu Object. Jeżeli zdejmujemy element ze stosu i zapiszemy go w zmiennej, kod skompiluje się i wykona bez błędów. Czy nie powinniśmy być w stanie użyć takiego kodu?

```

Stack<Number> numberStack = new Stack<>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);

```

Jeżeli spróbujemy skompilować ten kod dla przedstawionej wcześniej wersji popAll, otrzymamy błąd bardzo podobny do tego, który otrzymaliśmy przy pierwszej wersji pushAll: Collection<Object> nie jest nadtypem Collection<Number>. I tym razem szablony typu pozwalają rozwiązać problem. Typem parametru wejściowego popAll nie powinna być „kolekcja typu E”, ale „kolekcja typu bazowego dla E” (gdzie typ bazowy jest tak zdefiniowany, że E jest typem bazowym dla samego siebie [JLS, 4.10]). Typem szablonowym definiującym precyzyjnie tę zależność jest Collection<? super E>. Zmodyfikujemy teraz metodę popAll:

```

// Typ szablonowy dla parametru służącego jako konsument E
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}

```

Po tej zmianie zarówno Stack, jak i kod klienta kompilują się bez problemu.

Lekcja jest jasna. **W celu zachowania maksymalnej elastyczności należy użyć typu szablonowego dla parametrów wejściowych, które reprezentują producentów lub**

konsumentów. Jeżeli parametr wejściowy jest zarówno producentem, jak i konsumentem, to szablon typu niezbyt się przyda — konieczne jest dokładne dopasowanie typów, które możemy osiągnąć bez użycia typu szablonowego.

W literaturze anglojęzycznej wykorzystywane jest następujące zdanie pomagające zapamiętać, którego z typów szablonowych należy użyć:

PECS jest skrótem od producent nadrzędny, konsument bazowy (ang. *producer-extends, consumer-super*).

Inaczej mówiąc, jeżeli typ parametryzowany reprezentuje producenta *T*, to należy użyć `<? extends T>`; jeżeli *T* jest konsumentem, należy użyć `<? super T>`. W naszym przykładzie klasy *Stack*, parametr *src* metody *pushAll* produkuje obiekty *E* do wykorzystania przez *Stack*, więc odpowiednim typem dla *src* jest *Iterable*`<? extends E>`; parametr *dst* metody *popAll* konsumuje obiekty *E* z *Stack*, więc odpowiednim typem dla *dst* jest *Collection*`<? super E>`. Mnemonika PECS pozwala uchwycić podstawową zasadę rządzącą wykorzystaniem typów szablonowych. Naftalin i Wadler nazywają ją zasadą *Get i Put* [Naftalin07, 2.4].

Pamiętając o tej mnemonice, przyjrzyjmy się kilku deklaracjom metod i konstruktorów z poprzednich tematów. Konstruktor *Chooser* z tematu 28. ma następującą deklarację:

```
public Chooser(Collection<T> choices)
```

Konstruktor *Chooser* korzysta z kolekcji *choices* tylko jako z producenta wartości typu *T* (w celu zapamiętania na potrzeby późniejszego użycia), więc jego deklaracja powinna korzystać z typu szablonowego, który **rozszerza** *T*. Poniżej przedstawiona jest wynikowa deklaracja konstruktora:

```
// Typ szablonowy dla parametru służącego jako producent T
public Chooser(Collection<? extends T> choices)
```

Czy ta zmiana w praktyce wprowadza jakąś różnicę? Jak się okazuje, tak. Załóżmy, że mamy kolekcję *List*`<Integer>` i chcemy przekazać ją do konstruktora *Chooser* `<Number>`. Nie skompiluje się ona przy oryginalnej deklaracji, ale uda się to, jeżeli dodamy związany typ szablonowy.

Spójrzmy teraz na metodę *union* z tematu 30. Poniżej zamieszczona jest jej deklaracja:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Oba parametry, *s1* i *s2*, są producentami *E*, więc zgodnie z mnemoniką PECS deklaracja powinna wyglądać następująco:

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

Warto zwrócić uwagę, że zwracanym typem jest nadal `Set<E>`. **Nie należy używać typów szablonowych jak zwracanych typów.** Zamiast zapewnić użytkownikom większą elastyczność, zmuszamy ich do używania typów szablonowych w kodzie klienta. Po tej modyfikacji poniższy kod uda się skompilować bez żadnych błędów:

```
Set<Integer> integers = Set.of(1, 3, 5);
Set<Double> doubles = Set.of(2.0, 4.0, 6.0);
Set<Number> numbers = union(integers, doubles);
```

Prawidłowo używane typy szablonowe są niemal niewidoczne dla użytkowników klasy. Pozwalają one, aby metody akceptowały takie parametry, jakie powinny, i odrzucały takie, które nie są dozwolone. **Jeżeli użytkownik klasy musi myśleć o typach szablonowych, w API klasy prawdopodobnie tkwi błąd.**

Przed Javą 8 zasady wnioskowania typów nie były wystarczająco sprytnie, aby obsłużyć przedstawiony powyżej fragment kodu, który wymaga od kompilatora użycia zwracanego typu zależnego od kontekstu (lub **typu docelowego**), aby odgadnąć właściwy typ `E`. Typem docelowym wywołania `union` z przykładu jest `Set<Number>`. Jeśli spróbujemy powyższy kod skompilować we wcześniejszych wersjach Javy (wymaga to zastąpienia użycia fabryki `Set.of` innym kodem), otrzymamy długi i skomplikowany błąd:

```
Union.java:14: error: incompatible types
    Set<Number> numbers = union(integers, doubles);
                                ^
required: Set<Number>
found:    Set<INT#1>
where INT#1,INT#2 are intersection types:
    INT#1 extends Number,Comparable<? extends INT#2>
    INT#2 extends Number,Comparable<?>
```

Na szczęście istnieje sposób na poradzenie sobie z tego typu błędami. Jeżeli kompilator nie może wywnioskować typu, którego oczekujemy, można wskazać mu oczekiwany przez nas typ poprzez **jawny parametr typu** [JLS, 15.12]. Nawet przed Java 8 nie był to mechanizm wykorzystywany zbyt często, co jest korzystne, ponieważ jawne parametry typu nie są zbyt ładne. Gdy jest dodany jawny parametr typu, program kompiluje się bezbłędnie również w wersji sprzed Javy 8:

```
// Jawny parametr typu - wymagany przed Javą 8
Set<Number> numbers = union.<Number>union(integers, doubles);
```

Wróćmy teraz do metody `max` z tematu 30. Poniżej zamieszczona jest jej początkowa deklaracja:

```
public static <T extends Comparable<T>> T max(List<T> list)
```

Tutaj mamy jej deklarację wykorzystującą typy szablonowe:

```
public static <T extends Comparable<? super T>> T max(
    List<? extends T> list)
```

Aby uzyskać zmodyfikowaną deklarację z początkowej, dwukrotnie zastosowaliśmy zasadę PECS. Prostym przypadkiem jest parametr `list`. Produkuje on obiekty `T`, więc zmieniliśmy typ z `List<T>` na `List<? extends T>`. Bardziej skomplikowanym przypadkiem jest parametr `T`. Po raz pierwszy widzimy tu zastosowanie szablonu dla parametru typu. Typ `T` był początkowo zdefiniowany jako `Comparable<T>`, ale porównywanie `T` konsumuje obiekty `T` (i produkuje liczby całkowite wskazujące na relację uporządkowania). Z tego powodu typ parametryzowany `Comparable<T>` został zastąpiony związanym typem szablonowym `Comparable<? super T>`. Obiekty `Comparable` są zawsze konsumentami, więc najczęściej należy **używać `Comparable<? super T>` zamiast `Comparable<T>`**. To samo obowiązuje dla komparatorów, więc najczęściej należy **używać `Comparator<? super T>` zamiast `Comparator<T>`**.

Zmieniona deklaracja `max` jest prawdopodobnie najbardziej złożoną deklaracją metody w całej książce. Czy zwiększona złożoność faktycznie nam coś daje? Oczywiście. Przedstawiony poniżej przykład listy jest niedozwolony w przypadku zastosowania pierwszej deklaracji, ale w przypadku rozszerzonej deklaracji jest dozwolony.

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

Powodem braku możliwości zastosowania pierwszej deklaracji metody dla tej listy jest brak implementacji `Comparable<ScheduledFuture>` przez `ScheduledFuture`. Zamiast tego jest to podinterfejs `Delayed`, który dziedziczy po `Comparable<Delayed>`. Inaczej mówiąc, obiekty `ScheduledFuture` nie są nawet porównywalne z innymi obiektami `ScheduledFuture` — są one porównywalne z obiektami `Delayed` i w przypadku oryginalnej deklaracji wystarcza to do ich odrzucenia. Zastosowanie szablonu typu jest niezbędne do obsługi typów, które nie implementują `Comparable` (lub `Comparator`) bezpośrednio, ale rozszerzają typ, który to robi.

Istnieje jeszcze jedno zagadnienie odnoszące się do szablonów. Zachodzi pewna dwoistość pomiędzy parametrami typu i szablonami; wiele metod może być zadeklarowanych za pomocą jednego lub drugiego mechanizmu. Poniżej zamieszczone są na przykład dwie możliwe deklaracje metody statycznej, pozwalającej zamienić dwa elementy na liście, znajdujące się na podanych pozycjach. Pierwsza korzysta z niezwiązanego parametru typu (temat 30.), a druga z niezwiązanego szablonu typu:

```
// Dwie możliwe deklaracje metody swap
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Która z tych deklaracji jest zalecana i dlaczego? W publicznym API druga jest lepsza, ponieważ jest prostsza. Przekazujemy listę — dowolną listę — a metoda zamienia elementy. Nie ma tu elementu typu, o który należałoby zadbać. Można przyjąć zasadę, że **jeżeli parametr typu występuje w deklaracji tylko raz, należy zamienić go na szablon**. Jeżeli jest to niezwiązany parametr typu, należy go zastąpić niezwiązanym szablonem typu; jeżeli jest związanym parametrem typu, należy zamienić go na szablon związany.

Istnieje jeden problem z drugą deklaracją `swap`, korzystającą z szablonu zamiast parametru typu — prosta implementacja nie kompiluje się.

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Próba skompilowania tego kodu spowoduje wyświetlenie mało pomocnego komunikatu o błędzie:

```
Swap.java:5: error: incompatible types: Object cannot be
converted to CAP#1
```

```
list.set(i, list.set(j, list.get(i)));
                        ^
```

where CAP#1 is a fresh type-variable:

```
CAP#1 extends Object from capture of ?
```

Nieprawidłowe jest umieszczenie elementu z powrotem na liście, z której go wcześniej pobraliśmy. Problem wynika z tego, że typem `list` jest `List<?>`, więc nie możemy umieścić w `List<?>` żadnej wartości poza `null`. Na szczęście istnieje sposób na zaimplementowanie tej metody bez korzystania z mało bezpiecznego rzutowania lub typu surowego. Należy wykorzystać prywatną metodę pomocniczą do **przechwycenia** typu szablonowego. Aby metoda mogła przechwycić typ, musi być ona metodą ogólną. Wygląda ona następująco:

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}
```

// Prywatna metoda pomocnicza do przechwycenia szablonu

```
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Metoda `swapHelper` „wie”, że `list` jest typu `List<E>`. Z tego powodu wiadomo, że dowolna wartość wychodząca z listy jest typu `E` i można bezpiecznie umieścić w liście dowolną wartość typu `E`. Ta nieco zawiła implementacja `swap` kompiluje się bez problemów. Pozwala to nam wyeksportować elegancką deklarację `swap` bazującą na typach szablonowych, korzystając wewnętrznie z bardziej skomplikowanych metod ogólnych. Klienci metody `swap` nie muszą zmagać się z bardziej skomplikowaną deklaracją `swapHelper`, ale korzystają z jej zalet. Warto zauważyć, że metoda pomocnicza ma dokładnie taką sygnaturę, jaką wcześniej odrzuciliśmy jako zbyt złożoną dla metody publicznej.

Podsumujmy. Użycie typów szablonowych w API, choć nieco skomplikowane, zapewnia ich większą elastyczność. Jeżeli piszemy bibliotekę, która będzie powszechnie wykorzystywana, prawidłowe użycie typów szablonowych powinno być uznawane za obowiązkowe. Należy pamiętać o podstawowej zasadzie — producent to *extends*, konsument to *super* (PECS). Trzeba również pamiętać, że wszystkie obiekty `Comparable` i `Comparator` są konsumentami.

Temat 32. Ostrożne łączenie typów ogólnych i parametrów varargs

Metody z parametrami varargs (temat 53.), a także typy ogólne to dwa elementy wprowadzone do platformy jako część Javy 5, więc można by oczekiwać, że będą ze sobą dobrze współpracowały. Rzeczywistość niestety jest całkowicie inna. Celem tego rodzaju parametrów jest umożliwienie klientom przekazania do metody zmiennej liczby argumentów. Niestety, ta **abstrakcja ma luki** — w momencie wywołania metody z parametrem varargs powstaje tablica ze wszystkimi przekazanymi w ten sposób parametrami; tablica ta, choć powinna być tylko szczegółem implementacyjnym, jest widoczna. W konsekwencji kompilator zgłasza zaskakujące ostrzeżenia, jeśli parametry varargs okazują się typami ogólnymi lub parametryzowanymi.

Przypomnijmy, że w temacie 28. wskazaliśmy, iż typ nieuściślony to taki, którego reprezentacja w trakcie wykonywania ma mniej informacji niż reprezentacja na etapie kompilacji. Większość typów ogólnych i parametryzowanych to właśnie typy nieuściślane. Jeśli metoda deklaruje parametr varargs jako typ nieuściślony, kompilator w takiej sytuacji wygeneruje ostrzeżenie. Jeśli zostanie w kodzie wywołana metoda, która w parametrze varargs posiada typ nieuściślony, także i w tym przypadku kompilator wygeneruje ostrzeżenie. Ostrzeżenie wygląda podobnie do poniższego:

```
warning: [unchecked] Possible heap pollution from
parameterized vararg type List<String>
```

Skażenie sterty ma miejsce, jeśli zmienna typu parametryzowanego odnosi się do obiektu, który nie jest tego typu [JLS, 4.12.2]. Może ona spowodować, że nie powiedzie się automatyczne rzutowanie wygenerowane przez kompilator, co łamie podstawową gwarancję uogólnionego systemu typów.

Rozważmy poniższą metodę, która stanowi modyfikację fragmentu kodu, który pojawił się w temacie 28.

```
// Mieszanie uogólnień i varargs może zaburzyć bezpieczeństwo typów!
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList; // skażenie sterty
    String s = stringLists[0].get(0); // ClassCastException
}
```

Metoda ta nie zawiera żadnych widocznych rzutowań, a jednak zgłasza wyjątek `ClassCastException`, jeśli zostanie wywołana z co najmniej jednym argumentem. Ostatni wiersz zawiera niewidzialne rzutowanie generowane przez kompilator. Ponieważ rzutowanie to się nie udaje, bezpieczeństwo typów zostało zagrożone i **nie można bezpiecznie umieścić wartości w parametrze varargs wykorzystującym uogólnienia**.

Przykład ten pozwala zadać interesujące pytanie: dlaczego w ogóle możliwe jest zadeklarowanie metody z parametrem `varargs` wykorzystującym uogólnienia, gdy nie można utworzyć jawnie tablicy typu ogólnego? Innymi słowy, dlaczego metoda przedstawiona powyżej generuje jedynie ostrzeżenie, a kod z tematu 28. w ogóle się nie kompilował? Wynika to z faktu, iż metody z parametrami `varargs` wykorzystującymi typy ogólne lub parametryzowane są bardzo przydatne w praktyce, więc projektanci języka zdecydowali się żyć z tą niekonsekwencją. W rzeczywistości biblioteki Javy eksportują kilka tego rodzaju metod, włączając w to `Arrays.asList(T... a)`, `Collections.addAll(Collection<? super T> c, T... elements)` i `EnumSet.of(E first, E... rest)`. W odróżnieniu od metody `dangerous` przedstawionej powyżej metody z bibliotek są bezpieczne, jeśli chodzi o typy.

Przed Javą 7 autor metody wykorzystującej parametr `varargs` z uogólnieniem nie mógł nic zrobić z ostrzeżeniami w momencie użycia metody. Korzystanie z takich metod było więc mało przyjazne. Użytkownicy musieli z tym żyć lub wyciszać ostrzeżenia adnotacjami `@SuppressWarnings("unchecked")` w każdym miejscu wywołującym metodę (temat 27.). Było to zajęcie żmudne, zmniejszające czytelność kodu i ukrywające ostrzeżenia o innych, potencjalnie ważnych pomyłkach.

W Javie 7 do platformy wprowadzono adnotację `SafeVarargs`, która pozwala twórcom metody z parametrem `varargs` typu ogólnego na automatyczne wyłączenie ostrzeżeń po stronie klienckiej. **Adnotacja `SafeVarargs` to obietnica autora metody, że kod jest bezpieczny pod kątem typów.** W zamian za tę obietnicę kompilator zgadza się nie ostrzegać użytkowników metody o niebezpiecznym wywołaniu.

Niezwykle ważne jest, aby nie oznaczać metody jako `@SafeVarargs`, jeśli metoda nie jest **naprawdę** bezpieczna. Co trzeba zrobić, aby uzyskać pewność co do bezpieczeństwa? Przypomnijmy, że tablica ogólna powstaje w momencie wywołania metody, aby zawierać w sobie parametry `varargs`. Jeśli metoda nie przechowuje niczego w tablicy (powodowałoby to nadpisywanie parametrów) i nie pozwala na ucieczkę referencji do tablicy (powodowałoby to, że niezaufany kod miałby dostęp do tablicy), to metoda jest bezpieczna. Innymi słowy, jeśli parametr `varargs` służy jedynie do przeniesienia zmiennej liczby argumentów z części wywołującej do metody — czyli wykonuje tylko to, do czego przeznaczone są parametry `varargs` — to metoda jest bezpieczna.

Musimy w tym miejscu podkreślić, że istnieje sposób na złamanie bezpieczeństwa typów, i to bez próby zapisania czegokolwiek w tablicy parametrów `varargs`. Rozważmy następującą metodę z uogólnionym `varargs`, która zwraca tablicę przekazanych do niej parametrów. Na pierwszy rzut oka metoda wydaje się bardzo przydatna:

```
// NIEBEZPIECZNE - udostępnia referencję do ogólnej tablicy parametrów!
static <T> T[] toArray(T... args) {
    return args;
}
```

Metoda po prostu zwraca otrzymaną tablicę parametrów. Choć metoda wygląda niewinnie, jest niebezpieczna. Typ tablicy określa typy znane w momencie kompilacji, bo są to argumenty przekazane do metody. Czasem jednak kompilator nie ma wystarczającej wiedzy, by wszystko właściwie określić. Ponieważ metoda zwraca tablicę parametrów `varargs`, może przenieść skażenie sterty wyżej.

Aby skonkretyzować całe zagadnienie, rozważmy następującą metodę ogólną, która przyjmuje trzy argumenty typu `T` i zwraca tablicę zawierającą dwa z tych argumentów wybrane drogą losowania:

```
static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
        case 0: return toArray(a, b);
        case 1: return toArray(a, c);
        case 2: return toArray(b, c);
    }
    throw new AssertionError(); // nie można się tu dostać
}
```

Metoda ta nie jest sama w sobie niebezpieczna i nie wygeneruje ostrzeżenia, ale wywołuje metodę `toArray`, która korzysta z uogólnionego parametru `varargs`.

W trakcie kompilowania tej metody kompilator generuje kod tworzący tablicę parametrów `varargs`, która przekazuje dwie instancje `T` do `toArray`. Kod alokuje tablicę typu `Object[]`, która jest najbardziej ogólnym typem mogąym przechować obiekty każdego rodzaju, więc będzie działać niezależnie od tego, co trafi do metody `pickTwo`. Metoda `toArray` po prostu zwraca tablicę do `pickTwo`, a ta zwraca ją do kodu wywołującego, więc `pickTwo` zawsze zwróci tablicę typu `Object[]`.

Rozważmy teraz metodę `main`, która testuje metodę `pickTwo`:

```
public static void main(String[] args) {
    String[] attributes = pickTwo("Dobry", "Szybki", "Tani");
}
```

Ta metoda wygląda całkowicie poprawnie, więc kompiluje się bez generowania żadnych ostrzeżeń. Gdy jednak spróbujemy ją wykonać, otrzymamy wyjątek `ClassCastException`, choć w kodzie nie ma żadnych widocznych rzutowań. Nie widzimy, że kompilator wstawił ukryte rzutowanie na `String[]` dla wartości zwracanej przez `pickTwo`, aby ta mogła zostać umieszczona w `attributes`. Rzutowanie nie udaje się, bo `Object[]` nie jest podtypem `String[]`. Błąd jest bardzo niewdzięczny, ponieważ znajduje się dwa poziomy poza metodą, która tak naprawdę doprowadziła do skażenia sterty (`toArray`). W kodzie nie pojawiła się też żadna próba modyfikacji zawartości tablicy uzyskanej dzięki parametrowi `varargs`.

Przedstawiony przykład ma uświadomić, że **udostępnienie innej metodzie tablicy ogólnych parametrów `varargs` nie jest bezpieczne** poza dwoma wyjątkami: można bezpiecznie przekazać tablicę jako argument do innej metody oznaczonej za pomocą

@SafeVarargs lub też przekazać tablicę do zwykłej metody, która jedynie wylicza pewną wartość na podstawie zawartości tablicy.

Poniżej znajduje się przykład typowego, bezpiecznego użycia ogólnego parametru varargs. Metoda przyjmuje jako argumenty dowolną liczbę list i zwraca jedną listę zawierającą elementy ze wszystkich przekazanych list. Ponieważ metoda została oznaczona jako @SafeVarargs, nie generuje żadnych ostrzeżeń w miejscu deklaracji i w miejscu wywołania:

```
// Bezpieczna metoda wykorzystująca ogólny parametr varargs
@SafeVarargs
static <T> List<T> flatten(List<? extends T>... lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

Zasada określająca, czy można użyć adnotacji SafeVarargs, jest prosta — **użyj @SafeVarargs w każdej metodzie z parametrem varargs typu ogólnego lub parametryzowanego**, aby jej użytkownik nie musiał zmagać się z dziwnymi ostrzeżeniami kompilatora. Oznacza to, że nie należy nigdy tworzyć niebezpiecznych metod varargs takich jak dangerous lub toArray. Za każdym razem, gdy kompilator ostrzega o możliwym skażeniu sterty z powodu użycia parametru varargs dla typu ogólnego w metodzie, nad którą sprawujemy kontrolę, sprawdź, czy metoda jest bezpieczna. Przypomnijmy, że metoda jest bezpieczna, jeśli:

1. Nie stara się zapamiętać czegokolwiek w tablicy parametrów varargs.
2. Nie powoduje uwidocznienia tablicy (lub jej klonu) niezaufanemu kodowi.

Jeśli złamana jest którakolwiek z tych zasad, napraw kod.

Pamiętaj, że adnotacja SafeVarargs jest dopuszczalna tylko dla metod, których nie można przesłonić, bo nie jesteśmy w stanie zagwarantować bezpieczeństwa każdej z przesłoniętych wersji. W Javie 8 adnotacja jest dopuszczalna jedynie dla metod statycznych i metod instancyjnych typu final. W Javie 9 dopuszczono jej stosowanie również dla prywatnych metod instancyjnych.

Alternatywą dla adnotacji SafeVarargs jest skorzystanie z rady udzielonej w temacie 28. i zastąpienie parametrów varargs (czyli tak naprawdę ukrytej tablicy) parametrem typu List. Oto jak wyglądałaby metoda flatten po takiej modyfikacji. Zwróć uwagę, że zmieniła się jedynie deklaracja parametru:

```
// Klasa List jako bezpieczna pod kątem typów alternatywa dla ogólnego
// parametru varargs
static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
```

```

        result.addAll(list);
    return result;
}

```

Metodę tę można połączyć ze statyczną metodą fabryczną `List.of`, aby umożliwić stosowanie zmiennej liczby argumentów. Zauważ, że to podejście bazuje na fakcie, iż deklaracja `List.of` jest oznaczona z użyciem `@SafeVarargs`:

```
audience = flatten(List.of(friends, romans, countrymen));
```

Zaletą tego podejścia jest to, że kompilator potrafi **udowodnić**, iż metoda jest bezpieczna pod kątem typów. Nie trzeba samemu ręczyć za bezpieczeństwo adnotacją `SafeVarargs` ani zastanawiać się, czy przypadkiem nie pomyliło się w kwestii szacowania bezpieczeństwa metody. Główną wadą tego podejścia jest to, że kod kliencki jest bardziej rozwlekły i może działać nieco wolniej.

Tę sztuczkę można także zastosować w sytuacjach, gdy nie jesteśmy w stanie napisać bezpiecznej metody `varargs`, czyli w sytuacji prezentowanej wcześniej metody `toArray`. W świecie `List` odpowiednikiem tej metody **jest** metoda `List.of`, więc nawet nie musimy jej pisać, bo twórcy bibliotek Javy wykonali całą pracę za nas. W tej wersji metoda `pickTwo` ma postać:

```

static <T> List<T> pickTwo(T a, T b, T c) {
    switch(rnd.nextInt(3)) {
        case 0: return List.of(a, b);
        case 1: return List.of(a, c);
        case 2: return List.of(b, c);
    }
    throw new AssertionError();
}

```

a metoda `main` ma postać:

```

public static void main(String[] args) {
    List<String> attributes = pickTwo("Dobry", "Szybki", "Tani");
}

```

Wynikowy kod jest bezpieczny pod kątem typów, bo w całości polega na typach ogólnych, a nie na tablicach.

Podsumujmy. Parametry `varargs` i typy ogólne niezbyt dobrze współgrają ze sobą, bo `varargs` to dziurawa abstrakcja realizowana przy użyciu tablic, a tablice stosują inne reguły dotyczące typów niż uogólnienia. Choć parametry `varargs` z typami ogólnymi nie są bezpieczne pod kątem typów, są dopuszczone do użycia. Jeśli zdecydujesz się napisać metodę z parametrem `varargs` z typem ogólnym (lub parametryzowanym), upewnij się, że jest ona bezpieczna pod kątem typów, a następnie użyj dla niej adnotacji `@SafeVarargs`, aby nie była nieprzyjemna w użyciu.

Temat 33. Wykorzystanie heterogenicznych kontenerów bezpiecznych dla typów

Najczęstszym zastosowaniem typów ogólnych są kolekcje, takie jak `Set<E>` czy `Map<K, V>` oraz kontenery jednoelementowe, takie jak `ThreadLocal<T>` czy `AtomicReference<T>`. We wszystkich tych zastosowaniach kontenery są parametryzowane. Ogranicza to nas do stałej liczby parametrów na kontener. Normalnie jest to właśnie oczekiwane działanie. Kontener `Set` posiada jeden parametr typu reprezentujący typ elementu; `Map` posiada dwa, reprezentujące typ klucza i wartości i tak dalej.

Czasami jednak potrzebujemy większej elastyczności. Na przykład wiersze bazy danych mogą składać się z dowolnie wielu kolumn i dobrze byłoby mieć możliwość dostępu do nich w sposób bezpieczny dla typów. Na szczęście istnieje łatwy sposób osiągnięcia tego efektu. W tym celu należy sparametryzować **klucz** zamiast **kontenera**. Następnie należy przekazać sparametryzowany klucz do kontenera, aby wstawić lub pobrać wartość. System typów ogólnych jest wykorzystywany do zagwarantowania typu wartości zgadzających się z tym kluczem.

Prostym przykładem takiego podejścia jest klasa `Favorites`, która pozwala klientom przechowywać i odczytywać „ulubione” obiekty dowolnie wielu klas. Obiekt `Class` jest tu częścią klucza sparametryzowanego. Mechanizm ten działa, ponieważ klasa `Class` jest uogólniona. Typem literału `class` nie jest `Class`, ale `Class<T>`. Na przykład `String.class` jest typu `Class<String>`, natomiast `Integer.class` jest typu `Class<Integer>`. Gdy w czasie kompilacji oraz wykonywania literał jest przekazywany pomiędzy metodami w celu przesyłania informacji o typie, jest on nazywany **tokenem typu** [Bracha04].

API klasy `Favorites` jest proste. Wygląda jak proste odwzorowanie, ale parametryzowany jest klucz zamiast odwzorowania. Przy ustawianiu i pobieraniu ulubionych klient prezentuje obiekt `Class`. API jest następujące:

```
// Wzorzec kontenera heterogenicznego bezpiecznego dla typów - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Poniżej przedstawiony jest przykładowy program testujący klasę `Favorites`, który zapamiętuje, pobiera i wyświetla ulubione obiekty `String`, `Integer` oraz `Class`.

```
// Wzorzec kontenera heterogenicznego bezpiecznego dla typów - klient
public static void main(String[] args) {
    Favorites f = new Favorites();
}
```

```

        f.putFavorite(String.class, "Java");
        f.putFavorite(Integer.class, 0xcafebabe);
        f.putFavorite(Class.class, Favorites.class);
        String favoriteString = f.getFavorite(String.class);
        int favoriteInteger = f.getFavorite(Integer.class);
        Class<?> favoriteClass = f.getFavorite(Class.class);
        System.out.printf("%s %x %s%n", favoriteString,
                           favoriteInteger, favoriteClass.getName());
    }

```

Jak można się spodziewać, program ten wyświetla Java cafebabe Favorites. Zwróć uwagę na różnicę w działaniu metody printf w Javie w odróżnieniu od jej pierwowzoru z C, ponieważ w miejscach użycia `\n` w C należy wstawić `%n`. Zastosowanie `%n` spowoduje wygenerowanie separatora wiersza zależnego od platformy, czyli na wielu platformach, ale nie na wszystkich, będzie to `\n`.

Obiekt Favorites jest **bezpieczny dla typów** — nigdy nie zwróci on Integer, gdy pytamy o String. Jest on również **heterogeniczny** — w przeciwieństwie do zwykłego odwzorowania wszystkie klucze są różnych typów. Z tego powodu nazywamy Favorites **kontenerem heterogenicznym bezpiecznym dla typów**.

Implementacja Favorites jest zaskakująco niewielka. Możemy ją tu zamieścić w całości:

```

// Wzorzec kontenera heterogenicznego bezpiecznego dla typów - implementacja
public class Favorites {
    private Map<Class<?>, Object> favorites = new HashMap<>();

    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}

```

Wykonywanych jest w niej kilka ciekawych operacji. Każda klasa Favorites ma prywatne pole `Map<Class<?>, Object>` o nazwie favorites. Być może uważasz, że do tego pola nie można nic wstawić z powodu zastosowania niezwiązanego typu szablonowego, ale prawda jest inna. Należy zauważyć, że typ szablonowy jest zagnieżdżony — to nie Map jest typu szablonowego, ale jego klucz. Oznacza to, że każdy klucz może mieć **inny** typ parametryzowany — pierwszy może być `Class<String>`, następny `Class<Integer>` i tak dalej. Właśnie tak została osiągnięta heterogeniczność.

Należy również zauważyć, że wartością typu dla mapy favorites jest po prostu Object. Inaczej mówiąc, Map nie gwarantuje relacji typu pomiędzy kluczami i wartościami, ponieważ każda wartość jest typu reprezentowanego przez jej klucz. W rze-

czywistości system typów języka Java nie może wyrazić takiej zależności. Wiemy jednak, że jest to prawda i korzystamy z tego faktu w czasie odczytywania ulubionych.

Implementacja `putFavorite` jest bardzo prosta — po prostu umieszcza w `favorites` odwzorowanie danego obiektu `Class` na ulubioną instancję. Jak już wcześniej wspomnieliśmy, zrywa to „połączenie typu” pomiędzy kluczem i wartością; tracąca jest informacja o tym, że wartość jest obiektem klasy będącej kluczem. Jest to jednak dopuszczalne, ponieważ metoda `getFavorites` pozwala odtworzyć to połączenie.

Implementacja `getFavorite` jest nieco trudniejsza niż `putFavorite`. Pobiera ona z kontenera `favorites` wartość odpowiadającą danemu obiektowi `Class`. Jest to prawidłowa referencja obiektu do zwrócenia, ale ma ona niewłaściwy typ dla kompilacji. Typem tym jest po prostu `Object` (typ wartości odwzorowania `favorites`), a my musimy zwrócić `T`. Tak więc `getFavorite` implementuje **dynamiczne rzutowanie** referencji obiektu na typ reprezentowany przez obiekt `Class` przy użyciu metody `cast` z `Class`.

Metoda `cast` jest dynamicznym odpowiednikiem operatora rzutowania języka Java. Sprawdza ona, czy jej argument jest instancją typu reprezentowanego przez obiekt `Class`. Jeżeli tak, zwraca argument, a w przeciwnym razie zgłasza wyjątek `ClassCastException`. Wiemy, że wywołanie rzutowania w `getFavorite` nigdy nie zgłosi wyjątku `ClassCastException`, przy założeniu, że kod klienta został skompilowany bez problemów. Wiemy, że wartości w kontenerze `favorites` zawsze odpowiadają typom kluczy.

Co wykonuje dla nas metoda `cast`, jeżeli po prostu zwraca ona wartość? Sygnatura metody `cast` w pełni wykorzystuje fakt, że klasa `Class` jest uogólniona. Jej zwracany typ jest parametrem typu obiektu `Class`:

```
public class Class<T> {  
    T cast(Object obj);  
}
```

Tego dokładnie potrzebuje metoda `getFavorite`. Pozwala to nam zapewnić bezpieczeństwo dla typów klasy `Favorites` bez korzystania z niekontrolowanego rzutowania na `T`.

Klasa `Favorites` ma dwa ograniczenia, o których warto wspomnieć. Łatwo można naruszyć bezpieczeństwo typów obiektu `Favorites` poprzez użycie obiektu `Class` w swojej surowej postaci. Jednak taki kod klienta spowoduje w czasie kompilacji wygenerowanie ostrzeżenia o niekontrolowanym rzutowaniu. Nie różni się to od normalnej implementacji kolekcji, takich jak `HashSet` czy `HashMap`. Można bez trudu umieścić `String` w `HashSet<Integer>` przez użycie surowego typu `HashSet` (temat 26.). Można powiedzieć, że będziemy mieli zapewnione bezpieczeństwo typów w czasie działania, jeżeli będziemy chcieli o nie zadbać. Aby zapewnić brak możliwości

naruszenia niezmiennika typu przez obiekty Favorites, należy umieścić w metodzie putFavorite sprawdzenie, czy instance jest faktycznie typu reprezentowanego przez type. Wiemy już, jak można to zrobić. Wystarczy użyć rzutowania dynamicznego.

```
// Uzyskiwanie bezpieczeństwa typów w czasie działania
// przez użycie rzutowania dynamicznego
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

Dostępne są kolekcje osłonowe w java.util.Collections, które korzystają z tego mechanizmu. Są one nazwane checkedSet, checkedList, checkedMap i tak dalej. Ich statyczne metody fabryczne oprócz kolekcji oczekują podania obiektu (lub dwóch) Class. Statyczne metody fabryczne są ogólne, co zapewnia, że typ obiektu Class w czasie kompilacji odpowiada typowi kolekcji. Klasy osłonowe dodają uściślenie do osłanianych kolekcji. Na przykład klasa osłonowa zgłasza w czasie działania wyjątek ClassCastException, jeżeli ktoś spróbuje umieścić obiekt Coin w naszej kolekcji Collection<Stamp>. Te klasy osłonowe są przydatne do śledzenia dodawania do kolekcji elementów o nieprawidłowym typie w aplikacjach, w których wykorzystywany jest starszy oraz nowy, ogólny kod.

Drugim ograniczeniem klasy Favorites jest brak możliwości jej użycia dla typów nieuściślonych (temat 28.). Inaczej mówiąc, można zapamiętywać ulubiony String lub String[], ale nie List<String>. Jeżeli będziemy próbować zapamiętać ulubiony obiekt List<String>, program nie skompiluje się. Powodem jest brak możliwości uzyskania obiektu Class dla List<String> — List<String>.class jest błędem składniowym, co ma też swoje dobre strony. List<String> oraz List<Integer> współdzielą jeden obiekt Class, czyli List.class. Spowodowałoby to zniszczenie wewnętrznych mechanizmów obiektu Favorites, jeżeli „literały typu” List<String>.class i List<Integer>.class byłyby prawidłowe i zwracałyby tę samą referencję obiektu. Nie ma w pełni satysfakcjonującego rozwiązania tego ograniczenia.

Tokeny typu używane przez Favorites są niezwiązane — getFavorite oraz putFavorite akceptują dowolne obiekty Class. Czasami może zająć potrzeba ograniczenia typów przekazywanych do metody. Można to osiągnąć przy użyciu **tokena typu związanego**, który jest po prostu tokenem związanym z typem, który reprezentuje, co jest realizowane przez związany parametr typu (temat 30.) lub związany szablon (temat 31.).

API adnotacji (temat 39.) intensywnie wykorzystuje tokeny typu związanego. Poniżej znajduje się przykładowa metoda pozwalająca na odczytanie adnotacji w czasie działania programu. Metoda ta pochodzi z interfejsu AnnotatedElement, który jest implementowany przez typy refleksyjne reprezentujące klasy, metody, pola i inne elementy programu:

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

Argument `annoationType` jest tokenem typu związanego reprezentującym typ adnotacji. Metoda zwraca element adnotacji tego typu, jeżeli posiada taki, lub `null`, jeżeli go nie posiada. W rzeczywistości element z adnotacją jest bezpiecznym dla typów kontenerem heterogenicznym, którego kluczami są typy adnotacji.

Załóżmy, że mamy element typu `Class<?>` i chcemy przekazać go do metody wymagającej tokena typu związanego, na przykład `getAnnotation`. Można rzutować obiekt na `Class<? extends Annotation>`, ale rzutowanie to jest niekontrolowane, więc będzie w czasie kompilacji generowało ostrzeżenie (temat 27.). Na szczęście klasa `Class` posiada metodę instancyjną, która bezpiecznie (i dynamicznie) wykonuje takie rzutowanie. Metoda ta nosi nazwę `asSubclass` i rzutuje obiekt `Class` na klasę pochodną reprezentowaną przez jej argument. Jeżeli rzutowanie się powiedzie, metoda zwraca jej argument; jeżeli nie uda się, zgłasza `ClassCastException`.

Poniżej przedstawione jest zastosowanie metody `asSubclass` do odczytania adnotacji o typie nieznanym w czasie kompilacji. Metoda ta kompiluje się bez błędów i ostrzeżeń:

```
// Użycie asSubclass do bezpiecznego rzutowania na token typu związanego
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
    Class<?> annotationType = null; // niezwiązany token typu
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

Podsumujmy. Normalne wykorzystanie typów ogólnych, którego przykładem są API kolekcji, ogranicza nas do stałej liczby parametrów typu na kontener. Można obejść to ograniczenie przez umieszczenie parametru typu w kluczu zamiast w kontenerze. W takich bezpiecznych dla typu kontenerach heterogenicznych można korzystać z obiektów `Class` jako kluczy. Obiekt `Class` używany w ten sposób jest nazywany tokenem typu. Można również używać własnego typu klucza — na przykład wykorzystać typ `DatabaseRow` do reprezentowania wiersza bazy danych (kontenera) oraz typ ogólnego `Column<T>` jako klucza.

Typy wyliczeniowe i adnotacje

Java obsługuje dwie rodziny specjalnych typów referencyjnych — pewien rodzaj klas nazwany **typem wyliczeniowym** oraz pewien rodzaj interfejsu nazwany **typem adnotacyjnym**. W tym rozdziale przedstawimy najlepsze praktyki związane z wykorzystywaniem obu rodzin typów.

Temat 34. Użycie typów wyliczeniowych zamiast stałych int

Typ wyliczeniowy to typ, którego prawidłowe wartości tworzy stały zbiór, na przykład pory roku, planety w systemie słonecznym lub rodzaje kart w talii. Zanim do języka został dodany typ wyliczeniowy, standardową praktyką reprezentowania takich typów było deklarowanie grupy zmiennych typu `int`, po jednej dla każdej wartości typu.

// Wzorzec typu wyliczeniowego z użyciem int - bardzo niedoskonały!

```
public static final int APPLE_FUJI      = 0;  
public static final int APPLE_PIPPIN   = 1;  
public static final int APPLE_GRANNY_SMITH = 2;
```

```
public static final int ORANGE_NAVEL   = 0;  
public static final int ORANGE_TEMPLE = 1;  
public static final int ORANGE_BLOOD  = 2;
```

Technika ta, znana pod nazwą **wzorca wyliczeniowego int**, ma wiele wad. Nie zapewnia ona żadnej formy bezpieczeństwa typów i jest mało wygodna. Kompilator nie będzie protestował, jeżeli prześlemy jabłko do metody oczekującej pomarańczy, porównamy jabłko z pomarańczą za pomocą operatora `==` lub co gorsza:

```
// Smaczny sok jabłkowy z domieszką cytrusów!  
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Jak można zauważyć, nazwa każdej ze stałych dla jabłka zaczyna się od `APPLE_`, a nazwa każdej stałej pomarańczy zaczyna się od `ORANGE_`. Dzieje się tak, ponieważ Java nie zapewnia przestrzeni nazw dla grup wartości wyliczeniowych typu `int`. Prefiksy zapobiegają powtórzeniu nazw, gdy dwie grupy wyliczeniowe mają tak samo nazwane stałe, co pozwala odróżnić od siebie `ELEMENT_MERCURY` i `PLANET_MERCURY`.

Programy korzystające z wzorca wyliczeniowego `int` są niewygodne. Ponieważ stałe wyliczeniowe `int` są **zmiennymi stałymi** [JLS, 4.12.4], są kompilowane do programów klienckich, które je wykorzystują [JLS, 13.1]. Jeżeli wartość `int` związana ze stałą wyliczeniową zmieni się, klient musi być ponownie skompilowany. W przeciwnym razie będzie nadal działał, ale jego zachowanie będzie niezdefiniowane.

Nie istnieje prosty sposób przetłumaczenia stałej wyliczeniowej na napis do wyświetlenia. Jeżeli drukujemy taką stałą lub wyświetlamy ją w debugerze, widzimy tylko liczbę, co nie jest zbyt pomocne. Nie istnieje niezawodny sposób na iterowanie po wszystkich wartościach `int` w typie wyliczeniowym, a nawet określenie wielkości grupy stałych `int`.

Można również spotkać odmianę tego wzorca, w którym w miejsce stałych `int` są używane stałe `String`. Wariant ten, nazywany **wzorcem wyliczeniowym String**, jest nawet mniej polecany. Choć zapewnia on czytelne napisy dla stałych, może prowadzić do problemów z wydajnością, ponieważ korzysta z porównywania napisów. Co gorsza, może doprowadzić do tego, że niedoświadczeni użytkownicy na stałe wpiszą stałe znakowe do kodu klienta, zamiast korzystać z nazw pól. Jeżeli taki wpisany napis zawiera błąd typograficzny, wymknie się spod kontroli w czasie kompilacji i spowoduje błąd w czasie działania aplikacji.

Na szczęście Java zapewnia alternatywę pozwalającą na ominięcie wad wzorców wyliczeniowych `int` i `String`, dającą wiele nowych korzyści. Typ ten jest zdefiniowany w podręczniku jako **typ enum** [JLS, 8.9]. W najprostszej postaci wygląda następująco:

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }  
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

Na pierwszy rzut oka typ wyliczeniowy może wyglądać podobnie do typów znanych z innych języków, takich jak C, C++ i C#, ale podobieństwo jest mylące. Typ wyliczeniowy w języku Java jest w pełni wartościową klasą, znacznie bardziej zaawansowaną niż odpowiedniki w innych językach, w których typ wyliczeniowy jest właściwie zbiorem wartości `int`.

Podstawowe założenie leżące u podstaw typu wyliczeniowego w języku Java jest proste — są to klasy eksportujące po jednym obiekcie dla każdej stałej wyliczenia z użyciem finalnego statycznego pola publicznego. Typy wyliczeniowe są w efekcie finalne, dzięki temu, że nie jest dostępny konstruktor. Ponieważ klienci nigdy nie tworzą obiektów typu wyliczeniowego ani po nich nie dziedziczą, nie mogą istnieć instancje inne niż zadeklarowane stałe wyliczeniowe. Inaczej mówiąc, typy wyliczeniowe są kontrolowanymi instancjami (temat 1.). Są one generalizacją singletonów (temat 3.), które są w praktyce jednoelementowymi typami wyliczeniowymi.

Typ wyliczeniowy zapewnia bezpieczeństwo typów w czasie kompilacji. Jeżeli zadeklarujemy, że parametr będzie typu `Apple`, to gwarantowane jest, że przekazana do parametru referencja obiektu różna od `null` będzie jedną z trzech prawidłowych wartości typu `Apple`. Próba przekazania wartości niewłaściwego typu spowoduje błąd kompilacji, tak samo jak próba przypisania wyrażenia jednego typu wyliczeniowego do zmiennej innego albo użycia operatora `==` do porównywania wartości różnych typów wyliczeniowych.

Typy wyliczeniowe o identycznie nazwanych stałych mogą bez problemów współistnieć, ponieważ każdy typ posiada własną przestrzeń nazw. Można dodawać lub zmieniać kolejność stałych w typie wyliczeniowym bez konieczności ponownej kompilacji klientów, ponieważ pola eksportujące stałe zapewniają warstwę izolacyjną pomiędzy typem wyliczeniowym i jego klientami — stałe nie są wkompilowywane w klienty, tak jak miało to miejsce w przypadku wzorca typu wyliczeniowego `int`. Można również tłumaczyć wartości typu wyliczeniowego na czytelne napisy przez wywołanie ich metody `toString`.

Oprócz naprawiania niedociągnięć typu wyliczeniowego korzystającego z `int` nowe typy wyliczeniowe pozwalają dopisać dowolne metody i implementować dowolne interfejsy. Zapewniają wysokiej jakości implementacje wszystkich metod klasy `Object` (rozdział 3.), implementują `Comparable` (temat 14.) i `Serializable` (rozdział 12.), a ich serializowana postać jest zaprojektowana tak, aby przetrwać większość zmian typu wyliczeniowego.

Do czego jest potrzebna możliwość dodawania do typu wyliczeniowego metod i pól? Przykładem może być możliwość skojarzenia danych ze stałymi. Nasze typy `Apple` i `Orange` mogą na przykład zawierać metodę zwracającą kolor owocu lub nawet jego zdjęcie. Można wzbogacać typ wyliczeniowy o dowolną metodę, która wydaje się nam odpowiednia. Typ wyliczeniowy może rozpocząć życie jako prosta kolekcja stałych wyliczeniowych i przeobrazić się w rozbudowaną abstrakcję.

Dobrym przykładem takiego bogatego typu wyliczeniowego może być typ modelujący osiem planet naszego systemu słonecznego. Każda planeta ma masę i promień, a na podstawie tych dwóch parametrów można obliczyć grawitację na powierzchni.

To z kolei pozwala obliczyć ciężar obiektu na powierzchni planety na podstawie masy obiektu. Poniżej zamieszczony jest przykład takiego typu wyliczeniowego. Liczby w nawiasach po stałych wyliczeniowych są parametrami przekazywanymi do konstruktora. W tym przypadku jest to masa i promień planety:

```
// Typ wyliczeniowy z danymi i operacjami
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass; // w kilogramach
    private final double radius; // w metrach
    private final double surfaceGravity; // w m / s^2

    // uniwersalna stała grawitacyjna w m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // konstruktor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass() { return mass; }
    public double radius() { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity; // F = ma
    }
}
```

Jak widać, bardzo łatwo jest napisać bogaty typ wyliczeniowy, taki jak Planet. **Aby skojarzyć dane ze stałymi wyliczenia, należy zadeklarować pola instancyjne i napisać konstruktor pobierający dane i zapisujący je w polach.** Typy wyliczeniowe są z natury niezmiennie, więc wszystkie pola powinny być oznaczone final (temat 17.). Mogą być one publiczne, ale znacznie lepiej zadeklarować je jako prywatne i udostępnić publiczne akcesory (temat 16.). W przypadku typu Planet konstruktor oblicza i zapisuje grawitację na powierzchni, ale jest to tylko optymalizacja. Grawitacja może być wyliczana na podstawie masy i promienia przy każdym wywołaniu metody surfaceWeight, który pobiera masę obiektu i zwraca jego ciężar na planecie reprezentowanej przez stałą.

Choć typ wyliczeniowy `Planet` jest prosty, daje on zaskakująco dużo możliwości. Poniżej znajduje się krótki program pobierający wagę obiektu na Ziemi (w dowolnej jednostce) i wyświetla elegancką tabelę wag obiektu na wszystkich ośmiu planetach (w tej samej jednostce):

```
public class WeightTable {
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Waga na %s wynosi %f%n",
                              p, p.surfaceWeight(mass));
    }
}
```

Należy zwrócić uwagę, że `Planet`, podobnie jak inne typy wyliczeniowe, posiada metodę statyczną `values`, która zwraca tablicę wartości typu w kolejności ich zadeklarowania. Należy również zwrócić uwagę, że metoda `toString` zwraca zadeklarowaną nazwę każdej z wartości typu wyliczeniowego, co pozwala w łatwy sposób drukować ją za pomocą `println` i `printf`. Jeżeli nie jesteśmy usatysfakcjonowani reprezentacją tekstową stałej, można ją zmienić przez nadpisanie metody `toString`. Poniżej zamieszczony jest wynik działania naszego małego programu `WeightTable` (wyliczenie nie przysłania metody `toString`) z przekazaną w wierszu poleceń wartością 185:

```
Waga na MERCURY wynosi 69.912739
Waga na VENUS wynosi 167.434436
Waga na EARTH wynosi 185.000000
Waga na MARS wynosi 70.226739
Waga na JUPITER wynosi 467.990696
Waga na SATURN wynosi 197.120111
Waga na URANUS wynosi 167.398264
Waga na NEPTUNE wynosi 210.208751
```

Aż do roku 2006, czyli dwa lata po dodaniu typu wyliczeniowego do Javy, Pluton był planetą. Warto w tym momencie zadać sobie pytanie, co się stanie, jeśli usuniemy element z typu wyliczeniowego. Dowolny klient, który nie korzystał bezpośrednio z usuniętego elementu, nadal będzie działał prawidłowo. Nasz przykładowy program `WeightTable` nadal działałby prawidłowo — po prostu wyświetlałby jeden wiersz mniej. Co stanie się z programem, który jawnie korzystał z usuniętego elementu (w tym przypadku `Planet.PLUTO`)? Jeśli program będzie kompilowany ponownie, proces ten się nie powiedzie i zostanie wyświetlony komunikat o błędzie wskazujący na odniesienie do zdegradowanej planety. Jeśli nie dojdzie do ponownej kompilacji, program wyświetli podobny komunikat w trakcie działania. To najlepsze zachowanie, na jakie można liczyć, znacznie przyjemniejsze od tego, które dostarczyłby wzorzec wyliczenia `int`.

Niektóre operacje związane ze stałymi wyliczanymi mogą być używane wyłącznie w klasie lub pakiecie, w którym jest zdefiniowany typ wyliczeniowy. Takie operacje najlepiej zaimplementować jako metody prywatne lub prywatne w ramach pakietu. Każda ze stałych posiada wtedy ukrytą kolekcję operacji, które pozwalają na odpowiednie reagowanie w klasie lub pakiecie w przypadku manipulacji na stałych. Tak samo jak w przypadku innych klas, o ile nie mamy ważnej potrzeby udostępniania metod klientom, należy zadeklarować je jako prywatne lub, jeżeli jest to wymagane, prywatne w ramach pakietu (temat 15.).

Jeżeli typ wyliczeniowy jest ogólnie przydatny, powinien być klasą najwyższego poziomu, natomiast jeżeli jest używany w klasie najwyższego poziomu, powinien być jej składnikiem (temat 24.). Na przykład typ wyliczeniowy `java.math.RoundingMode` reprezentuje tryb zaokrąglania ułamków dziesiętnych. Tryby zaokrąglania są wykorzystywane w klasie `BigDecimal`, ale zapewniają one przydatną abstrakcję, która nie jest fundamentalnie związana z `BigDecimal`. Przez umieszczenie typu wyliczeniowego `RoundingMode` na najwyższym poziomie projektanci biblioteki zachęcają programistów potrzebujących trybu zaokrąglania do ponownego użycia tego typu, dzięki czemu uzyskuje się większą spójność API.

Techniki zademonstrowane w przykładzie `Planet` są wystarczające dla większości typów wyliczeniowych, ale czasami potrzeba więcej. Z każdą stałą `Planet` są związane różne dane, ale czasami potrzebujemy skojarzyć całkowicie inne **działanie** z każdą ze stałych. Załóżmy, że chcemy napisać typ wyliczeniowy reprezentujący operacje na prostym kalkulatorze czterodziałaniowym i chcemy zapewnić metodę do wykonania operacji arytmetycznych reprezentowanych przez każdą ze stałych. Jednym ze sposobów realizacji jest wybór wyrażenia w zależności od wartości wyliczenia:

```
// Typ wyliczeniowy z wyborem spośród własnych wartości - niedoskonały
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // wykonanie operacji arytmetycznej reprezentowanej przez stałą
    public double apply(double x, double y) {
        switch(this) {
            case PLUS: return x + y;
            case MINUS: return x - y;
            case TIMES: return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Nieznana operacja: " + this);
    }
}
```

Kod ten działa, ale nie jest zbyt ładny. Nie kompiluje się on bez instrukcji `throw`, ponieważ koniec metody jest technicznie osiągalny, nawet jeżeli nie zostanie nigdy osiągnięty [JLS, 14.21]. Co gorsza, kod ten jest delikatny. Jeżeli dodamy nową

stałą wyliczeniową, a zapomnimy dodać odpowiedni przypadek w switch, typ nadal będzie się kompilował, ale wystąpi błąd w czasie działania przy próbie wykonania nowej operacji.

Na szczęście istnieje lepszy sposób na kojarzenie różnych operacji z każdą stałą wyliczenia — można zadeklarować abstrakcyjną metodę `apply` w typie wyliczeniowym i nadpisywać ją w konkretnej metodzie dla stałej umieszczonej w **treści klasy specyficznej dla stałej**. Takie metody są znane jako **implementacje metod specyficznych dla stałych**.

// Typ wyliczeniowy z implementacją metod specyficznych dla stałej

```
public enum Operation {
    PLUS { public double apply(double x, double y){return x + y;} },
    MINUS { public double apply(double x, double y){return x - y;} },
    TIMES { public double apply(double x, double y){return x * y;} },
    DIVIDE { public double apply(double x, double y){return x / y;} };

    public abstract double apply(double x, double y);
}
```

Jeżeli dodamy nową stałą do drugiej wersji `Operation`, mało prawdopodobne jest, że zapomnimy napisać metody `apply`, ponieważ znajduje się ona zaraz po deklaracji każdej ze stałych. W mało prawdopodobnym przypadku, gdy jednak jej zapomnimy, kompilator przypomni nam o tym, ponieważ metody abstrakcyjne w typie wyliczeniowym muszą być przesłaniane konkretnymi metodami w każdej ze stałych.

Implementacja metody specyficzna dla stałej może być łączona z danymi specyficznymi dla stałej. Poniżej znajduje się na przykład kolejna wersja `Operation`, w której nadpisujemy metodę `toString` w celu zwrócenia symbolu skojarzonego z operacją.

// Typ wyliczeniowy z treścią klasy specyficzną dla stałej oraz z danymi

```
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

Na przykład przedstawiona powyżej implementacja `toString` pozwala łatwo wyświetlać wyrażenia arytmetyczne w sposób pokazany przez ten mały program.

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                           x, op, y, op.apply(x, y));
}
```

Uruchomienie tego programu z wartościami 2 i 4 przekazanymi w wierszu poleceń daje nam następujący wynik:

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

Typy wyliczeniowe mają automatycznie generowaną metodę `valueOf(String)`, która przekształca nazwę stałej na samą stałą. Jeżeli nadpiszemy metodę `toString` w typie wyliczeniowym, warto rozważyć również napisanie metody `fromString` w celu przekształcenia własnej reprezentacji na odpowiednią stałą. Poniższy kod (po odpowiedniej zmianie typu) realizuje to zadanie dla dowolnego typu wyliczeniowego, o ile każda stała będzie miała unikatową reprezentację znakową.

```
// Implementacja metody fromString dla typu wyliczeniowego
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));

// zwraca Operation dla napisu lub null, jeżeli napis jest nieprawidłowy
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

Należy zwrócić uwagę, że stałe `Operation` są umieszczane w odwzorowaniu `stringToEnum` w bloku statycznym, który jest wykonywany po utworzeniu stałych. Przedstawiony kod używa strumienia (rozdział 7.) dla tablicy zwróconej przez metodę `values()`. Przed Javą 8 musielibyśmy utworzyć pusty obiekt `HashMap`, a następnie przejść po każdej wartości z tablicy i wstawić ją do odwzorowania. Oczywiście jeśli chcesz, możesz nadal korzystać z takiego rozwiązania. Pamiętaj jednak, że próba umieszczenia każdej stałej w odwzorowaniu w jej konstruktorze **nie** zadziała. Spowoduje błąd kompilacji, co jest prawidłowym działaniem, ponieważ nastąpiłoby wygenerowanie `NullPointerException`, jeżeli takie przypisanie byłoby legalne. Konstruktory typu wyliczeniowego nie mogą korzystać z pól statycznych typu z wyjątkiem zmiennych stałych (temat 34.). Takie ograniczenie jest niezbędne, ponieważ te pola statyczne nie są zainicjowane w momencie wykonywania

konstruktorów. Specjalnym przypadkiem tego ograniczenia jest to, że stała wyliczeniowa nie może uzyskać dostępu do innej stałej wyliczenia z poziomu swojego konstruktora.

Zwróć uwagę, że metoda `fromString` zwraca `Optional<String>`. Umożliwia to metodzie wskazanie, że tekst, który został do niej przekazany, nie reprezentuje żadnej poprawnej operacji, co zmusza klienta do uwzględnienia takiej możliwości (temat 55.).

Wadą specyficznych dla stałych implementacji metod jest to, że powodują utrudnienie współdzielenia kodu pomiędzy stałymi wyliczeniami. Jako przykład weźmy typ wyliczeniowy reprezentujący dni tygodnia w pakiecie księgowym. Taki typ wyliczeniowy posiada metodę obliczającą wynagrodzenie pracownika za dany dzień na podstawie podstawowego wynagrodzenia (za godzinę) oraz liczby minut przepracowanych danego dnia. W pięciu dniach tygodnia każda minuta przekraczająca zwykłą zmianę powoduje wygenerowanie płacy w nadgodzinach, a w dwóch dniach weekendu cały czas pracy jest traktowany jako nadliczbowy. Przy użyciu instrukcji `switch` można łatwo wykonać takie obliczenia przez zastosowanie wielu etykiet dla dwóch fragmentów kodu.

```
// Typ wyliczeniowy wybierający na podstawie własnych wartości w celu
// współdzielenia kodu - wątpliwe
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    double pay(double minutesWorked, double payRate) {
        double basePay = minutesWorked * payRate;

        double overtimePay;
        switch(this) {
            case SATURDAY: case SUNDAY: // weekend
                overtimePay = basePay / 2;
                break;
            default: // zwykle dni tygodnia
                overtimePay = minutesWorked >= MINS_PER_SHIFT ?
                    0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        return basePay + overtimePay;
    }
}
```

Kod ten jest bez wątpienia spójny, ale z perspektywy jego utrzymania dosyć niebezpieczny. Załóżmy, że dodajemy element do wyliczenia, być może specjalną wartość oznaczającą urlop, ale zapomnimy dodać odpowiedniego przypadku do instrukcji `switch`. Program nadal będzie się kompilował, ale metoda `pay` po cichu zapłaci pracownikowi taką samą kwotę jak za zwykły dzień tygodnia.

Aby bezpiecznie wykonać obliczenie przy użyciu implementacji specyficznej dla stałej, konieczne będzie powielenie obliczenia nadgodzin dla każdej stałej lub przeniesienie obliczeń do dwóch metod pomocniczych (jednej dla weekendów i jednej dla dni roboczych) i wywoływanie odpowiedniej metody pomocniczej z każdej ze stałych. Oba podejścia powodują powstanie sporej ilości kodu narzędziowego, co znacznie zmniejsza czytelność i zwiększa możliwość popełnienia błędu.

Kod narzędziowy może być zmniejszony przez zastąpienie metody abstrakcyjnej `overtimePay` w `PayrollDay` metodą konkretną, w której wykonamy obliczenie nadgodzin dla dni powszednich. W takim przypadku konieczne jest nadpisanie metody wyłącznie dla weekendów. Ma to jednak takie same wady jak instrukcja `switch` — jeżeli dodamy kolejny dzień bez nadpisywania metody `overtimePay`, odziedziczy ona obliczenia dla dni tygodnia bez żadnej informacji o tym fakcie.

Potrzebujemy więc **wymuszenia** wyboru strategii płacy nadgodzin za każdym razem, gdy dodamy stałą wyliczenia. Na szczęście istnieje łatwy sposób osiągnięcia tego efektu. Polega to na przeniesieniu obliczenia nadgodzin do prywatnego zagnieżdżonego typu wyliczeniowego i przekazanie obiektu tego **typu wyliczeniowego strategii** do konstruktora typu wyliczeniowego `PayrollDay`. Typ wyliczeniowy `PayrollDay` deleguje obliczenie płacy w nadgodzinach do typu wyliczeniowego strategii, co eliminuje potrzebę zastosowania polecenia `switch` lub implementacji metod specyficznych dla stałych w `PayrollDay`. Choć ten wzorzec jest mniej zwięzły niż instrukcja `switch`, jest bezpieczniejszy i bardziej elastyczny:

// Wzorzec typu wyliczeniowego strategii

```
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;

    PayrollDay(PayType payType) { this.payType = payType; }
    PayrollDay() { this(PayType.WEEKDAY); } // domyślny

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }
}
```

// typ wyliczeniowy strategii

```
private enum PayType {
    WEEKDAY {
        int overtimePay(int minsWorked, int payRate) {
            return minsWorked <= MINS_PER_SHIFT ? 0 :
                (minsWorked - MINS_PER_SHIFT) * payRate / 2;
        }
    },
    WEEKEND {
        int overtimePay(int minsWorked, int payRate) {
```

```

        return minsWorked * payRate / 2;
    }
};

abstract int overtimePay(int mins, int payRate);
private static final int MINS_PER_SHIFT = 8 * 60;

int pay(int minsWorked, int payRate) {
    int basePay = minsWorked * payRate;
    return basePay + overtimePay(minsWorked, payRate);
}
}
}

```

Jeżeli instrukcje switch w typach wyliczeniowych nie są dobre do implementowania operacji specyficznych dla stałych w typach wyliczeniowych, to **do czego się one nadają? Instrukcje wyboru w typach wyliczeniowych są dobre do wzbogacania zewnętrznych typów wyliczeniowych o operacje specyficzne dla stałych.** Załóżmy, że typ wyliczeniowy Operation jest poza naszą kontrolą i chcemy dodać metodę instancyjną zwracającą odwrotność każdej operacji. Można symulować ten efekt za pomocą następującej metody statycznej:

```

// Wybór ze stałej do symulowania brakującej metody
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS: return Operation.MINUS;
        case MINUS: return Operation.PLUS;
        case TIMES: return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;

        default: throw new AssertionError("Nieznana operacja: " + op);
    }
}

```

Skorzystaj z tej techniki również dla typów wyliczeniowych, które są pod Twoją kontrolą, jeśli metoda po prostu nie należy do typu wyliczeniowego. Metoda może być potrzebna przy niektórych wyliczeniach, ale nie jest na tyle użyteczna, aby rozważyć dołączenie jej do typu wyliczeniowego.

Typy wyliczeniowe są porównywalne pod względem wydajności ze stałymi int. Niewielki spadek wydajności w stosunku do stałych int jest związany z ładowaniem i inicjowaniem typu wyliczeniowego. Poza urządzeniami o ograniczonych zasobach, takich jak telefony komórkowe czy tostery, mało prawdopodobne jest, aby było to w praktyce zauważalne.

Kiedy należy więc korzystać z typów wyliczeniowych? **Za każdym razem, gdy potrzebujemy stałego zestawu stałych znanych w momencie kompilacji.** Oczywiście, obejmuje to „naturalne typy wyliczeniowe”, takie jak planety, dni tygodnia

i figury szachowe. Oprócz tego mogą to być inne zbiory, których wszystkie możliwe wartości są znane w czasie kompilacji, takie jak pozycje menu, kody operacji i znaczniki wiersza poleceń. **Nie jest konieczne, aby zbiór stałych w typie wyliczeniowym był stały przez cały czas.** Mechanizm typów wyliczeniowych został zaprojektowany tak, aby pozwolić na binarną zgodność ewolucji typów wyliczeniowych.

Podsumujmy. Zalety typu wyliczeniowego w stosunku do stałych int są zachęcające. Typ wyliczeniowy jest znacznie bardziej czytelny, bezpieczny i daje większe możliwości. Wiele typów wyliczeniowych nie wymaga jawnego konstruktora ani składników, ale wiele innych korzysta z zalet skojarzenia danych z każdą stałą i zapewnienia metod, których działanie jest zależne od tych danych. Znacznie mniej typów wyliczeniowych korzysta z kojarzenia wielu operacji z jedną metodą. W tych względnie rzadkich przypadkach lepiej użyć metod specyficznych dla stałych, sterowanych własnymi wartościami. Jeżeli wiele stałych wyliczeniowych korzysta z tych samych operacji, warto rozważyć zastosowanie wzorca typu wyliczeniowego strategii.

Temat 35. Użycie pól instancyjnych zamiast kolejności

Wiele typów wyliczeniowych jest naturalnie skojarzonych z wartościami int. Wszystkie takie typy mają metodę ordinal, która zwraca pozycję numeryczną każdej stałej wyliczenia w tym typie. Można ulec pokusie skorzystania z wartości int metody ordinal:

```
// Nadużycie kolejności do uzyskania skojarzonej wartości - NIE RÓB TAK
public enum Ensemble {
    SOLO, DUET, TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET, DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

Choć ten typ wyliczeniowy działa, jest on koszmarem utrzymania. Jeżeli kolejność stałych zostanie zmieniona, metoda numberOfMusicians zawiedzie. Jeżeli chcemy dodać drugą stałą wyliczenia skojarzoną z wartością int, która była już użyta, mamy sporego pecha. Założmy, że chcemy dodać stałą do **podwójnego kwartetu**, który podobnie jak oktet składa się z ośmiu muzyków, ale nie ma sposobu na wykonanie tej operacji.

Dodatkowo nie można dodać stałej dla wartości int bez dodawania stałej dla kolejnych wartości int. Chcemy na przykład dodać stałą reprezentującą **potrójny kwartet**, który składa się z dwunastu muzyków. Nie istnieje standardowy termin

dla zespołu składającego się z jedenastu muzyków, więc jesteśmy zmuszeni do dodania dodatkowej stałej dla nieużywanej wartości `int` (11). Jest to co najmniej nieładne. Jeżeli mamy wiele nieużywanych wartości `int`, jest to niepraktyczne.

Na szczęście istnieje proste rozwiązanie tych problemów. **Nigdy nie należy kojarzyć wartości skojarzonej z wartością wyliczaną na podstawie kolejności; należy zamiast tego przechować ją w polu instancyjnym:**

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

Specyfikacja Enum zawiera następujące objaśnienie dla metody `ordinal`: „Większość programistów nie skorzysta z tej metody. Jest zaprojektowana do użycia przez ogólne struktury danych bazujące na typach wyliczeniowych, takie jak EnumSet i EnumMap”. Jeżeli nie piszemy tego typu struktur danych, najlepiej unikać metody `ordinal`.

Temat 36. Użycie EnumSet zamiast pół bitowych

Jeżeli element typu wyliczeniowego był używany przede wszystkim w zbiorach, zwykle wykorzystywany był wzorzec typu wyliczeniowego `int` (temat 34.), w którym każdej stałej przypisywana była inna potęga 2:

```
// Bitowe stałe typu wyliczeniowego - PRZESTARZALE!
public class Text {
    public static final int STYLE_BOLD = 1 << 0; // 1
    public static final int STYLE_ITALIC = 1 << 1; // 2
    public static final int STYLE_UNDERLINE = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

    // parametr jest bitową sumą zero lub więcej stałych STYLE_
    public void applyStyles(int styles) { ... }
}
```

Ta reprezentacja pozwala użyć bitowej operacji OR do połączenia kilku stałych ze zbioru, co jest znane pod nazwą **poła bitowego**:

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

Reprezentacja pola bitowego pozwala również na efektywne wykonywanie zbioru operacji, takich jak suma lub część wspólna, przy użyciu arytmetyki bitowej. Pola bitowe mają wszystkie wady stałych `int`, jak również swoje własne. Jest nawet trudniej zinterpretować pole bitowe niż zwykłą stałą wyliczeniową `int` po jej wyświetleniu w postaci liczby. Dodatkowo nie istnieje prosty sposób na iterowanie po wszystkich elementach reprezentowanych przez pole bitowe. Co więcej, trzeba przewidzieć maksymalną liczbę bitów, która kiedykolwiek będzie potrzebna, gdy pisze się API, i na tej podstawie dobrać typ pola (`int` lub `long`). Po wybraniu typu nie można przekroczyć jego szerokości (32 lub 64 bity) bez zmiany API.

Niektórzy programiści, korzystający z typów wyliczeniowych zamiast stałych `int`, nadal upierają się przy korzystaniu z pól bitowych, gdy muszą przekazywać zbiory stałych. Nie ma powodu, aby to robić, ponieważ istnieje lepsza alternatywa. W pakiecie `java.util` znajduje się klasa `EnumSet`, która pozwala efektywnie reprezentować zbiór wartości jednego typu wyliczeniowego. Klasa ta implementuje interfejs `Set`, dzięki czemu daje bogactwo, bezpieczeństwo typów i możliwości współpracy zapewniane przez wszystkie inne implementacje `Set`. Jednak wewnętrznie każdy `EnumSet` jest reprezentowany jako wektor bitowy. Jeżeli bazowy typ wyliczeniowy ma nie więcej niż sześćdziesiąt cztery elementy — czyli w większości przypadków — cały `EnumSet` jest reprezentowany za pomocą jednej liczby `long`, więc wydajność jest porównywalna do pól bitowych. Operacje masowe, takie jak `removeAll` i `retainAll`, są implementowane za pomocą arytmetyki bitowej, tak samo jak wykonujemy to ręcznie w przypadku pól bitowych. Jesteśmy jednak izolowani od brzydoty i podatności na błędy ręcznych manipulacji na bitach — `EnumSet` wykonuje za nas całą ciężką pracę.

Poniżej zamieszczony jest poprzedni przykład korzystający z typu wyliczeniowego zamiast pól bitowych. Jest krótszy, czytelniejszy i bezpieczniejszy:

```
// EnumSet - nowoczesny następca pól bitowych
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // może być przekazany dowolny Set, ale EnumSet jest najlepszy
    public void applyStyles(Set<Style> styles) { ... }
}
```

Poniżej mamy kod klienta, który przekazuje obiekt `EnumSet` do metody `applyStyles`. `EnumSet` zapewnia bogaty zestaw statycznych metod fabrycznych dla ułatwienia tworzenia zbioru i jedna z nich jest użyta w tym fragmencie kodu:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Należy zwrócić uwagę, że metoda `applyStyles` oczekuje `Set<Style>`, a nie `EnumSet` $\hookrightarrow$ `<Style>`. Choć wydaje się prawdopodobne, że wszystkie klienty będą przekazywać do metody obiekt `EnumSet`, to jednak dobrą praktyką jest akceptowanie typu interfejsu zamiast typu implementacji (temat 64.). Pozostawia to możliwość przekazania przez klienta w niektórych przypadkach innych implementacji `Set`.

Podsumujmy. **Użycie typu wyliczeniowego w zbiorach nie jest powodem jego reprezentacji przy użyciu pól bitowych.** Klasa EnumSet łączy w sobie spójność i wydajność pól bitowych z wieloma zaletami typów wyliczeniowych opisanych w temacie 34. Jedyną rzeczywistą wadą EnumSet jest to, że nadal w Java 9 niemożliwe jest utworzenie niezmiennego obiektu EnumSet, ale najprawdopodobniej zostanie to poprawione w kolejnej wersji. Na razie można opakować EnumSet za pomocą Collections.unmodifiableSet, ale wtedy zmniejszona zostanie zwężłość i wydajność.

Temat 37. Użycie EnumMap zamiast indeksowania kolejnościami

Czasami można napotkać kod korzystający z indeksowania tablicy lub listy za pomocą metody ordinal (temat 35.). Oto prosty przykład klasy reprezentującej roślinę:

```
class Plant {
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }

    final String name;
    final LifeCycle lifeCycle;

    Plant(String name, LifeCycle lifeCycle) {
        this.name = name;
        this.lifeCycle = lifeCycle;
    }

    @Override public String toString() {
        return name;
    }
}
```

Załóżmy teraz, że mamy tablicę reprezentującą rośliny w ogrodzie i chcemy wyświetlić te rośliny zorganizowane względem typu (roczne, wieloletnie lub dwuletnie). W tym celu utworzymy trzy zbiory, po jednym dla każdego cyklu życia i będziemy iterować po elementach ogrodu, umieszczając każdą roślinę w odpowiednim zbiorze. Niektórzy programiści umieszciliby te zbiory w tablicy indeksowanej wartością ordinal cyklu życia:

```
// Zastosowanie ordinal() do indeksowania tablicy - NIE RÓB TAK!
Set<Plant>[] plantsByLifeCycle =
    (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];
for (int i = 0; i < plantsByLifeCycle.length; i++)
    plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
```

```
// wyświetlenie wyników
for (int i = 0; i < plantsByLifeCycle.length; i++) {
    System.out.printf("%s: %s%n",
        Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

Technika ta działa, ale powoduje wiele problemów. Ponieważ tablice nie są zgodne z typami ogólnymi (temat 28.), program wymaga niekontrolowanego rzutowania, więc nie skompiluje się bez problemu. Ponieważ tablica nie ma żadnych informacji na temat tego, co reprezentuje indeks, konieczne jest ręczne nadawanie etykiet w czasie wyświetlania. Jednak najpoważniejszym problemem jest to, że przy odwoływaniu się do tablicy poprzez indeks będący kolejnością w typie wyliczeniowym spoczywa na nas odpowiedzialność za użycie właściwej wartości `int`; liczby `int` nie zapewniają bezpieczeństwa typów, tak jak typ wyliczeniowy. Jeżeli użyjemy niewłaściwej wartości, program wykona niewłaściwą operację lub — jeżeli będziemy mieli szczęście — zgłosi wyjątek `ArrayIndexOutOfBoundsException`.

Na szczęście istnieje znacznie lepszy sposób na osiągnięcie tego samego efektu. Tablica efektywnie służy jako odwzorowanie pomiędzy typem wyliczeniowym a wartością, więc można w takim przypadku zastosować `Map`. Dokładniej — dostępna jest bardzo szybka implementacja `Map`, zaprojektowana do wykorzystywania z kluczami typu wyliczeniowego, o nazwie `java.util.EnumMap`. Poprzedni program korzystający z `EnumMap` wygląda następująco:

```
// Zastosowanie EnumMap do skojarzenia typu wyliczeniowego z danymi
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class);
for (Plant.LifeCycle lc : Plant.LifeCycle.values())
    plantsByLifeCycle.put(lc, new HashSet<>());
for (Plant p : garden)
    plantsByLifeCycle.get(p.lifeCycle).add(p);
System.out.println(plantsByLifeCycle);
```

Program ten jest krótszy, czytelniejszy i bezpieczniejszy, a dodatkowo ma wydajność zbliżoną do pierwszej wersji. Nie ma tu niebezpiecznego rzutowania; nie ma potrzeby ręcznego wyświetlania etykiet, ponieważ klucze odwzorowania są typu wyliczeniowego, który może się przekształcić na czytelną postać, a dodatkowo nie ma możliwości popełnienia błędu przy obliczaniu indeksów tablicy. Obiekty `EnumMap` mają porównywalną wydajność z tablicami indeksowanymi za pomocą kolejności, ponieważ korzystają one wewnętrznie właśnie z tablic. Jednak ukrywają szczegóły implementacji przed programistą, łącząc bogactwo i bezpieczeństwo typów `Map` z wydajnością tablicy. Należy zwrócić uwagę, że konstruktor `EnumMap` oczekuje jako klucza obiektu `Class` — jest to **token typu związanego**, który zapewnia dostęp do typu ogólnego w czasie działania (temat 33.).

Przedstawiony powyżej program można skrócić jeszcze bardziej, wykorzystując strumienie (temat 45.) do zarządzania odwzorowaniem. Oto najprostszy kod bazujący na strumieniach, który w dużej mierze powiela działanie wcześniejszego przykładu:


```
// Uprozczone podejście wykorzystujące strumienie - jest wątpliwe,
// czy doprowadzi do utworzenia EnumMap!
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.lifeCycle)));
```

Problemem w tym kodzie jest to, że wybierze on własną implementację odwzorowania, czyli w praktyce inną klasę niż EnumMap, a tym samym nie zapewni wydajności i zwężłości pamięciowej rozwiązania stosującego EnumMap w sposób jawny. Aby rozwiązać ten problem, użyjmy trójparametrowej wersji Collectors.groupingBy, która pozwala kodowi wywołującemu wskazać implementację odwzorowania w parametrze mapFactory:

```
// Wykorzystanie strumienia i EnumMap do powiązania danych z wyliczeniem
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.lifeCycle,
        () -> new EnumMap<>(LifeCycle.class), toSet())));
```

Tego rodzaju optymalizacja nie jest warta dodatkowego nakładu pracy w tak prostym przykładzie jak prezentowany, ale może mieć duże znaczenie w programie wykorzystującym odwzorowania w sposób niezwykle intensywny.

Zachowanie wersji bazującej na strumieniu różni się nieco od wersji z EnumMap. Wersja EnumMap zawsze tworzy zagnieżdżone odwzorowanie dla każdego cyklu życia rośliny, a wersja strumieniowa tworzy zagnieżdżone odwzorowanie tylko wtedy, gdy ogród zawiera jedną roślinę lub więcej roślin o danym cyklu życia. Jeśli więc ogród zawiera rośliny jednoroczne i wieloletnie, ale nie dwuletnie, rozmiar plantsByLifeCycle będzie w wersji EnumMap wynosił trzy, a w obu wersjach strumieniowych dwa.

Można się spotkać również z tablicami tablic, indeksowanymi (dwukrotnie!) kolejnościami w typie, reprezentującymi odwzorowanie pomiędzy dwoma wartościami wyliczeniowymi. Na przykład poniższy program korzysta z takich tablic do reprezentowania przejść pomiędzy dwoma stanami (z ciekłego na stały to zamarzanie, z ciekłego do gazowego to parowanie i tak dalej).

```
// Zastosowanie ordinal() do indeksowania tablicy tablic - NIE RÓB TAK!
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

        // wiersze indeksowane kolejnością źródłową, kolumny docelową
        private static final Transition[][] TRANSITIONS = {
            { null, MELT, SUBLIME },
            { FREEZE, null, BOIL },
            { DEPOSIT, CONDENSE, null }
        };
    }
};
```

```

//zwraca przejście fazowe z jednego stanu do drugiego
public static Transition from(Phase from, Phase to) {
    return TRANSITIONS[from.ordinal()][to.ordinal()];
}
}
}

```

Program działa i może nawet wydawać się elegancki, ale wrażenie to jest złudne. Podobnie jak w przypadku wcześniejszego przykładu z ogrodem zielnym, kompilator nie ma możliwości poznać relacji pomiędzy kolejnością w typie i indeksami tablicy. Jeżeli popełnimy błąd w tablicy przejść lub zapomnimy zaktualizować ją przy modyfikacji typu wyliczeniowego Phase lub Phase.Transition, program ulegnie awarii w czasie działania. Awaria może przyjąć postać wyjątku `ArrayIndexOutOfBoundsException`, `NullPointerException` lub (co gorsza) nieprawidłowego działania. Dodatkowo wielkość tablicy jest kwadratem tablicy stanów, nawet jeżeli liczba niepustych wpisów jest mała.

I w tym przypadku lepiej zastosować `EnumMap`. Ponieważ każde przejście fazowe jest indeksowane parą stałych stanu, najlepiej reprezentować tę relację jako odwzorowanie z jednego typu wyliczeniowego (stan początkowy) na odwzorowanie z drugiego typu wyliczeniowego (stan końcowy) i na wynik (przejście fazowe). Dwa stany skupienia skojarzone z przejściem fazowym najlepiej modelować przez dołączenie danych do typu wyliczeniowego przejść fazowych, które są następnie wykorzystywane do inicjowania zagnieżdżonych obiektów `EnumMap`:

```

//Zastosowanie zagnieżdżonych EnumMap do skojarzenia danych z parą
//wartości wyliczeniowych
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

        private final Phase from;
        private final Phase to;

        Transition(Phase from, Phase to) {
            this.from = from;
            this.to = to;
        }
    }

    //inicjalizacja odwzorowania przejść między stanami
    private static final Map<Phase, Map<Phase, Transition>>
        m = Stream.of(values()).collect(groupingBy(t -> t.from,
            () -> new EnumMap<>(Phase.class),
            toMap(t -> t.to, t -> t,
                (x, y) -> y, () -> new EnumMap<>(Phase.class))));

```

```

        public static Transition from(Phase from, Phase to) {
            return m.get(from).get(to);
        }
    }
}

```

Kod inicjujący odwzorowanie przejść fazowych może wydawać się nieco skomplikowany, ale nie jest taki zły. Typem odwzorowania jest `Map<Phase, Map<Phase, Transition>>`, co oznacza „odwzorowanie ze stanu (źródłowego) na odwzorowanie ze stanu (docelowego) na przejście”. To odwzorowanie odwzorowań jest inicjalizowane kaskadową wersją dwóch kolektorów. Pierwszy kolektor grupuje przejścia na podstawie stanu początkowego, a drugi tworzy EnumMap z odwzorowaniami ze stanu docelowego do przejścia. Funkcja łącząca w drugim kolektorze ((x, y) -> y) nie jest używana; potrzebujemy jej tylko dlatego, że musimy wskazać fabrykę obiektów map w celu uzyskania EnumMap, a Collectors oferuje fabryki teleskopowe. Poprzednie wydanie książki stosowało jawną iterację w celu inicjalizacji odwzorowań. Tamten kod był dłuższy, ale łatwiejszy do zrozumienia.

Teraz założmy, że chcemy dodać do systemu nowy stan — **plazmę**, czyli zjonizowany gaz. Z tym stanem są związane tylko dwa przejścia — **jonizacja**, która powoduje przejście gazu w plazmę, oraz **dejonizacja**, czyli przejście plazmy w gaz. Aby zaktualizować program bazujący na tablicach, należy dodać nową stałą do Phase oraz dwie do Phase.Transition i zamienić oryginalną dziewięcioelementową tablicę tablic na nową, szesnastoelementową. Jeżeli dodamy zbyt dużo lub za mało elementów do tablicy albo umieścimy element w niewłaściwym miejscu, wszystko pójdzie źle — program się skompiluje, ale ulegnie awarii w czasie działania. Aby zaktualizować wersję bazującą na EnumMap, wystarczy dodać PLASMA do listy stanów oraz IONIZE(GAS, PLASMA) i DEIONIZE(PLASMA, GAS) do listy przejść fazowych.

```

// Dodanie nowej fazy za pomocą zagnieżdżonej implementacji EnumMap
public enum Phase {
    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
        ... // pozostały kod pozostaje bez zmian
    }
}

```

Program zajmie się resztą, a my niemal nie będziemy mieli możliwości popełnienia błędu. Wewnętrznie odwzorowanie odwzorowań jest implementowane jako tablica tablic, więc nie poświęcimy zbyt wiele przestrzeni ani narzutu czasowego w zamian za czytelność, bezpieczeństwo i łatwość utrzymania.

Warto w tym miejscu zastanowić się nad rozwiązaniem z wcześniejszych przykładów, w których używano `null` do wskazania braku zmiany stanu (czyli wartości `from` i `to` były identyczne). Nie jest to dobra praktyka, bo może doprowadzić do wystąpienia wyjątku `NullPointerException` w trakcie działania programu. Zaprojektowanie czystego, eleganckiego rozwiązania dla analizowanego problemu jest zadziwiająco trudne. Wynikowe programy byłyby dosyć długie, co odciągałoby uwagę od głównego celu tego tematu.

Podsumujmy. **Korzystanie z wartości `ordinal` jako indeksów tablicy jest rzadko właściwe — zamiast nich należy używać `EnumMap`.** Jeżeli reprezentowana relacja jest wielowymiarowa, należy skorzystać z `EnumMap<..., EnumMap<...>>`. Jest to specjalny przypadek ogólnej zasady, że programiści aplikacji powinni bardzo rzadko, o ile nie wcale, korzystać z `Enum.ordinal` (temat 35.).

Temat 38. Emulowanie rozszerzalnych typów wyliczeniowych za pomocą interfejsów

Typy wyliczeniowe są lepsze niemal we wszystkich aspektach od wzorca bezpiecznego dla typów typu wyliczeniowego opisywanego w pierwszym wydaniu tej książki [Bloch01]. Jednak jednym problemem może być rozszerzalność, która jest możliwa przy zastosowaniu wzorca, ale nie jest wspierana przez konstrukcje języka. Inaczej mówiąc, przy użyciu wzorca można mieć jeden typ wyliczeniowy będący rozszerzeniem innego, natomiast przy użyciu konstrukcji języka — nie. Nie jest to przypadek. W większości przypadków rozszerzalność typów wyliczeniowych okazuje się bardzo złym pomysłem. Mylące jest, że elementy typu rozszerzonego są obiektami typu bazowego i że nie zachodzi relacja odwrotna. Nie istnieje dobry sposób wyliczenia wszystkich elementów typu bazowego i jego rozszerzeń. Na koniec — rozszerzalność może skomplikować wiele aspektów projektu i implementacji.

Trzeba jednak pamiętać, że istnieje co najmniej jeden kuszący przypadek użycia rozszerzalnych typów wyliczeniowych — **kody operacji**. Kody operacji są typem wyliczeniowym, którego elementy reprezentują operacje w pewnej maszynie, którego przykładem jest typ `Operation` z tematu 34., reprezentujący funkcje prostego kalkulatora. Czasami pożądanym jest umożliwienie użytkownikom API dodawanie własnych operacji, co jest rozszerzaniem zbioru operacji udostępnianych przez API.

Na szczęście istnieje łatwy sposób osiągnięcia tego efektu z użyciem typów wyliczeniowych. Podstawą tego pomysłu jest wykorzystanie możliwości implementowania przez typy wyliczeniowe dowolnych interfejsów przez zdefiniowanie

interfejsu dla typu kodu operacji oraz typu wyliczeniowego będącego standardową implementacją interfejsu. Poniżej zamieszczona jest rozszerzalna wersja typu Operation z tematu 34.

```
// Emulowanie rozszerzalnego typu wyliczeniowego z użyciem interfejsu
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;

    BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

Choć typ wyliczeniowy (BasicOperation) nie jest rozszerzalny, typ interfejsu (Operation) jest, a ten typ interfejsu jest używany do reprezentowania operacji w API. Możemy zdefiniować kolejny typ wyliczeniowy, który implementuje ten interfejs, i użyć obiektów tego nowego typu w miejsce typu bazowego. Dla przykładu zdefiniujemy rozszerzenie typu operacji przedstawionego powyżej, składające się z operacji potęgowania i reszty z dzielenia. Wystarczy napisać typ wyliczeniowy implementujący interfejs Operation:

```
// Emulowanie rozszerzenia typu wyliczeniowego
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
```

```

        return x % y;
    }
};

private final String symbol;

ExtendedOperation(String symbol) {
    this.symbol = symbol;
}

@Override public String toString() {
    return symbol;
}
}

```

Można użyć naszych nowych operacji w każdym miejscu, gdzie korzystaliśmy z operacji podstawowych, pod warunkiem że API korzysta z typu interfejsu (`Operation`), a nie implementacji (`BasicOperation`). Warto zauważyć, że nie musimy definiować abstrakcyjnej metody `apply` w typie wyliczeniowym, tak jak musieliśmy to zrobić w nierozszerzalnym typie wyliczeniowym z implementacjami metod specyficznymi dla instancji (temat 34.). Możemy tak zrobić, ponieważ metoda abstrakcyjna (`apply`) jest składnikiem interfejsu (`Operation`).

Nie tylko możliwe jest przekazanie jednej instancji „rozszerzonego wyliczenia” w każdym miejscu, gdzie spodziewane jest „bazowe wyliczenie”, ale również możliwe jest przekazanie całego rozszerzonego typu wyliczeniowego i użycie jego elementów oprócz lub zamiast tych z typu bazowego. Poniżej znajduje się na przykład program testowy z tematu 34., który korzysta z wszystkich zdefiniowanych wcześniej rozszerzonych operacji:

```

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(
    Class<T> opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
            x, op, y, op.apply(x, y));
}

```

Należy zwrócić uwagę, że literal klasy dla rozszerzonego typu operacji (`ExtendedOperation.class`) jest przekazywany z `main` do `test`, co pozwala na opisanie zbioru operacji rozszerzonych. Ten literal klasy służy jako **token typu związanego** (temat 33.). Skomplikowana deklaracja parametru `opEnumType` (`<T extends Enum<T> & Operation> Class<T>`) zapewnia, że obiekt `Class` reprezentuje zarówno typ wyliczeniowy, jak i podtypy `Operation`, co jest wymagane do iterowania po elementach i wykonywania operacji związanych z każdą stałą.

Drugą możliwością jest przekazanie `Collection<? extends Operation>`, będącego **związanyam szablonem typu** (temat 31.), zamiast obiektu klasy:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
    double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
            x, op, y, op.apply(x, y));
}
```

Wynikowy kod jest nieco mniej skomplikowany, a metoda `test` jest bardziej elastyczna — pozwala to wywołującemu na połączenie operacji z wielu typów implementacji. Z drugiej strony uniemożliwiamy sobie użycie `EnumSet` (temat 36.) oraz `EnumMap` (temat 37.) dla wymienionych operacji.

Oba przedstawione powyżej programy zwrócą ten sam, zamieszczony poniżej wynik, gdy zostaną uruchomione z argumentami 2 i 4:

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

Niewielką wadą użycia interfejsu do emulowania rozszerzalnych typów wyliczeniowych jest to, że implementacje nie mogą dziedziczyć po jednym typie wyliczeniowym w innym. Jeśli kod implementacji nie zależy od żadnego stanu, można go umieścić w interfejsie, wykorzystując przy tym metody domyślne (temat 20.). W przypadku naszego przykładu typu `Operation` logika przechowywania i odczytywania symboli związanych z operacjami jest powielona w `BasicOperation` oraz `ExtendedOperation`. W tym przypadku nie ma to znaczenia, ponieważ powielone jest niewiele kodu. Jeżeli znajdowałoby się tam więcej wspólnych funkcji, w celu wyeliminowania powtarzania kodu konieczne byłoby umieszczenie ich w klasie pomocniczej lub statycznych metodach pomocniczych.

Opisany w tym miejscu wzorzec jest wykorzystywany w bibliotekach Javy. Przykładowo typ wyliczeniowy `java.nio.file.LinkOption` implementuje interfejsy `CopyOption` i `OpenOption`.

Podsumujmy. **Choć nie można pisać rozszerzalnych typów wyliczeniowych, można emulować je przez napisanie interfejsu dla bazowego typu wyliczeniowego i implementowanie tego interfejsu.** Pozwala to klientom na pisanie własnych typów wyliczeniowych implementujących ten interfejs. Te typy wyliczeniowe mogą być używane wszędzie tam, gdzie może być wykorzystywany bazowy typ wyliczeniowy, przy założeniu że API wykorzystuje typ interfejsu.

Temat 39. Korzystanie z adnotacji zamiast wzorców nazw

Dawniej standardowym działaniem było korzystanie z **wzorców nazw** w celu wskazania, że pewne elementy programu wymagają specjalnego traktowania przez narzędzia lub framework. Na przykład przed wersją 4. framework testowania JUnit wymagał od użytkowników wyznaczenia metod testowych przez rozpoczęcie ich nazwy od `test` [Beck04]. Technika ta działa, ale ma kilka dużych wad. Po pierwsze, błędy typograficzne mogą powodować niewykryte awarie. Na przykład możemy przypadkowo nazwać metodę testującą `tsetSafetyOverride` zamiast `testSafetyOverride`. JUnit 3 nie wykryje błędu, ale nie wykona tych testów, dając fałszywe poczucie bezpieczeństwa.

Drugą wadą wzorca nazw jest brak możliwości upewnienia się, że są użyte we właściwych elementach programu. Załóżmy, że nazwaliśmy klasę `testSafetyMechanisms` w nadziei, że JUnit 3 przetestuje automatycznie jej wszystkie metody, niezależnie od ich nazw. I w takim przypadku JUnit 3 nie będzie nic sygnalizował, ale nie wykona żadnych testów.

Trzecią wadą wzorców nazw jest brak dobrej metody łączenia wartości parametrów z elementami programu. Na przykład możemy chcieć obsługiwać kategorię testów, które są wykonane prawidłowo, jeżeli zgłoszą określony wyjątek. Typ wyjątku jest faktycznie parametrem testu. Moglibyśmy zakodować nazwę wyjątku w nazwie metody testowej z użyciem rozbudowanego wzorca nazwy, ale będzie to brzydkie i niepewne rozwiązanie (temat 62.). Kompilator nie będzie miał żadnej możliwości sprawdzenia, czy napis, który definiuje nazwę wyjątku jest prawidłowy. Jeżeli nazwana klasa nie istnieje lub nie jest wyjątkiem, nie dowiemy się o tym do momentu uruchomienia testu.

Adnotacje [JLS, 9.7] rozwiązują te problemy w elegancki sposób. Framework JUnit umożliwił korzystanie z nich od wersji 4. Załóżmy, że chcemy zdefiniować typ adnotacji do oznaczania prostych testów uruchamianych automatycznie, które nie udają się, jeżeli zgłoszą wyjątek. Poniżej pokazane jest, jak może wyglądać taki typ adnotacji o nazwie `Test`.

```
// Deklaracja znacznikowego typu adnotacji
import java.lang.annotation.*;

/**
 * Wskazuje, że adnotowana metoda jest metodą testującą.
 * Używać tylko dla bezparametrowych metod statycznych.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```


Deklaracja dla typu adnotacyjnego `Test` posiada również adnotacje `Retention` oraz `Target`. Takie adnotacje dla typu adnotacyjnego są również nazywane **metaadnotacjami**. Metaadnotacja `@Retention(RetentionPolicy.RUNTIME)` wskazuje, że adnotacja `Test` powinna być utrzymywana w czasie działania. Bez tego adnotacja `Test` nie byłaby widoczna dla narzędzia testującego. Metaadnotacja `@Target(ElementType.METHOD)` wskazuje, że adnotacja `Test` jest legalna tylko dla deklaracji metod — nie może być stosowana dla deklaracji klas, pól lub innych elementów programu.

Trzeba zauważyć, że przed deklaracją adnotacji `Test` znajduje się komentarz: „Używać tylko dla bezparametrowych metod statycznych”. Lepiej byłoby, gdyby kompilator pozwalał wymusić takie ograniczenie, ale tego nie robi, chyba że napiszemy **procesor adnotacji**. Aby dowiedzieć się więcej na ten temat, odwiedź stronę dokumentacji dotyczącą `javax.annotation.processing`. W przypadku braku procesora adnotacji, jeżeli umieścimy adnotację `Test` w metodzie instancyjnej lub metodzie mającej jeden lub więcej parametrów, program testujący nadal będzie się kompilował, ale narzędzie testujące będzie musiało poradzić sobie z problemem w czasie działania.

Przedstawimy teraz, jak adnotacja `Test` działa w praktyce. Jest ona nazywana adnotacją znacznikową, ponieważ nie posiada parametrów i tylko „oznacza” adnotowany element. Jeżeli programista źle wpisze `Test` lub zastosuje adnotację `Test` do innego elementu programu niż deklaracja metody, program się nie skompiluje:

```
// Program zawierający adnotacje znacznikowe
public class Sample {
    @Test public static void m1() { } // test powinien się udać
    public static void m2() { }
    @Test public static void m3() { // test powinien zawieść
        throw new RuntimeException("Bum!");
    }
    public static void m4() { }
    @Test public void m5() { } // NIEWŁAŚCIWE UŻYCIE:
                          // metoda niestaticzna
    public static void m6() { }
    @Test public static void m7() { // test powinien zawieść
        throw new RuntimeException("Awaria");
    }
    public static void m8() { }
}
```

Klasa `Sample` posiada osiem metod, z których cztery są adnotowane jako testowe. Dwie z nich, `m3` i `m7`, zgłaszają wyjątki, a dwie, `m1` i `m5` — nie. Jednak jedna z adnotowanych metod, które nie zgłaszają wyjątku, `m5`, jest metodą instancyjną, więc nie jest to prawidłowe użycie adnotacji. W sumie `Sample` zawiera cztery testy — jeden wykona się prawidłowo, dwa nieprawidłowo i jeden jest niewłaściwy. Cztery metody nieposiadające adnotacji `Test` zostaną zignorowane przez narzędzie testujące.

Adnotacja `Test` nie ma bezpośredniego wpływu na klasę `Sample`. Służy ona wyłącznie w celu dostarczania informacji dla „zainteresowanych” programów. Mówiąc bardziej ogólnie, adnotacje nigdy nie zmieniają semantyki adnotowanego kodu, ale pozwalają na jego specjalne traktowanie przez narzędzia, takie jak prosty program do uruchamiania testów:

```
// Program przetwarzający adnotacje znacznikowe
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " nieudany: " + exc);
                } catch (Exception exc) {
                    System.out.println("Niewłaściwy @Test: " + m);
                }
            }
        }
        System.out.printf("Udanych: %d, Nieudanych: %d\n",
            passed, tests - passed);
    }
}
```

Narzędzie do uruchamiania testów pobiera w pełni kwalifikowaną nazwę klasy z wiersza polecenia i uruchamia wszystkie metody adnotowane przez `Test` z użyciem refleksji przez wywołanie `Method.invoke`. Metoda `isAnnotationPresent` informuje program, którą metodę należy uruchomić. Jeżeli metoda testowa zgłosi wyjątek, mechanizmy refleksji opakowują go w `InvocationTargetException`. Narzędzie przechwytuje wyjątek i wyświetla raport o niewłaściwym wykonaniu, zawierający oryginalny wyjątek zgłoszony przez metodę testową, wyodrębniony z `InvocationTargetException` za pomocą metody `getCause`.

Jeżeli próba wywołania metody testowej za pomocą refleksji zgłosi wyjątek inny niż `InvocationTargetException`, oznacza to niewłaściwe zastosowanie adnotacji `Test`, które nie zostało wychwycone w czasie kompilacji. Takie użycia obejmują adnotację metody instancyjnej, metody z co najmniej jednym parametrem lub niedostępnej metody. Drugi blok `catch` w programie uruchamiającym testy przechwytuje błędy użycia i wyświetla odpowiedni komunikat o błędzie. Poniżej przedstawiony jest wynik wykonania programu `RunTests` z parametrem `Sample`:

```
public static void Sample.m3() nieudany: RuntimeException: Bum!
Niewłaściwy @Test: public void Sample.m5()
public static void Sample.m7() nieudany: RuntimeException: Awaria
Udanych: 1, Nieudanych: 3
```

Dodajmy teraz obsługę testów, które udają się, jeżeli zgłoszą określony wyjątek. Będziemy potrzebować dla nich nowej adnotacji:

```
// Typ adnotacji z parametrem
import java.lang.annotation.*;
/**
 * Wskazuje, że adnotowana metoda jest metodą testową,
 * która w razie powodzenia musi zgłosić wyznaczony wyjątek.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

Typem tego parametru dla adnotacji jest `Class<? extends Exception>`. Ten typ szablonowy jest oczywiście bardzo ogólny. Mówiąc po ludzku, oznacza on „obiekt `Class` pewnej klasy dziedziczącej po `Exception`”, co pozwala użytkownikowi adnotacji określić dowolny typ wyjątku. Jest to przykład zastosowania **tokena typu związanego** (temat 33.). Jak to działa w praktyce? Warto zwrócić uwagę, że literał klasy jest użyty jako wartość parametru adnotacji:

```
// Program zawierający adnotacje z parametrem
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() { // test powinien się powieść
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() { // test owinien zawieść (zły wyjątek)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { } // test powinien zawieść (brak wyjątku)
}
```

Zmodyfikujemy teraz nasz program do uruchamiania testów, aby przetwarzał nowe adnotacje. Wymaga to dodania do metody `main` następującego kodu:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s nieudany: brak wyjątku%n", m);
    }
```

```

    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Exception> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s nieudany: oczekiwano %s, otrzymano %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("Niewłaściwy @Test: " + m);
    }
}

```

Kod ten jest podobny do kodu używanego przy przetwarzaniu adnotacji `Test`, z jednym wyjątkiem — pobiera on wartość parametru adnotacji i korzysta z niej do sprawdzenia, czy zgłoszony wyjątek jest odpowiedniego typu. Nie ma tu jawnego rzutowania, dzięki czemu nie ma niebezpieczeństwa zgłoszenia `ClassCastException`. To, że program testowy kompiluje się, gwarantuje, że jego parametry adnotacji reprezentują prawidłowe typy wyjątków, poza jednym przypadkiem — możliwe jest, że parametry adnotacji są prawidłowe w czasie kompilacji, ale plik klasy reprezentującej typ wyjątku nie jest dostępny w czasie działania. W tym dosyć rzadkim przypadku program zgłosi wyjątek `TypeNotPresentException`.

Rozszerzając nasz przykład testowania wyjątków, możliwe jest, że test będzie sprawdzał, czy metoda zgłasza jeden z kilku wymienionych wyjątków. Mechanizm adnotacji posiada mechanizmy ułatwiające obsługę takich przypadków. Załóżmy, że zmienimy typ parametru adnotacji `ExceptionTest` na tablicę obiektów `Class`:

```

// Typ adnotacji z parametrem tablicowym
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}

```

Składnia parametrów tablicowych w adnotacjach jest elastyczna. Jest zoptymalizowana dla tablic jednoelementowych. Wszystkie poprzednie adnotacje `Exception` ➔ `Test` są nadal prawidłowe dla nowej wersji z parametrem tablicowym i będą dawać w wyniku tablice jednoelementowe. Aby przekazać tablicę wieloelementową, należy ująć elementy w nawiasy klamrowe i rozdzielić je przecinkami:

```

// Kod zawierający adnotację z parametrem tablicowym
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<>();
}

```

```

// specyfikacja pozwala tej metodzie zgłosić
// IndexOutOfBoundsException lub NullPointerException
list.addAll(5, null);
}

```

Dosyć łatwo jest zmienić nasz program testujący, aby przetwarzać nową wersję `ExceptionTest`. Poniższy kod zastępuje oryginalną wersję:

```

if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s nieudany: brak wyjątku%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s nieudany: %s %n", m, exc);
    }
}

```

Wraz z Javą 8 pojawił się nowy sposób tworzenia adnotacji wielowartościowych. Zamiast deklarować typ adnotacyjny z parametrem w postaci tablicy, można oznaczyć deklarację adnotacji metaadnotacją `@Repeatable`, która oznacza, że do jednego elementu można tę samą adnotację stosować wielokrotnie. Ta metaadnotacja przyjmuje jeden parametr, który jest obiektem klasy **typu adnotacyjnego zawierającego**, którego jedynym parametrem jest tablica typu adnotacyjnego [JLS, 9.6.3]. Poniżej przedstawiamy, jak wyglądałaby adnotacja, gdybyśmy kontynuowali przykład z adnotacją `ExceptionTest`. Zwróć uwagę, że typ adnotacji zawierającej musi zostać oznaczony odpowiednią regułą retencji i celu, bo w przeciwnym razie deklaracji nie uda się skompilować:

```

// Typ adnotacji pozwalający na powtarzalność adnotacji
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

```

```
public @interface ExceptionTestContainer {
    ExceptionTest[] value();
}
```

Oto jak wyglądałby test doublyBad wykorzystujący adnotację powtarzalną zamiast adnotacji przyjmującej tablicę:

```
// Kod wykorzystujący adnotację powtarzalną
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

Przetwarzanie powtarzalnych adnotacji wymaga uwagi. Adnotacja powtarzalna generuje sztuczny zawierający typ adnotacyjny. Metoda `getAnnotationsByType` jest tego świadoma i może być stosowana do dostępu zarówno do powtórzonych, jak i niepowtórzonych adnotacji typu powtarzalnego. Z drugiej strony `isAnnotationPresent` jawnie wskazuje, że adnotacja powtarzalna nie jest typu adnotacyjnego, ale zawierającego typu adnotacyjnego. Jeśli element posiada adnotację powtarzalną pewnego typu i użyje się metody `isAnnotationPresent` do sprawdzenia, czy element posiada adnotację tego typu, okaże się, że tak nie jest. Użycie tej metody do sprawdzania istnienia typu adnotacyjnego spowoduje, że program będzie po cichu ignorował adnotacje powtarzalne. Z drugiej strony, jeśli użyjemy tej metody do sprawdzania zawierającego typu adnotacyjnego, program będzie po cichu ignorował adnotacje nieobsługujące powtarzania. Aby za pomocą `isAnnotationPresent` wykrywać wersje z powtarzaniem i bez niego, trzeba sprawdzać zarówno typ adnotacji jak i typ adnotacji zawierającej. Oto jak wyglądałby fragment programu `RunTests` po wprowadzeniu poprawek pozwalających na obsługę powtarzanej wersji adnotacji `ExceptionTest`:

```
// Przetwarzanie adnotacji powtarzalnych
if (m.isAnnotationPresent(ExceptionTest.class)
    || m.isAnnotationPresent(ExceptionTestContainer.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s nieudany: brak wyjątku", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        ExceptionTest[] excTests =
            m.getAnnotationsByType(ExceptionTest.class);
        for (ExceptionTest excTest : excTests) {
            if (excTest.value().isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s nieudany: %s %n", m, exc);
    }
}
```

Adnotacje powtarzalne wprowadzono, aby poprawić czytelność kodu źródłowego, który w sposób logiczny stosuje wiele instancji tego samego typu adnotacyjnego dla danego elementu programu. Jeśli uważasz, że faktycznie będzie to poprawiało czytelność kodu źródłowego, korzystaj z nich, ale pamiętaj, że deklaracja i przetwarzanie adnotacji powtarzalnych wymaga więcej kodu, a przetwarzanie jest też narażone na błędy.

Framework testujący opracowany w tym temacie jest tylko zabawką, ale jasno demonstrować przewagę adnotacji nad wzorcami nazw. Pokazuje on tylko czubek góry lodowej możliwości, jakie dają adnotacje. Jeżeli napiszemy narzędzie wymagające od programisty dodania informacji do plików źródłowych, wystarczy zdefiniować zbiór odpowiednich adnotacji. **Nie istnieje dobry powód używania wzorców nazw, gdy mamy do dyspozycji adnotacje.**

Jednak oprócz twórców narzędzi większość programistów nie będzie musiała definiować typów adnotacyjnych. **Wszyscy programiści powinni jednak używać predefiniowanych typów adnotacji oferowanych przez platformę Java** (tematy 40. oraz 27.). Dodatkowo należy rozważyć użycie adnotacji definiowanych przez używane IDE lub narzędzia analizy statycznej. Takie adnotacje pozwalają poprawić jakość informacji diagnostycznych udostępnianych przez te narzędzia. Należy jednak pamiętać, że te adnotacje nie zostały ustandaryzowane, więc być może będzie potrzebna włożyć nieco pracy przy zmianie narzędzi lub po powstaniu standardu.

Temat 40. Spójne użycie adnotacji Override

Biblioteki Javy zawierają kilka typów adnotacji. Dla typowego programisty najważniejszy jest `@Override`. Adnotacja ta może być używana tylko w deklaracjach metod i wskazuje, że adnotowana deklaracja metody przesłania deklarację w typie bazowym. Jeżeli będziemy spójnie korzystać z tej adnotacji, zabezpieczymy się przed dużą grupą nieprzyjemnych błędów. Przeanalizujemy poniższy program, w którym klasa `Bigram` reprezentuje **bigram**, czyli uporządkowaną parę liter:

```
// Czy możesz wskazać błąd?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
}
```

```

    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}

```

Główny program w pętli dodaje do zbioru dwadzieścia sześć bigramów, z których każdy składa się z dwóch identycznych małych liter. Następnie wyświetlana jest wielkość zbioru. Można oczekiwać, że program wyświetli 26, ponieważ zbiory nie mogą zawierać duplikatów. Jeżeli spróbujemy uruchomić ten program, okaże się, że wyświetla nie 26, ale 260. Gdzie tkwi błąd?

Jasne jest, że autor klasy `Bigram` miał zamiar nadpisać metodę `equals` (temat 10.) i pamiętał nawet o nadpisaniu metody `hashCode` (temat 11.). Niestety nasz nieszczyśny programista nie nadpisał `equals`, ale ją przeciążył (temat 52.). Aby nadpisać `Object.equals`, należy zdefiniować metodę `equals`, której parametrem jest `Object`, ale parametrem metody `equals` w klasie `Bigram` nie jest `Object`, więc `Bigram` dziedziczy metodę `equals` z `Object`. Ta metoda `equals` testuje **identyczność** obiektów, podobnie jak operator `==`. Każda z dziesięciu kopii każdego bigramu jest inna od pozostałych dziewięciu, więc dla metody `Object.equals` są różne, co wyjaśnia, dlaczego program wyświetlił 260.

Na szczęście kompilator może pomóc nam znaleźć ten błąd, ale tylko wtedy, gdy nieco mu w tym pomożemy przez wskazanie, że nadpisujemy `Object.equals`. W tym celu adnotujemy `Bigram.equals` za pomocą `@Override`, tak jak poniżej:

```

@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}

```

Jeżeli wstawimy tę adnotację i spróbujemy skompilować program, kompilator wygeneruje komunikat o błędzie, taki jak ten:

```

Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^

```

Można natychmiast zorientować się, co zrobiliśmy źle, klepnąć się w czoło i zamienić błędną implementację `equals` na prawidłową (temat 10.):


```
@Override public boolean equals(Object o) {  
    if (!(o instanceof Bigram))  
        return false;  
    Bigram b = (Bigram) o;  
    return b.first == first && b.second == second;  
}
```

Z tego powodu **powinno się używać adnotacji Override przy każdej deklaracji metody, która ma nadpisywać deklarację w klasie bazowej**. Istnieje jeden niewielki wyjątek od tej reguły. Jeżeli piszemy klasę, która nie jest oznaczona jako abstrakcyjna, i uważamy, że będzie nadpisywać metodę abstrakcyjną, nie trzeba dodawać adnotacji Override dla tej metody. W klasie, która nie jest zadeklarowana jako abstrakcyjna, kompilator będzie generował komunikat o błędzie, jeżeli nie nadpiszemy abstrakcyjnej metody z klasy bazowej. Jednak można również skupić uwagę na wszystkich metodach swojej klasy, które nadpisują metody bazowe, i w takim przypadku można bez żadnych problemów dodać do nich adnotacje. Większość środowisk IDE można skonfigurować tak, aby wstawiały adnotacje @Override automatycznie, gdy decydujemy się przesłonić metodę.

Nowoczesne środowiska IDE dają kolejny powód na spójne korzystanie z adnotacji Override. Środowiska te wykonują automatyczne testy nazywane inspekcją kodu. Jeżeli włączymy odpowiednią inspekcję, IDE będzie generowało ostrzeżenia w przypadku napotkania metody, która nie ma adnotacji Override, ale nadpisuje metodę w klasie bazowej. Jeżeli będziemy konsekwentnie korzystać z adnotacji Override, ostrzeżenie to będzie szybko informować o nieoczekiwanym nadpisywaniu. Ostrzeżenia te uzupełniają komunikaty o błędach kompilatora, które ostrzegają nas o nieoczekiwanym błędzie nadpisania. Dzięki IDE i kompilatorowi możemy być pewni, że nadpisujemy metody wszędzie tam, gdzie chcemy je nadpisać, i nigdzie indziej.

Można używać adnotacji Override w deklaracjach metod, które nadpisują deklaracje z interfejsów, jak również klas. Wraz z pojawieniem się metod domyślnych dobrą praktyką stało się korzystanie z Override na konkretnych implementacjach interfejsów metod, aby upewnić się co do poprawności sygnatury. Jeśli wiesz, że interfejs nie zawiera metod domyślnych, możesz pominąć adnotacje Override w konkretnych implementacjach metod interfejsu, aby niepotrzebnie nie zaśmiecać kodu.

W klasie abstrakcyjnej lub interfejsie **warto** jednak adnotować **wszystkie** metody, które będą nadpisywały metody klasy lub interfejsu nadrzędnego, niezależnie, czy konkretne, czy abstrakcyjne. Na przykład interfejs Set dodaje nowe metody do interfejsu Collection, więc powinien zawierać adnotacje Override we wszystkich deklaracjach metod w celu upewnienia się, czy przypadkowo do interfejsu Collection nie zostały dodane nowe metody.

Podsumujmy. Kompilator może ochronić nas przed wieloma błędami, o ile będziemy używać adnotacji `Override` w każdej deklaracji metody, która będzie nadpisywała deklarację w typie bazowym — z jednym wyjątkiem. W klasie konkretnej nie ma potrzeby adnotować metod, które będą nadpisywały deklaracje metod abstrakcyjnych (choć nic nie przeszkadza to zrobić).

Temat 41. Użycie interfejsów znacznikowych do definiowania typów

Interfejs znacznikowy jest interfejsem niezawierającym deklaracji metod. Oznacza on klasę implementującą ten interfejs jako mającą pewne właściwości. Przykładem może być interfejs `Serializable` (rozdział 12.). Przez implementowanie tego interfejsu klasa wskazuje, że te obiekty mogą być zapisywane do `ObjectOutput` `Stream` (lub serializowane).

Można spotkać się z opinią, że adnotacje znacznikowe (temat 39.) spowodowały, że interfejsy znacznikowe są przestarzałe. Założenie to jest błędne. Interfejsy znacznikowe mają dwie zalety w stosunku do adnotacji znacznikowych. **Pierwszą i najważniejszą jest to, że interfejsy znacznikowe definiują typ implementowany przez obiekty oznaczanej klasy, a adnotacje znacznikowe tego nie zapewniają.** Istnienie tego typu pozwala przechwytywać błędy w czasie kompilacji, które w przypadku użycia adnotacji znacznikowej mogą być przechwycone dopiero w czasie działania programu.

Mechanizm serializacji Javy (rozdział 11.) używa interfejsu znacznikowego `Serializable` do oznaczenia typu, który można serializować. Metoda `ObjectOutput` `Stream.write`, która odpowiada za serializację przekazanego do niej obiektu, wymaga zastosowania tego interfejsu w przekazywanym obiekcie. Gdyby argumentem metody był typ `Serializable`, próbę serializacji nieodpowiedniego obiektu można by wykryć już na etapie kompilacji. Wykrywanie błędów na etapie kompilacji to główne zadanie interfejsów znacznikowych. Z niewyjaśnionych przyczyn metoda `ObjectOutputStream.write` nie skorzysta z interfejsu `Serializable`. Typem argumentu metody jest `Object`. Jak się okazuje, próba wywołania `ObjectOutputStream.write` na obiekcie, który nie implementuje `Serializable`, spowoduje błąd tylko w czasie wykonania.

Inną przewagą interfejsu znacznikowego nad adnotacjami znacznikowymi jest to, że mogą być bardziej precyzyjnie kierowane. Jeżeli typ adnotacji jest zadeklarowany z docelowym `ElementType.TYPE`, może być stosowany do **dowolnej** klasy lub interfejsu. Załóżmy, że mamy znacznik stosowany tylko do implementacji określonego interfejsu. Jeżeli zdefiniujemy go jako interfejs znacznikowy, będziemy mogli rozszerzać jedyny interfejs, do którego się on odnosi, gwarantując, że wszystkie oznaczone typy są również podtypami jedynego interfejsu, do którego się odnosi.

Można się spierać, że interfejs `Set` jest takim **ograniczonym interfejsem znacznikowym**. Jest stosowany tylko do podtypów `Collection`, ale nie dodaje żadnych innych metod poza zdefiniowanymi przez `Collection`. Nie jest on uważany powszechnie za interfejs znacznikowy, ponieważ precyzuje założenia kilku metod `Collection`, w tym `add`, `equals` i `hashCode`. Jednak łatwo sobie wyobrazić interfejs znacznikowy, który jest stosowany tylko do podtypów pewnego interfejsu i który **nie** precyzuje założeń żadnej z metod interfejsu. Takie interfejsy znacznikowe mogą opisywać pełny niezmiennik całego obiektu lub wskazywać, że obiekty mogą być przetwarzane przez metodę pewnej innej klasy (w sposób, w jaki interfejs `Serializable` oznacza, że obiekt może być przetwarzany przez `ObjectOutputStream`).

Główną zaletą adnotacji znacznikowych względem interfejsów znacznikowych jest to, że są częścią większego mechanizmu adnotacji. Dzięki temu adnotacje znacznikowe pozwalają na zachowanie spójności we frameworkach pozwalających na adnotowanie różnych elementów programu.

Kiedy powinniśmy korzystać z adnotacji znacznikowych, a kiedy z interfejsów znacznikowych? Jasne jest, że konieczne jest używanie adnotacji, jeżeli znacznik odnosi się do dowolnego elementu programu innego niż klasa lub interfejs, ponieważ tylko klasy i interfejsy mogą implementować lub rozszerzać interfejs. Jeżeli znacznik odnosi się wyłącznie do klasy lub interfejsu, należy odpowiedzieć sobie na pytanie, czy potrzebuję napisać jedną lub więcej metod akceptujących wyłącznie obiekty z tym znacznikiem? Jeżeli tak, powinniśmy użyć interfejsu znacznikowego zamiast adnotacji. Umożliwia to użycie interfejsu jako typu parametru metody, co pozwoli skorzystać z kontroli typów w czasie kompilacji. Jeżeli odpowiedź na pierwsze pytanie brzmi „nie”, zadajemy sobie drugie — czy chcemy ograniczyć użycie tego znacznika do elementów określonego interfejsu na zawsze. Jeżeli tak, sensowne jest zdefiniowanie znacznika jako podinterfejsu tego interfejsu. Jeżeli odpowiedź na oba pytania brzmi „nie”, powinniśmy prawdopodobnie użyć adnotacji.

Podsumujmy. Interfejsy znacznikowe i adnotacje znacznikowe mają swoje zastosowania. Jeżeli chcemy zdefiniować typ, który nie ma skojarzonych żadnych metod, pozwoli na to typ znacznikowy. Jeżeli chcemy oznaczyć elementy programu inne niż klasy i interfejsy, pozostawiając możliwość dodania w przyszłości do znacznika większej ilości informacji, albo dodać znacznik do frameworku intensywnie wykorzystującego typ adnotacyjny, to właściwym wyborem jest adnotacja znacznikowa. **Jeżeli okazuje się, że piszemy typ znacznikowy, którego elementem docelowym jest `ElementType.TYPE`, warto poświęcić nieco czasu na zorientowanie się, czy faktycznie powinien to być typ znacznikowy, czy też lepszy byłby interfejs znacznikowy.**

W pewnym sensie jest to odwrotność porady z tematu 22.: „Jeżeli nie potrzebujesz typu, nie używaj interfejsu”. Można powiedzieć, że jest to odpowiednik wskazówki: „Jeżeli chcesz zdefiniować typ, użyj interfejsu”.

Lambdy i strumienie

W Javie 8 pojawiło się kilka nowych elementów takich, jak interfejsy funkcyjne, lambdy i referencje do metod, których celem jest ułatwienie tworzenia obiektów funkcyjnych. Dodatkowo do wymienionych zmian językowych jest API strumieni, które zapewnia obsługę przetwarzania sekwencji elementów już na poziomie bibliotek języka. W tym rozdziale omówimy, jak w najlepszy sposób wykorzystać nowe funkcjonalności.

Temat 42. Stosuj lambdy zamiast klas anonimowych

Dawniej jako **typy funkcyjne** wykorzystywano interfejsy (lub rzadziej klasy abstrakcyjne) posiadające tylko jedną metodę abstrakcyjną. Ich instancje, nazywane **obiektami funkcyjnymi**, reprezentowały akcje lub funkcje. Od momentu pojawienia się JDK 1.1 w roku 1997 podstawowym sposobem tworzenia obiektów funkcyjnych były **klasy anonimowe** (temat 24.). Oto fragment kodu, który sortuje listę tekstów na podstawie ich długości, wykorzystując przy tym klasę anonimową w celu określenia funkcji porównującej stosowanej przy sortowaniu:

```
// Instancja klasy anonimowej jako obiekt funkcyjny - to rozwiązanie przestarzałe!
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
```

Klasy anonimowe były odpowiednim rozwiązaniem w klasycznych, obiektowych wzorcach projektowych wymagających obiektów funkcyjnych, w szczególności we wzorcu **strategii** (*Strategy*) [Gamma95]. Interfejs `Comparator` reprezentuje

strategię abstrakcyjną dotyczącą sortowania, a powyższa klasa anonimowa to **konkretna strategia** dotycząca sortowania tekstów. Zbyt rozbudowana składnia związana z klasami anonimowymi powodowała jednak, że programowanie funkcyjne w Javie nie należało do zadań szczególnie przyjemnych.

W Javie 8 język sformalizował pojęcie interfejsu z pojedynczą metodą abstrakcyjną jako pewien przypadek szczególnie, wymagający specjalnego traktowania. Interfejsy tego typu nazywa się obecnie **interfejsami funkcyjnymi**, a język pozwala na tworzenie ich instancji za pomocą **wyrażeń lambda** nazywanych skrótowo **lambdami**. Lambdy odgrywają podobną rolę jak klasy anonimowe, ale ich zapis jest dużo bardziej zwięzły. Oto fragment kodu realizujący dokładnie to samo zadanie co przedstawiona wcześniej klasa anonimowa, ale za pomocą lambda. Zniknął cały nadmiarowy zapis, więc od razu widać, co jest celem funkcji:

```
// Wyrażenie lambda jako obiekt funkcyjny (zastępuje klasę anonimową)
Collections.sort(words, (s1, s2) -> Integer.compare(s1.length(),
↳s2.length()));
```

Zwróć uwagę, że zarówno typ lambda (`Comparator<String>`), jak i typy parametrów lambda (`s1` i `s2` jako `String`) oraz typ zwracanej wartości (`int`) nie pojawiają się w kodzie. Kompilator odgaduje te typy na podstawie kontekstu, wykorzystując proces nazywany **wnioskowaniem typów**. Czasem zdarza się, że kompilator nie potrafi wywnioskować typu i trzeba wskazać go ręcznie. Reguły wnioskowania są dosyć złożone, o czym świadczy chociażby to, że zajmują cały rozdział w specyfikacji języka [JLS, 18]. Niewielu programistów dokładnie rozumie cały proces wnioskowania, ale to nie problem. **Pomijaj typy we wszystkich parametrach lambda, chyba że ich pojawienie się ułatwi zrozumienie programu.** Jeśli kompilator zgłosi błąd z informacją o tym, że nie jest w stanie odgadnąć typu, to **wskaz** go ręcznie. Czasem zajdzie potrzeba rzutowania zwracanej wartości lub wskazania typu całego wyrażenia lambda, ale to bardzo rzadkie przypadki.

W tym miejscu musimy podkreślić pewien aspekt dotyczący wnioskowania typów. Temat 26. doradza, aby nie używać typów surowych, temat 29. zaleca stosowanie typów ogólnych, a temat 30. proponuje skupienie się na metodach ogólnych. Wszystkie te rady są szczególnie istotne, jeśli korzysta się z lambda, ponieważ kompilator większość informacji napędzających wnioskowanie typów czerpie właśnie z typów ogólnych. Jeśli nie dostarczymy odpowiednich informacji, kompilator nie będzie w stanie wykonać wnioskowania i trzeba będzie ręcznie wskazać wszystkie typy w lambda, co znacznie zaciemni ich zapis. Przykładowo powyższy fragment kodu nie skompiluje się, jeśli zmienna `words` zostanie zadeklarowana jako typ surowy `List` zamiast typu parametryzowanego `List<String>`.

Co ciekawe, komparator w przedstawionym przykładzie można zapisać jeszcze zwięźlej, jeśli zamiast lambda użyjemy **metody konstruującej komparator** (temat 43.):

```
Collections.sort(words, comparingInt(String::length));
```

Tak naprawdę kod można skrócić jeszcze bardziej, wykorzystując nową metodę `sort` dodaną do interfejsu `List` w Javie 8:

```
words.sort(comparingInt(String::length));
```

Wprowadzanie do języka `lambda` powoduje, że sensowne staje się stosowanie obiektów funkcyjnych tam, gdzie wcześniej nie było to szczególnie wygodne. Rozważmy na przykład typ wyliczeniowy `Operation` z tematu 34. Ponieważ każdy element wyliczenia wymagał innego działania w metodzie `apply`, wykorzystaliśmy osobne treści klas dla każdej stałej i nadpisaliśmy metodę `apply` osobno dla każdego elementu wyliczenia. Przytoczmy jeszcze raz fragment, o którym mówimy:

```
// Typ wyliczeniowy z osobną treścią i danymi dla każdej stałej (temat 34.)
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

W temacie 34. wskazaliśmy, że pola instancyjne wyliczeń są lepsze niż osobne treści klasy dla każdej stałej. Lambdy ułatwiają implementację różnego działania dla każdej stałej przy użyciu pierwszej techniki zamiast drugiej. Wystarczy, że przekażemy lambdę implementującą logikę każdej ze stałych do konstruktora. Konstruktor umieści lambdę w polu instancji, a metoda `apply` przekaże wywołanie do lambdy. Wynikowy kod jest prostszy i czytelniejszy niż oryginał:

```
// Wyliczenie z polami obiektów funkcyjnych i działaniem unikatowym dla każdej stałej
public enum Operation {
    PLUS("+", (x, y) -> x + y),
    MINUS("-", (x, y) -> x - y),
    TIMES("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;
```

```
Operation(String symbol, DoubleBinaryOperator op) {
    this.symbol = symbol;
    this.op = op;
}

@Override public String toString() { return symbol; }

public double apply(double x, double y) {
    return op.applyAsDouble(x, y);
}
}
```

Zwróć uwagę, że w przypadku lambda reprezentujących działanie stałych wyliczenia korzystamy z interfejsu `DoubleBinaryOperator`. To jeden z wielu predefiniowanych interfejsów funkcyjnych dostępnych w pakiecie `java.util.function` (temat 44.). Reprezentuje on funkcję, która przyjmuje dwa argumenty typu `double` i zwraca argument typu `double`.

Patrząc na wersję wyliczenia `Operation` wykorzystującą lambda, moglibyśmy dojść do wniosku, że osobne treści metod dla każdej stałej przestały być potrzebne, ale tak nie jest. W odróżnieniu od metod i klas, **lambdy nie mają nazw i dokumentacji; jeśli obliczenia nie są oczywiste na pierwszy rzut oka lub zajmują więcej niż kilka wierszy kodu, nie umieszczaj ich w lambdzie**. Dla lambda najlepsze są operacje jednowierszowe, a trzy wiersze to rozsądne maksimum. Złamanie tej zasady może prowadzić do poważnych problemów z czytelnością programów. Jeśli lambda jest długa lub skompilowana, znajdź sposób na jej uproszczenie lub zrefaktoryzuj kod, aby się jej pozbyć. Argumenty przekazywane do konstruktorów wyliczeń są wyliczane w kontekście statycznym. Lambdy w konstruktorach wyliczeń nie mają dostępu do składowych instancji wyliczenia. Niezależne treści klas dla każdej stałej to nadal właściwe rozwiązanie, jeśli typ wyliczeniowy wymaga złożonego działania, którego nie można zawrzeć w kilku wierszach kodu, lub też potrzebuje dostępu do pól i metod instancji.

Można by również pomyśleć, że w erze lambda klasy anonimowe są przeżytkiem. Nie jest to dalekie od prawdy, ale istnieje kilka zadań, które są możliwe do zrealizowania za pomocą klas anonimowych, a nie można ich wykonać za pomocą lambda. Lambdy ograniczają się tylko do interfejsów funkcyjnych. Jeśli musisz otworzyć instancję klasy abstrakcyjnej, musisz użyć klasy anonimowej zamiast lambda. Klasy abstrakcyjne pozwalają na utworzenie instancji interfejsów zawierających wiele metod abstrakcyjnych. Lambda nie jest w stanie pobrać referencji do samej siebie. W lambdzie `this` dotyczy otaczającej instancji, bo najczęściej właśnie na tym nam zależy. W klasie anonimowej `this` odnosi się do instancji klasy anonimowej. Jeśli z jakiegoś powodu musisz mieć dostęp do obiektu funkcyjnego z poziomu jego kodu, klasa anonimowa to jedyna droga.

Lambdy mają też pewną wspólną cechę z klasami anonimowymi, a mianowicie nie można ich w sposób niezawodny serializować i deserializować między różnymi implementacjami. Z tego powodu **bardzo rzadko, jeśli kiedykolwiek, decyduj się na serializację lambdy** (lub instancji klasy anonimowej). Jeśli jakiś obiekt funkcyjny chcesz uczynić serializowalnym (np. obiekt `Comparator`), użyj instancji prywatnej, statycznej klasy zagnieżdżonej (temat 24.).

Podsumujmy. Od Javy 8 lambdy to zdecydowanie najlepszy sposób tworzenia niewielkich obiektów funkcyjnych. **Nie używaj klas anonimowych do tworzenia obiektów funkcyjnych, chyba że musisz tworzyć instancje typów, które nie są interfejsami funkcyjnymi.** Pamiętaj, że dzięki lambdom tworzenie małych obiektów funkcyjnych stało się wyjątkowo proste, co otwiera drzwi do wielu technik programowania funkcyjnego, które dawniej nie były w Javie szczególnie wygodne w stosowaniu.

Temat 43. Wybieraj referencje do metod zamiast lambd

Podstawową zaletą lambd względem klas anonimowych jest zwięzłość. Java umożliwia jednak generowanie obiektów funkcyjnych jeszcze bardziej zwięzłych w zapisie niż lambdy — to **referencje do metod**. Poniżej znajduje się kod stanowiący fragment programu, który wykorzystuje odwzorowanie z dowolnych wartości kluczy na wartości typu `Integer`. Jeśli wartość jest interpretowana jako liczba wystąpień danego klucza, to program stanowi implementację wielozbioru. Funkcja przedstawiona poniżej ma za zadanie ustawić wartość 1, jeśli klucz nie znajduje się jeszcze w odwzorowaniu, lub zwiększyć istniejącą wartość o jeden, jeśli już się znajduje:

```
map.merge(key, 1, (count, incr) -> count + incr);
```

Zwróć uwagę, że kod korzysta z metody `merge` dodanej do interfejsu `Map` w Javie 8. Jeśli w odwzorowaniu nie ma wskazanego klucza, metoda po prostu wstawia podaną wartość. Jeśli jednak klucz już istnieje, metoda `merge` uruchamia wskazaną funkcję, aby połączyć istniejącą wartość i podaną, a następnie zapisać wynik jako nową wartość dla klucza. Kod ilustruje typowy sposób użycia metody `merge`.

Kod działa poprawnie, ale nadal można twierdzić, że jest za długi. Parametry `count` i `incr` zajmują miejsce w kodzie, choć nie wnoszą wiele do zrozumienia funkcji. Tak naprawdę lambda wykonuje najprostsze na świecie sumowanie dwóch przekazanych wartości i zwrócenie wyniku. Od Javy 8 klasa `Integer` (i inne klasy osłonowe dla liczbowych typów prostych) zapewniają metodę statyczną `sum`, która

wykonuje dokładnie to samo zadanie co powyższa lambda. Możemy więc przekazać referencję do tej metody i uzyskać taki sam wynik, choć w bardziej zwięzły sposób:

```
map.merge(key, 1, Integer::sum);
```

Im więcej parametrów ma metoda, tym więcej kodu można się pozbyć, jeśli zamieni się ją na referencję do metody. Czasem jednak nazwy parametrów lambda stanowią swego rodzaju dokumentację, co czyni lambda czytelniejszymi i łatwiejszymi w konserwacji niż referencje do metod, nawet jeśli efektem jest dłuższy kod.

W zasadzie nie istnieją zadania, których nie może wykonać referencja do metody, a które może zrealizować lambda (poza jednym wyjątkiem — znajdziesz go w JLS, 9.9-2). Zazwyczaj referencje do metod pozwalają skrócić kod i uczynić go łatwiejszym w analizie. Są wyjściem bezpieczeństwa, jeśli lambda staje się zbyt długa lub złożona — można wydobyć kod lambda i przenieść go do nowej metody, a lambda zastąpić referencją do metody. Nowa metoda otrzyma klarowną nazwę i w razie potrzeby może również zostać w pełni udokumentowana.

Jeśli w trakcie programowania korzystamy ze środowiska IDE, zaproponuje ono najczęściej zastąpienie lambda referencją do metody, jeśli to dopuszczalne. Zazwyczaj, choć nie zawsze, powinieneś przystać na taką ofertę. Zdarzają się jednak sytuacje, gdy to lambda będzie bardziej zwięzła od referencji do metody. Sytuacja taka zachodzi najczęściej wtedy, gdy metoda znajduje się w tej samej klasie co lambda. Rozważmy poniższy fragment, który istnieje w klasie o nazwie `GoshThis` ↪ `ClassNameIsHumongous`:

```
service.execute(GoshThisClassNameIsHumongous::action);
```

Równoważna temu zapisowi lambda ma postać:

```
service.execute(() -> action());
```

W takiej sytuacji referencja do metody nie zapewnia większej klarowności kodu ani nie jest krótsza, więc stosuj drugie rozwiązanie. Warto w tym miejscu wspomnieć o tym, że interfejs `Function` oferuje uogólnioną, statyczną metodę fabryczną, która zwraca funkcję tożsamościową: `Function.identity()`. W większości przypadków bardziej czytelny i krótszy będzie kod **nieużywający** tej metody, ale jego równoważnika w postaci lambda: `x -> x`.

Wiele referencji do metod odnosi się do metod statycznych, ale cztery rodzaje dotyczą innych sytuacji. Dwa z nich to **związane** i **niezwiązane** referencje do metody instancyjnych. W referencji związanej obiekt otrzymujący jest określony w referencji do metody. Referencje związane działają podobnie do referencji statycznych — obiekt funkcyjny przyjmuje takie same argumenty jak metoda, której dotyczy referencja. W referencjach niezwiązanych obiekt otrzymujący jest wskazywany w momencie wykonywania obiektu funkcyjnego jako dodatkowy parametr, który pojawi się przed parametrami zadeklarowanymi przez metodę. Referencji niezwią-

zanych używa się najczęściej jako funkcji odwzorowujących i filtrujących w potokach strumieni (temat 45.). Istnieją również dwa rodzaje referencji do konstruktorów: po jednym dla klas i tablic. Działają one jako obiekty fabryczne. Wszystkie pięć rodzajów referencji do metod zostało podsumowane w tabeli 7.1.

Tabela 7.1. Rodzaje referencji do metod

Rodzaje referencji do metody	Przykład	Równoważna wersja jako lambda
statyczna	<code>Integer::parseInt</code>	<code>str -> Integer.parseInt(str)</code>
związana	<code>Instant.now()::isAfter</code>	<code>Instant then = Instant.now(); t -> then.isAfter(t)</code>
niezwiązana	<code>String::toLowerCase</code>	<code>str -> str.toLowerCase()</code>
konstruktor klasy	<code>TreeMap<K,V>::new</code>	<code>() -> new TreeMap<K,V></code>
konstruktor tablicy	<code>int[]::new</code>	<code>len -> new int[len]</code>

Podsumujmy. Referencje do metod najczęściej stanowią bardziej zwięzłą alternatywę dla lambda. **Jeśli referencja do metody jest krótsza i czytelniejsza od lambda, użyj jej; jeśli nie, pozostañ przy lambda.**

Temat 44. Korzystaj ze standardowych interfejsów funkcyjnych

Wraz z wprowadzeniem do języka Java lambda znaczącej zmianie uległy najlepsze praktyki związane z pisaniem API. Na przykład wzorzec **metody szablonowej** (*Template Method*) [Gamma95], w którym podklasa nadpisuje metodę prostą, aby uszczegółowić zachowanie klasy nadrzędnej, staje się dużo mniej atrakcyjny. Nowoczesną alternatywą jest zapewnienie statycznej fabryki lub konstruktora, który przyjmuje obiekt funkcyjny w celu uzyskania tego samego efektu. Ogólnie rzecz biorąc, dużo częściej tworzy się teraz konstruktory i metody przyjmujące w parametrach obiekty funkcyjne. Z tego względu dodatkową uwagę trzeba poświęcić wyborowi właściwego parametru funkcyjnego.

Rozważmy klasę `LinkedHashMap`. Można wykorzystywać tę klasę jako bufor lub pamięć podręczną po nadpisaniu zabezpieczonej metody `removeEldestEntry`, która jest wywoływana przy każdym użyciu metody `put` w celu wstawienia nowego elementu. Gdy metoda zwróci wartość `true`, odwzorowanie usuwa swój najstarszy wpis, który został przekazany do metody. Poniższy fragment kodu z nadpisywaną metodą pozwala odwzorowaniu urosnąć do stu wpisów, a następnie powoduje usunięcie najstarszego wpisu za każdym razem, gdy pojawi się nowy klucz, co pozwala utrzymywać stałą wartość stu najnowszych wpisów:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return size() > 100;
}
```

Przedstawiona technika sprawdza się, ale dzięki lambdom można ją udoskonalić. Gdyby klasę `LinkedHashMap` tworzono obecnie z uwzględnieniem zmian w języku, najprawdopodobniej posiadałaby ona statyczną fabrykę lub konstruktor przyjmujący obiekt funkcyjny. Po przyjrzeniu się deklaracji metody `removeEldestEntry` moglibyśmy sądzić, że obiekt funkcyjny musi jedynie przyjąć parametr `Map.Entry<K,V>` i zwrócić wartość typu `boolean`, ale to nie wystarczy — metoda wywołuje metodę `size()`, aby pobrać aktualną liczbę wpisów w odwzorowaniu, co oczywiście działa, bo `removeEldestEntry` jest metodą instancyjną odwzorowania. Przekazywany obiekt funkcyjny nie jest metodą instancyjną odwzorowania i nie może nią być, bo w momencie wywoływania statycznej fabryki lub konstruktora obiekt odwzorowania jeszcze nie istnieje. Z tego powodu odwzorowanie musi przekazać same siebie do obiektu funkcyjnego, więc musi on przyjmować nie tylko najstarszy wpis, ale również obiekt odwzorowania. Gdybyśmy chcieli zadeklarować odpowiedni interfejs funkcyjny, wyglądałby on podobnie do poniższego:

```
// Zbyteczny interfejs funkcyjny; skorzystaj ze standardowego
@FunctionalInterface interface EldestEntryRemovalFunction<K,V>{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

Taki interfejs działałby poprawnie, ale nie używaj go, bo nie trzeba deklarować w tym celu nowych interfejsów. Pakiet `java.util.function` zapewnia duży zbiór standardowych interfejsów funkcyjnych. **Jeśli jeden ze standardowych interfejsów funkcyjnych potrafi wykonać zadanie, stosuj go, zamiast tworzyć interfejs funkcyjny dedykowany do konkretnej sytuacji.** W ten sposób API będzie łatwiejsze do opanowania, bo zmniejszy się liczba nowych elementów, a dodatkowo rozwiązanie zwiększy interoperacyjność, ponieważ wiele standardowych interfejsów funkcyjnych zawiera przydatne metody domyślne. Przykładowo interfejs `Predicate` zapewnia metody pozwalające łączyć predykaty. W przedstawionym przykładzie z klasą `LinkedHashMap` powinniśmy zastosować standardowy interfejs `BiPredicate<Map<K,V>, Map.Entry<K,V>>` zamiast własnego interfejsu `EldestEntryRemovalFunction`.

Pakiet `java.util.function` zawiera czterdzieści trzy interfejsy. Nikt nie oczekuje, że zapamiętasz je wszystkie, ale jeśli zapamiętasz sześć najbardziej podstawowych, to będziesz mógł na ich podstawie wyprowadzić pozostałe. Interfejsy podstawowe operują na typach stanowiących referencje do obiektów. Interfejs `Operator` reprezentuje funkcje, w których typ wyniki i typy argumentów są takie same. Interfejs `Predicate` reprezentuje funkcje, które przyjmują argument i zwracają wartość typu `boolean`. Interfejs `Function` reprezentuje funkcje, których typy argumentu i zwracanej wartości różnią się. Interfejs `Supplier` reprezentuje funkcje, które nie przyjmują argumentów, ale zwracają („dostarczają”) pewną wartość. Interfejs `Consumer`

reprezentuje funkcje, które przyjmują pewien argument, ale nic nie zwracają, czyli można powiedzieć, że go „konsumują”. Tabela 7.2 podsumowuje podstawowe interfejsy funkcyjne.

Tabela 7.2. Podstawowe interfejsy funkcyjne

Interfejs	Sygnatura funkcji	Przykład
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

Istnieją trzy odmiany każdego z sześciu interfejsów podstawowych działające dla typów prostych `int`, `long` i `double`. Ich nazwy wywodzą się z nazw typów podstawowych, ale są poprzedzone nazwą typu prostego. Oznacza to, że predykat przyjmujący typ `int` to `IntPredicate`, a operator binarny przyjmujący dwie wartości i typu `long` i zwracający typ `long` to `LongBinaryOperator`. Żadna z tych odmian nie jest parametryzowana poza odmianami `Function`, które pozwalają wskazać typ zwracanej wartości. Przykładowo `LongFunction<int[]>` przyjmuje wartość typu `long` i zwraca wynik typu `int[]`.

Dodatkowo istnieje dziewięć odmian interfejsu `Function` mających zastosowanie, gdy zwracanym typem jest typ prosty. Typ źródłowy i docelowy zawsze są różne, bo gdyby były takie same, interfejsem byłby `UnaryOperator`. Jeśli zarówno typ źródłowy, jak i docelowy jest typem prostym, poprzedź `Function` tekstem *ŹródłoToCel*, np. `LongToIntFunction` (tych odmian jest sześć). Jeśli źródło jest typem prostym, ale wynik jest referencją do obiektu, poprzedź `Function` tekstem *ŹródłoToObj*, np. `DoubleToObjFunction` (są trzy takie odmiany).

Istnieją dwuargumentowe wersje trzech podstawowych interfejsów funkcyjnych, dla których mają one sens: `BiPredicate<T,U>`, `BiFunction<T,U,R>` i `BiConsumer<T,U>`. Istnieją również odmiany `BiFunction` zwracające odpowiednie typy proste: `ToIntBiFunction<T,U>`, `ToLongBiFunction<T,U>` i `ToDoubleBiFunction<T,U>`. Istnieją dwuargumentowe odmiany `Consumer`, które przyjmują jedną referencję do obiektu i jeden typ prosty: `ObjDoubleConsumer<T>`, `ObjIntConsumer<T>` i `ObjLongConsumer<T>`. Łącznie istnieje dziewięć dwuargumentowych wersji interfejsów podstawowych.

Istnieje również interfejs `BooleanSupplier`, czyli odmiana interfejsu `Supplier` zwracająca wartości typu `boolean`. To jedyne miejsce, gdzie w dokumentacji oficjalnie pojawia się typ `boolean` w którejkolwiek z nazw standardowych interfejsów. Pamiętaj, że typ `boolean` jako zwracana wartość jest obsługiwany przez `Predicate` i jego cztery odmiany. Interfejs `BooleanSupplier` i jego czterdzieści dwie odmiany opisane we wcześniejszych akapitach tworzą łącznie czterdzieści trzy standardowe

interfejsy funkcyjne. Oczywiście taką liczbę nowych elementów niełatwo przyswoić. Z drugiej strony większość niezbędnych w typowych działaniach interfejsów funkcyjnych została w ten sposób udostępniona, a ich nazwy są na tyle spójne, że odgadnięcie właściwej nazwy interfejsu, z którego chce się skorzystać, nie powinno być szczególnie trudne.

Większość standardowych interfejsów funkcyjnych istnieje tylko po to, aby zapewnić wsparcie dla typów prostych. **Nie daj się skusić na stosowanie podstawowych interfejsów funkcyjnych z opakowaniem typów prostych zamiast dedykowanych interfejsów funkcyjnych dla typów prostych.** Choć taki kod będzie działał, łamie on zalecenie z tematu 61., czyli „stosowanie typów prostych zamiast opakowanych typów prostych”. Konsekwencje wydajnościowe związane ze stosowaniem opakowanych typów w przypadku hurtowego wykonywania operacji mogą być naprawdę poważne.

Wiemy już, że powinniśmy w większości typowych sytuacji korzystać ze standardowych interfejsów funkcyjnych, zamiast tworzyć własne. Kiedy więc **powinniśmy** napisać własny interfejs funkcyjny? Oczywiście musimy to zrobić, gdy jeden z istniejących interfejsów nie spełnia stawianych wymagań, bo na przykład musimy użyć trzech parametrów lub kod metody musi zgłaszać przechwytywany wyjątek. Zdarzają się jednak przypadki, gdy powinniśmy napisać własny interfejs funkcyjny, choć istnieje interfejs standardowy o identycznej strukturze.

Przyjrzyjmy się naszemu staremu znajomemu, `Comparator<T>`, który jest strukturalnie równoważny interfejsowi `ToIntBiFunction<T,T>`. Nawet gdyby ten drugi istniał już w bibliotece, gdy dodawano ten pierwszy, użycie standardowego interfejsu byłoby błędem. Jest kilka powodów, dla których `Comparator` zasługuje na własny interfejs. Po pierwsze, jego nazwa stanowi doskonałą dokumentację przy każdym użyciu w API, a jest stosowana bardzo często. Po drugie, interfejs `Comparator` wyraźnie określa wymagania dla prawidłowych instancji, czyli definiuje ogólny kontrakt. Implementując ten interfejs, wskazujemy, że zastosujemy się do kontraktu. Po trzecie, interfejs jest obecnie najeżony metodami domyślnymi pozwalającymi na przekształcanie i łączenie komparatorów.

Warto rozważyć napisanie własnego interfejsu funkcyjnego nakierowanego na konkretny cel, zamiast korzystać z istniejącego, gdy interfejs funkcyjny musi posiadać przynajmniej jedną z cech, które wskazaliśmy dla interfejsu `Comparator`, jeśli:

- będzie powszechnie stosowany, a posiadanie unikatowej nazwy przyniesie korzyść;
- ma silny, dobrze zdefiniowany kontrakt;
- dodanie metod domyślnych interfejsu byłoby korzystne.

Jeśli zdecydujesz się na napisanie własnego interfejsu funkcyjnego, pamiętaj, że jest to interfejs i jako taki powinien być projektowany bardzo rozważnie (temat 21.).

Zwróć uwagę, że interfejs `EldestEntryRemovalFunction` zaprezentowany kilka stron wcześniej został oznaczony adnotacją `@FunctionalInterface`. Adnotacja ma podobne znaczenie jak adnotacja `@Override`. Wskazuje ona na intencje programisty i ma trzy cele: informuje przeglądających klasę i jej dokumentację, że interfejs został zaprojektowany na potrzeby lambda; wymusza niemożność kompilacji, jeśli interfejs nie będzie składał się z dokładnie jednej metody abstrakcyjnej; i zapobiega przypadkowemu dodaniu przez osoby konserwujące kod nowych metod abstrakcyjnych. **Zawsze stosuj adnotację `@FunctionalInterface` we własnych interfejsach funkcyjnych.**

Ostatnia uwaga w tym temacie dotyczyć będzie zastosowania interfejsów funkcyjnych w API. Nie dostarczaj metody z wieloma przeciążeniami, które przyjmują różne interfejsy funkcyjne na tej samej pozycji argumentu, jeśli po stronie klienckiej mogłoby to doprowadzić do dwuznaczności. Nie jest to jedynie problem teoretyczny. Metoda `submit` z `ExecutorService` może przyjmować jako argument `Callable<T>` lub `Runnable`. Można tak napisać program kliencki, że niezbędne będzie rzutowanie wskazujące właściwe przeciążenie (temat 52.). Najprostszym sposobem uniknięcia tego problemu jest powstrzymanie się od tworzenia przeciążeń, które przyjmują na tej samej pozycji argumentu różne interfejsy funkcyjne. To szczególnie przypadkiem zalecenia z tematu 52.

Podsumujmy. Skoro w Javie pojawiły się wyrażenia lambda, należy projektować API z uwzględnieniem ich istnienia. Przyjmuj typy interfejsów funkcyjnych na wejściu i zwracaj je na wyjściu. W większości sytuacji zaleca się stosowanie standardowych interfejsów zapewnianych przez pakiet `java.util.function`, ale istnieją rzadkie przypadki, w których lepiej napisać własny interfejs funkcyjny.

Temat 45. Rozważnie korzystaj ze strumieni

API strumieni zostało wprowadzone w Javie 8, aby ułatwić realizację operacji grupowych wykonywanych sekwencyjnie lub w sposób równoległy. API korzysta z dwóch kluczowych abstrakcji: strumienia, który reprezentuje skończony lub nieskończony ciąg elementów danych, oraz potoku strumienia, który reprezentuje wieloetapowe obliczenia realizowane na elementach. Elementy w strumieniu mogą pochodzić z dowolnego źródła. Typowymi źródłami są kolekcje, tablice, pliki, wyniki dopasowania do wyrażenia regularnego, generatory liczb pseudolosowych itp. Elementami danych w strumieniu mogą być zarówno referencje do obiektów, jak i wartości proste. Obsługiwane są trzy typy proste: `int`, `long` i `double`.

Potok strumienia składa się ze strumienia źródłowego, po którym pojawia się zero lub więcej **operacji pośrednich** i jedna **operacja kończąca**. Każda operacja pośrednia w jakiś sposób przekształca strumień, np. odwzorowuje element na funkcję tego elementu lub filtruje wszystkie elementy niespełniające pewnego warunku. Operacje pośrednie przekształcają jeden strumień w inny, którego typ elementów może być taki sam jak strumienia wejściowego lub inny. Operacja kończąca wykonuje ostatnie obliczenia na strumieniu z ostatniej operacji pośredniej, np. zapamiętuje elementy w kolekcji, zwraca konkretny element lub wyświetla wszystkie elementy.

Potoki strumieni są obliczane w sposób **leniwy** — obliczenia nie będą realizowane do momentu wywołania operacji kończącej, a elementy danych, które nie są wymagane do wykonania operacji kończącej, nie zostaną nigdy obliczone. To właśnie leniwe obliczanie pozwala na korzystanie ze strumieni nieskończonych. Pamiętaj, że potok strumienia bez operacji kończącej to w zasadzie pusta operacja, więc nie zapomnij dołączyć odpowiedniej operacji.

API strumieni zaprojektowano jako **płynne** (lub łańcuchowe), czyli takie, w którym poszczególne wywołania w potoku są łączone w łańcuch tworzący jedno wyrażenie. Co więcej, nawet kilka potoków można tak połączyć, aby tworzyły jedno wyrażenie.

Domyślnie potoki strumienia są realizowane szeregowo. Zamiana ich na wersję równoległą jest tak prosta, jak wywołanie metody `parallel` na dowolnym strumieniu, ale rzadko jest to właściwe działanie (temat 48.).

API strumieni jest na tyle elastyczne, że w zasadzie za pomocą strumieni można zrealizować dowolne obliczenia. To, że można to zrobić, nie oznacza, że należy. Właściwe użycie strumieni powoduje, że program jest krótszy i czytelniejszy. Niewłaściwe użycie strumieni powoduje, że program jest trudny w analizie i utrzymaniu. Nie istnieją żadne sztywne lub szybkie reguły dotyczące stosowania strumieni — sprawdza się raczej podejście heurystyczne.

Rozważmy poniższy program, który odczytuje słowa ze słownika umieszczonego w pliku i wyświetla wszystkie grupy anagramowe, które spełniają wskazane przez użytkownika minimum. Przypomnijmy, że dwa słowa są anagramami, jeśli składają się z tych samych liter, ale inaczej ułożonych. Program odczytuje każde słowo ze słownika wskazanego przez użytkownika i umieszcza je w obiekcie `Map`. Kluczem w tym obiekcie są litery danego słowa posortowane alfabetycznie, więc kluczem dla "piecza" jest "aceipz", a dla "zapiec" również jest "aceipz". Z tego powodu oba słowa to anagramy, bo wszystkie anagramy współdzielą tę samą postać po posortowaniu liter. Wartością w obiekcie `Map` dla danego klucza będą więc wszystkie słowa współdzielące tę samą postać alfabetyczną. Po przetworzeniu słownika każda lista to osobna grupa anagramowa. Program przechodzi następnie przez widok wartości obiektu `Map` (metoda `values`) i wyświetla te listy, które spełniają wymagania co do długości:


```
// Wyświetla wszystkie duże grupy anagramów ze słownika w sposób iteracyjny
public class Anagrams {
    public static void main(String[] args) throws IOException {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        Map<String, Set<String>> groups = new HashMap<>();
        try (Scanner s = new Scanner(dictionary)) {
            while (s.hasNext()) {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                    (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

Jeden krok tego programu wart jest dodatkowego wyjaśnienia. Wstawianie każdego słowa do odwzorowania (pogrubiony fragment) używa metody `computeIfAbsent` dodanej dopiero w Javie 8. Metoda ta wyszukuje klucz w odwzorowaniu (obiekcie `Map`) — jeśli klucz istnieje, metoda po prostu zwraca powiązaną z nim wartość. Jeśli klucz nie istnieje, metoda oblicza wartość przez wykonanie podanego obiektu funkcyjnego, wiąże tę wartość z kluczem i zwraca wynik obliczeń. Metoda `computeIfAbsent` upraszcza implementację odwzorowań, które łączą pod jednym kluczem wiele wartości.

Rozważmy teraz poniższy program, który rozwiązuje ten sam problem, ale bardzo intensywnie korzysta ze strumieni. Zwróć uwagę, że cały program (z wyjątkiem kodu otwierającego plik słownika) znajduje się w jednym wyrażeniu. Jedynym powodem otwierania słownika w osobnym wyrażeniu jest użycie konstrukcji `try` z zamykami, co pozwala upewnić się co do zamknięcia pliku niezależnie od sytuacji:

```
// Nadużycie strumieni - nie rób tak!
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(
```

```

        groupingBy(word -> word.chars().sorted()
            .collect(StringBuilder::new,
                (sb, c) -> sb.append((char) c),
                StringBuilder::append).toString()))
        .values().stream()
        .filter(group -> group.size() >= minGroupSize)
        .map(group -> group.size() + ": " + group)
        .forEach(System.out::println);
    }
}
}

```

Jeśli przedstawiony kod wyda Ci się trudny w analizie, nie przejmuj się — nie jesteś jedyny. Jest krótszy, ale również mniej czytelny, szczególnie w przypadku programistów, którzy nie są jeszcze ekspertami od strumieni. **Nadużywanie strumieni czyni programy trudnymi w analizie i konserwacji.**

Na szczęście istnieje kompromis. Poniższy program rozwiązuje ten sam problem i korzysta ze strumieni, ale ich nie nadużywa. W wyniku powstaje kod, który jest zarówno krótszy, jak i bardziej czytelny niż oryginał:

```

// Rozważne użycie strumieni poprawia czytelność i zwięzłość kodu
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }

    // metoda alphabetize jest taka sama jak w wersji oryginalnej
}

```

Nawet jeśli nigdy wcześniej nie korzystałeś ze strumieni, taki zapis kodu programu nie jest szczególnie trudny do zrozumienia. Otwarcie pliku ze słownikiem następuje w bloku try z zasobami. Następnie powstaje strumień składający się ze wszystkich wierszy pliku. Zmienna strumienia nosi nazwę words, aby podkreślić, że każdy element strumienia jest słowem. Potok tego strumienia nie zawiera żadnych operacji pośrednich; operacja kończąca zbiera wszystkie słowa i wstawia je do odwzorowania, które grupuje je na podstawie wersji z literami ułożonymi alfabetycznie (temat 46.). Konstruowany jest dokładnie ten sam obiekt Map co w obu poprzednich wersjach programu. Następnie program otwiera nowy strumień Stream<List<String>> na podstawie wartości zwracanej przez metodę values() odwzorowania. Elementami

tego strumienia są oczywiście grupy anagramowe. Strumień jest poddany filtracji, aby pominąć wszystkie grupy, których rozmiar jest mniejszy od `minGroupSize`. Grupy, które pozostaną, wyświetla operacja kończąca `forEach`.

Zwróć uwagę, że bardzo ostrożnie dobrano nazwy parametrów `lambd`. Parametr `g` powinien tak naprawdę nazywać się `group`, ale wynikowy kod byłby za długi, aby zmieścić się w jednym wierszu w książce. **W przypadku braku jawnych typów ostrożne dobieranie nazw `lambd` to klucz do uzyskania dobrej czytelności potoków strumienia.**

Zauważ, że zamiana wyrazów na postać z posortowaną alfabetycznie listą liter ma miejsce w osobnej metodzie `alphabetize`. Użycie nazwy poprawia czytelność i pozwala pozbyć się szczegółów implementacji z głównej wersji programu. **Metody pomocnicze są w przypadku potoków strumieni istotniejsze w kwestii poprawy czytelności niż w standardowym kodzie iteracyjnym**, bo potokom brakuje jawnej informacji o typie i nazwanych zmiennych tymczasowych.

Choć metodę `alphabetize` można napisać w sposób wykorzystujący strumienie, taka wersja byłaby mniej przejrzysta, trudniejsza do prawidłowego napisania i prawdopodobnie wolniejsza. Wymienione wady wynikają z braku obsługi w Javie strumieni typu prostego `char` (nie sugeruję tu, że Java powinna posiadać strumienie tego typu; byłoby to niewykonalne). Aby pokazać ryzyka związane z przetwarzaniem wartości `char` za pomocą strumieni, rozważmy następujący kod:

```
"Witaj, świecie!".chars().forEach(System.out::print);
```

Moglibyśmy oczekiwać, że kod wyświetli tekst `Witaj, świecie!`, ale po uruchomieniu ukaże nam się tekst `871051169710644323471191051019910510133`. Wynika to z faktu, iż elementami strumienia zwróconego przez `"Hello world!".chars()` nie są wartości typu `char`, ale typu `int`, więc kompilator użył przeciążenia metody `print` dotyczącego typu `int`. Z pewnością zwracanie przez metodę `chars()` strumienia wartości typu `int` jest mylące. **Moglibyśmy** poprawić program, wymuszając rzutowanie na właściwy typ w momencie wywoływania metody:

```
"Witaj, świecie!".chars().forEach(x -> System.out.print((char) x));
```

ale najlepiej po prostu **powstrzymać się od stosowania strumieni do przetwarzania wartości typu `char`.**

Gdy zaczniesz korzystać ze strumieni, z pewnością po pewnym czasie poczujesz potrzebę zamiany wszystkich pętli na strumienie — nie daj się jej. Choć taka zamiana jest możliwa, w wielu przypadkach doprowadzi do zmniejszenia czytelności i utrudni konserwację kodu. Warto przyjąć zasadę, że nawet średnio złożone zadania najlepiej realizować jako połączenie strumieni i iteracji, co doskonale ilustruje przedstawiony przykład z anagramami. Z tego powodu **refaktoryzuj istniejący kod na wersję strumieniową lub korzystaj ze strumieni w nowym kodzie tylko wtedy, gdy to ma sens.**

Jak łatwo się przekonać, analizując zaprezentowane przykłady, potoki strumieni wyrażają powtarzające się obliczenia za pomocą obiektów funkcyjnych (lambda lub referencji do metod), natomiast kod iteracyjny wyraża powtarzające się obliczenia przy użyciu bloków kodu. Istnieje kilka operacji, które można zrealizować za pomocą bloków kodu, ale nie można przy użyciu obiektów funkcyjnych.

- Z poziomu bloku kodu możemy odczytywać lub modyfikować dowolną zmienną lokalną znajdującą się w zasięgu; z poziomu lambda można odczytywać tylko zmienne oznaczone `final` lub efektywnie `final` [JLS 4.12.4] i nie można modyfikować żadnych zmiennych lokalnych.
- W bloku kodu można użyć instrukcji `return`, aby wyjść z metody lub też zastosować instrukcje `break` lub `continue`, aby przerwać iterację pętli. Można też zgłosić dowolny wyjątek przechwytywany, o ile jest zadeklarowany w klauzuli `throws`. W przypadku lambda nie jest dostępne żadne z tych rozwiązań.

Jeśli obliczenia najlepiej wyrazić za pomocą wymienionych wyżej technik, strumienie nie są najlepszą drogą do realizacji zadania. Z drugiej strony strumienie znacząco ułatwiają realizację poniższych zadań:

- przekształcanie w ten sam sposób zbioru elementów;
- filtrowanie zbioru elementów;
- łączenie ciągów elementów za pomocą pojedynczej operacji (np. dodawanie ich, złączanie lub wyliczanie minimum);
- zbieranie ciągu elementów w kolekcji, być może przy jednoczesnym grupowaniu ich na podstawie wskazanego atrybutu;
- poszukiwanie w zbiorze elementów wartości spełniającej pewne kryteria.

Jeśli obliczenia najlepiej wyrazić za pomocą tych technik, strumienie dobrze się nadają do ich zrealizowania.

Jednym z zadań notorycznie trudnych do zrealizowania za pomocą strumieni jest dostęp do odpowiednich elementów z wcześniejszych etapów potoku — gdy odwzorowujemy wartość na pewną inną wartość, tracimy oryginał. Jednym z obejść tego problemu jest odwzorowanie każdej wartości na **obiekt pary** zawierający wartość oryginalną i nową, ale nie jest to satysfakcjonujące rozwiązanie, szczególnie jeśli para obiektów wymagana jest dla wielu etapów potoku. Tworzony w taki sposób kod jest zbyt długi i skomplikowany, co przeczy powodowi stosowania strumieni. Jeżeli to możliwe, lepszym obejściem problemu bywa odwrócenie odwzorowania, jeśli potrzebna jest oryginalna wartość.

Napiszmy na przykład program, który wyświetla pierwsze 20 **liczb pierwszych Mersenne’a**. Przypomnijmy, że liczba Mersenne’a to liczba w postaci $2^p - 1$. Jeśli p jest liczbą pierwszą, odpowiadająca mu liczba Mersenne’a **może** być liczbą pierwszą. W takiej sytuacji jest to liczba pierwsza Mersenne’a. Jako strumień początkowy dla potoku wybierzmy wszystkie liczby pierwsze. Oto metoda zwracająca ten (nieskończony) strumień. Zakładamy, że w celu ułatwienia dostępu do składowych statycznych BigInteger użyliśmy wcześniej importu statycznego:

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

Nazwa metody (`primes`) to rzeczownik w liczbie mnogiej opisujący elementy strumienia. Taka konwencja nazewnictwa jest polecana dla wszystkich metod zwracających strumienie, bo poprawia czytelność potoków strumieni. Metoda używa statycznej fabryki `Stream.iterate`, która przyjmuje dwa parametry: pierwszy element strumienia i funkcję generującą następny element strumienia na podstawie poprzedniego. Oto fragment programu, który wyświetla pierwsze 20 liczb pierwszych Mersenne’a:

```
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

Program to w zasadzie bezpośrednie przełożenie przedstawionego wcześniej opisu: zaczynamy od liczb pierwszych, obliczamy liczbę Mersenne’a, filtrujemy wszystkie wartości poza liczbami pierwszymi (magiczna liczba 50 steruje prawdopodobieństwem w teście liczby pierwszej), ograniczamy wyniki do pierwszych 20 i wyświetlamy je.

Przypuśćmy teraz, że chcemy poprzedzić każdą liczbę pierwszą Mersenne’a wykładnikiem (p). Wartość ta istnieje tylko w początkowym strumieniu, więc operacja kończąca nie ma do niej dostępu. Na szczęście istnieje prosty sposób na obliczenie wykładnika liczby Mersenne’a przez odwrócenie działania, które miało miejsce w pierwszej operacji pośredniej. Wykładnik jest po prostu liczbą bitów w reprezentacji binarnej, więc operacja kończąca realizująca całe zadanie będzie miała postać:

```
.forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

Istnieje wiele zadań, w których nie można na początku określić, czy lepiej skorzystać ze strumieni, czy iteracji. Rozważmy zadanie inicjalizacji nowej talii kart. Załóżmy, że `Card` (karta) to niezmienna klasa wartości, która zawiera w sobie wartości `Rank` (figura) i `Suit` (kolor), które są typami wyliczeniowymi. Zadanie to jest przedstawicielem dowolnego zadania, które wymaga wyliczenia wszystkich par elementów

pochodzących z dwóch wybranych zbiorów. Matematycy nazywają taki przypadek **iloczynem kartezjańskim** dwóch zbiorów. Oto implementacja iteracyjna wykorzystująca zagnieżdżone pętle for-each. Jej postać z pewnością wygląda znajomo:

```
// Iteracyjne obliczenie iloczynu kartezjańskiego
private static List<Card> newDeck() {
    List<Card> result = new ArrayList<>();
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            result.add(new Card(suit, rank));
    return result;
}
```

Poniżej znajduje się implementacja bazująca na strumieniu, która korzysta z operacji pośredniej flatMap. Operacja ta odwzorowuje każdy element strumienia na strumień, a następnie łączy wszystkie nowe strumienie w jeden strumień (można powiedzieć, że go **splaszcz**). Zwróć uwagę, że implementacja zawiera zagnieżdżoną lambda (pogrubiony fragment):

```
// Obliczenie iloczynu kartezjańskiego bazujące na strumieniach
private static List<Card> newDeck() {
    return Stream.of(Suit.values())
        .flatMap(suit ->
            Stream.of(Rank.values())
                .map(rank -> new Card(suit, rank)))
        .collect(toList());
}
```

Która z dwóch przedstawionych wersji metody newDeck jest lepsza? Wszystko zależy od osobistych preferencji i środowiska, w których tworzony jest kod. Pierwsza wersja jest prostsza i wielu będzie wydawała się bardziej naturalna. Duży odsetek programistów Javy będzie w stanie ją zrozumieć i w razie potrzeby poprawić, ale dla niektórych osób bardziej komfortowa będzie wersja druga (bazująca na strumieniach). Jest nieco bardziej zwięzła i nie tak trudna do zrozumienia, jeśli w miarę dobrze zna się strumienie i programowanie funkcyjne. Jeśli nie wiesz, którą wersję preferujesz, bezpiecznym wyborem będzie pozostanie przy wersji iteracyjnej. Jeżeli lubisz wersję strumieniową i sądzisz, że inni programiści w zespole będą w stanie ją zrozumieć i poprawić, skorzystaj z niej.

Podsumujmy. Niektóre zadania najlepiej wykonuje się przy użyciu strumieni, a inne za pomocą iteracji. Wiele zadań najlepiej wykonać, łącząc obie techniki. Nie istnieją żadne sztywne zasady wyboru jednego lub drugiego rozwiązania, ale istnieją pewne ogólne wskazówki. W wielu przypadkach już na pierwszy rzut oka wiadomo, które podejście jest lepsze, choć zdarzają się sytuacje, gdy nie jest to takie łatwe. **Jeśli nie potrafisz zdecydować, czy podejście iteracyjne lub strumieniowe sprawdzi się w konkretnym zadaniu, spróbuj obu i przekonaj się, które działa lepiej.**

Temat 46. Stosuj w strumieniach funkcje bez efektów ubocznych

Jeśli dopiero zaczynasz przygodę ze strumieniami, niełatwo się do nich przyzwyczaić. Często już samo wyrażenie obliczeń jako potoku strumienia potrafi być trudne. Gdy operacja się powiedzie, program będzie działał, ale możesz dojść do wniosku, że korzyść z takiego podejścia jest niewielka. Strumień to nie tylko API, to paradygmat bazujący na programowaniu funkcyjnym. Aby móc w pełni wykorzystać siłę wyrazu, szybkość, a czasem nawet możliwość zrównoleglenia działań, musisz nie tylko użyć API, ale także postępować zgodnie z paradygmatem.

Najważniejszym elementem paradygmatu strumieni jest takie zdefiniowanie struktury obliczeń, aby stanowiły ciąg przekształceń, gdzie wynik każdego z etapów jest tak blisko **funkcji czystej** realizowanej na podstawie wyniku poprzedniego etapu, jak to tylko możliwe. Funkcja czysta to funkcja, której wynik zależy tylko i wyłącznie od danych wejściowych — nie ma znaczenia żaden zmienny stan, a sama funkcja nie aktualizuje żadnego stanu. Aby to osiągnąć, wszystkie obiekty funkcyjne przekazywane do operacji na strumieniu (czyli operacje pośrednie i kończące) powinny być pozbawione efektów ubocznych.

Czasem zdarza się znaleźć kod używający strumieni w poniższy sposób. Ten konkretny przykład buduje tablicę częstości słów w pliku tekstowym:

```
// Używa API strumieni, ale niezgodnie z paradygmatem - nie rób tak!
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
    words.forEach(word -> {
        freq.merge(word.toLowerCase(), 1L, Long::sum);
    });
}
```

Dlaczego ten kod jest nieprawidłowy? Przecież używa strumieni, lambd i referencji do metod, a uzyskiwany wynik jest poprawny. Po prostu nie jest to tak naprawdę kod strumieniowy — to kod iteracyjny kryjący się pod płaszczykiem kodu strumieniowego. Nie korzysta z żadnych zalet API strumieni, a także jest nieco dłuższy, trudniejszy w utrzymaniu i mniej czytelny niż równoważna mu wersja iteracyjna. Problem wynika z faktu, iż kod wykonuje całą pracę w operacji kończącej `forEach`, wykorzystując lambdę zmieniającą zewnętrzny stan (tablicę częstości). Operacja `forEach` wykonująca inne zadanie niż prezentacja wyniku obliczeń zrealizowanych przez strumień to „zły zapach” w kodzie. Podobna sytuacja dotyczy lambdy modyfikującej stan. Jak więc wygląda właściwa wersja strumieniowa?

```
// Właściwe użycie strumieni do inicjalizacji tablicy częstości
Map<String, Long> freq;
try (Stream<String> words = new Scanner(file).tokens()) {
```

```

    freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
}

```

Powyższy kod wykonuje to samo zadanie co wcześniejszy, ale tym razem w odpowiedni sposób korzysta ze strumieni. Jest krótki i przejrzysty. Dlaczego więc ktokolwiek tworzyłby inną wersję? Ponieważ inna wersja używa znajomych narzędzi. Programiści Javy znają pętle `for-each`, a operacja kończąca `forEach` bardzo ją przypomina. W rzeczywistości operacja `forEach` ma jedne z gorszych możliwości z wielu operacji kończących i jest najmniej przyjazna strumieniom. Jest jawnie iteracyjna, więc nie pozwala na zrównoleglenie. **Operację `forEach` należy stosować tylko do zgłoszenia wyników obliczeń strumienia, a nie do ich realizacji.** Czasami zdarza się, że operacja `forEach` okazuje się wygodnym środkiem realizacji innych zadań, np. dodawania wyników obliczeń do istniejącej już kolekcji.

Ulepszony kod wykorzystuje **kolekcionera**, który jest nową koncepcją wymagającą poznania, jeśli chce się używać strumieni. API `Collectors` potrafi przytłoczyć — zawiera trzydzieści dziewięć metod, z których niektóre mają aż pięć parametrów typów. Dobrą wiadomością jest to, że większość zalet tego API można wykorzystać bez zajmowania się jego złożonością. Na początku zignoruj interfejs `Collector` i potraktuj kolekcionera jako niewidoczny obiekt, który zawiera w sobie strategię **redukcji**. W tym kontekście redukcja oznacza łączenie elementów strumienia w jeden obiekt. Obiektem tworzonym przez kolekcionera jest najczęściej kolekcja (stąd też wzięła się nazwa mechanizmu).

Kolekcionery zbierające elementy strumienia do prawdziwych obiektów `Collection` są proste w użyciu. Istnieją trzy główne kolekcionery: `toList()`, `toSet()` i `toCollection(↪(collectionFactory))`. Zwracają one odpowiednio listę, zbiór i typ kolekcji wskazany przez programistę. Dzięki tej wiedzy bardzo łatwo napiszemy potok strumienia, który wydobywa 10 najważniejszych wyrazów z tablicy częstości:

```

// Potok pobierający 10 najważniejszych wyrazów z tablicy częstości
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    .collect(toList());

```

Zwróć uwagę, że metody `toList` nie poprzedza nazwa klasy `Collectors`. **Wynika to z faktu, iż przyjęło się importować statycznie wszystkich członków klasy `Collectors`, aby w ten sposób uczynić potoki strumienia czytelniejszymi.**

Jednym z bardziej zaawansowanym fragmentem powyższego kodu jest komparator przekazywany do metody `sorted`: `comparing(freq::get).reversed()`. Metoda `comparing` to metoda konstruująca komparator (temat 14.), która przyjmuje funkcję wydobywającą klucz. Funkcja przyjmuje słowo, a „wydobywanie” polega po prostu na pobraniu wartości klucza — związana referencja do metody `freq::get` wyszukuje

słowo w tablicy częstości i zwraca liczbę wystąpień słowa w pliku. Dodatkowo wywołujemy metodę `reversed` komparatora, aby posortować wyrazy od najczęściej do najrzadziej występujących. Teraz pozostaje już tylko ograniczyć strumień do 10 elementów i otrzymać wyniki w postaci listy.

Powyższy kod używa metody `stream` dla `Scanner`, aby otrzymać strumień operujący na skanerze. Metodę tę wprowadzono dopiero w Javie 9. Jeśli korzystasz z wcześniejszego wydania języka, możesz przekształcić skaner, który implementuje interfejs `Iterator`, na strumień za pomocą adaptera podobnego do tego z tematu 47. (`streamOf(Iterable<E>)`).

Do czego służy pozostałe trzydzieści sześć metod z `Collectors`? Większość z nich istnieje, aby zapewnić właściwe zbieranie wyników strumienia do odwzorowań, co jest zadaniem dużo bardziej skomplikowanym niż zbieranie do zwykłych kolekcji. Każdy element strumienia jest powiązany z kluczem i wartością, a wiele elementów strumienia może dotyczyć tego samego klucza.

Najprostszym kolekcjonerem dotyczącym odwzorowań jest `toMap(keyMapper, valueMapper)`, który przyjmuje dwie funkcje. Pierwsza z nich zamienia element strumienia na klucz, a druga na wartość. Użyliśmy tego kolekcjonera w implementacji `fromString` w temacie 34., aby uzyskać odwzorowanie z postaci tekstowej wyliczenia na rzeczywiste wyliczenie:

```
// Użycie kolekcjonera toMap w celu uzyskania odwzorowania z tekstu
// na wyliczenie
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));
```

Ta prosta postać `toMap` sprawdza się doskonale, jeśli każdy element odwzorowania strumienia ma własny, unikatowy klucz. Jeśli wiele elementów strumienia zostanie odwzorowanych na ten sam klucz, potok zakończy działanie zwróceniem wyjątku `IllegalStateException`.

Bardziej wyrafinowane wersje `toMap`, a także metoda `groupingBy` pozwalają na określenie różnych strategii radzenia sobie z kolizjami. Jednym z rozwiązań jest przekazanie do metody `toMap` **funkcji łączącej** jako dodatku do funkcji odwzorowującej na klucz i wartość. Funkcja łącząca to interfejs `BinaryOperator<V>`, w którym `V` to typ wartości odwzorowania. Wszystkie dodatkowe wartości powiązane z kluczem są łączone z istniejącą wartością przy użyciu funkcji łączącej. Jeśli na przykład funkcja łącząca to mnożenie, końcową wartością dla danego klucza będzie iloczyn wszystkich wartości posiadających ten sam klucz.

Trójargumentowa wersja `toMap` przydatna jest również do utworzenia odwzorowania z klucza na wybrany element powiązany z tym kluczem. Przypuśćmy, że mamy strumień albumów muzycznych różnych wykonawców i chcemy wykonać

odzworowanie z artysty na najlepiej sprzedający się album. Zadanie to zrealizuje odpowiedni kolekcjoner:

```
// Kolektor generujący odzworowanie z klucza do wybranego elementu klucza
Map<Artist, Album> topHits = albums.collect(
    toMap(Album::artist, a->a, maxBy(comparing(Album::sales))));
```

Komparator używa statycznej metody fabrycznej `maxBy`, która została zaimportowana statycznie z `BinaryOperator`. Metoda ta zamienia `Comparator<T>` na `BinaryOperator<T>`, który znajduje maksymalną wartość na podstawie podanego komparatora. W tym przypadku komparator zwraca metoda konstruuująca komparator `comparing`, która otrzymuje funkcję wydobywającą klucz o nazwie `Album::sales`. Rozwiązanie jest nieco zawile w opisie, ale kod okazuje się czytelny. Cały kod informuje kompilator „zamień strumień albumów na odzworowanie, które przekształca każdego wykonawcę na jego najlepiej sprzedający się album”. To stwierdzenie zadziwiająco wiernie oddaje rzeczywisty kod.

Innym zastosowaniem trójargumentowej postaci `toMap` jest kolekcjoner, który wymusza użycie reguły „ostatni zapis wygrywa” w przypadku wystąpienia kolizji. Dla wielu strumieni wynik takiej operacji będzie niedeterministyczny, ale jeśli wszystkie wartości przypisywane do klucza są takie same lub wszystkie są dopuszczalne, takie działanie kolekcjonera może okazać się wystarczające:

```
// Kolektor wymuszający zasadę ostatni zapis wygrywa
toMap(keyMapper, valueMapper, (v1, v2) -> v2)
```

Trzecia i ostatnia wersja `toMap` przyjmuje czwarty argument, którym jest fabryka obiektów `Map`, która pozwala wskazać konkretną implementację odzworowania, jeśli na przykład chcemy użyć `EnumMap` lub `TreeMap`.

Istnieją również odmiany trzech pierwszych wersji `toMap` o nazwie `toConcurrentMap`, które działają wydajnie w sposób równoległy i tworzą instancje klasy `ConcurrentHashMap`.

Poza metodą `toMap` API `Collectors` oferuje również metodę `groupingBy`, która zwraca kolekcjonery tworzące odzworowania grupujące elementy do poszczególnych kategorii na podstawie funkcji klasyfikującej. Funkcja klasyfikująca przyjmuje element i zwraca kategorię, do której przynależy. Kategoria ta stanie się kluczem w odzworowaniu. Najprostsza wersja metody `groupingBy` przyjmuje tylko klasyfikator i zwraca odzworowanie, którego wartościami są listy wszystkich elementów należących do poszczególnych kategorii. Oto kolekcjoner, z którego skorzystaliśmy w programie `Anagram` z tematu 45., aby wygenerować odzworowanie z wyrazów z literami ułożonymi alfabetycznie na listę wyrazów współdzielących ten sam anagram:

```
words.collect(groupingBy(word -> alphabetize(word)))
```

Jeśli chcemy, aby metoda `groupBy` zwracała kolekcjonera, który produkuje odwzorowanie z wartościami innymi niż listy, możemy wskazać **kolekcjonera wartości** jako dodatek do klasyfikatora. Taki kolekcjoner wartości generuje wartość na podstawie strumienia zawierającego wszystkie elementy z danej kategorii. Najprostszym sposobem użycia tego parametru jest przekazanie `toSet()`, co spowoduje, że wartości w oryginalnym odwzorowaniu nie będą listami, ale zbiorami.

Alternatywne rozwiązanie polega na przekazaniu `toCollection(collectionFactory)`, co pozwala określić samemu rodzaj kolekcji, do której trafią elementy należące do tej samej kategorii. W ten sposób możemy wskazać dowolny rodzaj kolekcji. Innym przykładem użycia dwuargumentowej wersji metody `groupBy` jest przekazanie `counting()` jako kolekcjonera wartości. Wynikiem w takiej sytuacji będzie odwzorowanie z **liczbą** elementów należących do poszczególnych kategorii zamiast z wartościami należącymi do poszczególnych kategorii. Właśnie to rozwiązanie pojawiło się w przykładzie z tablicą częstości wyrazów na początku tego tematu:

```
Map<String, Long> freq = words
    .collect(groupingBy(String::toLowerCase, counting()));
```

Trzecia wersja metody `groupBy` poza kolekcjonerem wartości pozwala zdefiniować również fabrykę dla odwzorowania. Zwróć uwagę, że metoda ta łamie podstawową zasadę teleskopowej listy argumentów — parametr `mapFactory` znajduje się przed parametrem `downStream` (zamiast po nim). Ten rodzaj grupowania zapewnia pełną kontrolę zarówno nad tworzonym obiektem odwzorowania, jak i nad kolekcjami zawartymi jako wartości. W ten sposób jesteśmy na przykład w stanie wskazać kolekcjonera zwracającego obiekt `TreeMap`, którego wartościami są obiekty `TreeSet`.

Metoda `groupByConcurrent` udostępnia wszystkie trzy odmiany metody `groupBy`. Odmiany te działają wydajnie w sposób zrównoleglony i tworzą obiekty `ConcurrentHashMap`. Istnieje również rzadko używana metoda podobna do `groupBy` o nazwie `partitioningBy`. Jako metodę klasyfikującą przyjmuje predykat i zwraca odwzorowanie, którego kluczami są wartości typu `Boolean`. Istnieją dwa przeciążenia tej metody, a jedno z nich przyjmuje kolekcjonera wartości jako dodatek do predykatu.

Kolekcjonery zwracane przez metodę `counting` są przeznaczone **tylko** do stosowania jako kolekcjonery wartości. Tę samą funkcjonalność oferuje klasa `Stream` w sposób bezpośredni poprzez metodę `count`, więc **nie ma powodu, aby tworzyć kod typu `collect(counting())`**. Istnieje jeszcze piętnaście innych metod `Collectors` o podobnych właściwościach. Dziewięć z tych metod zaczyna się od nazw `summing`, `averaging` i `summarizing` (ich funkcjonalności są dostępne dla odpowiednich strumieni dotyczących typów prostych). Znajdziemy tu również wszystkie przeciążenia metody `reducing`, jak również metody `filtering`, `mapping`, `flatMapMapping`

i `collectingAndThen`. Większość programistów może z czystym sumieniem pominąć wiele z wymienionych tu metod. Z perspektywy projektanta tego rodzaju kolekcjonery reprezentują próbę częściowego powielenia funkcjonalności strumieni w kolekcjonerach, aby kolekcjonery wartości działały podobnie do „ministrumieni”.

Istnieją jeszcze trzy metody `Collectors`, o których do tej pory nie wspomnieliśmy. Choć znajdują się we wspomnianej klasie, nie mają związku z kolekcjami. Pierwsze dwie to `minBy` i `maxBy`, które przyjmują komparator i zwracają minimalny lub maksymalny element strumienia na podstawie wskazań przekazanego komparatora. To niewielkie uogólnienia metod `min` i `max` z interfejsu `Stream`. Stanowią one kolekcjonerski odpowiednik operatorów binarnych zwracanych przez metody takie jak `BinaryOperator`. Przypomnijmy, że w przykładzie z najlepiej sprzedającymi się albumami użyliśmy `BinaryOperator.maxBy`.

Ostatnią z metod `Collectors` jest `joining`, która działa tylko dla strumieni instancji `CharSequence` takich jak teksty. W wersji bezparametrowej zwraca kolekcjonera, który po prostu łączy ze sobą poszczególne elementy. W wersji z jednym parametrem przyjmuje wartość `delimiter` i zwraca kolekcjonera, który łączy ze sobą poszczególne elementy, wstawiając między nimi przekazany tekst. Jeśli wartością parametru będzie przecinek, kolekcjoner spowoduje zwrócenie tekstu z wartościami oddzielonymi przecinkami (pamiętaj, że taki tekst będzie dwuznaczny, jeśli którykolwiek z elementów strumienia również zawiera przecinki). Wersja trójparametrowa przyjmuje poza elementem wstawianym pomiędzy również przedrostek i przyrostek. Wynikowy kolekcjoner wygeneruje w takiej sytuacji (po użyciu odpowiednich wartości) tekst przypominający wyświetlanie zawartości kolekcji, np. `[przybyłem, zobaczyłem, zwyciężyłem]`.

Podsumujmy. Podstawą programowania z użyciem potoków kolekcji są obiekty funkcyjne bez efektów ubocznych. Dotyczy to każdego z wielu obiektów funkcyjnych przekazywanych do strumieni i powiązanych obiektów. Operacja kończąca `forEach` powinna służyć tylko i wyłącznie do zgłaszania wyników obliczeń przeprowadzonych przez strumień, a nie do wykonywania działań. Aby poprawnie korzystać ze strumieni, trzeba poznać kolekcjonery. Najważniejszymi fabrykami kolekcjonerów są metody `toList`, `toSet`, `toMap`, `groupingBy` i `joining`.

Temat 47. Zwracaj kolekcje zamiast strumieni

Wiele metod zwraca ciągi elementów. Przed Javą 8 oczywistym typem zwracanym przez tego rodzaju metody były interfejsy kolekcji `Collection`, `Set` i `List`, a także interfejsy `Iterable` lub typy tablicowe. Najczęściej dosyć łatwo można było podjąć

decyzję co do zwracanego typu. Interfejsy kolekcji stanowiły ogólnie przyjętą normę. Jeśli metoda istniała tylko i wyłącznie na potrzeby pętli for-each lub zwracany ciąg nie był w stanie zaimplementować niektórych metod interfejsu Collection (najczęściej contains(Object)), korzystało się z interfejsu Iterable. Jeśli zwracanymi wartościami były typy proste lub pojawiała się konieczność zapewnienia maksymalnej wydajności, używało się tablic. W Javie 8 do platformy dołączyły strumienie, co znacząco utrudniło zadanie wyboru właściwego typu dla zwracanych ciągów elementów.

Usłyszeć można głosy, że obecnie to strumienie są oczywistym wyborem, jeśli musimy zwrócić ciąg elementów, ale zgodnie z tematem 45. strumienie nie spowodowały odejścia iteracji w niebyt. Jeśli API zwraca tylko strumień, a użytkownicy chcą iterować po zwróconym ciągu za pomocą pętli for-each, z pewnością nie będą zadowoleni. To szczególnie frustrujące, bo interfejs Stream zawiera jedyną metodę abstrakcyjną z interfejsu Iterable, a specyfikacja Stream dla tej metody jest zgodna z Iterable. Jedyną rzeczą powstrzymującą programistów korzystających z pętli for-each przed iteracją po strumieniu jest to, że Stream nie rozszerza interfejsu Iterable.

Niestety, nie istnieje dobry sposób obejścia tego problemu. Na pierwszy rzut oka wydaje się, że przekazanie referencji do metody iterator ze Stream powinno zadziałać. Powstający w takiej sytuacji kod jest nieco zaciemniony i dziwaczny, ale nie jest pozbawiony sensu:

```
// Kompilacja nie powiedzie się z powodu ograniczeń wnioskowania
// typów w języku Java
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    // przetwórz proces
}
```

Próba kompilacji takiego kodu zakończy się niestety niepowodzeniem i poniższym błędem:

```
Test.java:6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    ^
```

Aby umożliwić kompilację kodu, musimy rzutować referencję do metody na odpowiednią parametryzowaną wersję Iterable:

```
// Pokraczne obejście problemu pozwalające iterować po strumieniu
for (ProcessHandle ph : (Iterable<ProcessHandle>)
    ProcessHandle.allProcesses()::iterator)
```

Tym razem kod działa, ale jest jeszcze bardziej zawiły niż poprzedni. Lepszym obejściem problemu byłoby użycie metody działającej jako adapter. JDK nie udostępnia takiej metody, ale łatwo ją napisać, wykorzystując tę samą technikę, użytą

w powyższym przykładzie. Zwróć uwagę, że tym razem rzutowanie nie jest konieczne, bo w tym kontekście mechanizm wnioskowania typów Javy działa poprawnie:

```
// Adapter ze Stream<E> na Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}
```

Dzięki takiemu adapterowi możemy łatwo iterować po każdym strumieniu przy użyciu pętli for-each:

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
    // przetwórz proces
}
```

Zwróć uwagę, że strumieniowa wersja programu Anagram z tematu 34. używa metody `Files.lines`, aby odczytać słownik, a wersja iteracyjna używa skanera. Metoda `Files.lines` jest lepsza od skanera, bo ten drugi po cichu połyka wszystkie wyjątki napotkane w trakcie odczytu pliku. W idealnej sytuacji w wersji iteracyjnej również skorzystalibyśmy z `Files.lines`. To rodzaj kompromisu, na który musi przystać programista, jeśli API zapewnia tylko strumieniowy dostęp do ciągu elementów, a chce iterować po elementach za pomocą instrukcji for-each.

Z drugiej strony programista, który chce przetworzyć ciąg elementów przy użyciu strumienia, będzie zawiedziony, gdy okaże się, że API udostępnia tylko interfejs `Iterable`. Także i w tym przypadku JDK nie udostępnia odpowiedniego adaptera, ale napisanie własnego nie jest kłopotliwe:

```
// Adapter z Iterable<E> na Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

Jeśli tworzysz metodę, która zwraca ciąg obiektów, i wiesz, że jedynym użytkownikiem takiego ciągu będzie potok strumienia, oczywiście zwróć strumień. Podobnie, jeśli metoda zwracająca pewien ciąg będzie używana tylko przez iterację, powinna zwrócić `Iterable`. Jeżeli jednak powstaje publiczne API zwracające ciąg elementów, warto zadowolić zarówno użytkowników potrzebujących strumienia, jak i tych chcących skorzystać z instrukcji for-each, chyba że posiadasz informację, że zdecydowana większość użytkowników będzie skłonna korzystać tylko z jednego z tych mechanizmów.

Interfejs `Collection` jest podtypem `Iterable` i posiada metodę `stream`, więc zapewnia zarówno iterację, jak i strumień. Z tego powodu **interfejs `Collection` lub jego bardziej odpowiedni podtyp to najczęściej najlepszy zwracany typ dla publicznych metod zwracających ciągi elementów**. Tablice również zapewniają łatwy dostęp zarówno do iteracji, jak i strumienia poprzez metody `Arrays.asList` i `Stream.of`. Jeśli zwracana sekwencja jest na tyle mała, aby łatwo zmieścić się

w pamięci, najlepiej przy zwracaniu wartości użyć jednej ze standardowych implementacji kolekcji, np. `ArrayList` lub `HashSet`. Pamiętaj jednak, aby **nie przechowywać dużych ciągów w pamięci tylko i wyłącznie na potrzeby ich zwrócenia jako kolekcji**.

Jeśli zwracany ciąg elementów jest duży, ale łatwo go przedstawić w sposób zwężły, rozważ zaimplementowanie kolekcji specyficznej dla tego ciągu. Przypuśćmy, że chcemy zwrócić **zbiór potęgowy** wskazanego zbioru, który składa się ze wszystkich jego podzbiorów. Zbiór potęgowy dla $\{a, b, c\}$ ma postać $\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Jeśli zbiór ma n elementów, jego zbiór potęgowy ma 2^n elementów. Z tego powodu nawet nie warto zastanawiać się nad przechowywaniem zbioru potęgowego w standardowej implementacji kolekcji. Łatwo jednakże zaimplementować własną kolekcję wykonującą właściwe zadanie, jeśli tylko skorzysta się z pomocy `AbstractList`.

Sztuczka polega na użyciu indeksu każdego elementu zbioru potęgowego jako wektora bitowego, w którym n -ty bit elementu wskazuje istnienie bądź jego brak n -tego elementu w zbiorze źródłowym. Innymi słowy, istnieje naturalne odwzorowanie między liczbami binarnymi od 0 do $2^n - 1$ a zbiorem potęgowym wykonanym na podstawie zbioru n elementowego. Oto kod:

```
// Zwraca zbiór do potęgi dla zbioru wejściowego jako własną kolekcję
public class PowerSet {
    public static final <E> Collection<Set<E>> of(Set<E> s) {
        List<E> src = new ArrayList<>(s);
        if (src.size() > 30)
            throw new IllegalArgumentException("Zbiór zbyt duży " + s);
        return new AbstractList<Set<E>>() {
            @Override public int size() {
                return 1 << src.size(); // 2 do potęgi srcSize
            }

            @Override public boolean contains(Object o) {
                return o instanceof Set && src.containsAll((Set)o);
            }

            @Override public Set<E> get(int index) {
                Set<E> result = new HashSet<>();
                for (int i = 0; index != 0; i++, index >>= 1)
                    if ((index & 1) == 1)
                        result.add(src.get(i));
                return result;
            }
        };
    }
}
```

Zwróć uwagę, że `PowerSet.of` zgłasza wyjątek, jeśli zbiór wejściowy ma więcej niż 30 elementów. Podkreśla to wadę użycia `Collection` zamiast `Stream` lub `Iterable` jako zwracanego typu, bo kolekcja posiada zwracającą typ `int` metodę `size`, co ogranicza długość zwracanego ciągu do `Integer.MAX_VALUE`, czyli $2^{31} - 1$. Choć specyfikacja `Collection` dopuszcza, aby metoda `size` zwróciła $2^{31} - 1$, jeśli kolekcja jest dłuższa, a nawet nieskończona, nie jest to szczególnie satysfakcjonujące rozwiązanie.

Aby napisać implementację `Collection` na podstawie `AbstractCollection`, konieczne jest zaimplementowanie tylko dwóch metod poza tą wymaganą przez `Iterable`: `contains` i `size`. Bardzo często łatwo napisać wydajne implementacje obu metod. Jeśli tak nie jest, bo na przykład nie jesteśmy w stanie określić zawartości ciągu przed rozpoczęciem iteracji, lepiej jest zwrócić iterację lub strumień (wybierz wersję najbardziej naturalną dla danych okoliczności). Oczywiście nic nie stoi na przeszkodzie, aby zwracać obie wersje po zastosowaniu dwóch różnych metod.

Zdarzają się przypadki, gdy wybór zwracanego typu uzależniony jest przede wszystkim od łatwości implementacji. Przypuśćmy, że chcemy napisać metodę zwracającą wszystkie podlisty listy wejściowej (w sposób ciągły). Choć wygenerowanie podlist i umieszczenie ich w standardowej kolekcji wymaga napisania tylko trzech wierszy kodu, pamięć potrzebna do przechowywania danych jest proporcjonalna do kwadratu rozmiaru listy źródłowej. Choć nie jest to tak zła wiadomość jak zbiór potęgowy, który rośnie wykładniczo, również nie jest do zaakceptowania. Implementacja własnej kolekcji, tak jak to zrobiliśmy dla zbioru potęgowego, byłaby żmudna w realizacji, głównie z powodu braku w JDK szkieletowej implementacji `Iterator`.

Z drugiej strony można łatwo zaimplementować strumień ze wszystkimi podlistami oryginalnej listy, choć wymaga to dodatkowych przygotowań. Nazwijmy podlistę zawierającą pierwszy element listy **przedrostkiem** listy. Na przykład przedrostkami dla (a, b, c) są (a) , (a, b) i (a, b, c) . W podobny sposób zdefiniujemy podlistę zawierającą ostatni element listy jako przyrostek, więc dla (a, b, c) będą to (a, b, c) , (b, c) i (c) . Sztuczka polega na tym, że podlisty listy są po prostu przyrostkami przedrostków (lub też przedrostkami przyrostków) oraz pustą listą. Ta obserwacja prowadzi do czytelniej i stosunkowo zwięzłej implementacji:

```
// Zwraca strumień ze wszystkimi podlistami listy wejściowej
public class SubLists {
    public static <E> Stream<List<E>> of(List<E> list) {
        return Stream.concat(Stream.of(Collections.emptyList()),
                               prefixes(list).flatMap(SubLists::suffixes));
    }

    private static <E> Stream<List<E>> prefixes(List<E> list) {
        return IntStream.rangeClosed(1, list.size())
            .mapToObj(end -> list.subList(0, end));
    }
}
```



```

private static <E> Stream<List<E>> suffixes(List<E> list) {
    return IntStream.range(0, list.size())
        .mapToObj(start -> list.subList(start, list.size()));
}

```

Metoda `Stream.concat` służy do dodania pustej listy do zwróconego strumienia. Zauważ, że metoda `flatMap` (temat 45.) posłużyła do wygenerowania pojedynczego strumienia składającego się ze wszystkich przedrostków wszystkich przyrostków. Zwróć również uwagę, że za odpowiednie generowanie przedrostków i przyrostków odpowiada odwzorowanie strumienia kolejnych wartości `int` zwracane przez `IntStream.range` i `IntStream.rangeClosed`. Ten idiom to w zasadzie strumieniowy odpowiednik standardowej pętli `for` wykorzystującej indeksy w postaci liczb całkowitych. Implementacja `podlist` jest więc podobna w działaniu do prostej, zagnieżdżonej pętli `for`:

```

for (int start = 0; start < src.size(); start++)
    for (int end = start + 1; end <= src.size(); end++)
        System.out.println(src.subList(start, end));

```

Można przekształcić powyższą pętlę `for` bezpośrednio w strumień. Wynik tej operacji będzie bardziej zwięzły niż wcześniejsza implementacja, ale i mniej czytelny. Przypomina on w działaniu kod strumieniowy wyliczający iloczyn kartezjański w temacie 45.:

```

// Zwraca strumień ze wszystkimi podlistami listy wejściowej
public static <E> Stream<List<E>> of(List<E> list) {
    return IntStream.range(0, list.size())
        .mapToObj(start ->
            IntStream.rangeClosed(start + 1, list.size())
                .mapToObj(end -> list.subList(start, end)))
        .flatMap(x -> x);
}

```

Podobnie jak wcześniejsza pętla `for`, także i ten kod **nie** emituje pustej listy. Aby naprawić ten brak, musimy użyć metody `concat` (jak we wcześniejszym przykładzie) lub zastąpić 1 wartością `(int) Math.signum(start)` w wywołaniu `rangeClosed`.

Obie przedstawione strumieniowe implementacje `podlist` są dobre, ale obie będą wymagały od użytkowników wprowadzenia adaptera ze strumienia do iteratora lub zastosowania strumienia tam, gdzie bardziej naturalna wydawałaby się iteracja. Tego rodzaju adapter nie tylko zaśmieca kod kliencki, ale również spowalnia pętlę 2,3 razy w trakcie testów na moim komputerze. Specjalnie przygotowana implementacja wykorzystująca kolekcje (nie jest tu prezentowana) jest zdecydowanie dłuższa jako kod, ale działa na moim komputerze 1,4 razy szybciej niż wersja strumieniowa.

Podsumujmy. Pisząc metodę, która zwraca ciąg elementów, pamiętaj, że niektórzy jej użytkownicy będą preferowali przetwarzanie danych jako strumienia, a inni będą woleli użyć standardowej iteracji. Staraj się dogodzić obu grupom. Jeśli zwrócenie kolekcji jest możliwe, zrób to. Jeśli elementy już znajdują się w kolekcji lub też ich liczba jest na tyle niewielka, że utworzenie nowej kolekcji jest dopuszczalne, zwróć standardową kolekcję jako `ArrayList`. W przeciwnym razie rozważ zwrócenie własnej kolekcji, jak to zrobiliśmy dla zbioru potęgowego. Jeśli zwrócenie kolekcji nie jest sensowne, zwróć strumień lub iterator w zależności od tego, co wydaje się bardziej naturalne. Jeśli w przyszłych wydaniach Javy deklaracja interfejsu `Stream` zostanie wzbogacona o rozszerzenie interfejsu `Iterable`, będziesz mógł z czystym sumieniem zwracać strumienie, bo będzie je można wykorzystać również w iteracji.

Temat 48. Ostrożnie korzystaj ze strumieni zrównoleglonych

Wśród najpopularniejszych języków programowania Java zawsze pojawiała się w czołówce tych, które starały się ułatwić zadanie programowania współbieżnego. Gdy wydano Javę w 1996 roku, na poziomie języka istniało wsparcie dla wątków, synchronizacji i obsługi `wait` oraz `notify`. Java 5 wprowadziła bibliotekę `java.util.concurrent`, kolekcje współbieżne i framework `Executor`. Java 7 wzbogaciła język o pakiet dotyczący mechanizmu rozdzielania i łączenia, czyli wysoko-wydajnego sposobu na dekompozycję zadań i ich zrównoleglone wykonywanie. Java 8 wprowadziła strumienie, które można zrównoleglić jednym wywołaniem metody `parallel`. Tworzenie programów współbieżnych w Javie staje się coraz prostsze, ale pisanie współbieżnych programów, które są jednocześnie poprawne i szybkie, okazuje się trudniejsze niż kiedykolwiek wcześniej. Naruszenia bezpieczeństwa i żywotności są częścią programowania współbieżnego i zrównoleglone potoki strumieni nie są tu wyjątkiem.

Rozważmy program z tematu 45.:

```
// Strumieniowa wersja programu generującego pierwszych 20 liczb pierwszych
// Mersenne'a
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

Na moim komputerze program od razu zaczyna wyświetlać liczby pierwsze, a jego pełne wykonanie zajmuje 12,5 sekundy. Przypuśćmy, że w nieco naiwny sposób spróbuję go przyspieszyć, dodając wywołanie metody `parallel()` do potoku strumienia. Jak sądzisz, co się stanie z wydajnością? Czy uzyskamy kilkuprocentowe przyspieszenie? Czy może będzie to kilkuprocentowe spowolnienie? Niestety, taki program przestanie cokolwiek wyświetlać, ale poziom zajętości CPU skoczy do 90% i pozostanie taki już w nieskończoność (**naruszenie żywotności**). Choć program być może zakończyłby się po jakimś czasie, nie miałem wystarczająco cierpliwości, aby to sprawdzić — zatrzymałem go po półgodzinie.

Co się stało? Biblioteka strumieni nie wie, w jaki sposób zrównoleglić taki potok, i zawodzi heurystyka. Nawet w przypadku najlepszych okoliczności **zrównoleglenie potoku nie przyniesie wzrostu wydajności, jeśli źródłem jest `Stream.iterate` lub operacją pośrednią jest `limit`**. Potok musi uwzględniać **obie** wymienione kwestie. Co gorsza, domyślna strategia zrównoleglania stara się rozwiązać problem nieprzewidywalności limitu, zakładając, że przetworzenie kilku dodatkowych elementów i ich późniejsze pominięcie w wynikach nie wyrządzi żadnej szkody. W przypadku liczb pierwszych Mersenne’a znalezienie każdej następnej liczby zajmuje mniej więcej dwa razy więcej czasu niż znalezienie poprzedniej. Oznacza to, że koszt wyliczenia następnego elementu jest mniej więcej równy kosztowi wyliczenia wszystkich poprzednich, co powoduje, że ten niewinnie wyglądający potok rozkłada cały algorytm zrównoleglenia. Morał z tej historii jest prosty — **nie staraj się bezmyślnie zrównoleglić każdego potoku strumienia**. Konsekwencje wydajnościowe tego mogą być dotkliwe.

Ogólnie rzecz biorąc, **wzrost wydajności spowodowany zrównolegleniem jest najlepiej widoczny w przypadku strumieni działających na instancjach `ArrayList`, `HashMap`, `HashSet` i `ConcurrentHashMap`, tablicach, a także zakresach `int` lub `Long`**. Cechą wspólną tych wszystkich struktur danych jest możliwość ich dokładnego i taniego podzielenia na podzakresy o pożądanej wielkości, co pozwala podzielić prace do wykonania między kilka wątków. Abstrakcją wykorzystywaną przez bibliotekę strumieni do realizacji tego zadania jest rozdzielnik zwracany przez metodę `splitter` dostępną dla `Stream` i `Iterable`.

Innym ważnym czynnikiem wspólnym dla wszystkich tych struktur danych jest to, że zapewniają dobrą lub nawet doskonałą lokalność referencji w trakcie przetwarzania sekwencyjnego — referencje do kolejnych elementów są przechowywane we wspólnym obszarze pamięci. Obiekty wskazywane przez te referencje nie muszą znajdować się obok siebie w pamięci, co ogranicza lokalność referencji. Lokalność referencji okazuje się wyjątkowo ważnym elementem w zrównoleglaniu operacji grupowych — jeśli jej brakuje, wątki spędzają większość czasu na oczekiwaniu na przesłanie danych z pamięci operacyjnej do pamięci podręcznej procesora. Strukturami danych o najlepszej lokalności referencji są oczywiście tablice typów prostych, ponieważ wszystkie potrzebne dane znajdują się w ciągłym obszarze pamięci.

Rodzaj operacji kończącej w potoku strumienia również ma wpływ na efektywność zrównoleglenia. Jeśli zdecydowana większość prac do wykonania znajduje się w operacji kończącej, a nie w operacjach pośrednich, a sama operacja jest z natury sekwencyjna, zrównoleglenie strumienia będzie miało ograniczoną efektywność. Najlepszymi operacjami kończącymi dla zrównolegnień są redukcje, gdzie wszystkie elementy pojawiające się w strumieniu są łączone za pomocą jednej z metod `reduce` z `Stream` lub też korzystają z przygotowanych wcześniej redukcji typu `min`, `max`, `count` i `sum`. Operacje dokonujące **wcześniejszego zakończenia operacji**, np. `anyMatch`, `allMatch` i `noneMatch`, również mogą skorzystać na zrównolegleniu. Operacje wykonywane przez metodę `collect` z `Stream` (nazywane **redukcjami zmiennymi**) niezbyt nadają się do zrównoleglenia, bo narzut związany z łączeniem jest kosztowny.

Jeśli tworzysz własną implementację `Stream`, `Iterable` lub `Collection` i chcesz zapewnić dobrą wydajność zrównoleglenia, musisz przesłonić metodę `splitterator`, a następnie bardzo intensywnie przetestować wydajność wynikowych strumieni. Napisanie dobrej jakości metody `splitterator` jest trudne i wykracza poza ramy tej książki.

Zrównoleglenie strumienia może prowadzić nie tylko do marnej wydajności i czasem do naruszenia żywotności, ale również do niepoprawnych wyników, a nawet nieprzewidywalnego zachowania (naruszenie bezpieczeństwa). Naruszenie bezpieczeństwa może być wynikiem zrównoleglenia potoku, który używa odwzorowań, filtrów lub innych dostarczonych przez programistę obiektów funkcyjnych, które nie dostosowały się do ich specyfikacji. Specyfikacja `Stream` dokładnie określa wymagania stawiane obiektom funkcyjnym. Na przykład funkcje akumulatora lub łączenia przekazywane do metody `reduce` z `Stream` muszą być łączne, niezależne od siebie i bezstanowe. Złamanie tych wymagań (niektóre z nich zostały omówione w temacie 46.), ale działanie w sposób sekwencyjny najprawdopodobniej doprowadzi do uzyskania właściwych wyników. Jeśli jednak dokonamy zrównoleglenia, zapewne pojawi się błąd, czasem nawet katastrofalnych rozmiarów.

Warto w tym miejscu podkreślić, że nawet jeśli zrównoleglona wersja programu liczącego liczby pierwsze Mersenne'a zadziałałaby i zwróciła wyniki, zapewne nie wyświetliłaby liczb w poprawnej, czyli rosnącej, kolejności. Aby zapewnić właściwą kolejność oferowaną domyślnie przez wersję sekwencyjną, musielibyśmy zastąpić operację kończącą `forEach` operacją `forEachOrdered`, która gwarantuje przejście przez strumienie zrównoleglone w **porządku zgodnym z pojawianiem się nowych elementów**.

Nawet przy założeniu, że właściwie dobraliśmy funkcję rozdzielającą strumień źródłowy, a także użyliśmy łatwej do zrównoleglenia lub taniej operacji kończącej oraz działających niezależnie obiektów funkcyjnych, nie uzyskamy dobrego przyspieszenia poprzez zrównoleglenie, jeśli potok nie wykonuje wystarczająco dużo rzeczywistej pracy, aby zrównoważyć koszt samego zrównoleglenia. Przy **bardzo**

zgrubnym szacowaniu należy przyjąć, że liczba elementów w strumieniu przemnożona przez liczbę wierszy kodu wykonywanego dla każdego elementu powinna wynosić przynajmniej sto tysięcy [Lea14].

Pamiętajmy, że zrównoleglenie strumienia to optymalizacja wydajnościowa. Podobnie jak w przypadku każdej innej optymalizacji musimy wykonać test wydajności przed i po wprowadzeniu zmian, aby upewnić się, że zmiana była warta wprowadzenia (temat 67.). Oczywiście najlepiej tego rodzaju test przeprowadzać w realistycznej konfiguracji systemu. Standardowo wszystkie zrównoleglone potoki strumienia w programie działają w jednej wspólnej puli dzielenia i łączenia. Jeden źle zachowujący się potok może zaszkodzić wydajności innych potoków w całkowicie niepowiązanych częściach systemu.

Jeśli zaczynasz mieć wrażenie, że wszystko przeciwstawia się zastosowaniu zrównoleglonych potoków strumienia, to masz rację. Znajomy, który zarządza bazą kodu zawierającą miliony wierszy kodu intensywnie korzystającego ze strumieni, stwierdził, że jedynie w niewielkiej liczbie miejsc zrównoleglenie okazało się przydatne. **Nie** oznacza to, że powinieneś unikać zrównoleglenia. **W sprzyjających okolicznościach można osiągnąć prawie liniowy wzrost wydajności wraz ze zwiększaniem liczby rdzeni procesora przez proste wywołanie metody `parallel` na potoku strumienia.** Pewne rodzaje obliczeń, najczęściej związane z uczeniem maszynowym i przetwarzaniem danych, są szczególnie podatne na tego rodzaju przyspieszenia.

Jako prosty przykład sytuacji, w której zrównoleglenie potoku przynosi wzrost wydajności, przeanalizujemy funkcję obliczającą $\pi(n)$ na podstawie liczb pierwszych mniejszych lub równych n :

```
// Potok strumieniowy zliczający liczby pierwsze - mógłby skorzystać
// ze zrównoleglenia
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

Na moim komputerze wyliczenie $\pi(10^8)$ za pomocą przedstawionej funkcji zajęło 31 sekund. Po dodaniu wywołania `parallel()` czas potrzebny na wykonanie zadania uległ skróceniu do 9,2 sekundy:

```
// Potok strumieniowy zliczający liczby pierwsze - wersja zrównoleglona
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

Oznacza to, że zrównoleglenie operacji przyspieszyło obliczenia 3,7 razy na komputerze czterordzeniowym. Podkreślmy, że nie jest to zbyt praktyczny sposób wyliczania $\pi(n)$ dla dużych wartości n . Istnieją dużo wydajniejsze algorytmy, w szczególności wzór Lehmera.

Jeśli zamierzamy zrównoleglić strumień liczb losowych, warto zacząć od użycia instancji `SplittableRandom` zamiast klasy `ThreadLocalRandom` (lub wycofywanego `Random`). Klasa `SplittableRandom` powstała właśnie w tym celu i pozwala na uzyskanie liniowego przyspieszenia. Klasa `ThreadLocalRandom` jest zaprojektowana do użycia w pojedynczym wątku i jest w stanie działać jako źródło dla strumienia zrównoleglonego, ale nie będzie tak szybka jak `SplittableRandom`. Zastosowanie klasy `Random`, która synchronizuje się przy każdej operacji, mogłoby skutkować ogromnymi zatorami, wprost wykluczającymi efektywne zrównoleglenie.

Podsumujmy. Nie próbuj stosować zrównoleglonych wersji potoku strumienia, chyba że masz uzasadnione powody wierzyć, iż pozwoli to jednocześnie zachować poprawność obliczeń i zwiększyć szybkość ich realizacji. Kosztem niewłaściwego zrównoleglenia strumienia będzie błąd w programie lub katastrofalna wydajność. Jeśli wierzysz, że zrównoleglenie jest uzasadnione, upewnij się, że kod działa poprawnie po takiej operacji. Zadbaj też o porównanie wydajności w możliwie realistycznych warunkach. Tylko w sytuacji, gdy kod zapewnia właściwe wyniki, a testy wykazały, że podejrzenia co do wzrostu wydajności są słuszne, użyj strumienia zrównoleglonego w kodzie produkcyjnym.

Metody

W rozdziale tym omówimy kilka aspektów projektowania metod — w jaki sposób obsługiwać parametry i zwracane wartości, jak projektować sygnatury metod i w jaki sposób należy dokumentować metody. Większość materiału zawartego w tym rozdziale odnosi się zarówno do konstruktorów, jak i do metod. Podobnie jak w rozdziale 4., w rozdziale tym skupiamy się na użyteczności, solidności i elastyczności.

Temat 49. Sprawdzanie poprawności parametrów

Większość metod i konstruktorów posiada pewne ograniczenia wartości, jakie można przekazywać do ich parametrów. Na przykład rzadkie jest korzystanie z ujemnych wartości indeksów, a referencje do obiektów muszą być różne od null. Powinno jasno udokumentować wszystkie tego typu ograniczenia i zapewnić ich stosowanie za pomocą odpowiednich testów na początku metody. Jest to specjalny przypadek ogólnej zasady, mówiącej, że błędy należy wykrywać tak szybko, jak jest to możliwe. Niespełnienie tego warunku powoduje, że zmniejsza się prawdopodobieństwo szybkiego wykrycia błędu, a gdy zostanie on już wykryty, utrudnia to określenie przyczyny błędu.

Jeżeli do metody zostanie przekazana nieprawidłowa wartość parametru, a metoda przed rozpoczęciem pracy kontroluje wartości parametrów, szybko zostanie ona zakończona z odpowiednim komunikatem o błędzie. Jeżeli metoda nie kontroluje wartości parametrów, w przypadku wystąpienia błędu mogą wystąpić różne efekty. Wykonanie metody może zostać zatrzymane przez kłopotliwy wyjątek. Co gorsza, metoda może normalnie zwrócić wynik, ale będzie on całkowicie nieprawidłowy.

Najgorszym przypadkiem jest normalne zakończenie metody przy pozostawieniu niektórych obiektów w nieprawidłowym stanie, co w przyszłości powoduje wystąpienie błędu w przypadkowych miejscach programu. Innymi słowy, brak właściwego sprawdzania poprawności parametrów może spowodować złamanie zasady **atomowości w przypadku błędu** (temat 76.).

W przypadku metod publicznych lub zabezpieczonych do dokumentowania wyjątków, które może zgłosić metoda w przypadku naruszenia zakresu wartości parametrów, należy skorzystać ze znacznika `@throws` (temat 74.). Najczęściej wyjątkiem tym będzie `IllegalArgumentException`, `IndexOutOfBoundsException` lub `NullPointerException` (temat 72.). Po udokumentowaniu ograniczeń wartości parametrów metody oraz wyjątków, zgłaszanych w przypadku naruszenia tych ograniczeń, należy wymusić te ograniczenia. Typowy przykład wygląda następująco:

```
/**
 * Zwraca BigInteger, którego wartość to (this mod m). Metoda ta
 * różni się od metody remainder tym, że zawsze zwraca
 * nieujemną liczbę BigInteger.
 *
 * @param m moduł, który zawsze musi być dodatni
 * @return this mod m
 * @throws ArithmeticException, jeżeli m jest mniejsze lub równe 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Moduł <= 0: " + m);
    ... //wykonanie obliczeń
}
```

Zwróć uwagę, że komentarz dokumentujący nie informuje o tym, iż „mod zgłasza `NullPointerException`, jeśli `m` jest równe `null`”, choć metoda taki wyjątek zgłosi właśnie w efekcie próby wywołania metody `m.signum()`. Ten wyjątek jest udokumentowany w komentarzu dokumentującym klasę `BigInteger`. Komentarz na poziomie klasy dotyczy wszystkich parametrów wszystkich klas publicznych metody. To dobry sposób na pozbycie się bałaganu powodowanego dodawaniem komentarza dotyczącego `NullPointerException` do każdej metody. Można to rozwiązanie połączyć z użyciem `@Nullable` lub innej adnotacji, aby wskazać, że ten konkretny parametr może jednak przyjmować wartość `null`. Rozwiązanie to nie jest jeszcze standardem i w tym celu stosuje się obecnie kilka różnych adnotacji.

Metoda `Objects.requireNonNull` (dodana w Javie 7) jest elastyczna i wygodna, więc nie ma już potrzeby samodzielnego sprawdzania wartości `null`. Można w razie potrzeby wskazać własny komunikat dla wyjątku. Metoda zwraca przekazaną wartość, więc mamy okazję jednocześnie sprawdzić niewystąpienie wartości `null` i przypisać parametr do pola:

```
// Krótki i wygodny sposób upewnienia się co do wartości różnej od null
this.strategy = Objects.requireNonNull(strategy, "strategy");
```


Można też zignorować zwracaną wartość i potraktować wywołanie `Objects.requireNonNull` ↪ `NonNull` jako wartownika przed użyciem wartości w przyszłości.

W Javie 9 do `java.util.Objects` dodano obsługę sprawdzania zakresów. Składa się ona z trzech metod: `checkFromIndexSize`, `checkFromToIndex` i `checkIndex`. Nie są one jednak tak elastyczne jak metoda sprawdzająca wartość `null`. Nie pozwalają na określenie własnego komunikatu dla wyjątku i są zaprojektowane do działania tylko na listach i tablicach. Metody nie obsługują zakresów zamkniętych (które zawierają obie wskazane końcówki). Z drugiej strony metody robią dokładnie to, co zawiera ich opis, więc bywają przydatne, jeśli zgadzają się z naszym przypadkiem.

W przypadku metod nieudostępnianych to autor pakietu określa, kiedy wywołana jest dana metoda, wystarczy zapewnić jedynie, że do metody przekazane będą właściwe wartości parametrów. Dlatego metody niepubliczne powinny sprawdzać poprawność parametrów, korzystając z asercji, tak jak w tym fragmencie kodu:

```
// Prywatna metoda pomocnicza do sortowania rekurencyjnego
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // wykonanie obliczeń
}
```

W rzeczywistości asercje te są deklaracjami, że warunek asercji **będzie** prawdziwy, niezależnie od sposobu używania pakietu przez jego klienty. W przeciwieństwie do normalnych testów poprawności asercje zgłaszają wyjątek `AssertionError`, jeżeli ich warunek nie jest dotrzymany. W przeciwieństwie do zwykłych testów poprawności nie mają żadnego efektu i nie powodują dodatkowego narzutu, o ile się ich jawnie nie aktywuje, co jest realizowane przez użycie znacznika `-ea` (lub `-enableassertions`) w interpreterze java. Więcej informacji na temat asercji można znaleźć w samouczku [Asserts].

Szczególnie ważne jest sprawdzanie poprawności parametrów, które nie są wykorzystywane w metodzie, a jedynie zapamiętywane w celu późniejszego użycia. Dla przykładu, wróć do statycznej metody fabrycznej z tematu 20., która jako parametru wymaga tablicy liczb `int` i zwraca obiekt `List`, odpowiadający tej tablicy. Jeżeli użytkownik tej metody przekaże wartość `null`, zgłosi ona wyjątek `NullPointerException` ↪ `Exception`, ponieważ metoda sprawdza wartość parametru. Jeżeli kontrola ta zostanie usunięta, metoda zwróci referencję do nowego obiektu `List`, ale w momencie, gdy użytkownik się odwoła do tej referencji, zostanie zgłoszony wyjątek `NullPointerException`. Gdy błąd się ujawni, określenie pochodzenia obiektu `List` może być trudne, co skomplikuje proces uruchamiania programu.

W przypadku konstruktorów należy szczególnie ściśle stosować zasadę kontroli wartości parametrów, ponieważ wartości te są zapamiętywane do późniejszego wykorzystania. Kontrola wartości parametrów pozwoli uniknąć tworzenia obiektów, naruszających niezmienniki klasy.

Istnieją również wyjątki od zasady sprawdzania wartości parametrów przed rozpoczęciem obliczeń. Ważnym wyjątkiem jest przypadek, w którym sprawdzanie poprawności parametrów jest czasochłonne lub niepraktyczne, a kontrola poprawności jest przeprowadzana niejawnie w czasie działania metody. Dla przykładu przyjrzyjmy się metodzie sortującej listę obiektów, takiej jak `Collections.sort(List)`. Wszystkie obiekty na liście muszą być wzajemnie porównywalne. W czasie procesu sortowania listy każdy obiekt jest porównywany z innymi elementami. Jeżeli obiekty nie są wzajemnie porównywalne, jedno z porównań zwróci wyjątek `ClassCastException`, co jest prawidłowym działaniem metody sortującej. Dlatego nie ma potrzeby wcześniejszego sprawdzania, czy elementy są porównywalne. Jeżeli jednak aplikacja nie obsługuje takiego zachowania, może to spowodować błędne działanie i utratę zasady **atomowości w przypadku błędu** (temat 76.).

Czasami obliczenia powodują niejawne wykonanie sprawdzenia poprawności na niektórych parametrach, ale w przypadku niepowodzenia tych testów zgłaszają nieprawidłowe wyjątki. To znaczy, że wyjątek, który naturalnie zgłasza wyrażenie, w przypadku napotkania nieprawidłowego argumentu nie zgadza się z wyjątkiem zamieszczonym w dokumentacji metody. Należy wtedy skorzystać z opisanego w temacie 73. **tłumaczenia wyjątków**, zmieniając naturalne wyjątki wyrażenia na właściwe dla niego metody.

Nie należy wnioskować, że ograniczanie wartości parametrów jest najlepszą metodą. Odwrotnie, powinienś projektować metody tak, aby były one możliwie ogólne. Im mniej nałożysz ograniczeń na wartości parametrów, tym lepiej, o ile oczywiście metoda może zwracać rozsądne wyniki dla wszystkich dopuszczalnych wartości parametrów. Czasami jednak ograniczenia są bezpośrednio związane z abstrakcją, którą opisuje klasa.

Podsumujmy. Gdy piszesz metodę lub konstruktor, powinienś określić, jakie należy wprowadzić ograniczenia wartości parametrów. Należy udokumentować te ograniczenia i wymusić je przez jawne testy na początku metody. Ważne jest, aby przyzwyczaić się do takiego tworzenia kodu. Ta nudna praca odpłaci się już za pierwszym razem, gdy zadziałają wprowadzone mechanizmy zabezpieczeń.

Temat 50. Defensywne kopiowanie

Jedną z cech, które powodują, że programowanie w języku Java jest takie przyjemne, jest to, że jest to **język bezpieczny**. Dzięki wyeliminowaniu metod natywnych nie występują przepełnienia bufora, przepełnienia tablicy, nieokreślone wskaźniki i inne błędy dostępu do pamięci, które są plagą w językach takich, jak C i C++. Korzystając z języków bezpiecznych, można pisać klasy, których niezmienniki pozostają spełnione, bez względu na to, co się dzieje z pozostałą częścią systemu. Nie jest to możliwe w językach, które traktują pamięć jak jedną, gigantyczną tablicę.

Nawet w językach bezpiecznych nie da się odizolować klas bez dodatkowego wsparcia z ich strony. **Musisz programować defensywnie, zakładając, że klienci Twojej klasy zrobią wszystko, aby naruszyć jej niezmienniki.** Może być to prawdą w przypadkach, gdy ktoś próbuje złamać zabezpieczenia systemu bezpieczeństwa, ale najczęściej Twoja klasa będzie musiała radzić sobie z nieprzewidywanymi sytuacjami, spowodowanymi przez zwykłe pomyłki programistów, korzystających z Twojego API. Warto jest poświęcić nieco czasu, aby utworzyć klasę, która będzie odporna na różne działania klientów.

Choć niemożliwe jest zmodyfikowanie wewnętrznego stanu obiektu innej klasy bez wsparcia ze strony tego obiektu, niespodziewanie łatwo udostępnić takie wsparcie, nie zdając sobie z tego sprawy. Na przykład popatrzmy na poniższą klasę, która reprezentuje niezmienny odcinek czasu.

```
//Naruszona "niezmiennność" klasy reprezentującej odcinek czasu
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start początek odcinka czasu
     * @param end koniec odcinka czasu, nie może być wcześniejszy niż start
     * @throws IllegalArgumentException jeżeli start jest większy niż end
     * @throws NullPointerException jeżeli start lub end ma wartość null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " po " + end);
        this.start = start;
        this.end = end;
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }

    ... //pozostała część pominięta
}
```

Na pierwszy rzut oka klasa wydaje się niezmienna i wymusza założenie, że początek odcinka czasu nie może być późniejszy niż jego koniec. Łatwo można naruszyć ten niezmiennik, korzystając z tego, że obiekty `Date` są modyfikowalne.

```
//Atak na wewnętrzne składowe obiektu Period
Date start = new Date();
Date end = new Date();
```

```
Period p = new Period(start, end);
end.setYear(78); // Modyfikacja wewnętrznej składowej p!
```

Od Javy 8 istnieje bardzo proste rozwiązanie tego problemu w postaci użycia klas `Instant` (lub `LocalDateTime` albo `ZonedDateTime`) w miejsce klas `Date`, ponieważ `Instant` (i inne klasy z `java.time`) są niezienne (temat 17.). **Klasa `Date` jest wycofywana z użycia i nie powinna być wykorzystywana w nowym kodzie.** Oczywiście sam problem jako taki nie przestanie istnieć, bo jest wiele sytuacji, w których musimy skorzystać z modyfikowalnych typów wartości w API lub wewnętrznej reprezentacji. W takich przypadkach użyj technik proponowanych w tym temacie.

Aby zabezpieczyć wewnętrzne składowe obiektu `Period` przed tego rodzaju atakami, **należy w konstruktorze defensywnie kopiować każdy modyfikowalny parametr** i korzystać z tych kopii jako składowych obiektu `Period`.

```
// Poprawiony konstruktor - defensywne kopiowanie parametrów
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end    = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start + " po " + end);
}
```

W przypadku nowego konstruktora poprzedni typ ataku nie będzie przynosił efektów. Zwróć uwagę, że **defensywne kopiowanie jest wykonane przed sprawdzaniem poprawności parametrów (temat 49.), a sprawdzanie jest przeprowadzane na kopiach obiektów.** Choć może się to wydać nienaturalne, jest jednak niezbędne. Zabezpiecza to klasę przed zmianą parametrów przez inny wątek w czasie „**okna wrażliwości na uszkodzenia**”, pomiędzy sprawdzeniem parametrów a ich skopiowaniem (jest to atak znany pod nazwą **czasu kontroli/czasu użycia** lub **TOCTOU** [Viega01]).

Zauważ, że do defensywnego kopiowania nie korzystamy z metody `clone` klasy `Date`. Ponieważ klasa `Date` nie jest zadeklarowana jako `final`, nie ma gwarancji, że zwróci obiekt klasy `java.util.Date` — może zwrócić obiekt specjalnie przygotowanej podklasy, która jest zaprojektowana w celu umożliwienia kolejnego ataku. Podklasa taka może na przykład zapamiętywać referencje do obiektów w prywatnej statycznej liście, w celu umożliwienia dostępu do zapamiętanych obiektów. Aby zabezpieczyć się przed takimi atakami, **nie można używać metody `clone` do defensywnego kopiowania parametrów, których typy można dziedziczyć.**

Choć poprawiony konstruktor broni klasę przed poprzednim typem ataku, nadal można zmodyfikować obiekt `Period`, ponieważ akcesory pozwalają na dostęp do modyfikowalnych składników.

```
// Drugi atak na składowe obiektu Period
```

```
Date start = new Date();
```

```
Date end = new Date();
```

```
Period p = new Period(start, end);
```

```
p.end().setYear(78); // Modyfikacja wewnętrznej składowej p!
```

Aby bronić się przed drugim rodzajem ataku, wystarczy zmienić akcesor, aby **zwracał defensywne kopie modyfikowalnych pól wewnętrznych**.

```
// Poprawiony akcesor - defensywne kopiowanie pól wewnętrznych
```

```
public Date start() {
```

```
    return new Date(start.getTime());
```

```
}
```

```
public Date end() {
```

```
    return new Date(end.getTime());
```

```
}
```

Klasa `Period` z nowym konstruktorem i akcesorami jest w pełni niezmienna. Ani złośliwy, ani niekompetentny programista nie może naruszyć niezmiennika klasy, ponieważ po prostu nie ma na to sposobu (nie uwzględniamy tutaj rozwiązań wyjątkowo nietypowych, takich jak metody natywne lub mechanizm refleksji). Dzieje się tak, ponieważ żadna inna klasa poza `Period` nie ma dostępu do modyfikowalnych składników obiektów `Period`. Pola te są w pełni hermetyzowane w obiekcie.

Zwróć uwagę, że nowe akcesory, w przeciwieństwie do nowego konstruktora, mogłyby korzystać z metody `clone` do wykonywania defensywnych kopii. Jest to dopuszczalne, ponieważ wiemy, że obiekty składowe klasy `Period` są prawdziwymi obiektami `java.util.Date`, a nie potencjalnie niebezpiecznymi podklasami. Dlatego, zwykle z powodów przedstawionych w temacie 13., lepiej jest używać konstruktorów lub statycznych metod fabrycznych.

Defensywne kopiowanie parametrów nie jest stosowane tylko w przypadku klas niezmiennych. Za każdym razem, gdy tworzysz metodę lub konstruktor zapamiętujący w wewnętrznych strukturach przekazany przez użytkownika obiekt, pomyśl, czy obiekt ten jest modyfikowalny. Jeżeli tak jest, trzeba określić, czy klasa będzie tolerować zmiany stanu tego obiektu po wbudowaniu w swoje struktury danych. Jeżeli odpowiedź brzmi „nie”, trzeba utworzyć kopię obiektu i do wewnętrznych struktur wbudować kopię zamiast oryginału. Na przykład jeżeli chcesz wykorzystać dostarczoną przez klienta referencję do obiektu jako element wewnętrznego obiektu `Set` lub jako klucz wewnętrznego obiektu `Map`, powinieneś wiedzieć, że niezmienniki tego obiektu zostaną zniszczone, jeżeli po wstawieniu obiekt ten zostanie zmieniony.

Te same uwagi są prawdziwe w przypadku defensywnego kopiowania komponentów przed ich zwróceniem do klientów. Niezależnie od tego, czy Twoja klasa jest niezmienna czy nie, powinieneś pomyśleć przed zwróceniem referencji do

wewnętrznego modyfikowalnego składnika klasy. Najprawdopodobniej najlepiej zwrócić kopię tego obiektu. Należy również pamiętać, że tablice o rozmiarze większym od zera są zawsze modyfikowalne, dlatego zawsze powinny być defensywnie kopiować wewnętrzne tablice zwracane klientowi. Obie te techniki są opisane w temacie 15.

Powinno się więc korzystać w swoich programach, o ile jest to możliwe, z obiektów niezmiennych jako składników klas, dzięki temu nie trzeba pamiętać o defensywnym kopiowaniu (temat 17.). W przedstawionym przykładzie z klasą `Period` użyj klasy `Instant` (lub `LocalDateTime` albo `ZonedDateTime`), chyba że potrzebujesz wersji działającej w maszynie wirtualnej sprzed Javy 8. W przypadku starszych wersji języka można korzystać z pierwotnej wartości `long` zwracanej przez `Date.getTime()` zamiast z referencji do obiektu `Date`.

Kopiowanie defensywne jest często obciążone zmniejszeniem wydajności, co nie zawsze jest uzasadnione. Jeśli klasa ufa kodowi ją wywołującemu w kwestii tego, że nie będzie modyfikował jej wewnętrznych komponentów (bo na przykład klasa i kod jej używający należą do tego samego pakietu), można zdecydować się na poluzowanie zasady kopiowania defensywnego. W takiej sytuacji zaznacz jednak wyraźnie w dokumentacji, że kodowi wywołującemu nie wolno modyfikować wskazanych parametrów lub zwracanych wartości.

Nawet pomiędzy granicami pakietów nie zawsze jest prawidłowe wykonywanie defensywnej kopii zmiennych parametrów przed zintegrowaniem ich w obiekcie. Istnieją metody i konstruktory, których wywołania wskazują na jawne **przekazanie** obiektu, do którego odwołuje się parametr. Przy wywoływaniu takich metod klient obiecuje, że nie będzie bezpośrednio modyfikował obiektu. Metoda lub konstruktor oczekujący przejęcia na własność dostarczanego przez klienta obiektu modyfikowalnego musi jasno to zdefiniować w dokumentacji.

Klasy zawierające metody i konstruktory, których wywołania powodują przekazanie kontroli, nie mogą się bronić przed złośliwymi klientami. Klasy takie można stosować, jeżeli istnieje obopólne zaufanie pomiędzy klasą i jej klientem lub gdy naruszenie niezmienników klasy nie wpłynie na inne obiekty oprócz klienta. Przykładem takiej sytuacji są klasy osłonowe, przedstawione w temacie 18. W zależności od natury klasy osłonowej klient może zniszczyć niezmienniki klasy, bezpośrednio odwołując się do osłanianego obiektu, ale zwykle wpłynie to tylko na klienta.

Podsumujmy. Jeżeli klasa posiada modyfikowalne komponenty, które są pobierane z klienta lub zwracane do niego, musi ona defensywnie kopiować te komponenty. Jeżeli koszt kopiowania komponentów będzie zbyt duży **oraz** w klasie można zaufać swoim klientom, że nie będą oni niewłaściwie modyfikować komponentów, to kopiowanie defensywne może być zastąpione przez udokumentowanie odpowiedzialności klienta za to, że nie może on modyfikować komponentów.

Temat 51. Projektowanie sygnatur metod

Temat ten zawiera kilka wskazówek, dotyczących projektowania API. Wskazówki te, zebrane razem, pozwolą Ci stworzyć API odporniejsze na błędy oraz łatwiejsze do nauczenia się i stosowania.

Starannie wybieraj nazwy metod. Nazwy zawsze powinny być zgodne z konwencją nazewnictwa (temat 68.). Twoim podstawowym celem powinien być zawsze wybór nazwy, która jest zrozumiała i spójna z innymi nazwami z tego samego pakietu. Celem drugorzędnym jest zapewnienie spójności z bardziej ogólnymi zasadami nazewnictwa, o ile takie są przyjęte. Unikaj długich nazw metod. Jeżeli nie masz pomysłów, spróbuj zastosować zgodność z bibliotekami języka Java. Choć istnieje w nich wiele niespójności, które są nieuchronne z powodu rozmiaru i zakresu tych bibliotek, stanowią one pewien wzorzec.

Nie przesadzaj z tworzeniem metod pomocniczych. Zbyt duża liczba metod powoduje, że trudno się nauczyć używania, dokumentowania, testowania i konserwacji klasy. Sprawdza się to zwłaszcza w przypadku interfejsów, gdy zbyt wiele metod komplikuje implementowanie tego interfejsu. Dla każdej operacji obsługiwanej przez Twoją klasę należy napisać w pełni funkcjonalną metodę. Wprowadzanie metod zapewniających łatwiejsze wykonanie operacji jest dopuszczalne jedynie wtedy, gdy jest ona wykonywana często. **Jeżeli nie masz tej pewności, pozostaw klasę bez zmian.**

Unikaj długich list parametrów. Jako zasadę powinno się przyjąć, że metoda nie powinna mieć więcej jak cztery parametry, a im ich jest mniej, tym lepiej. Większość programistów nie potrafi zapamiętać dłuższej listy parametrów. Jeżeli wiele metod przekracza to ograniczenie, Twoje API będzie mało użyteczne bez ciągłego sięgania do dokumentacji. Nowoczesne środowiska IDE trochę pomagają opanować ten problem, ale i tak łatwiej posługiwać się krótkimi listami parametrów. **Długie sekwencje parametrów jednakowego typu są szczególnie dokuczliwe.** Nie tylko dlatego, że użytkownicy takiego API nie będą pamiętać kolejności parametrów, ale gdy omyłkowo zamienią ich kolejność, program nadal będzie się kompilował i uruchamiał. Nie będzie jednak działał zgodnie z zamierzeniami autora.

Istnieją trzy techniki skracania długich list parametrów. Pierwszą jest podział metody na wiele metod, z których każda wymaga jedynie podzbioru oryginalnej listy parametrów. Jeżeli jednak zostanie to wykonane nieuważnie, może prowadzić do powstania zbyt wielu metod. Jednak dzięki zwiększeniu ortogonalności liczba metod może się **zmniejszyć**. Przykładem takim może być interfejs `java.util.List`. Nie zawiera on metod potrzebnych do wyszukania pierwszego lub ostatniego indeksu elementu z podlisty. Obie takie metody wymagałyby trzech parametrów. Zamiast tego dostępna jest metoda `subList`, która wymaga dwóch parametrów i zwraca **widok** podlisty. Metoda ta może być połączona z metodami `indexOf` lub

`lastIndexOf`, z których każda do wykonania swojego zadania wymaga jednego parametru. Co więcej, aby wykonać różne operacje na podliście, metoda `subList` może być łączona z innymi metodami, operującymi na obiektach `List`. Otrzymane API ma świetny stosunek wydajności do objętości.

Drugą techniką jest skracanie długich list parametrów przez tworzenie **klas pomocniczych** przechowujących zbiory parametrów. Najczęściej klasy pomocnicze są statycznymi klasami zagnieżdżonymi (temat 24.). Technika ta jest zalecana w przypadku częstego występowania sekwencji parametrów opisujących jeden obiekt. Rozważmy dla przykładu klasę reprezentującą grę w karty, gdzie stale przekazujesz sekwencje dwóch parametrów, opisujących wagę karty i jej kolor. Twoje API, jak również wewnętrzne struktury klasy, mogą być najprawdopodobniej ulepszone, jeżeli dodasz klasę pomocniczą reprezentującą kartę i zamienisz wszystkie wystąpienia sekwencji parametrów na jeden parametr klasy pomocniczej.

Trzecia technika, łącząca cechy dwóch pierwszych, polega na zaadaptowaniu wzorca *Builder* (temat 2.) z tworzenia obiektów na wywołanie metody. Jeżeli mamy metodę z wieloma parametrami, szczególnie jeżeli część z nich jest opcjonalna, przydatne może być zdefiniowanie obiektu reprezentującego wszystkie te parametry i umożliwienie klientowi wykonanie wielu wywołań metod ustawiających parametry na tym obiekcie, z których każda ustawia jeden parametr lub małą grupę parametrów związanych logicznie ze sobą. Po ustawieniu odpowiednich parametrów klient wywołuje metodę „wykonaj” tego obiektu, która wykonuje kontrolę poprawności parametrów, a następnie oczekiwane obliczenia.

W przypadku typu parametrów lepiej używać interfejsów zamiast klas (temat 64.). Jeżeli istnieje odpowiedni interfejs opisujący parametr, należy go użyć zamiast klasy implementującej ten interfejs. Na przykład nie ma powodu pisać metody oczekującej parametru `HashMap` — zamiast tego należy skorzystać z `Map`. Pozwoli to przekazać do metody obiekt `HashMap`, `TreeMap`, `ConcurrentHashMap`, fragment mapy `TreeMap` lub dowolną inną implementację interfejsu `Map`. Korzystając z klasy zamiast interfejsu, ograniczasz klienta do określonej implementacji i wymuszasz stosowanie niepotrzebnych, a często kosztownych operacji kopiowania, jeżeli dane wejściowe są przekazane w innej postaci.

Korzystaj z dwuelementowych typów wyliczeniowych zamiast parametrów `boolean`. Powoduje to ułatwienie czytania i pisania kodu, szczególnie gdy używane IDE obsługuje autouzupełnianie. Dodatkowo ułatwia to dodanie w przyszłości większej liczby opcji. Na przykład mamy typ `Thermometer` posiadający statyczną metodę fabryczną, która oczekuje wartości naszego typu wyliczeniowego:

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Nie tylko `Thermometer.newInstance(TemperatureScale.CELSIUS)` jest bardziej sensowne niż `Thermometer.newInstance(true)`, ale również można dodać do typu wyliczeniowego `TemperatureScale` wartość `KELVIN` bez konieczności dodawania

nowej metody fabrycznej do `Thermometer`. Można również przenieść zależności między skalami temperatur do stałych typu wyliczeniowego (temat 34.). Na przykład każda stała typu wyliczeniowego może mieć metodę oczekującą wartości `double` i normalizującą temperaturę do skali Celsjusza.

Temat 52. Rozsądne korzystanie z przeciążania

Poniżej przedstawiamy kod przeprowadzający próbę podziału kolekcji na zbiory, listy i inne rodzaje kolekcji:

```
// Błąd - co wyświetli program?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> l) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Nieznana kolekcja";
    }

    public static void main(String[] args) {
        Collection<?>[] tests = collections {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

Mógłbyś oczekiwać, że program wypisze „Set”, następnie „List” i na koniec „Nieznana kolekcja”, ale tak się nie dzieje. Wynikiem tego programu jest trzykrotnie „Nieznana kolekcja”. Co się stało? Powodem takiego zachowania programu jest **przeciążenie** metody `classify`, a **wyбір, która przeciążona metoda zostanie wywołana, jest dokonywany w czasie kompilacji programu**. Dla wszystkich trzech przebiegów pętli typ parametru w czasie kompilacji jest taki sam — `Collection<?>`. W czasie wykonania typ ten jest różny dla każdego z przebiegów pętli, ale nie wpływa już na wybór przeciążonej metody. Ponieważ w czasie kompilacji typ

parametru został określony na `Collection<?>`, jedyną właściwą przeciążoną wersją metody jest trzecia — `classify(Collection<?>)`. Ta wersja jest zatem wywoływana w każdym przebiegu pętli.

Działanie tego programu nie jest intuicyjne, ponieważ **wybór metod przeciążanych jest statyczny, natomiast wybór metod przesłanianych jest dynamiczny**. Właściwa wersja **przesłanianych** metod jest wybierana w czasie wykonania, w oparciu o aktualny typ obiektu, na rzecz którego metoda ta została wywołana. Przypomnijmy: metoda jest przesłonięta, jeżeli klasa pochodna zawiera metodę o tej samej sygnaturze, co metoda klasy bazowej. Jeżeli metoda jest przesłonięta w klasie pochodnej i jest wywoływana na rzecz obiektu klasy pochodnej, wykonana zostanie **przesłaniająca metoda** z klasy pochodnej, niezależnie od typu obiektu na etapie kompilacji. Aby to zilustrować, przeanalizujmy fragment kodu:

```
class Wine {
    String name() { return "wino"; }
}

class SparklingWine extends Wine {
    String name() { return "wino musujące"; }
}

class Champagne extends SparklingWine {
    String name() { return "szampan"; }
}

public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.print(wine.name());
    }
}
```

Metoda `name` jest zadeklarowana w klasie `Wine` i przesłonięta (przedefiniowana) w klasie `SparklingWine` oraz `Champagne`. Tak, jak można było się spodziewać, program wypisze ciąg „wino wino musujące i szampan”, pomimo że na etapie kompilacji typem obiektu w każdym przebiegu jest `Wine`. Typ określony na etapie kompilacji nie ma wpływu na wybór wersji przesłoniętej metody — zawsze wywołana jest „najbardziej zagłębiona” przesłonięta metoda. Jest to działanie odwrotne do przeciążania, gdzie typ na etapie wykonania nie ma wpływu na wybór wersji metody — wybór wersji jest dokonywany na etapie kompilacji, w oparciu o typy parametrów na tym właśnie etapie.

W przykładzie klasy `CollectionClassifier` założeniem programu było rozróżnienie typu parametrów dzięki automatycznemu wywołaniu metody w oparciu o typ parametru na etapie wykonania, podobnie do metody `name` w przykładzie klasy

Wine. Przeciążanie metod nie pozwala jednak na zrealizowanie tego założenia. Jedyńm sposobem poprawienia tego programu jest zamiana trzech przeciążonych metod `classify` na jedną, która sprawdza typ za pomocą metody `instanceOf`:

```
public static String classify(Collection<?> c) {  
    return (c instanceof Set ? "Zbiór" :  
           (c instanceof List ? "Lista" : "Nieznana kolekcja"));  
}
```

Ponieważ przesłanianie jest normą, a przeciążanie wyjątkiem, to przesłanianie określa oczekiwania programistów do sposobu wywoływania metod. Tak jak zdemostrzowaliśmy na przykładzie klasy `CollectionClassifier`, przeciążanie może łatwo zawieść te oczekiwania. Złą praktyką jest tworzenie kodu, którego działanie nie jest oczywiste dla średnio zaawansowanego programisty. Jest to szczególnie ważne w przypadku API. Jeżeli typowy użytkownik API nie będzie wiedział, która z kilku przeciążonych metod zostanie wykonana w przypadku określonego zbioru parametrów, korzystanie z takiego API będzie się wiązało z ciągłym szukaniem błędów. Błędy te mogą ujawniać się jako nieprawidłowe działanie programu i wielu programistów nie będzie ich potrafiło zdiagnozować. Dlatego powinieneś **unikac mylących zastosowań przeciążania**.

Nie można jasno określić, jakie zastosowanie przeciążania jest mylące. **Bezpieczną, konserwatywną polityką jest unikanie eksportowania dwóch wersji metody z tą samą liczbą parametrów**. Jeżeli metoda korzysta z argumentów `varargs`, konserwatywną polityką jest unikanie jej przeciążania, poza przypadkami opisanymi w temacie 53. Jeżeli będziesz stosował się do tego ograniczenia, programista nigdy nie będzie miał wątpliwości, która przeciążona metoda pasuje dla danego zbioru parametrów. Ograniczenie to nie jest szczególnie dotkliwe, ponieważ **zamiast przeciążać metodę, zawsze możesz nadać jej inną nazwę**.

Rozważmy klasę `ObjectOutputStream`. Posiada ona warianty metody `write` dla każdego typu prostego oraz kilku typów obiektowych. Zamiast przeciążać metodę `write`, stworzone są osobne warianty o nazwach `writeBoolean(boolean)`, `writeInt(int)` i `writeLong(long)`. Dodatkową korzyścią tego schematu nazewnictwa, w porównaniu z przeciążaniem, jest możliwość utworzenia metod `read` z odpowiednimi nazwami, na przykład: `readBoolean()`, `readInt()` i `readLong()`. Klasa `ObjectInputStream` faktycznie zawiera metody `read` o podanych przez nas nazwach.

W przypadku konstruktorów nie możesz skorzystać z innej nazwy — tworząc kilka konstruktorów klasy, **zawsze** korzysta się z przeciążania. W niektórych przypadkach zamiast tworzyć konstruktory, można udostępnić statyczne metody fabryczne (temat 1.), ale nie zawsze jest to praktyczne. W przypadku konstruktorów jednak nie trzeba się przejmować różnicą pomiędzy przesłanianiem a przeciążaniem, ponieważ konstruktory nie mogą być przesłaniane. Ponieważ prawdopodobnie miałeś okazję udostępnić kilka konstruktorów z tą samą liczbą parametrów, warto wiedzieć, kiedy można to wykonać bezpiecznie.

Udostępnianie kilku przeciążonych wersji metod z tą samą liczbą parametrów niekoniecznie musi być mylące dla programisty, o ile zawsze jasno jest określone, która wersja zostanie wywołana dla określonego zestawu parametrów. Jest to przypadek, gdy co najmniej jeden parametr formalny z każdej pary przeciążonych metod posiada „radykalnie różny” typ. Dwa typy są radykalnie różne, jeżeli nie można rzutować jednego typu na drugi. W takim przypadku wybór odpowiedniej wersji przeciążonej metody jest jasno określony przez typy parametrów określone w czasie wykonania i nie może być zmieniony przez inną interpretację typów na etapie kompilacji. Dzięki temu główne źródło pomyłek zostaje wyeliminowane. Na przykład `ArrayList` posiada pierwszy konstruktor, który oczekuje wartości `int`, i drugi, który oczekuje `Collection`. Trudno wyobrazić sobie pomyłki związane z wyborem jednego z tych konstruktorów.

W wersjach wcześniejszych niż Java 5 wszystkie typy proste były radykalnie inne od typów referencyjnych, ale wprowadzenie mechanizmu autoboxingu spowodowało, że nie jest to już prawdą, co wywołało spore problemy. Weźmy pod uwagę następujący program:

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}
```

Program dodaje do sortowanego zbioru oraz listy liczby całkowite od `-3` do `2`, a następnie wykonuje trzy identyczne wywołania metody `remove` na zbiorze oraz liście. Większość ludzi oczekuje, że program usunie wszystkie wartości nieujemne (`0`, `1` i `2`) z listy i ze zbioru, a następnie wyświetli `[-3 -2 -1]` `[-3 -2 -1]`. W rzeczywistości program usuwa wartości nieujemne ze zbioru i wartości nieparzyste z listy, po czym wyświetla `[-3, -2, -1]` `[-2, 0, 2]`. Mylące działanie jest związane z realizacją wywołań.

Program działa w następujący sposób: wywołanie `set.remove(i)` wybiera przeciążoną metodę `remove(E)`, gdzie `E` jest typem elementu `set(Integer)`, i powoduje opakowanie wartości `i` z `int` do `Integer`. Jest to oczekiwane działanie, więc program usunie wartości dodatnie ze zbioru. Z kolei wywołanie `list.remove(i)` wybiera

przeciążoną metodę `remove(i)`, która usuwa element z podanej **pozycji** listy. Jeżeli zaczniemy od listy `[-3, -2, -1, 0, 1, 2]` i usuniemy zerowy element, następnie pierwszy, a na koniec drugi, pozostanie nam lista `[-2, 0, 2]` — i tajemnica została rozwiązana. Aby usunąć ten problem, wystarczy rzutować argument `list.remove` na `Integer`, co wymusi wywołanie prawidłowej przeciążonej wersji metody. Można również wywołać `Integer.valueOf` dla `i` oraz przekazać wynik do `list.remove`. W obu przypadkach program wyświetli `[-3, -2, -1]` `[-3, -2, -1]`, tak jak oczekiwaliśmy:

```
for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i); // lub remove(Integer.valueOf(i))
}
```

Mylące działanie demonstrowane w poprzednim przykładzie jest spowodowane tym, że interfejs `List<E>` posiada dwie przeciążone wersje metody `remove`: `remove(E)` oraz `remove(int)`. Przed Javą 5, w której zostało przeprowadzone uogólnienie, interfejs `List` posiadał zamiast `remove(E)` metodę `remove(Object)`, więc typy parametrów `Object` i `int` były radykalnie różne. Jednak po dodaniu typów ogólnych i mechanizmu autoboxingu te dwa typy parametru przestały być radykalnie różne. Inaczej mówiąc, dodanie do języka typów ogólnych i autoboxingu uszkodziło interfejs `List`. Na szczęście problem ten występuje jeszcze tylko w kilku API bibliotek Java, ale przykład ten jasno pokazuje, że gdy w języku dostępne są typy ogólne i autoboxing, przeciążanie należy stosować z rozwagą.

Dodanie w Javie 8 lambd i referencji do metod jeszcze bardziej wzmocniło prawdopodobieństwo pomyłki związanej z przeciążeniami. Rozważmy dwa poniższe fragmenty:

```
new Thread(System.out::println).start();

ExecutorService exec = Executors.newCachedThreadPool();
exec.submit(System.out::println);
```

Choć wywołanie konstruktora `Thread` i wywołanie metody `submit` wyglądają podobnie, pierwsze kompiluje się bez błędów, a drugie nie. Argumenty są identyczne (`System.out::println`) i zarówno konstruktor, jak i metoda posiada przeciążenie przyjmujące typ `Runnable`. Co się stało? Odpowiedź jest nieco zaskakująca — metoda `submit` ma jeszcze przeciążenie przyjmujące `Callable<T>`, którego nie posiada konstruktor `Thread`. Wydawać by się mogło, że nie powinno to wprowadzać żadnej różnicy, bo wszystkie przeciążenia metody `println` zwracają `void`, więc referencja do metody nie może być `Callable`. Choć to rozumowanie ma sens, nie odpowiada sposobowi działania algorytmu rozwiązującego przeciążenia. Równie zadziwiający może być fakt, iż wywołanie metody `submit` byłoby poprawne, gdyby metoda `println` nie była przeciążona. To właśnie połączenie przeciążenia referencji do metody (`println`) i przeciążenia wywoływanej metody (`submit`) powoduje, że algorytm przestaje się zachowywać tak, jak byśmy się tego spodziewali.

Mówiąc językiem technicznym, problemem jest to, że `System.out::println` jest **nieściśłą referencją do metody** [JLS, 15.13.1] oraz że „pewne wyrażenia argumentów zawierające niejawnie typowane wyrażenia lambda lub nieściśle referencje do metod są ignorowane przez testy zastosowalności, bo ich znaczenia nie da się ustalić aż do momentu poznania typu docelowego” [JLS, 15.12.2]. Nie przejmuj się, jeśli nie zrozumiałeś tego cytatu — jego docelowym odbiorcą są twórcy kompilatorów. Kluczowe jest to, że przeciążanie metod i konstruktorów różnymi interfejsami funkcyjnymi na tej samej pozycji argumentu powoduje niejasności. Dlatego **nie przeciążaj metod przyjmujących różne interfejsy funkcyjne na tej samej pozycji argumentu**. W kontekście omawianego tematu interfejsy funkcyjne nie są jakoś radykalnie różne. Kompilator Javy potrafi ostrzec o tego rodzaju problematycznych przeciążeniach, jeśli użyjemy w wierszu polecenia przełącznika `-Xlint:overloads`.

Typy tablicowe i klasy inne niż `Object` są również radykalnie różne. Radykalnie różne są dodatkowo typy tablicowe i interfejsy inne niż `Serializable` i `Cloneable`. Dwie różne klasy są traktowane jako **niezwiązane**, jeżeli żadna z klas nie jest przodkiem drugiej [JLS, 5.5]. Na przykład `String` i `Throwable` są niezwiązane. Niemożliwe jest, aby obiekt był instancją dwóch niezwiązanych klas, więc klasy niezwiązane są radykalnie inne.

Istnieją również inne pary typów, które nie mogą być konwertowane w dowolnym kierunku [JLS, 5.1.12], ale poza zamieszczonymi tu prostymi przypadkami większość programistów może mieć wątpliwości, która przeciążona wersja odnosi się do aktualnego zestawu parametrów. Zasady sterujące wyborem przeciążonych wersji są niezwykle skomplikowane. Zajmują one trzydzieści trzy strony w specyfikacji języka [JLS, 15.12.1-3] i niewielu programistów rozumie wszystkie ich subtelności.

Istnieją przypadki, gdy czujemy potrzebę naruszenia wskazówek z tego tematu, szczególnie przy rozwoju istniejących klas. Na przykład klasa `String` posiada od Javy 4 metodę `contentEquals(StringBuffer)`. W Javie 5 został dodany nowy interfejs o nazwie `CharSequence`, który jest wspólnym interfejsem dla `StringBuffer`, `StringBuilder`, `String`, `CharBuffer` i innych podobnych typów, które zostały zmodyfikowane, aby implementowały ten interfejs. W tym samym czasie, gdy do platformy został dodany `CharSequence`, do `String` została dodana przeciążona wersja metody `contentEqual`, która oczekuje `CharSequence`.

Choć wynikowa przeciążona wersja w jawny sposób narusza zasady z tego tematu, nie powoduje to problemów, ponieważ obie przeciążone metody realizują tę samą operację, jeżeli zostaną wywołane z referencją tego samego obiektu. Programista może nie wiedzieć, która przeciążona wersja metody została wywołana, ale nie pociąga to za sobą konsekwencji, ponieważ działają one identycznie. Standardową metodą zapewnienia takiego działania jest przeciążanie specyficznych przypadków:

```
// Upewnienie się, że obie metody działają identycznie przez przekazanie
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

Choć w bibliotekach języka Java zwykle stosowane są przedstawione tu zalecenia, istnieje wiele miejsc, w których zostały one naruszone. Na przykład klasa `String` udostępnia dwie przeciążone statyczne metody fabryczne: `valueOf(char[])` i `valueOf(Object)`, które realizują całkowicie różne zadania. Przykłady takie nie mogą być dla nas żadnym usprawiedliwieniem i powinny być traktowane jako anomalie oraz potencjalne źródło pomyłek.

Podsumujmy. Istnienie możliwości przeciążania metod nie oznacza, że powinniśmy ją wykorzystywać. Powinieneś unikać przeciążania metod z wieloma sygnaturami, mającymi tę samą liczbę parametrów. W niektórych przypadkach, szczególnie jeżeli chodzi o konstruktory, niemożliwe jest postępowanie według tych wskazówek. W takich przypadkach powinieneś unikać sytuacji, w których ten sam zbiór parametrów wejściowych może być przekazany do różnych wersji metody, przy dodatkowym wykorzystaniu rzutowania. Jeżeli nie można uniknąć takiej sytuacji, na przykład gdy wzbogacasz istniejącą klasę o implementację nowego interfejsu, powinieneś zapewnić, że wszystkie przeciążone wersje działają identycznie dla tych samych parametrów. Jeżeli nie uda Ci się tego zapewnić, programiści nie będą mogli efektywnie korzystać z przeciążonych metod i konstruktorów oraz nie będą mogli zrozumieć, dlaczego napisany kod źle działa.

Temat 53. Rozsądne korzystanie z metod varargs

Metody varargs, formalnie nazywane metodami o **zmiennej liczbie argumentów** [JLS, 8.4.1], oczekują zera lub więcej argumentów o zdefiniowanym typie. Podczas wywołania tych metod tworzona jest tablica o rozmiarze równym liczbie przekazanych argumentów, wartości argumentów są przepisywane do tej tablicy, a na koniec tablica jest przekazywana do metody.

Zamieszczona poniżej przykładowa metoda varargs oczekuje serii argumentów `int` i zwraca ich sumę. Jak można oczekiwać, wartością `sum(1, 2, 3)` jest 6, a wartością `sum()` jest 0.

```
// Prosty przykład metody varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Czasami zachodzi potrzeba napisania metody wymagającej **jednego** lub więcej argumentów określonego typu, a nie **zero** lub więcej. Dla przykładu weźmy obliczenie minimum z wartości argumentów `int`. Funkcja ta nie jest zdefiniowana, jeżeli klient nie przekaże argumentów. Możemy sprawdzić wielkość tablicy w czasie działania programu:

```
// NIEPRAWIDŁOWY sposób użycia varargs z co najmniej jednym argumentem!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Za mało argumentów");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

Takie rozwiązanie zawiera kilka problemów. Najpoważniejszy z nich polega na tym, że gdy klient wywoła tę metodę bez argumentów, ulegnie ona awarii w czasie działania, a nie kompilacji. Dodatkowo jest to brzydkie rozwiązanie. Konieczne jest dodanie jawnych kontroli poprawności `args`, jak również nie można używać pętli `for-each`, o ile nie zainicjuje się `min` wartością `Integer.MAX_VALUE`, co jest również brzydkie.

Na szczęście istnieje znacznie lepszy sposób na osiągnięcie tego samego efektu. Należy zadeklarować metodę oczekującą dwóch parametrów, jednego normalnego o oczekiwanym typie i drugiego parametru `varargs` o tym samym typie. Rozwiązanie to koryguje wszystkie niedoskonałości poprzedniego:

```
// Prawidłowy sposób użycia varargs do przekazania co najmniej jednego argumentu
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Jak widać w tym przykładzie, metody `varargs` są efektywne, gdy faktycznie potrzebujemy metody o zmiennej liczbie argumentów. Metody `varargs` zostały zaprojektowane dla `printf`, która została dodana do platformy w tym samym czasie, oraz dla bazowych mechanizmów refleksji (temat 65.), które zostały ulepszone, aby wykorzystać zalety zmiennej liczby argumentów. Zarówno `printf`, jak i refleksja znacznie skorzystały z użycia metod `varargs`.

Należy zachować szczególną ostrożność przy stosowaniu zmiennej liczby parametrów w sytuacjach wymagających wysokiej wydajności. Każde wywołanie metody `varargs` powoduje przydzielenie pamięci na tablicę i jej zainicjowanie. Jeżeli empirycznie stwierdzimy, że nie możemy sobie pozwolić na elastyczność metod

o zmiennej liczbie parametrów, należy zastosować wzorzec pozwalający jednocześnie zjeść ciastko i je mieć. Załóżmy, że określiliśmy, że 95% wywołań metody posiada najwyżej trzy parametry. Następnie deklarujemy pięć przeciążonych wariantów metody, posiadających od zera do trzech parametrów oraz jedną metodę ze zmienną liczbą parametrów, wykorzystywaną, gdy liczba argumentów przekroczy trzy.

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

Po takiej operacji będziemy musieli ponieść koszt tworzenia tablicy jedynie w 5% wszystkich wywołań, gdy liczba parametrów przekracza trzy. Podobnie jak większość optymalizacji wydajności, technika ta zwykle jest niepotrzebna, ale w niektórych przypadkach jest nieoceniona.

Technika ta jest zastosowana w statycznych metodach fabrycznych klasy EnumSet, co pozwoliło ograniczyć koszt tworzenia zbiorów wyników do minimum. Było to konieczne, ponieważ wydajność zbiorów wyliczeniowych musiała być konkurencyjna w stosunku do pól bitowych (temat 36.).

Podsumujmy. Metody varargs są dogodnym sposobem definiowania metod o zmiennej liczbie argumentów, ale nie powinny być nadużywane. Poprzedzaj parametr varargs dowolną liczbą parametrów wymaganych i bądź świadom konsekwencji wydajnościowych związanych z metodami varargs.

Temat 54. Zwracanie pustych tablic lub kolekcji zamiast wartości null

Często można zobaczyć metody, które wyglądają następująco:

```
// Zwraca null, aby wskazać pustą kolekcję. Nie rób tak!
private final List<Cheese> cheesesInStock = ...;

/**
 * @return zwraca tablicę zawierającą wszystkie rodzaje serów w sklepie
 *      lub null, jeżeli nie można kupić żadnego sera
 */
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null
        : new ArrayList<>(cheesesInStock);
}
```

Nie ma żadnego powodu, aby odróżniać sytuację, gdy w sklepie nie ma żadnego sera. Dodając takie wyróżnienie, powodujemy, że kod klienta musi obsługiwać również wartości null, na przykład:

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Bardzo dobry, akurat dla mnie.");
```

Taka obsługa jest niezbędna niemal we wszystkich przypadkach, gdy metoda zwraca wartość null zamiast tablicy o długości zero. Jest to metoda powodująca powstanie błędów, ponieważ programista tworzący kod klienta może zapomnieć o dopisaniu fragmentu kodu, obsługującego wartość null. Błąd taki może nie ujawniać się latami, ponieważ metoda zwykle zwraca jeden obiekt lub więcej obiektów. Mniej ważne, choć również warte zauważenia jest niepotrzebne komplikowanie także metody zwracającej wartość null.

Istnieją opinie, że zwracanie wartości null zamiast pustej tablicy jest lepsze, ponieważ pozwala uniknąć wykonywania operacji tworzenia tablicy. Argument ten można obalić z dwóch powodów. Po pierwsze, nie jest zalecane zajmowanie się wydajnością aplikacji na tym poziomie, jeżeli profilowanie nie wykaże, że metoda ta faktycznie powoduje problemy wydajnościowe (temat 67.). Po drugie, możliwe **jest** zwracanie stale tej samej pustej tablicy lub kolekcji w przypadku każdego wywołania bez wielokrotnej alokacji miejsca w pamięci. Oto typowy kod, który zwraca pustą kolekcję. Najczęściej to wszystko, czego potrzebujemy:

```
// Właściwy sposób zwrócenia prawdopodobnie pustej kolekcji
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

Jeśli w jakiejś rzadkiej sytuacji profilowanie wykaże, że alokacja pustych kolekcji wpływa negatywnie na wydajność, możemy zdecydować się na uniknięcie alokacji przez wielokrotne zwracanie tej samej, **niezmiennej** i pustej kolekcji, ponieważ obiekty niezmiennicze można dowolnie współdzielić (temat 17.). Poniżej znajduje się fragment kodu realizujący opisane zadanie z wykorzystaniem metody `Collections.emptyList()`. Gdyby zwracanym elementem był zbiór, należałoby użyć metody `Collections.emptySet()`; gdyby było zwracane odwzorowanie, odpowiednią metodą byłoby `Collections.emptyMap()`. Pamiętaj, że jest to jedynie optymalizacja, po którą nie trzeba sięgać zbyt często. Jeśli sądzisz, że jest niezbędna, zmierz wydajność przed jej wprowadzeniem i po jej wprowadzeniu, aby mieć pewność, że faktycznie pomaga:

```
// Optymalizacja - unikamy alokacji pamięci dla nowych, pustych kolekcji
public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? Collections.emptyList()
    : new ArrayList<>(cheesesInStock);
}
```

W przypadku tablic sytuacja jest taka sama jak w kolekcjach. Nigdy nie zwracaj `null` zamiast pustej tablicy. Standardowo zawsze zwracaj tablicę o właściwej długości, która akurat w tej sytuacji wynosi zero. Zwróć uwagę, że tablica o długości zero przekazywana jest do metody `toArray` w celu określenia wymaganego typu, którym w tym przypadku jest `Cheese[]`:

```
// Prawidłowy sposób zwracania pustej tablicy
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(new Cheese[0]);
}
```

Jeśli wierzysz, że alokacja tablicy o zerowej długości wpływa negatywnie na wydajność, możesz zawsze zwracać tę samą tablicę o zerowej długości, ponieważ tablice o długości zero są niezmiennie:

```
// Optymalizacja - unikamy alokacji pamięci dla nowych, pustych tablic
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

W zoptymalizowanej wersji do każdego wywołania `toArray` przekazujemy **tę samą**, pustą tablicę. To właśnie ta tablica zostanie zwrócona z `getCheese`, jeśli `cheesesInStock` jest puste. Nie dokonuj wcześniejszej alokacji tablicy przekazywanej do `toArray` w nadziei na poprawę wydajności. Badania pokazały, że efekt jest odwrotny od zamierzonego [Shipilëv16]:

```
// Tak nie rób - wczesna alokacja tablicy pogarsza wydajność!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

Podsumujmy. **Nie ma powodu, by z metody zwracającej tablicę lub kolekcję zwracać wartość `null` zamiast tablicy o długości zero lub pustej kolekcji.** Komplikuje to wynikowe API, naraża programistów na większą liczbę pomyłek i nie oferuje żadnych zysków pod kątem wydajności.

Temat 55. Rozsądne zwracanie obiektów opcjonalnych

Przed Javą 8 istniały dwa podejścia do tworzenia metod, które w pewnych okolicznościach nie mogły zwrócić żadnej sensownej wartości. Mogliśmy zgłosić wyjątek lub zwrócić wartość typu `null` (o ile zwracany typ był typem referencyjnym). Żadne z tych rozwiązań nie jest idealne. Wyjątki powinny być zarezerwowane dla sytuacji nadzwyczajnych (temat 69.). Zgłoszenie wyjątku jest operacją kosztowną, bo wymaga zebrania całego śladu stosu w momencie jego tworzenia. Zwracanie `null` nie

ma tych wad, ale posiada inne. Jeśli metoda zwraca `null`, trzeba w kodzie ją używającym korzystać ze specjalnego przypadku obsługującego tę sytuację, chyba że programista jest w stanie **dowieść**, że wartość `null` jest niemożliwa do zwrócenia. Jeżeli klienty nie zapewniają sprawdzenia zwrócenia wartości `null` i zapisują ją w jakiejś strukturze danych, wyjątek `NullPointerException` zostanie zgłoszony w późniejszym terminie w miejscu, które zapewne nie ma nic wspólnego z powstaniem błędu.

Od Javy 8 istnieje trzecie podejście do tworzenia metod, które mogą nie być w stanie zwrócić wartości. Klasa `Optional<T>` reprezentuje niezmienny kontener, który przechowuje pojedynczą, różną od `null` referencję do `T` lub nie przechowuje niczego. Obiekt opcjonalny, który niczego nie przechowuje, nazywamy **pustym**. Mówimy, że w niepustym obiekcie opcjonalnym wartość jest **obecna**. W zasadzie obiekt opcjonalny to niezmienna kolekcja, która potrafi przechowywać co najwyżej jeden element. Choć `Optional<T>` nie implementuje `Collection<T>`, w zasadzie mógłby to robić.

Metoda, która standardowo zwraca typ `T`, ale w pewnych okolicznościach nie może tego zrobić, może zostać zadeklarowana jako metoda zwracająca `Optional<T>`. W ten sposób metoda ma szansę zwrócić pusty wynik, który oznacza brak możliwości zwrócenia wyniku poprawnego. Metoda zwracająca typ `Optional` jest bardziej elastyczna i łatwiejsza w użyciu niż metoda zgłaszająca wyjątek, a przy tym mniej naraża kod kliencki na błędy związane z wartością `null`.

W temacie 30. przedstawiliśmy poniższą metodę obliczającą maksymalną wartość w kolekcji na podstawie porządku naturalnego zawartych w niej elementów.

```
// Zwraca maksymalną wartość w kolekcji - zgłasza wyjątek, gdy kolekcja jest pusta
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Pusta kolekcja");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return result;
}
```

Metoda zgłasza wyjątek `IllegalArgumentException`, jeśli przekazana do niej kolekcja jest pusta. W tamtym temacie wspomnieliśmy, że lepszym rozwiązaniem byłoby zwrócenie `Optional<E>`. Oto wygląd metody po wprowadzeniu takiej poprawki:

```
// Zwraca maksymalną wartość w kolekcji jako Optional<E>
public static <E extends Comparable<E>> Optional<E>
max(Collection<E> c) {
    if (c.isEmpty())
        return Optional.empty();
}
```

```

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return Optional.of(result);
}

```

Jak możemy się przekonać, zwrócenie wartości opcjonalnej nie jest niczym trudnym. Wystarczy otworzyć odpowiedni obiekt, wykorzystując statyczną metodę fabryczną. W przedstawionym programie używamy dwóch takich metod: `Optional.empty()` zwraca pusty obiekt opcjonalny, a `Optional.of()` zwraca obiekt opcjonalny z wartością inną niż `null`. Błędem programistycznym będzie przekazanie `null` do `Optional.of()`. W takiej sytuacji metoda zareaguje zgłoszeniem wyjątku `NullPointerException`. Metoda `Optional.ofNullable(value)` dopuszcza przekazanie wartości `null`, ale w takiej sytuacji zwróci pusty obiekt opcjonalny. **Nigdy nie zwracaj wartości `null` z metody zwracającej typ `Optional`** — przeczy to przesłankom przemawiającym za powstaniem tego typu.

Wiele operacji kończących związanych ze strumieniami zwraca obiekt opcjonalny. Jeżeli skonwertujemy metodę `max` w taki sposób, aby korzystała ze strumieni, operacja `max` z `Stream` automatycznie wygeneruje dla nas obiekt opcjonalny (w tym rozwiązaniu musimy jednak jawnie wskazać komparator):

```

// Zwraca maksymalną wartość w kolekcji jako Optional<E> - używa strumienia
public static <E extends Comparable<E>> Optional<E>
    ↪max(Collection<E> c) {
    return c.stream().max(Comparator.naturalOrder());
}

```

Kiedy warto zdecydować się na zwracanie obiektu opcjonalnego zamiast na zwracanie `null` lub zgłaszanie wyjątków? **Obiekty opcjonalne mają w zamyśle działać podobnie do wyjątków przechwytywanych** (temat 71.), bo **zmuszają** wykorzystującego kod do zmierzenia się z faktem, iż wartość może nie zostać zwrócona. Zgłoszenie wyjątku nieprzechwytywanego lub wartości `null` pozwala programiście na zignorowanie takiej ewentualności, co może mieć katastrofalne konsekwencje. Z drugiej strony zgłoszenie wyjątku przechwytywanego wymaga od programisty dodatkowego, dosyć mało wygodnego kodu otaczającego.

Jeśli metoda zwraca obiekt opcjonalny, klient musi określić, jaką akcję podejmie, gdy metoda nie jest w stanie zwrócić wartości. Można wskazać wartość domyślną:

```

// Wykorzystanie obiektu opcjonalnego do wskazania wartości domyślnej
String lastWordInLexicon = max(words).orElse("Brak słów...");

```

Ewentualnie można zgłosić najbardziej odpowiedni wyjątek. Zwróć uwagę, że przekazujemy fabrykę wyjątków, a nie rzeczywisty wyjątek. W ten sposób unikamy kosztownego tworzenia wyjątku do momentu, aż stanie się to niezbędnym.

```
// Wykorzystanie obiektu opcjonalnego do zgłoszenia wybranego wyjątku
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

Jeśli jesteś w stanie **dowieść**, że obiekt opcjonalny nie może być pusty, po prostu pobierz zawartą w nim wartość bez podejmowania żadnych akcji w sytuacji, gdy jest pusty. Jeśli jednak Twoje założenie okaże się błędne, kod zgłosi wyjątek `NoSuchElementException`:

```
// Wykorzystanie obiektu opcjonalnego, gdy wiemy, że istnieje w nim wartość
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

Czasami zdarzają się sytuacje, w których pobranie wartości domyślnej jest kosztowne, więc chcemy uniknąć tego kosztu, o ile to możliwe. Klasa `Optional` jest na takie sytuacje przygotowana i oferuje metodę przyjmującą `Supplier<T>`, którego wywołuje tylko wtedy, gdy jest to niezbędne. Metoda ta nosi nazwę `orElseGet`, choć byłoby lepiej, aby nazywała się `orGetCompute`, bo jest blisko związana z trzema metodami interfejsu `Map`, których nazwy rozpoczynają się od `compute`. Klasa `Optional` zawiera również kilka innych metod wykorzystywanych w bardziej wyspecjalizowanych przypadkach: `filter`, `map`, `flatMap` i `ifPresent`. W Javie 9 dodano dwie nowe metody: `or` i `ifPresentOrElse`. Jeśli podstawowe metody przedstawione w tym temacie nie pasują do wykonywanego zadania, zajrzyj do dokumentacji, aby sprawdzić, czy nie mogą wykonać właściwego zadania metody zaawansowane.

Jeśli żadna z dostępnych w klasie metod nie spełnia potrzeb, klasa `Optional` oferuje metodę `isPresent()`, która działa jako sieć bezpieczeństwa. Zwraca wartość `true`, jeśli obiekt opcjonalny zawiera wartość, lub `false`, jeśli jest pusty. Dzięki tej metodzie można w dowolny sposób przetworzyć wynik, ale pamiętaj, aby korzystać z niej mądrze. Wiele standardowych użyć metody `isPresent` można zastąpić jedną z wymienionych wcześniej metod, co spowoduje, że wynikowy kod będzie krótszy, czystszy i bardziej idiomatyczny.

Przyjrzyjmy się poniższemu fragmentowi, który wyświetla identyfikator procesu nadrzędnego względem aktualnego procesu lub tekst nie dotyczy, jeśli proces nie ma procesu nadrzędnego. Fragment korzysta z klasy `ProcessHandle` wprowadzonej w Javie 9:

```
Optional<ProcessHandle> parentProcess = ph.parent();
System.out.println("ID procesu nadrzędnego: " +
    ↳ parentProcess.isPresent() ?
        String.valueOf(parentProcess.get().pid()) : "nie dotyczy");
```

Powyższy fragment można zastąpić poniższym, który używa metody `map` z `Optional`:

```
System.out.println("ID procesu nadrzędnego: " +
    ph.parent().map(h -> String.valueOf(h.pid())).orElse
    ↳ ("nie dotyczy"));
```

Pisząc kod wykorzystujący strumienie, nierzadko znajdziemy się w sytuacji, w której będziemy mieć do czynienia z `Stream<Optional<T>>`, co wymaga umieszczenia w `Stream<T>` wszystkich elementów w niepustych obiektach opcjonalnych, aby zapewnić ich przetworzenie. W Javie 8 odpowiednią konwersję zapewnia kod:

```
streamOfOptionals
    .filter(Optional::isPresent)
    .map(Optional::get)
```

W Javie 9 do klasy `Optional` dodano metodę `stream()`. Metoda ta jest adapterem, który zamienia `Optional` w `Stream` zawierający element, jeśli takowy istnieje, lub pusty strumień, jeśli elementu nie ma. W połączeniu z metodą `flatMap` z `Stream` (temat 45.) metoda ta zapewnia bardzo zwężony kod realizujący to samo zadanie co kod powyższy:

```
streamOfOptionals.
    .flatMap(Optional::stream)
```

Nie wszystkie zwracane typy skorzystają na zamianie na obiekty opcjonalne. **Typy kontenerowe, czyli kolekcje, odwzorowania, strumienie, tablice i obiekty opcjonalne nie powinny być otaczane obiektem opcjonalnym.** Zamiast zwracać pusty `Optional<List<T>>`, lepiej po prostu zwrócić pusty obiekt `List<T>` (temat 54.). W ten sposób wyeliminujemy w kodzie klienckim potrzebę obsługi wartości opcjonalnej. Klasa `ProcessHandle` posiada co prawda metodę `arguments`, która zwraca `Optional<String[]>`, ale metodę tę należy traktować jako aberrację, której nie należy naśladować.

W takim razie w jakich sytuacjach należy zadeklarować metodę jako zwracającą `Optional<T>` zamiast `T`? Ogólna zasada jest następująca: **zadeklaruj metodę jako zwracającą `Optional<T>`, jeśli może nie być w stanie zwrócić wyniku i klienci muszą dokonywać dodatkowego przetwarzania w przypadku braku wartości.** Trzeba sobie zdawać sprawę, że zwracanie `Optional<T>` nie jest bezkosztowe. Obiekt `Optional` musi zostać zaalokowany i zainicjalizowany, a odczyt wartości z obiektu wymaga dodatkowego wywołania. Elementy te czynią obiekty opcjonalne nieodpowiednimi w przypadku niektórych rozwiązań wymagających ogromnej wydajności. To, czy konkretna metoda zalicza się do tej kategorii, powinno zostać sprawdzone na podstawie odpowiednich pomiarów (temat 67.).

Zwrócenie obiektu opcjonalnego zawierającego otoczony typ prosty jest wyjątkowo kosztowne, bo w porównaniu z bezpośrednim zwróceniem wartości mamy tworzenie dwóch obiektów zamiast zera. Dlatego projektanci biblioteki zauważyli potrzebę dodania klas analogicznych do `Optional<T>`, ale dotyczących typów prostych `int`, `long` i `double`. Tymi klasami są `OptionalInt`, `OptionalLong` i `OptionalDouble`. Klasy te zawierają większość, ale nie wszystkie, metod `Optional<T>`. Z tego powodu nigdy nie zwracaj standardowego obiektu opcjonalnego dla otaczanych typów prostych, chyba że korzystasz z rzadziej stosowanych typów prostych, takich jak `Boolean`, `Byte`, `Character`, `Short` i `Float`.

Do tej pory omówiliśmy zwracanie obiektów opcjonalnych i ich przetworzenie tuż po zwróceniu. Nie opisywaliśmy innych potencjalnych użycí, ponieważ w większości sytuacji są one bardzo podejrzanym wykorzystaniem opisywanego mechanizmu. Przykładowo — nigdy nie używaj obiektów opcjonalnych jako wartości w odwzorowaniach. Jeśli to zrobisz, będziesz miał dwa sposoby wyrażenia braku elementu w odwzorowaniu: klucz dla elementu może w odwzorowaniu nie istnieć lub istnieć, ale zwracać pusty obiekt opcjonalny. Wprowadza to zbędną złożoność, co jedynie prowadzi do błędów i niepotrzebnych pytań. Możemy powiedzieć, że **w zasadzie nigdy nie jest odpowiednim zastosowanie obiektu opcjonalnego jako klucza, wartości lub elementu w kolekcji albo tablicy.**

Jedno istotne pytanie pozostało do tej pory bez odpowiedzi. Czy kiedykolwiek sensowne jest umieszczenie obiektu opcjonalnego w polu instancji? Często to „zły znak”, bo sugeruje, że najprawdopodobniej lepiej będzie utworzyć podklasę zawierającą pola opcjonalne. Rozważmy przykładową klasę `NutritionFacts` z tematu 2. Instancja tej klasy ma wiele pól, które nie są wymagane. Nie możemy utworzyć podklasy dla każdej możliwej kombinacji pól opcjonalnych. Poza tym typami tych pól są typy proste, co uczyniłoby ich brak dosyć dziwnym. Najlepszym API dla klasy `NutritionFacts` byłoby zwracanie obiektu opcjonalnego przez metodę pobierającą pole opcjonalne. W takiej sytuacji przechowywanie obiektów opcjonalnych w polach instancji ma sens.

Podsumujmy. Jeśli tworzymy metodę, która nie zawsze będzie w stanie zwrócić wartość, i wierzymy, że dla użytkowników metody ważne będzie każdorazowe uwzględnienie takiej ewentualności, warto rozważyć zwrócenie obiektu opcjonalnego. Musimy jednak pamiętać o prawdziwych konsekwencjach wydajnościowych związanych ze zwracaniem takich obiektów — w metodach wymagających szczególnie dużej wydajności zwrócenie `null` lub zgłoszenie wyjątku może okazać się lepszym wyborem. Prawie nigdy nie stosuj obiektów opcjonalnych w sytuacjach innych niż zwracanie wartości.

Temat 56. Tworzenie komentarzy dokumentujących dla udostępnianych elementów API

Jeżeli API ma być użyteczne, musi posiadać dokumentację. Najczęściej dokumentacja API była generowana ręcznie i synchronizacja dokumentacji z aktualnym kodem była kłopotliwa. Środowisko programowania języka Java upraszcza to zadanie za pomocą programu o nazwie **Javadoc**. Program ten automatycznie generuje dokumentację API na podstawie kodu źródłowego, połączonego z odpowiednio sformatowanymi **komentarzami dokumentującymi**, często zwanymi **komentarzami doc**.

Choć konwencje te nie są częścią języka Java, tworzą de facto API, które powinien znać każdy programista. Konwencje te są zdefiniowane na stronie *How to Write Doc Comments* [Javadoc-guide]. Choć strona ta nie została zaktualizowana od Javy 4, jest nadal niezastąpionym źródłem informacji. W Javie 9 zostały dodane jeden ważny znacznik, `{@index}`, w Javie 8 również jeden ważny znacznik, `{@implSpec}`, a w Javie 5 dwa ważne znaczniki `{@literal}` oraz `{@code}`. Choć brakuje ich na wspomnianej stronie WWW, znaczniki te zostaną przedstawione dokładniej w dalszej części tematu.

Aby właściwie udokumentować API, każdą deklarację udostępnianej klasy, interfejsu, konstruktora, metody i pola należy poprzedzić komentarzem dokumentującym. Jeżeli klasa jest serializowana, należy udokumentować również jej postać serializowaną (temat 87.). Jeżeli komentarz dokumentujący nie zostanie utworzony, Javadoc może co najwyżej umieścić w dokumentacji deklarację danego elementu. Korzystanie z API, które nie posiada odpowiedniej dokumentacji, jest frustrujące i powoduje powstawanie błędów. Aby kod był łatwy w utrzymaniu, należy również napisać komentarze dokumentujące dla nieudostępnianych klas, interfejsów, konstruktorów, metod i pól, choć taka dokumentacja nie musi być tak rozbudowana jak dla udostępnianego API.

Komentarze dokumentujące dla metod powinny zwięźle opisywać założenia komunikacji pomiędzy metodą a jej klientem. Z wyjątkiem metod w klasach zaprojektowanych jako baza dziedziczenia (temat 19.) założenia powinny opisywać **co** dana metoda robi, a nie **jak** to robi. Komentarze dokumentujące powinny zawierać wszystkie **warunki wstępne**, czyli warunki, jakie muszą być spełnione przed wykonaniem metody, oraz **warunki końcowe**, czyli warunki, jakie są spełnione po jej wykonaniu. Warunki wstępne są najczęściej opisywane niejawnie w znacznikach `@throws`, opisujących wyjątki zgłaszane przez metodę — każdy wyjątek jest związany z naruszeniem warunku wstępnego. Warunki wstępne mogą być również wymieniane wraz z opisem parametrów w znaczniku `@param`.

Oprócz warunków wstępnych i końcowych metoda powinna dokumentować wszystkie **efekty uboczne**. Efektem ubocznym nazywamy każdą, dającą się zaobserwować zmianę stanu systemu, która nie jest w sposób oczywisty związana z warunkami końcowymi. Na przykład jeżeli metoda rozpoczyna wątek tła, dokumentacja powinna zawierać informację o tym fakcie.

Aby w pełni opisać założenia metody, komentarz dokumentujący metody powinien zawierać znacznik `@param` dla każdego parametru, znacznik `@return` (chyba że metoda deklaruje zwracany typ `void`) oraz znaczniki `@throws` dla wszystkich wyjątków zgłaszanych przez metodę, niezależnie od tego, czy są weryfikowane, czy nie (temat 74.). Gdyby treść znacznika `@return` była w efekcie taka sama jak opis metody, w zależności od przyjętej w projekcie konwencji można taki opis pominąć.

Konwencja zaleca, aby po znaczniku @param lub @return umieszczone było zdanie z rzeczownikiem opisującym wartość reprezentowaną przez parametr lub zwracaną wartość. W rzadkich przypadkach zamiast zdań stosowane są wyrażenia matematyczne (przykładów szukaj w klasie BigInteger). Tekst po znaczniku @throws powinien składać się ze słowa „jeżeli”, po którym następuje zdanie z rzeczownikiem opisującym warunki, w których zgłaszany jest wyjątek. Konwencja zaleca, aby po zdaniu dotyczącym znacznika @param, @return lub @throws nie umieszczać znaku kropki. Wszystkie te konwencje są zamieszczone w poniższym krótkim komentarzu:

```
/**
 * Zwraca element z określonej pozycji tej listy.
 *
 * <p>Metoda ta <i>nie</i> gwarantuje wykonania w stałym czasie.
 * W niektórych implementacjach może działać w czasie
 * proporcjonalnym do pozycji elementu.
 *
 * @param index indeks elementu do zwrócenia; musi być nieujemny
 * i mniejszy niż rozmiar tej listy
 * @return element na podanej pozycji tej listy
 * @throws IndexOutOfBoundsException jeżeli indeks jest poza zakresem
 * ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

Zwróć uwagę na znaczniki HTML zastosowane w tym komentarzu (<p> oraz <i>). Program Javadoc zamienia komentarze dokumentujące na HTML i w wynikowym dokumencie HTML znajdują się odpowiednie jednostki kodu HTML. Czasami programiści są zbyt gorliwi i samodzielnie umieszczają tabele HTML w swoich komentarzach — nie jest to jednak zbyt częste.

Warto również zwrócić uwagę na znacznik Javadoc {@code} wokół fragmentów kodu w klauzuli @throws. Służy on do dwóch celów — powoduje, że fragment kodu jest generowany z użyciem znacznika <code> (czcionka o stałej szerokości) oraz wyłącza przetwarzanie znaczników HTML i zagnieżdżonych znaczników Javadoc w danym fragmencie kodu. Ta druga własność pozwala nam użyć znaku mniejszości (<) we fragmencie kodu, pomimo że jest to metaznak HTML. Aby dołączyć wielowierszowy przykład kodu do komentarza dokumentującego, należy ująć znacznik Javadoc {@code} w znaczniki HTML <pre>. Inaczej mówiąc, należy umieścić przed wielowierszowym przykładem kodu sekwencję <pre>{@code}, a po nim sekwencję }</pre>. W ten sposób zachowane zostaną wszystkie przejścia do nowych wierszy i nie trzeba będzie konwertować metaznaków HTML. Niestety, znak @ nadal trzeba traktować w sposób szczególny, jeśli w przykładach pojawiają się adnotacje.

Na koniec zwróć uwagę na wykorzystanie słowa „ten” w komentarzu dokumentującym. Zgodnie z konwencją, słowo „ten”, wykorzystane w komentarzu dokumentującym do metody, zawsze odnosi się do obiektu, na rzecz którego wywoływana jest metoda.

Jak wspomniano w temacie 15., gdy projektujemy klasę na potrzeby dziedziczenia, musimy wyraźnie opisać **wzorce użycia metod samego siebie** (czyli wywołania innych metod tej samej klasy), aby programiści implementujący podklasy znali semantykę przesłanianych metod. Wzorce te warto oznaczyć znacznikiem `@implSpec` wprowadzonym w Javie 8. Przypomnijmy, że zwykły komentarz dokumentujący opisuje kontrakt między metodą a jej klientem; komentarz `@implSpec` opisuje kontrakt między metodą a jej podklasą, co pozwala podklasie polegać na zachowaniu implementacji, jeśli metoda jest dziedziczona lub wywoływana poprzez konstrukcję `super`.

```
/**
 * Zwraca true, jeśli kolekcja jest pusta.
 *
 * @implSpec
 * Ta implementacja zwraca {@code this.size() == 0}.
 *
 * @return true, jeśli kolekcja jest pusta
 */
public boolean isEmpty() { ... }
```

W Javie 9 narzędzie Javadoc nadal ignoruje znacznik `@implSpec`, chyba że użyje się w wierszu poleceń dodatkowy przełącznik `-tag "implSpec:a:Implementation Requirements:"`. Miejmy nadzieję, że w następnym wydaniu języka to niedopatrzenie zostanie poprawione.

Nie należy zapominać, że trzeba podjąć specjalne akcje w celu wygenerowania dokumentacji zawierającej metaznaki HTML, takie jak znaki większości (`>`) i mniejszości (`<`) oraz ampersand (`&`). Najlepszym sposobem na uzyskanie tych znaków w dokumentacji jest ich ujęcie w znacznik `{@literal}`, który wstrzymuje przetwarzanie znaczników HTML i zagnieżdżonych znaczników Javadoc. Działa on podobnie jak znacznik `{@code}`, ale tekst nie jest wyświetlany za pomocą czcionki o stałej szerokości. Na przykład poniższy fragment Javadoc:

```
* Szereg geometryczny jest zbieżny, gdy {@literal |r| < 1 }.
```

daje wynik: „Szereg geometryczny jest zbieżny, gdy $|r| < 1$ ”. Znacznik `{@literal}` mógłby być, z tym samym efektem w dokumentacji, umieszczony tylko wokół znaku mniejszości, a nie wokół całej nierówności, ale komentarz dokumentujący w kodzie źródłowym byłby mniej czytelny. Ilustruje to ogólną zasadę mówiącą, że **komentarze dokumentujące powinny być czytelne zarówno w kodzie źródłowym, jak i w wygenerowanej dokumentacji**. Jeżeli nie można tego osiągnąć, czytelność wygenerowanej dokumentacji jest ważniejsza niż czytelność kodu źródłowego.

Pierwsze zdanie każdego komentarza dokumentującego jest krótkim **podsumowaniem** elementu, który jest opisywany w komentarzu. Podsumowaniem w komentarzu zamieszczonego powyżej przykładu jest: „Zwraca element z określonej pozycji tej listy”. Podsumowanie musi samodzielnie opisywać działanie elementu. Aby nie było pomyłek, **żadne dwa składniki lub konstruktory w klasie czy interfejsie nie mogą mieć tego samego podsumowania**. Szczególnie zwróć uwagę na przeciążanie, w przypadku którego naturalne może się wydawać użycie tego samego opisu dla wszystkich wariantów (co nie jest odpowiednie dla komentarzy dokumentujących).

Uważaj na stosowanie kropek w pierwszym zdaniu komentarza dokumentującego. Jeżeli w środku zdania umieścisz kropkę, zakończy ona podsumowanie. Na przykład komentarz dokumentujący może rozpoczynać się od: „Stopnie naukowe, takie jak dr, prof. lub mgr”, a w opisie znajdzie się tylko „Stopnie naukowe, takie jak dr, prof.”. Problem wynika z tego, że krótki opis kończy się na pierwszej kropce, po której znajduje się spacja, tabulator lub koniec wiersza (lub pierwszy znacznik bloku) [Javadoc-ref]. W tym przypadku po kropce w skrócie „prof.” znajduje się spacja. Najlepszym rozwiązaniem jest ujęcie przeszkadzającej kropki oraz całego skojarzonego z nią tekstu w znacznik `{@literal}` — przestanie ona wtedy poprzedzać spację w kodzie źródłowym:

```
/**
 * Stopnie naukowe, takie jak dr, {@literal prof.} lub mgr.
 */
public class Degree { ... }
```

Nieco mylące może być stwierdzenie, że podsumowanie jest pierwszym **zdaniem** w komentarzu dokumentującym. Rzadko jest to jednak pełne zdanie. W przypadku metod i konstruktorów podsumowanie powinno mieć formę równoważnika zdania (zawierającego dowolny obiekt), opisującego zadania wykonywane przez metodę, na przykład:

- `ArrayList(int initialCapacity)` — tworzy pustą listę o określonej początkowej pojemności.
- `Collection.size()` — zwraca liczbę elementów w tej kolekcji.

Zauważ, że przykłady wykorzystują zdanie oznajmujące w trzeciej osobie („zwraca liczbę”), a nie zdanie rozkazujące w drugiej osobie („zwróć liczbę”).

W przypadku klas, interfejsów i pól podsumowanie powinno być zdaniem, opisującym element reprezentowany przez obiekt klasy, interfejsu lub przez samo pole. Na przykład:

- `Instant` — konkretny punkt na linii czasu.
- `Math.PI` — wartość `double` przybliżająca liczbę pi, czyli stosunek obwodu koła do jego średnicy.

W Javie 9 do kodu HTML generowanego przez Javadoc dodawany jest indeks nazw. Indeks ten ma za zadanie ułatwić poruszanie się po rozbudowanej dokumentacji API i przyjmuje postać pola wyszukiwania w prawym górnym narożniku strony. Wpisując tekst w polu, otrzymujemy podpowiedzi dotyczące pasujących stron. Elementy API takie jak klasy, metody lub pola są indeksowane automatycznie. Czasem warto dodać do indeksu dodatkowe terminy istotne z punktu widzenia API. Temu celowi służy znacznik `{@index}`. Wystarczy wybrany termin otoczyć tym znacznikiem, aby zaczął być uwzględniany w indeksie. Oto przykład:

** Ta metoda jest zgodna ze standardem {@index IEEE 754}.*

Typy ogólne, wyliczeniowe i adnotacje wymagają specjalnej obsługi w komentarzach dokumentujących. **Przy dokumentowaniu typu lub metody ogólnej należy pamiętać o udokumentowaniu wszystkich parametrów typu:**

```
/**
 * Obiekt odwzorowujący klucze na wartości. Mapa nie może zawierać
 * powielonych kluczy; każdy z nich może odwzorowywać najwyżej jedną wartość.
 *
 * (pozostała część pominięta)
 *
 * @param <K> typ kluczy przechowywanych w tym odwzorowaniu
 * @param <V> typ odwzorowanych wartości
 */
public interface Map<K, V> { ... }
```

Przy dokumentowaniu typu wyliczeniowego należy pamiętać, aby udokumentować stałe, jak również typ i wszystkie publiczne metody. Jeżeli komentarz jest krótki, można go umieścić w całości w jednym wierszu.

```
/**
 * Sekcja instrumentów orkiestry symfonicznej.
 */
public enum OrchestraSection {
    /** Instrumenty dęte, takie jak flet, klarnet i obój. */
    WOODWIND,

    /** Instrumenty miedziowe, takie jak róg i puzon. */
    BRASS,

    /** Instrumenty perkusyjne takie jak bębny i dzwonki. */
    PERCUSSION,

    /** Instrumenty strunowe, takie jak wiolonczela i altówka. */
    STRING;
}
```

Przy dokumentowaniu typu adnotacyjnego należy pamiętać o dokumentowaniu jego składników, jak również samego typu. Składniki należy udokumentować

z użyciem równoważników zdań, tak jakby były polami. W krótkim opisie typu należy użyć zdania informującego o przeznaczeniu elementu mającego adnotację danego typu.

```
/**
 * Wskazuje, że adnotowana metoda jest metodą testową,
 * która musi zgłosić wyjątek, aby skończyła się sukcesem.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * Wyjątek, który musi być zgłoszony przez adnotowaną metodę
     * aby przejść test (test może zgłosić dowolny
     * podtyp wyjątku opisanego przez ten obiekt klasy).
     */
    Class<? extends Throwable> value();
}
```

Komentarze na poziomie pakietu powinny być umieszczane w pliku o nazwie *package-info.java*. Oprócz komentarzy dokumentujących na poziomie pakietu *package-info.java* może zawierać deklarację pakietu i adnotacje pakietu (ale nie jest to wymagane). Jeśli zdecydujesz się korzystać z systemu modułów (temat 15.), komentarze na poziomie modułu powinny się znaleźć w pliku *module-info.java*.

Dwoma często pomijanymi aspektami eksportowanego API klas są bezpieczeństwo dla typów i możliwość serializacji. **Niezależnie od tego, czy klasa jest bezpieczna dla wątków, czy nie, powinniśmy udokumentować jej poziom bezpieczeństwa** zgodnie z opisem w temacie 82. Jeżeli klasa jest możliwa do serializacji, należy udokumentować jej postać serializowaną zgodnie z opisem w temacie 87.

Javadoc ma możliwość „dziedziczenia” komentarzy metod. Jeżeli element API nie posiada komentarza komentującego, Javadoc wyszukuje najbliższy komentarz, preferując przy tym interfejsy nad klasami bazowymi. Szczegóły działania algorytmu wyszukującego można znaleźć w *The Javadoc Reference Guide* [Javadoc-ref]. Można również **dziedziczyć** części komentarzy z typów bazowych przez użycie znacznika `@inheritDoc`. Oznacza to między innymi, że klasa może powtórnie wykorzystywać komentarze dokumentujące z implementowanych interfejsów, co eliminuje konieczność ich kopiowania. Mechanizm ten potencjalnie pozwala zredukować problemy z utrzymaniem wielu zbiorów niemal identycznych komentarzy dokumentujących, ale jest kłopotliwy w użyciu i ma pewne ograniczenia. Szczegóły wykraczają poza ramy tej książki.

Warto w tym miejscu podjąć pewien istotny temat związany z komentarzami dokumentującymi. Choć niezbędne jest dostarczenie odpowiednich komentarzy dla wszystkich eksportowanych elementów API, w wielu sytuacjach to nie wystarcza.

W przypadku złożonych API zawierających wiele zależnych od siebie klas warto wzbogacić komentarze dokumentujące o zewnętrzną dokumentację omawiającą ogólną architekturę API. Jeśli taka dokumentacja istnieje, odpowiednia klasa lub pakiet powinny zawierać łącze do niej.

Javadoc automatycznie sprawdza stosowanie się do wielu rekomendacji omawianych w tym temacie. W Javie 7 do włączenia odpowiednich testów konieczne było dodanie w wierszu poleceń przełącznika `-Xdoclint`. W Javie 8 i 9 jest on domyślnie włączony. Istnieją dodatki do IDE takie jak checkstyle, które przeprowadzają dodatkowe sprawdzenia zgodności komentarzy z rekomendacjami [Burn01]. Prostym sposobem na ograniczenie prawdopodobieństwa wystąpienia błędów w komentarzach dokumentujących jest skontrolowanie plików HTML wygenerowanych przez Javadoc za pomocą **walidatora HTML**. Pozwala on wykryć wiele niewłaściwych zastosowań znaczników HTML. Dostępnych jest kilka programów kontrolujących poprawność HTML (można je pobrać z internetu), jak również można użyć programu działającego w sieci [W3C-validator]. W trakcie sprawdzania poprawności warto pamiętać, że od Javy 9 Javadoc potrafi generować kod HTML5, ale nadal domyślnie generuje kod HTML 4.01. Aby włączyć generowanie kodu HTML5, użyj w wierszu poleceń przełącznika `-html5`.

Konwencje przedstawione w tym temacie obejmują podstawy. Choć minęło już piętnaście lat od momentu jego aktualizacji, nadal wyczerpującym podręcznikiem pisania komentarzy dokumentujących jest *How to Write Doc Comments* [Javadoc-guide].

Gdy zastosujemy się do zaleceń, wygenerowana dokumentacja powinna stanowić przyjazny i czytelny opis powstałego API. Jedynym sposobem, aby przekonać się, że rzeczywiście tak się stało, jest **przeczytanie stron WWW wygenerowanych przez narzędzie Javadoc**. Warto to zrobić dla każdego API, które ma być stosowane przez innych. Podobnie jak testowanie programu prawie zawsze prowadzi do pewnych zmian w kodzie, tak czytanie dokumentacji w większości przypadków doprowadza do zmian w komentarzach dokumentujących.

Podsumujmy. Komentarze dokumentujące są najlepszym i najefektywniejszym sposobem dokumentowania API. Ich zastosowanie powinno być uważane za obowiązkowe dla wszystkich eksportowanych elementów API. Należy przyjąć spójny styl pisania, zgodny ze standardowymi konwencjami. Trzeba pamiętać, że w komentarzach dokumentujących można używać dowolnego kodu HTML, a metaznaki HTML muszą być dodatkowo przetwarzane.

Programowanie

Rozdział ten jest w większości poświęcony podstawowym mechanizmom języka. Omawia stosowanie zmiennych lokalnych, wykorzystanie bibliotek, użycie różnych typów danych oraz dwóch mechanizmów pozajęzykowych: **refleksyjności** i **metod natywnych**. Na koniec omówimy optymalizację i konwencje nazewnictwa.

Temat 57. Ograniczanie zasięgu zmiennych lokalnych

Temat ten jest podobny do tematu 15., „Ograniczanie dostępności klas i ich składników”. Ograniczając zasięg zmiennych lokalnych, można zwiększyć czytelność kodu i zmniejszyć prawdopodobieństwo wystąpienia błędów.

W języku programowania C zmienne lokalne musiały być deklarowane na początku bloku i programiści przyzwyczaili się do takiego stylu tworzenia kodu. Jednak warto zarzucić to przyzwyczajenie. Przypomnijmy, że język Java pozwala na deklarowanie zmiennych wszędzie tam, gdzie można zastosować wyrażenia (język C od wersji C99 również to dopuszcza).

Najlepszą techniką ograniczania zasięgu zmiennych lokalnych jest ich deklarowanie w miejscu, gdzie są po raz pierwszy użyte. Jeżeli zmienna jest zadeklarowana przed jej użyciem, zwiększa tylko bałagan i jest dodatkowym elementem rozpraszaającym czytelnika, który próbuje zrozumieć, jak program działa. W miejscu, gdzie zmienna jest zastosowana, czytelnik może już zapomnieć, jaki był typ lub wartość początkowa zmiennej.

Zbyt wczesne deklarowanie zmiennych lokalnych powoduje niepotrzebne rozszerzanie zasięgu w obie strony. Zasięg zmiennej lokalnej rozciąga się od miejsca jej zadeklarowania do końca zawierającego ją bloku. Jeżeli zmienna jest zadeklarowana poza blokiem, w którym jest używana, pozostaje widoczna po zakończeniu wykonywania tego bloku. Jeżeli zmienna zostanie przypadkowo wykorzystana przed lub po bloku, w którym zamierzone było jej stosowanie, konsekwencje mogą być katastrofalne.

Niemal każda deklaracja zmiennej lokalnej powinna zawierać wyrażenie inicjujące. Jeżeli nie masz jeszcze wszystkich danych do właściwego zainicjowania zmiennej, powinieneś opóźnić deklarację do momentu, w którym będziesz mógł wykonać inicjalizację. Jedynym wyjątkiem od tej reguły jest wyrażenie try-catch. Jeżeli zmienna jest inicjowana przez metodę, która zgłasza obsługiwany wyjątek, zmienna ta musi być zainicjowana wewnątrz bloku try. Jeżeli wartość ta ma być wykorzystana poza blokiem try, musi być zadeklarowana przed jego rozpoczęciem, kiedy nie może być jeszcze odpowiednio zainicjowana. Przykład takiej inicjalizacji można znaleźć w przykładzie do tematu 65.

W przypadku pętli istnieje dodatkowa możliwość ograniczenia zasięgu zmiennych. Pętla for pozwala na deklarowanie **zmiennych pętli**, ograniczając jej zasięg jedynie do obszaru, w którym jest potrzebna (obszar ten składa się z ciała pętli, inicjalizacji, warunku zakończenia pętli i modyfikacji poprzedzających ciało pętli). Dlatego **należy stosować pętle for w miejsce pętli while**, pod warunkiem, że wartość zmiennej pętli nie będzie potrzebna po zakończeniu pętli.

Do przeglądania całej kolekcji zaleca się stosowanie następującego idiomu (temat 58.):

```
// Zalecany idiom do przeglądania całej kolekcji
for (Element e : c) {
    ... // wykonaj pewne operacje na e
}
```

Jeśli potrzebujesz dostępu do iteratora, bo na przykład chcesz wywołać na nim metodę remove, preferowanym rozwiązaniem okazuje się tradycyjna pętla for zamiast pętli for-each:

```
// Rozwiązanie do stosowania, gdy potrzebny jest dostęp do iteratora
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // wykonaj pewne operacje na e lub i
}
```

Aby przekonać się, dlaczego pętle for są zalecane w miejsce pętli while, przeanalizuj następujący fragment kodu, zawierającego dwie pętle while i jeden błąd:

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
```

```

    }

    ...

    Iterator<Element> i2 = c2.iterator();
    while (i.hasNext()) {    // BŁĄD
        doSomethingElse(i2.next());
    }

```

Druga pętla zawiera błąd, który może wkraść się podczas kopiowania i wklejania fragmentu kodu. Inicjowana jest nowa zmienna pętli `i2`, ale wykorzystana jest stara zmienna `i`, która niestety nadal jest w zasięgu. Kod taki jest bezbłędnie kompilowany i działa bez generowania żadnego wyjątku, ale daje nieprawidłowe wyniki. Zamiast przeglądania nowej tablicy `c2`, druga pętla natychmiast zostaje zakończona, dając fałszywe złudzenie, że tablica `c2` jest pusta. Ponieważ nie są zgłaszane żadne ostrzeżenia o błędzie, może on długo pozostać nie wykryty.

Jeżeli popełnimy analogiczny błąd podczas edycji kodu programu, korzystając z zalecanego idiomu pętli `for` (czy to w wersji tradycyjnej, czy `for-each`), to kod taki nie skompiluje się. Zmienna pętli z pierwszej pętli będzie poza zasięgiem w miejscu, gdzie znajduje się druga pętla. Oto jak wyglądałby przykład w przypadku użycia tradycyjnej pętli `for`:

```

    for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
        Element e = i.next();
        ... // wykonaj pewne operacje na e lub i
    }

    ...

    // Błąd kompilacji - nierozpoznany symbol i
    for (Iterator<Element> i2 = c2.iterator(); i2.hasNext(); ) {
        Element e2 = i2.next();
        ... // wykonaj pewne operacje na e2 lub i2
    }

```

Dodatkowo, jeżeli skorzystasz z tego idiomu, istnieje mniejsze prawdopodobieństwo, że popełnisz błąd przy kopiowaniu i wklejaniu tekstu, ponieważ nie ma konieczności użycia innej nazwy zmiennej w drugiej pętli. Pętle te są całkowicie niezależne, więc nie ma żadnego niebezpieczeństwa związanego z powtórным użyciem nazwy zmiennej (lub iteratora) pętli. Co więcej, jest to jak najbardziej zgodne z dobrym stylem programowania.

Przedstawiony idiom posiada jeszcze jedną, choć niewielką, przewagę nad pętlą `while`. Pętla `for` jest o jeden wiersz krótsza, zwiększając tym samym czytelność kodu.

Do minimalizowania zakresu zmiennych lokalnych można zastosować również następujący idiom:

```
for (int i = 0; n = expensiveComputation(); i < n ; i++) {  
    ... // wykonaj pewne operacje na i  
}
```

Trzeba tu zauważyć, że wykorzystywane są dwie zmienne pętli *i* oraz *n*, które mają dokładnie taki zasięg, jak potrzeba. Druga z nich, *n*, jest wykorzystywana w celu uniknięcia wykonywania nadmiarowych obliczeń w każdym przebiegu pętli. Powinieneś stosować ten idiom, jeżeli warunek pętli wymaga wywołania metody, która będzie zwracała tę samą wartość dla wszystkich przebiegów pętli.

Ostatnią techniką ograniczania zasięgu zmiennych lokalnych jest **pisanie małych, specjalizowanych metod**. Jeżeli połączysz w jednej metodzie dwa zadania, zmienne lokalne właściwe dla jednego zadania mogą znajdować się w zasięgu kodu wykonującego drugie zadanie. Aby temu zapobiec, wystarczy podzielić metodę na dwie, po jednej dla każdego zadania.

Temat 58. Stosowanie pętli for-each zamiast tradycyjnych pętli for

Jak wspomnieliśmy w temacie 45., są zadania, które najlepiej wykonywać za pomocą strumieni, ale są i takie, które lepiej sprawdzają się z iteracjami. Oto tradycyjna pętla for dotycząca przeglądania kolekcji:

```
// Już nie najlepszy idiom przeglądania kolekcji!  
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {  
    Element e = i.next();  
    ... // wykonaj pewne zadania na e  
}
```

Z kolei tradycyjnym sposobem przeglądania tablicy był idiom:

```
// Już niezalecany idiom przeglądania tablicy!  
for (int i = 0; i < a.length; i++) {  
    ... // wykonaj pewne zadania na a[i]  
}
```

Idiomy te są lepsze niż pętle *while* (temat 57.), ale nie są doskonałe. Iterator i zmienna indeksowa po prostu zaciemniają kod — potrzebne są nam tylko elementy. Dodatkowo stwarzają one możliwość popełnienia błędu. Iterator i zmienna indeksowa występują w każdej pętli trzykrotnie, co daje większą szansę pomyłki. Jeżeli się to zdarzy, nie ma gwarancji, że kompilator wychwyci błąd. Obie przedstawione pętle są dosyć różne, co niepotrzebnie skupia uwagę na rodzaju używanego kontenera, a także utrudnia (choć w niewielkim stopniu) jego zamianę.

Pętla for-each (oficjalnie znana „rozszerzona instrukcja for”) usuwa tę nadmiarowość i możliwość popełnienia błędu przez całkowite ukrycie iteratora lub zmiennej indeksowej. Wynikowy idiom można w identycznej formie stosować dla kolekcji i tablic, co ułatwia późniejszą zmianę kontenera z jednej odmiany na inną:

```
// Zalecany idiom do przeglądania tablicy oraz kolekcji
for (Element e : elements) {
    ... // wykonaj pewne zadania na e
}
```

Dwukropek (:) występujący w tym kodzie należy czytać jako „w”, czyli powyższą pętlę należy czytać jako „dla każdego elementu *e* w *elements*”. W przypadku korzystania z pętli for-each nie ma żadnego spadku wydajności, nawet w przypadku tablic. Generowany kod jest w praktyce taki sam jak ten tworzony samodzielnie przez programistę.

Przewaga zagnieżdżonej pętli for-each nad tradycyjną pętlą for jest nawet większa w przypadku zagnieżdżonego przeglądania wielu kolekcji. W poniższym kodzie znajduje się często popełniany błąd w przypadku zagnieżdżonego przeglądania dwóch kolekcji:

```
// Czy zauważasz błąd?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }

...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

Jeżeli nie zauważyłeś błędu, nie musisz czuć się źle. Wielu ekspertów co pewien czas popełnia ten błąd. Problem polega na zbyt dużej liczbie wywołań iteratora dla zewnętrznej kolekcji (suits). Powinien być on wywoływany z zewnętrznej pętli, więc jeden raz dla każdego koloru, ale zamiast tego jest wywoływany w pętli wewnętrznej, czyli dla każdej karty. Po wyczerpaniu kolorów, pętla zgłasza wyjątek NoSuchElementException.

Jeżeli nie będziemy mieli szczęścia i wielkość zewnętrznej kolekcji jest wielokrotnością wewnętrznej kolekcji — na przykład jest to ta sama kolekcja — pętla zakończy się normalnie, ale nie otrzymamy oczekiwanych wyników. Przeanalizujemy na przykład nieudaną próbę wydrukowania wszystkich możliwych wyników przy rzucie parą kostek:

```
// Ten sam błąd, inny objaw!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = Arrays.asList(Face.values());

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

Program nie zgłosi wyjątku, ale wydrukuje tylko sześć „par” (od ONE ONE do SIX SIX) zamiast spodziewanych trzydziestu sześciu kombinacji.

Aby poprawić błąd z tych przykładów, należy dodać zmienną do zakresu pętli zewnętrznej, która będzie przechowywała zewnętrzny element:

```
// Poprawione, ale nieładnie - można to zrobić lepiej!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

Jeżeli zamiast tego skorzystamy z zagnieżdżonych pętli for-each, problem po prostu zniknie. Wynikowy kod jest tak zwięzły, jak tego oczekiwaliśmy:

```
// Zalecany idiom dla zagnieżdżonego przeglądania kolekcji i tablic
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

Niestety, istnieją trzy często spotykane sytuacje, w których **nie można** użyć pętli for-each:

- 1. Destrukcyjne filtrowanie** — jeżeli potrzebujemy przeglądać kolekcję i usuwać wybrane elementy, to konieczne będzie jawne użycie iteratora i wywołanie jego metody remove. Można uniknąć jawnego przejścia przez kolekcję, używając metody `Collection.removeIf` dodanej w Javie 8.
- 2. Transformacja** — jeżeli potrzebne jest przeglądanie listy lub tablicy i zamiana niektórych lub wszystkich elementów, konieczne będzie użycie iteratora listy lub indeksu tablicy w celu ustawienia wartości elementu.
- 3. Równoległa iteracja** — jeżeli konieczne jest równoległe przeglądanie wielu kolekcji, konieczne będzie ręczne sterowanie iteratorem lub zmienną indeksową, dzięki czemu wszystkie iteratory lub zmienne indeksowe będą zwiększane jednocześnie (jak przypadkowo było pokazane w błędnych programach wyświetlających karty i kostki).

Jeżeli znajdziemy się w jednej z tych sytuacji, należy skorzystać ze zwykłej pętli for, ale trzeba pamiętać o wspomnianych w tym temacie pułapkach i wiedzieć, że robimy wszystko, co możemy.

Pętla `for-each` pozwala przeglądać nie tylko kolekcje i tablice, ale wszystkie obiekty implementujące interfejs `Iterable`. Ten prosty interfejs składa się z jednej metody. Wygląda ona następująco:

```
public interface Iterable<E> {  
    //zwraca iterator po elementach tego obiektu  
    Iterator<E> iterator();  
}
```

Implementacja interfejsu `Iterable` nie jest trudna. Jeżeli napiszemy typ reprezentujący grupę elementów, zaimplementujemy `Iterable`, nawet jeżeli nie zdecydujemy się na implementowanie `Collection`. Pozwoli to użytkownikom na przeglądanie obiektów naszego typu za pomocą pętli `for-each`, za co będą nam dożywotnie wdzięczni.

Podsumujmy. Pętla `for-each` daje znacznie większe możliwości w stosunku tradycyjnej pętli `for` w obszarze jasności i zapobiegania błędom — bez utraty wydajności. Powinna być używana wszędzie, gdzie jest to możliwe.

Temat 59. Poznanie i wykorzystywanie bibliotek

Założmy, że chcesz wygenerować losowe liczby całkowite z przedziału od 0 do wybranej górnej granicy. W przypadku tego często spotykanego zadania wielu programistów napisze krótką metodę, która wygląda podobnie do przedstawionej poniżej.

```
// Często spotykane, ale wadliwe rozwiązanie!  
static Random rnd = new Random();  
  
static int random(int n) {  
    return Math.abs(rnd.nextInt()) % n;  
}
```

Metoda ta nie może wyglądać dobrze — posiada trzy wady. Pierwsza z nich ujawnia się, gdy `n` jest małą potęgą dwójki. Generowana sekwencja liczb losowych powtarza się po dosyć krótkim czasie. Druga wada jest zauważalna, gdy `n` nie jest potęgą dwójki — niektóre liczby są generowane częściej niż inne. Jeżeli `n` jest duże, wada ta staje się łatwo zauważalna. Jest to demonstrowane przez program, który generuje milion liczb losowych w odpowiednio wybranym przedziale, a następnie wypisuje, ile z wylosowanych liczb trafiło do dolnej połowy przedziału:

```
public static void main(String[] args) {  
    int n = 2 * (Integer.MAX_VALUE / 3);  
    int low = 0;
```

```
for (int i = 0; i < 1000000; i++)
    if (random(n) < n/2)
        low++;

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

Jeżeli metoda `random` działa prawidłowo, program powinien wypisać liczbę bliską pół miliona, ale jeżeli go uruchomisz, przekonasz się, że liczba ta jest bliższa 666 666. Dwie trzecie z liczb wygenerowanych przez metodę `random` trafiło do dolnej połowy przedziału!

Trzecia wada jest związana z metodą `random`, która może w rzadkich przypadkach zupełnie zawieść, zwracając liczbę spoza określonego przedziału. Jeżeli wystąpi taka sytuacja, metoda próbuje odwzorować wartość `rnd.nextInt()` za pomocą metody `Math.abs()` na nieujemną wartość typu całkowitego. Jeżeli `nextInt()` zwróci `Integer.MIN_VALUE`, `Math.abs` również zwróci `Integer.MIN_VALUE`, a operator reszty z dzielenia (%) zwróci liczbę ujemną — o ile `n` nie jest potęgą dwójki. Niemal zawsze spowoduje to błąd w programie, który będzie bardzo trudny do powtórzenia.

Aby z metody `random` usunąć wymienione przez nas trzy błędy, musisz wiele wiedzieć na temat przystawiania liniowego generatorów liczb pseudolosowych, teorii liczb i rachunku uzupełnieniowym do dwójki. Na szczęście nie musisz zaczynać od podstaw — zadanie to zostało już wykonane. Istnieje metoda `Random.nextInt(int)`. Nie musisz szczegółowo wiedzieć, w jaki sposób działa metoda `nextInt(int)` (choć jeśli jesteś chorobliwie ciekawy, możesz przestudiować dokumentację i kod źródłowy). Nad powstaniem tej metody pracował doświadczony inżynier o dużej wiedzy na temat algorytmów, a następnie przekazał ją ekspertom w dziedzinie liczb losowych, aby wydali o niej swoją opinię. Następnie biblioteka przeszła przez fazę testów beta, była udostępniona i kilka lat intensywnie wykorzystywana przez tysiące programistów. W metodzie tej do tej pory nie znaleziono błędów, ale jeżeli odkryto by jakąś usterkę, zostałaby ona usunięta w następnej wersji. **Stosując biblioteki standardowe, korzystasz z wiedzy ekspertów, którzy ją stworzyli, i doświadczenia wszystkich, którzy ją przed Tobą wykorzystywali.**

Od Javy 7 nie powinieneś już stosować klasy `Random`. W większości sytuacji **polecanym obecnie generatorem liczb losowych jest `ThreadLocalRandom`**. Tworzy lepszej jakości liczby losowe, a sam algorytm jest bardzo szybki. Na moim komputerze jest 3,6 razy szybszy niż wersja z `Random`. W przypadku strumieni równoległych lub tworzenia puli rozłączanych i złączanych używaj `SplitTableRandom`.

Drugą zaletą stosowania bibliotek jest to, że nie musisz tracić czasu na rozwiązywanie problemów jedynie marginalnie związanych z Twoją pracą. Jeżeli jesteś podobny do większości programistów, to prawdopodobnie wolisz udoskonalać swoją aplikację, a nie mechanizmy, z których ona korzysta.

Trzecia zaleta bibliotek standardowych to stopniowy wzrost ich wydajności bez konieczności Twojej ingerencji w program. Ponieważ wielu ludzi korzysta z bibliotek i są one wykorzystywane w standardowych testach wydajności, organizacje dostarczające te biblioteki często poważnie je optymalizują. Wiele z bibliotek platformy Java zostało w czasie jej życia powtórnie napisanych, co poskutkowało drastycznym wzrostem ich wydajności.

Czwartą zaletą jest to, że biblioteki zwykle są z upływem czasu rozbudowywane. Jeżeli w klasie z biblioteki brakuje ważnej funkcji, środowisko programistów szybko zgłasza takie niedociągnięcie. Przy tworzeniu platformy języka Java zawsze korzystano ze wsparcia tego środowiska.

Ostatnią zaletą stosowania bibliotek standardowych jest tworzenie typowego kodu. Kod taki jest dla większości programistów łatwiejszy do zrozumienia, konserwacji i powtórnego użycia.

Jeśli znamy te wszystkie zalety, wydaje się logiczne stosowanie bibliotek standardowych zamiast implementacji tworzonych w razie potrzeby, jednak spora część programistów nie korzysta z nich w dostatecznym stopniu. Dlaczego? Być może nie znają wszystkich dostępnych możliwości oferowanych przez biblioteki. **W każdej głównej wersji do bibliotek dodawanych jest wiele funkcji i opłaca się być z nimi na bieżąco.** Wraz z udostępnieniem każdej głównej wersji platformy Java pojawia się strona opisująca wszystkie nowe funkcjonalności. Warto się zapoznać z tymi stronami [Java8-feat, Java9-feat]. Aby lepiej zobrazować sytuację, przypuśćmy, że chcemy napisać program wyświetlający zawartość adresu URL wpisanego w wierszu poleceń (odpowiada to mniej więcej funkcjonalności narzędzia `curl` znanego z systemu Linux). Przed Javą 9 kod był nieco skomplikowany, ale Java 9 wprowadziła metodę `transferTo` do `InputStream`. Oto kompletny program wykonujący opisane zadanie z wykorzystaniem wspomnianej metody:

```
// Wyświetlanie zawartości adresu URL dzięki metodzie transferTo dodanej w Javie 9
public static void main(String[] args) throws IOException {
    try (InputStream in = new URL(args[0]).openStream()) {
        in.transferTo(System.out);
    }
}
```

Biblioteki są zbyt obszerne, aby czytać całą ich dokumentację [Java9-api], ale **każdy programista powinien poznać zawartość bibliotek `java.lang`, `java.util` i `java.io`, a także ich podpakietów.** Wiedza na temat innych bibliotek może być uzupełniana w razie potrzeby. Podsumowanie wszystkich możliwości bibliotek wykracza poza zakres tego tematu, bo przez lata ich możliwości znacznie się zwiększyły.

Kilka bibliotek wymaga specjalnego zainteresowania. Biblioteka kolekcji oraz biblioteka strumieni (tematy od 45. do 48.) powinny być częścią podstawowego zestawu narzędzi każdego programisty, podobnie jak zbiór narzędzi do obsługi współbieżności dostępny w pakiecie `java.util.concurrent`. Pakiet ten zawiera

zarówno narzędzia wysokiego poziomu, pozwalające uprościć zadanie programowania wielowątkowego, jak i niskopoziomowe narzędzia, pozwalające ekspertom na pisanie własnych wysokopoziomowych współbieżnych abstrakcji. Wysokopoziomowa część pakietu `java.util.concurrent` została opisana w tematach 80. i 81.

Czasami możliwości biblioteki nie spełniają naszych potrzeb. Im bardziej specjalizowane są nasze potrzeby, tym większe prawdopodobieństwo, że się to zdarzy. Choć naszym pierwszym impulsem powinno być użycie bibliotek, to jeżeli po analizie możliwości w interesującym nas obszarze okaże się, że nie spełniają one naszych potrzeb, to należy wykorzystać alternatywne rozwiązanie. Zawsze istnieją luki w zestawie funkcji oferowanych przez skończony zbiór bibliotek. Jeśli to, czego potrzebujesz, nie jest dostępne w standardowych bibliotekach Javy, skorzystaj z innych bibliotek wysokiej jakości, na przykład doskonałej biblioteki Guava oferowanej przez firmę Google [Guava]. Jeżeli potrzebne funkcje nie są dostępne, nie będziemy mieli innego wyjścia jak ich samodzielne zaimplementowanie.

Podsumujmy. Nie ma potrzeby ponownego wymyślania koła. Jeżeli potrzebujesz mechanizmu, który wydaje się często stosowany, prawdopodobnie w bibliotekach istnieje klasa, która realizuje Twój problem. Jeżeli taka klasa istnieje, należy jej użyć, a jeżeli nie wiesz, czy istnieje, sprawdź to. Najczęściej kod z biblioteki jest lepszy niż kod, jaki napiszesz samodzielnie, a dodatkowo często jest usprawniany w kolejnych wersjach. Nie uważamy, że Twoje umiejętności są niewystarczające — ekonomia i skala sprawia, że bibliotekom poświęca się więcej uwagi, niż przeciętny programista może przeznaczyć dla konserwacji tego samego zestawu funkcji.

Temat 60. Unikanie typów `float` i `double`, gdy potrzebne są dokładne wyniki

Typy `float` i `double` zostały zaprojektowane przede wszystkim do obliczeń naukowych i technicznych. Korzystają one z binarnej arytmetyki zmiennoprzecinkowej, która została zaprojektowana do szybkiego dostarczania precyzyjnych przybliżeń dla szerokiego zakresu wartości. Nie pozwalają na wyliczanie dokładnych wyników i nie powinny być stosowane tam, gdzie potrzebne są dokładne wyniki. Typy `float` i `double` w szczególności nie nadają się do obliczeń finansowych, ponieważ niemożliwe jest dokładne reprezentowanie wartości 0.1 (lub dowolnej ujemnej potęgi dziesięciu).

Załóżmy, że mamy 1,03 zł w kieszeni i wydamy 42 gr. Ile pieniędzy nam pozostało? Poniżej przedstawiamy niewielki fragment programu, który odpowie nam na to proste pytanie.

```
System.out.println(1.03 - 0.42);
```

Niestety, otrzymamy wartość 0.6100000000000001. Nie jest to przypadek odośobniony. Załóżmy, że w kieszeni mamy złotówkę i kupujemy dziewięć uszczerek po 10 gr sztuka. Ile dostaniemy reszty?

```
Ssystem.out.println(1.00 - 9*.10);
```

Zgodnie z naszym obliczeniem powinniśmy otrzymać 0.09999999999995 zł.

Mógłbyś pomyśleć, że problem ten może być rozwiązany przez zaokrąglenie wyników w czasie drukowania, ale niestety nie zawsze to pomaga. Załóżmy, że mamy złotówkę i na półce widzimy cukierki o cenach 10 gr, 20 gr, 30 gr i tak aż do złotych. Kupujesz po jednym cukierku, rozpoczynając od najtańszego, aż na kolejny braknie Ci pieniędzy. Ile kupisz cukierków i ile zostanie Ci reszty? Prosty program może pomóc Ci obliczyć rozwiązanie tego problemu:

// Błąd - zastosowanie liczb zmiennoprzecinkowych do obliczeń finansowych!

```
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = 0.10; funds >= price; price += 0.10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " zakupionych towarów.");
    System.out.println("Reszta: " + funds);
}
```

Jeżeli uruchomisz ten program, okaże się, że będziesz mógł kupić trzy cukierki i dostaniesz 0,3999999999999999 zł reszty. Oczywiście jest to nieprawidłowa odpowiedź! Prawidłowym sposobem rozwiązania tego problemu jest **skorzystanie z typu `BigDecimal`, `int` lub `long` do obliczeń finansowych**.

Poniżej przedstawiamy prostą modyfikację poprzedniego programu, wykorzystującą w miejsce typu `double` typ `BigDecimal`. Zwróć uwagę, że korzystamy z konstruktora `BigDecimal` w wersji `String`, a nie w wersji `double`. W ten sposób unikamy wprowadzenia do systemu niedokładnej wartości [Bloch05, Puzzle 2].

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal( ".10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
        funds.compareTo(price) >= 0;
        price = price.add(TEN_CENTS)) {
        funds = funds.subtract(price);
        itemsBought++;
    }
    System.out.println(itemsBought + " zakupionych towarów.");
    System.out.println("Reszta: " + funds);
}
```

Po uruchomieniu poprawionego programu otrzymamy prawidłowy wynik czterech cukierków i 0.00 zł reszty.

Istnieją jednak dwie niedogodności przy stosowaniu typu `BigDecimal`: jest on mniej wygodny od typów prostych i jest powolniejszy. Druga wada nie jest dokuczliwa, jeżeli rozwiązujemy niewielkie problemy, ale pierwsza może Cię irytować.

Zamiast typu `BigDecimal` możemy skorzystać z typu `int` lub `long` (w zależności od wielkości obsługiwanych kwot) i samemu obsługiwać kropkę dziesiętną. W naszym przykładzie oczywiste wydaje się przeprowadzenie wszystkich obliczeń w groszach zamiast w złotówkach. Poniżej przedstawiamy kolejną wersję naszego programu, która operuje groszami.

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " zakupionych towarów.");
    System.out.println("Reszta: " + funds + " groszy");
}
```

Podsumujmy. Nie należy korzystać z typów `float` ani `double` do obliczeń, w których wymagany jest dokładny wynik. Jeżeli chcesz, aby system obsługiwał kropkę dziesiętną, skorzystaj z typu `BigDecimal`; niestety jest on mniej wygodny w porównaniu z typami prostymi. Przy korzystaniu z typu `BigDecimal` mamy dodatkowo możliwość pełnej kontroli zaokrąglania. Do wyboru mamy osiem trybów zaokrąglania. Jest to wygodne, jeżeli wykonujemy obliczenia finansowe, w których sposób zaokrąglania jest usankcjonowany prawnie. Jeżeli ważna jest szybkość działania, to jeżeli możesz samodzielnie obsługiwać kropkę dziesiętną, a wartości nie są zbyt duże, możesz skorzystać z typu `int` lub `long`. Jeżeli wartości nie przekraczają dziewięciu cyfr, można skorzystać z typu `int`, jeżeli nie przekraczają osiemnastu — `long`. Jeżeli wartości przekraczają osiemnaście cyfr, musisz skorzystać z typu `BigDecimal`.

Temat 61. Stosowanie typów prostych zamiast opakowanych typów prostych

Java posiada dwuczęściowy system typów, składający się z **typów prostych**, takich jak `int`, `double` i `boolean` oraz **typów referencyjnych**, takich jak `String` i `List`. Każdy typ prosty ma odpowiadający mu typ referencyjny, nazywany **opakowanym typem prostym**. Opakowanymi typami prostymi odpowiadającymi `int`, `double` oraz `boolean` są `Integer`, `Double` oraz `Boolean`.

Jak zostało wspomniane w temacie 6., mechanizmy automatycznego pakowania i odpakowania (ang. *autoboxing* i *auto-unboxing*) rozmywiają, ale nie likwidują różnic pomiędzy typami prostymi i ich opakowanymi odpowiednikami. Istnieją spore różnice pomiędzy tymi dwoma rodzajami typów i ważne jest, aby uważać, gdzie są one wykorzystywane, jak również uważnie je wybierać.

Istnieją trzy główne różnice pomiędzy typami prostymi a opakowanymi typami prostymi. Po pierwsze, typy proste mają wyłącznie swoją wartość, natomiast opakowane typy proste nie są identyfikowane przez ich wartości. Inaczej mówiąc, dwa obiekty opakowanych typów prostych mają te same wartości, ale są identyfikowane przez inne. Po drugie, typy proste mają wyłącznie wartości funkcjonalne, natomiast opakowane typy proste mają jedną wartość нефункциональную (jest nią `null`), jak również wszystkie wartości funkcjonalne odpowiedniego typu prostego. Po trzecie, typy proste są zwykle wydajniejsze czasowo i pamięciowo w porównaniu z opakowanymi typami prostymi. Wszystkie różnice mogą spowodować spore problemy, jeżeli nie zachowamy uwagi.

Weźmy pod uwagę poniższy komparator, który jest zaprojektowany do reprezentowania rosnącego porządku wartości `Integer`. (Dla przypomnienia — metoda `compare` z komparatora zwraca liczbę ujemną, zero lub dodatnią, w zależności od tego, czy pierwszy argument jest mniejszy niż drugi, równy mu lub większy niż on). W praktyce nie ma potrzeby pisania tego komparatora, ponieważ implementuje on naturalny porządek typu `Integer`, co można uzyskać bez komparatora, ale jest to dosyć interesujący przykład:

```
// Niedziałający komparator - czy zauważasz błąd?  
Comparator<Integer> naturalOrder =  
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

Ten komparator wygląda nieźle i przejdzie wiele testów. Na przykład może być użyty w `Collections.sort` do prawidłowego posortowania miliona elementów listy, niezależnie od tego, czy zawiera ona zduplikowane elementy, czy nie. Jednak komparator ten ma poważną wadę. Aby się o tym przekonać, wystarczy wyświetlić wartość `naturalOrder.compare(new Integer(42), new Integer(42))`. Oba obiekty `Integer` reprezentują tę samą wartość (42), więc wartością tego wyrażenia powinno być 0, ale faktycznie jest 1, co wskazuje, że pierwsza wartość `Integer` jest większa niż druga.

W czym więc tkwi problem? Pierwszy test `naturalOrder` działa świetnie. Wyliczenie wyrażenia `i < j` powoduje, że obiekty `Integer` odpowiadające zmiennym `i` oraz `j` są poddawane procesowi **autounboxingu**, czyli pobierane są ich wartości proste. Następnie jest wykonywane sprawdzenie, czy pierwsza z wyników wartości `int` jest mniejsza niż druga. Załóżmy, że nie jest. Następny test sprawdza wyrażenie `i == j`, które wykonuje **porównanie identyfikatorów** dwóch referencji obiektów. Jeżeli `i` oraz `j` reprezentują różne obiekty `Integer`, który reprezentuje

te same wartości `int`, to porównanie zwróci `false`, a komparator niewłaściwie zwróci `1`, wskazując, że pierwsza wartość `Integer` jest większa niż druga. **Zastosowanie operatora `==` do opakowanych typów prostych jest niemal zawsze niewłaściwe.**

Jeśli w praktyce potrzebujemy komparatora stosującego naturalny porządek, najprostszym sposobem rozwiązania problemu będzie wywołanie `Comparator.naturalOrder()`. Jeśli komparator piszesz samodzielnie, użyj metod konstruujących komparator lub statycznych metod `compare` dostępnych dla typów prostych (temat 14.). W przedstawionym przykładzie prostym sposobem rozwiązania problemu jest dodanie dwóch zmiennych lokalnych do zapamiętania wartości prostych `int` odpowiadających zmiennym `i` oraz `j` i wykonywanie wszystkich porównań na tych zmiennych. Aby uniknąć błędnego porównywania identyfikatorów, można zastosować następującą wersję:

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

Następnie weźmy pod uwagę następujący mały program:

```
public class Unbelievable {
    static Integer i;

    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Niemożliwe!");
    }
}
```

Nie, nie powoduje on wyświetlenia `Niemożliwe!`, ale jego działanie jest równie dziwne. Podczas wartościowania wyrażenia `(i==42)` zgłasza wyjątek `NullPointerException` ➤ `Exception`. Problem polega na tym, że `i` jest typu `Integer`, a nie `int`, więc podobnie jak wszystkie pola referencyjne zmienna ta ma wartość początkową `null`. Gdy program wartościuje wyrażenie `(i==42)`, porównuje `Integer` z `int`. W niemal zawsze, gdy **mieszmamy typy proste i opakowane typy proste w jednej operacji, opakowane typy proste są automatycznie odpakowywane i taki przypadek nie jest wyjątkiem**. Jeżeli referencja do obiektu `null` jest odpakowywana, otrzymujemy `NullPointerException`. Jak pokazuje ten program, może się to zdarzyć niemal wszędzie. Naprawienie tego programu wymaga tylko zadeklarowania `i` jako `int`, a nie `Integer`.

Na koniec przeanalizujmy program z tematu 6.:

```
//Nieoczekiwane powolny program! Czy zauważasz tworzenie obiektów?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
```

```
        sum += i;
    }
    System.out.println(sum);
}
```

Program ten jest znacznie wolniejszy, niż mógłby być, ponieważ przypadkowo zmienna lokalna (`sum`) została zadeklarowana jako opakowany typ prosty `Long` zamiast `long`. Program kompiluje się bez błędów i ostrzeżeń, a zmienna jest w pętli pakowana i rozpakowywana, co powoduje zauważalną utratę wydajności.

We wszystkich trzech programach przedstawianych w tym temacie występuje ten sam problem — programista zignorował różnice pomiędzy typami prostymi a opakowanymi typami prostymi i poniósł tego konsekwencje. W pierwszych dwóch programach konsekwencją tą była awaria, a w trzecim znaczne problemy wydajnościowe.

Kiedy należy więc korzystać z opakowanych typów prostych? Mają one kilka właściwych zastosowań. Pierwszym jest ich użycie jako kluczy, elementów i wartości w kolekcjach. Nie można umieścić typów prostych w kolekcji, więc jesteśmy zmuszeni do użycia opakowanych typów prostych. Jest to specjalny przypadek bardziej ogólnej zasady. Konieczne jest stosowanie opakowanych typów prostych jako parametrów typu w typach parametryzowanych (rozdział 5.), ponieważ język nie pozwala na użycie w tych miejscach typów prostych. Na przykład nie można zadeklarować zmiennej typu `ThreadLocal<int>`, więc trzeba użyć `ThreadLocal` ↪ `<Integer>`. Trzeba również używać opakowanych typów prostych przy refleksyjnym wywoływaniu metod (temat 65.).

Podsumujmy. Należy używać typów prostych zamiast typów opakowanych wszędzie tam gdzie jest to możliwe. Typy proste są mniej skomplikowane i szybsze. Jeżeli zachodzi konieczność użycia opakowanych typów prostych, należy zachować ostrożność. **Autoboxing redukuje objętość kodu, ale nie zagrożenia związane z użyciem opakowanych typów prostych.** Gdy program porównuje dwa opakowane typy proste za pomocą operatora `==`, dokonuje porównywania identyfikatorów, co niemal na pewno **nie** jest przez nas oczekiwane. Gdy nasz program wykonuje obliczenia na mieszanych typach składających się z opakowanych i nieopakowanych typów prostych oraz **gdy nasz program wykonuje operację unboxingu, może zgłosić wyjątek `NullPointerException`.** Na koniec, gdy nasz program opakowuje wartości proste, może to powodować kosztowne i niepotrzebne tworzenie obiektów.

Temat 62. Unikanie typu String, gdy istnieją bardziej odpowiednie typy

Typ String został zaprojektowany do reprezentowania ciągów znaków i świetnie spełnia swoje zadanie. Ponieważ ciągi znaków są tak powszechne i bardzo dobrze obsługiwane przez język programowania, istnieje naturalna tendencja do stosowania ciągów do zadań, dla których nie były zaprojektowane. Omówimy teraz kilka przypadków, w których nie powinieneś stosować ciągów znaków.

Ciąg znaków jest słabym zamiennikiem dla innych typów posiadających wartość. Gdy pewna dana trafia do programu z pliku, sieci lub klawiatury, często jest zapisana w postaci ciągu znaków. Naturalne wydaje się pozostawienie jej w tej postaci, ale jest to prawidłowe jedynie wtedy, gdy dana ma naturę tekstową. Jeżeli jest to liczba, powinna być zmieniona na właściwy typ numeryczny, taki jak `int`, `float` lub `BigInteger`. Jeżeli jest to odpowiedź na pytanie o możliwych odpowiedziach typu tak lub nie, powinna być zmieniona na wartość `boolean`. Ogólnie mówiąc, jeżeli istnieje właściwy typ, niezależnie od tego, czy jest to typ prosty czy obiektowy, powinieneś z niego skorzystać. Choć wskazówka ta wydaje się oczywista, często jest ignorowana.

Ciąg znaków jest słabym zamiennikiem dla typów wyliczeniowych. W temacie 34. wspominaliśmy, że typ wyliczeniowy jest o wiele lepszym rozwiązaniem niż zastosowanie ciągów znaków.

Ciąg znaków jest słabym zamiennikiem dla typu złożonego. Jeżeli element składa się z wielu składników, zwykle reprezentowanie go w postaci ciągu znaków jest bardzo złym pomysłem. Na przykład poniższy wiersz kodu pochodzi z istniejącego systemu — nazwa identyfikatora została zmieniona w celu ochrony winnych:

```
// Niewłaściwe wykorzystanie ciągu jako typu złożonego
String compoudKey = className + "#" + i.next();
```

Podejście takie posiada wiele wad. Jeżeli znak wykorzystywany do rozdzielania pól wystąpi w jednym z tych pól, spowoduje to chaos. Aby dostać się do poszczególnych pól, musisz przeanalizować ciąg, co jest powolne, nużące i łatwo przy tym popełnić błąd. Nie możesz utworzyć metod `equals`, `toString` lub `compareTo` i musisz zaakceptować działanie charakterystyczne dla typu String. Lepszym rozwiązaniem jest utworzenie klasy reprezentującej to złożenie, często w postaci prywatnej statycznej klasy zagnieżdżonej (temat 24.).

Ciąg znaków jest słabym zamiennikiem dla uprawnień. Czasami ciągi znaków są wykorzystywane do umożliwiania dostępu do niektórych funkcji. Rozważmy na przykład projekt obsługi zmiennej lokalnej dla wątku. Mechanizm taki udostępnia zmienną, która ma dla indywidualne wartości dla różnych wątków. Biblioteki Java posiadają od wersji 1.2 możliwości zmiennych lokalnych dla wątków, ale

wcześniej kilku programistów niezależnie utworzyło ten sam projekt, gdzie ciąg znaków przekazywany przez klienta otwierał dostęp do zawartości zmiennej lokalnej dla wątku:

```
// Błąd - niewłaściwe wykorzystanie typu String jako uprawnień!
public class ThreadLocal {
    private ThreadLocal() { } // brak możliwości utworzenia obiektu

    // ustawia wartość zmiennej dla bieżącego wątku
    public static void set(String key, Object value);

    // zwraca wartość zmiennej dla bieżącego wątku
    public static Object get(String get);
}
```

W przypadku takiego podejścia klucz jest reprezentowany we współdzielonej, globalnej przestrzeni nazw. Jeżeli dwa niezależne klienty tego samego pakietu zdecydują się na korzystanie z tej samej nazwy dla swojej zmiennej lokalnej dla wątku, nieumyślnie będą współdzielić tę zmienną, co może spowodować nieprawidłowe działanie obu klientów. Bezpieczeństwo takiego rozwiązania również jest niewielkie, złośliwy klient może skorzystać z tego samego klucza w celu uzyskania nielegalnego dostępu do danych innego klienta.

Przedstawione powyżej API może być poprawione przez wymianę ciągu znaków na niemożliwy do sfalszowania klucz (zwany czasami **uprawnieniem**):

```
public class ThreadLocal {
    private ThreadLocal() { } // brak możliwości utworzenia obiektu

    public static class Key {
        Key() { }
    }

    // generowanie unikalnego, niemożliwego do sfalszowania klucza
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

Choć rozwiązuje to problem z API korzystającym z ciągów znaków, problem można rozwiązać lepiej. Nie potrzebujesz w rzeczywistości żadnych metod statycznych. Mogą one stać się metodami instancyjnymi klucza; w ten sposób klucz przestaje być kluczem — **staje się** zmienną lokalną dla wątku. Nie jest potrzebna klasa wysokiego poziomu, ponieważ nie posiada ona żadnej nowej funkcji, można ją więc usunąć i zmienić nazwę klasy zagnieżdżonej na `ThreadLocal`:

```
public class ThredLocal {
    public ThreadLocal() {}
    public void set(Object value);
    public Object get();
}
```

API to nie jest bezpieczne dla typów, ponieważ przy pobieraniu zmiennej lokalnej dla wątków konieczne jest rzutowanie wartości `Object` na faktyczny typ. Niemożliwe jest doprowadzenie oryginalnego API bazującego na typie `String` do postaci bezpiecznej dla typów, a API bazującego na `Key` bardzo trudne, ale można uzyskać API bezpieczne dla typów przez uogólnienie klasy `ThreadLocal` (temat 29.):

```
public final class ThreadLocal<T> {
    public ThreadLocal() {}
    public void set(T value);
    public T get();
}
```

Jest to w przybliżeniu API klasy `java.util.ThreadLocal`. Oprócz rozwiązywania problemów API korzystającego z ciągów znaków jest ono szybsze i bardziej eleganckie od wszystkich API korzystających z kluczy.

Podsumujmy. Należy unikać naturalnej tendencji do reprezentowania obiektów jako ciągów znaków, jeżeli istnieją lub można utworzyć lepsze typy danych. Jeżeli ciągi znaków zostaną nieprawidłowo wykorzystane, są niewygodne, mniej elastyczne, powolniejsze i bardziej narażone na błędy niż inne typy danych. Najczęstszymi typami, jakie nieprawidłowo są zamieniane na ciągi, są typy proste, wyliczeniowe i złożone.

Temat 63. Problemy z wydajnością przy łączeniu ciągów znaków

Operator złączenia ciągów (+) jest wygodnym narzędziem do łączenia kilku ciągów w jeden. Doskonale nadaje się do generowania pojedynczych wierszy wyniku lub do tworzenia reprezentacji znakowej małych obiektów o stałej wielkości. **Wielokrotne wykorzystanie operatora złączenia ciągów do złączenia n ciągów jest operacją o koszcie n^2 .** Jest to konsekwencja założenia, że ciągi znaków są **niezmienne** (temat 17.). Gdy łączone są dwa ciągi, zawartość ich jest kopiowana.

Dla przykładu, poniższa metoda tworzy reprezentację znakową rachunku przez dołączanie wierszy dla kolejnych pozycji.

```
// Niewłaściwe zastosowanie złączenia - bardzo niska wydajność!
public String statement() {
```

```
String result = "";
for (int i = 0; i < numItems(); i++)
    result += lineForItem(i); //złączenie ciągów
return result;
}
```

Metoda ta działa ta niezmiernie powoli, gdy liczba pozycji jest duża. **Aby osiągnąć akceptowalną wydajność, do przechowywania tymczasowego ciągu należy skorzystać ze `StringBuilder` zamiast ze `String`, co przedstawia poniższy kod:**

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}
```

Choć od Javy 6 dużo pracy włożono w przyspieszenie łączenia tekstów, różnica wydajności między oboma rozwiązaniami jest ogromna. Jeżeli metoda `numItems` zwraca 100-, a `lineForItem` 80-znakowy ciąg, na moim komputerze druga metoda jest 6,5 razy szybsza niż pierwsza. Ponieważ pierwsza metoda ma wydajność kwadratową w zależności od liczby elementów, natomiast druga — wydajność liniową, to ze wzrostem liczby elementów różnica wydajności będzie jeszcze bardziej widoczna. Zwróć uwagę, że druga metoda wstępnie alokuje `StringBuilder` o takiej wielkości, aby zmieścić się wynikowy ciąg. Nawet jeżeli zmniejszymy wydajność, korzystając z bufora o standardowej wielkości, metoda ta jest nadal 5,5 razy szybsza niż pierwsza.

Konkluzja jest prosta: **Nie należy korzystać z operatora złączenia ciągów dla więcej niż kilku ciągów**, chyba że wydajność nie jest istotna. Zamiast tego należy korzystać z metody `append` klasy `StringBuilder`. Można również skorzystać z tablicy znaków lub przetwarzać ciągi po kolei, zamiast łączyć je w jeden.

Temat 64. Odwoływanie się do obiektów poprzez interfejsy

W temacie 51. zawarta jest porada mówiąca, że jako typy parametrów powinny być zamiast klas wykorzystywać interfejsy. Ogólniej mówiąc, odwołując się do obiektów powinny być, o ile jest to możliwe, korzystać z interfejsów zamiast klas. **Jeżeli istnieje odpowiedni interfejs dla parametru, zwracanej wartości, zmiennej i pola, powinien być on używany podczas deklaracji tych elementów.** Jedynym miejscem, w którym naprawdę musisz odwołać się do klasy obiektu, jest moment jego tworzenia. Rozważmy przykład klasy `LinkedHashSet`, która jest implementacją interfejsu `Set`. Powinno się przyzwyczaić się do takiego deklarowania zmiennych:

```
// Dobrze - wykorzystanie interfejsu jako typu  
Set<Son> sonSet = new LinkedHashSet<>();
```

Nie powinno się stosować następującego zapisu:

```
// Źle - wykorzystanie klasy jako typu!  
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

Jeżeli przyzwyczaisz się do wykorzystywania interfejsów jako typów, Twoje programy będą bardziej elastyczne. Jeżeli zdecydujesz się na zmianę implementacji, wystarczy zmienić nazwę klasy w konstruktorze (lub wykorzystać inną, statyczną metodę fabryczną). Dla przykładu, pierwsza deklaracja może być zmieniona na:

```
Set<Son> sonSet = new HashSet<>();
```

Cały kod korzystający z tej deklaracji będzie działał prawidłowo, ponieważ nie zależy od właściwego typu implementacji, więc można ją bez problemu zmienić.

Istnieje tylko jeden wyjątek — jeżeli oryginalna implementacja zawiera dodatkowe funkcje niewymagane przez założenia interfejsu, a kod korzysta z tych funkcji, niezbędne jest udostępnienie tej funkcji w nowej implementacji. Na przykład jeżeli kod używający pierwszej implementacji korzysta z tego, że `LinkedHashSet` posiada ustaloną kolejność iteracji elementów, zastąpienie jej wersją `HashSet` nie byłoby właściwe, bo druga z klas nie daje gwarancji odpowiedniej kolejności elementów.

Z jakiego powodu możemy chcieć zmienić implementację? Być może dlatego, że nowe implementacje będą miały lepszą wydajność lub będą zawierać nowe funkcje. Rzeczywistym przykładem może być pole zawierające instancję `HashMap`. Zamiana na `EnumMap` poprawi wydajność i zapewni iterację przez treść obiektu w naturalnym porządku, ale może być zastosowana tylko w sytuacji, gdy kluczem jest typ wyliczeniowy. Zmiana z `HashMap` na `LinkedHashMap` zapewniłaby przewidywalną kolejność w trakcie iteracji przy wydajności podobnej do `HashMap` i to bez wymagań co do rodzaju klucza.

Mógłbyś uważać, że poprawne jest zadeklarowanie zmiennej z wykorzystaniem typu implementacyjnego, bo przecież będziesz dokonywał jednoczesnej zmiany typu deklaracyjnego i typu implementacyjnego, ale nie ma gwarancji, że po takiej zmianie kod uda się skompilować. Jeśli kod kliencki wykorzystywał metody z oryginalnego typu implementacyjnego, które nie istnieją w nowej klasie, lub też kod kliencki przekazywał instancję w miejsce wymagające oryginalnego typu, to po zmianie kod przestałby się kompilować. Dzięki zadeklarowaniu pola o typie definiowanym przez interfejs można uniknąć tych kłopotów.

Całkowicie dopuszczalne jest odwoływanie się do obiektów poprzez klasy zamiast poprzez interfejsy, jeżeli nie istnieje odpowiedni interfejs. Przykładem takim mogą być klasy posiadające wartość, takie jak `String` lub `BigInteger`.

Klasy posiadające wartość rzadko posiadają wiele implementacji. Najczęściej są one zadeklarowane jako `final` i rzadko posiadają skojarzone z nimi interfejsy. Całkowicie poprawne jest wykorzystywanie klas posiadających wartość jako parametrów, zmiennych, pól lub zwracanych typów.

Drugim przypadkiem, w którym nie istnieje odpowiedni interfejs, jest przynależność obiektu do modelu, którego podstawowe typy są klasami, a nie interfejsami. Jeżeli obiekt należy do takiego **modelu bazującego na klasach**, zalecane jest odwoływanie się do obiektów poprzez odpowiednią klasę podstawową, która zwykle jest abstrakcyjna, zamiast przez jej klasę implementacji. Do tej kategorii należy wiele klas pakietu `java.io`, między innymi `OutputStream`.

Ostatnim przykładem, gdy nie istnieje odpowiedni interfejs, jest klasa implementująca interfejs, ale zawierająca dodatkowe metody, nieistniejące w interfejsie — na przykład klasa `PriorityQueue` posiada metodę `comparator`, której nie posiada interfejs `Queue`. Do takiej klasy należy się odwoływać **tylko** w tych przypadkach, gdy program korzysta z dodatkowych metod. To powinna być rzadko występująca sytuacja.

Przypadki te nie wyczerpują zagadnienia, ale przybliżają sytuacje, w których właściwe jest odwoływanie do klasy obiektu. W praktyce powinno być oczywiste, czy interesujący nas obiekt posiada odpowiedni interfejs. Jeżeli go posiada, program będzie bardziej elastyczny, jeżeli wykorzystasz ten typ w celu odwoływania się do obiektów. **Jeżeli nie, skorzystaj po prostu z klasy najwyższej w hierarchii, która posiada już wszystkie potrzebne funkcje.**

Temat 65. Stosowanie interfejsów zamiast refleksyjności

Refleksyjność, `java.lang.reflect`, pozwala na programowy dostęp do informacji na temat załadowanych klas. Na podstawie obiektu `Class` można uzyskać obiekty `Constructor`, `Method` i `Field`, reprezentujące konstruktory, metody oraz pola klasy reprezentowanej przez obiekt `Class`. Obiekty te pozwalają na programowy dostęp do nazw składników klasy, typów pól, sygnatur metod itd.

Dodatkowo obiekty `Constructor`, `Method` i `Field` pozwalają na **refleksyjne** manipulowanie ich odpowiednikami: można tworzyć obiekty, wywoływać metody i odwoływać się do pól podstawowej klasy, korzystając z metod obiektów `Constructor`, `Method` i `Field`. Na przykład `Method.invoke` pozwala na wywołanie dowolnej metody obiektu dowolnej klasy (zgodnie z ograniczeniami dostępu). Refleksyjność pozwala jednej klasie wykorzystać inną nawet wtedy, gdy wywoływana klasa jeszcze nie istnieje w czasie kompilacji klasy do niej się odwołującej. Niestety nie da się tego zrobić bez ponoszenia dodatkowych kosztów.

- **Tracisz wszystkie zalety kontroli typów w czasie kompilacji**, w tym sprawdzanie wyjątków. Jeżeli program próbuje refleksyjnie wywołać nieistniejącą lub niedostępną metodę, błąd zostanie zgłoszony w czasie wykonywania programu, chyba że zostaną zastosowane specjalne zabezpieczenia.
- **Kod wymagany do refleksyjnego dostępu jest niezgrabny i rozwlekły**. Jest nudny w pisaniu i trudny w czytaniu.
- **Tracisz na wydajności**. Refleksyjne wywołanie metod jest znacznie powolniejsze niż ich normalne wywoływanie. Na ile jest wolniejsze, trudno powiedzieć, ponieważ wpływa na to wiele czynników. Na moim komputerze różnica wydajności w przypadku wywołania metody bez parametrów zwracającej typ `int` wyniosła jedenaście razy.

Istnieje kilka zaawansowanych zastosowań wymagających mechanizmu refleksyjności. Przykładami mogą tu być narzędzia do analizy kodu i frameworki wstrzykiwania zależności. Nawet jednak te narzędzia odchodzą powoli od refleksyjności, bo coraz wyraźnie ujawniają się jej wady. Jeżeli nie jesteś pewien, czy Twoja aplikacja pasuje do wymienionych tu kategorii, to prawdopodobnie nie powinieneś korzystać z refleksyjności.

Jeżeli refleksyjność zostanie wykorzystana tylko w ograniczonym zakresie, można skorzystać z wielu jej zalet bez ponoszenia dużych kosztów. W przypadku wielu programów trzeba skorzystać z klasy, która jest niedostępna w czasie kompilacji. Dostępny jest jednak interfejs lub klasa bazowa, dzięki czemu można odwoływać się do niedostępnej klasy (temat 64.). Jeżeli występuje taki przypadek, można **utworzyć obiekt w sposób refleksyjny i odwoływać się do niego w standardowy sposób poprzez interfejs lub klasę bazową**.

Przedstawimy teraz krótki program tworzący obiekt na podstawie interfejsu `Set<String>`; klasa implementująca go jest podawana jako pierwszy argument w wierszu poleceń. Program umieszcza pozostałe argumenty wiersza poleceń w tym kontenerze i wypisuje je na ekran. Niezależnie od wartości pierwszego argumentu program eliminuje powtórzenia wartości pozostałych argumentów. Kolejność wypisywania argumentów zależy od klasy podanej jako pierwszy argument. Jeżeli podamy `java.util.HashSet`, będą one wypisane w pozornie losowej kolejności. Jeżeli skorzystamy z klasy `java.util.TreeSet`, argumenty będą wypisane w kolejności alfabetycznej, ponieważ elementy `TreeSet` są sortowane.

```
// Konkretyzacja refleksyjna z dostępem przez interfejs
public static void main(String[] args) {
    // zmiana nazwy klasy na obiekt Class
    Class<? extends Set<String>> cl = null;
    try {
        cl = (Class<? extends Set<String>>) // Niesprawdzone rzutowanie!
            Class.forName(args[0]);
```

```

    } catch(ClassNotFoundException e) {
        fatalError("Nie znaleziono klasy.");
    }

    //pobranie konstruktora
    Constructor<? extends Set<String>> cons = null;
    try {
        cons = cl.getDeclaredConstructor();
    } catch (NoSuchMethodException e) {
        fatalError("Brak konstruktora bezparametrowego.");
    }

    //utworzenie obiektu
    Set<String> s = null;
    try {
        s = cons.newInstance();
    } catch (IllegalAccessException e) {
        fatalError("Klasa niedostępna.");
    } catch (InstantiationException e) {
        fatalError("Błąd tworzenia obiektu klasy.");
    } catch (InvocationTargetException e) {
        fatalError("Konstruktor zgłosił " + e.getCause());
    } catch (ClassCastException e) {
        fatalError("Klasa nie implementuje Set.");
    }

    //wypisanie zawartości zbioru
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}

private static void fatalError(String msg) {
    System.err.println(msg);
    System.exit(1);
}

```

Choć program ten jest jedynie zabawką, to demonstrowana technika ta daje ogromne możliwości. Nasz program można łatwo przebudować, aby stał się testem zbiorów, który kontroluje implementacje interfejsu `Set`, sprawdzając, czy spełnia on założenia interfejsu. Można również na jego podstawie zbudować program analizujący wydajność poszczególnych implementacji. Technika ta jest wystarczająca do realizacji zaawansowanej **biblioteki dostawcy usług** (temat 1.). W większości przypadków nie będziesz musiał sięgać do innych możliwości refleksyjności.

Analizując nasz przykład, można zauważyć dwie niedogodności przy stosowaniu refleksyjności. Po pierwsze, w czasie wykonania programu możliwe jest generowanie błędów, które w przypadku zwykłego wywołania zostałyby wykryte na etapie kompilacji. (W ramach ćwiczenia spróbuj przekazywać do programu takie parametry wiersza poleceń, aby spowodować wystąpienia każdego z sześciu wyjątków).

Po drugie, utworzenie obiektu na podstawie nazwy klasy wymaga napisania ponad dwudziestu wierszy kodu, podczas gdy wywołanie konstruktora mieści się zwykle w jednym wierszu. Program można by skrócić, wyłapując wyjątek `ReflectiveOperationException` stanowiący klasę nadrzędną dla różnych wyjątków związanych z mechanizmem refleksji (wprowadzono ją w Javie 7). Niedogodności te są jednak ograniczone jedynie do fragmentu, w którym następuje utworzenie obiektu. Po utworzeniu obiektu nie można go odróżnić od innych obiektów `Set`. Przy korzystaniu z ograniczonej refleksyjności ogromna większość kodu normalnych programów będzie identyczna, jak w przypadku normalnego tworzenia obiektów.

Jeżeli spróbujemy skompilować program, otrzymamy ostrzeżenie o rzutowaniu bez możliwości weryfikacji. Ostrzeżenie to jest właściwe w tym sensie, że rzutowanie na `Class<? extends Set<String>>` powiedzie się nawet wtedy, gdy nazwana klasa nie będzie implementacją `Set`, ale wyjątek `ClassCastException` zostanie zgłoszony w momencie tworzenia instancji klasy. Opis najlepszych sposobów uniknięcia tych ostrzeżeń znajduje się w temacie 27.

Dopuszczalnym, rzadkim przypadkiem skorzystania z refleksyjności jest usunięcie zależności pomiędzy klasami, metodami lub polami, jeżeli elementy te mogą nie być dostępne w czasie wykonania programu. Może być to użyteczne, gdy pakiet musi współdziałać z różnymi wersjami innego pakietu. W takim przypadku zwykle pakiet jest kompilowany z minimalną wersją środowiska, najczęściej najstarszą obsługiwaną wersją, a do nowszych klas i metod odwołuje się refleksyjnie. Aby konstrukcja taka mogła działać prawidłowo, należy odpowiednio obsługiwać sytuacje, w których nastąpi odwołanie do nowszej metody lub klasy, która nie jest dostępna w czasie działania programu. Obsługa taka może polegać na zastosowaniu innych metod do zrealizowania tego samego zadania lub na ograniczeniu liczby wykonywanych czynności.

Podsumujmy. Refleksyjność ma ogromne możliwości, które są wymagane przy niektórych skomplikowanych zadaniach programistycznych, ale ma również wiele wad. Jeżeli piszesz program, który musi działać z klasami, które są niedostępne w czasie kompilacji, powinieneś, o ile jest to możliwe, korzystać z refleksyjności jedynie do tworzenia obiektów, a następnie odwoływać się do nich, korzystając z interfejsu lub klasy bazowej dostępnej w czasie kompilacji.

Temat 66. Rozważne wykorzystywanie metod natywnych

Java posiada interfejs *Java Native Interface* (JNI), który pozwala aplikacjom wywoływać **metody natywne**, będące specjalnymi metodami napisanymi za pomocą **innych języków programowania**, takich jak C lub C++. Historycznie, metody

natywne posiadały trzy główne zastosowania. Pozwalały skorzystać z mechanizmów specyficznych dla platformy, na przykład z rejestru. Zapewniały dostęp do bibliotek istniejącego kodu, który z kolei odwoływał się do istniejących danych. Na koniec, metody natywne były używane do tworzenia krytycznych pod względem wydajności fragmentów aplikacji.

Dopuszczalne jest skorzystanie z metod natywnych do korzystania z udogodnień charakterystycznych dla platformy systemowej, ale wraz z rozwojem środowiska Java dostępne jest coraz więcej funkcji, poprzednio występujących jedynie w środowisku określonego systemu. Przykładem może być dodane w Javie 9 API pozwalające na dostęp do procesów systemowych. Dopuszczalne jest również zastosowanie metod natywnych w celu uruchamiania istniejącego kodu, gdy nie istnieje żadna równoważna biblioteka o podobnej funkcjonalności dostępna w Javie.

Rzadko konieczne jest stosowanie metod natywnych w celu poprawienia wydajności. We wczesnych wersjach (wcześniejszych niż Java 3) było to niezbędne, ale nowoczesne implementacje JVM są już o **wiele** szybsze. W większości zadań możliwe jest osiągnięcie wydajności porównywalnej ze zwykłymi językami. Dla przykładu, po dodaniu pakietu `java.math` w wersji 1.1 platformy klasa `BigInteger` była zbudowana przy wykorzystaniu szybkiej biblioteki arytmetyki o dowolnej dokładności, napisanej w języku C. Było to wtedy niezbędne do uzyskania odpowiedniej wydajności. W wersji 1.3 klasa `BigInteger` została napisana od początku w języku Java i dokładnie zoptymalizowana. Nowa wersja okazywała się szybsza od oryginału dla większości operacji i rozmiarów operandów.

Dalsza część tej historii nie jest już tak przyjemna, bo od tamtego czasu klasa `BigInteger` niewiele się zmieniała, z wyjątkiem szybszego mnożenia wprowadzonego w Javie 8. W tym samym czasie prace nad bibliotekami natywnymi nadal trwały, w szczególności nad biblioteką GMP (*GNU Multiple Precision*). Okazuje się, że obecnie programiści Javy potrzebujący wysokiej wydajności przy działaniach na wielkich liczbach mogą uzasadnić skorzystanie z GMP poprzez metody natywne [Blum14].

Wykorzystanie metod natywnych niesie ze sobą **poważne** niedogodności. Ponieważ języki wykorzystywane do tworzenia tych metod nie są **bezpieczne** (temat 50.), aplikacje korzystające z nich przestają być odporne na błędy dostępu do pamięci. Ponieważ języki natywne nie są niezależne od platformy, aplikacje korzystające z metod natywnych nie mogą być dowolnie przenoszone pomiędzy platformami. Ich debugowanie jest znacznie trudniejsze. Jeśli się jest nieostrożnym, metody natywne mogą **obniżyć** wydajność aplikacji, ponieważ mechanizm odzyskiwania nieużytków nie może zautomatyzować lub nawet śledzić zużycia pamięci (temat 8.). Istnieje też pewien dodatkowy koszt wejścia do metody natywnej i wyjścia z niej. Ponadto metody natywne są trudne w tworzeniu i analizie.

Podsumujmy. Dokładnie przeżyjmy zasadność zastosowania metod natywnych. W rzadkich przypadkach osiągniesz dzięki nim wzrost wydajności. Jeżeli musisz wykorzystać metody natywne do odwołania się do zasobów niskiego poziomu lub istniejących bibliotek, próbuj zminimalizować ilość kodu natywnego i bardzo dokładnie go przetestuj. Jeden błąd w kodzie natywnym może uszkodzić całą aplikację.

Temat 67. Unikanie przesadnej optymalizacji

Na temat optymalizacji powstały trzy aforyzmy, które wszyscy powinni znać.

W imię efektywności popełniono więcej grzechów programowania (niekoniecznie ją osiągając) niż z jakiegokolwiek innego powodu — w tym jawnej głupoty.

William A. Wulf [Wulf72]

Powinniśmy zapomnieć o małych usprawnieniach na mniej więcej 97% czasu — przedwczesna optymalizacja jest przyczyną wszelkiego zła.

Donald E. Knuth [Knuth74]

Powinniśmy postępować zgodnie z dwiema zasadami optymalizacji:

Zasada 1. Nie optymalizować.

Zasada 2. (tylko dla ekspertów). Jeszcze jej nie wykonuj — to znaczy, dopóki nie otrzymasz perfekcyjnie jasnego i nieoptymalizowanego rozwiązania.

M.A. Jackson [Jackson75]

Wszystkie te aforyzmy wyprzedzają język Java o dwie dekady. Przekazują one głęboką prawdę o optymalizacji — łatwo można zrobić wiele złego zamiast poprawić, szczególnie zbyt wcześnie rozpoczynając optymalizację. Optymalizując w ten sposób, można wytworzyć program, który nie jest ani szybki, ani prawidłowy, ani łatwy do poprawienia.

Nie poświęcaj podstawowych zasad architektury w imię wydajności. **Staraj się pisać programy poprawne, a nie szybkie.** Jeżeli dobry program nie jest wystarczająco szybki, jego budowa pozwoli go łatwo zoptymalizować. Dobre programy konstruowane są zgodnie z zasadą **ukrywania informacji**. Gdziekolwiek jest to możliwe, ograniczaj decyzje projektowe do pojedynczych modułów, dzięki czemu będzie można je zmienić bez potrzeby zmieniania pozostałych części systemu (temat 15.).

Nie oznacza to, że możesz ignorować zagadnienia wydajności do czasu zakończenia programu. Problemy implementacji mogą być poprawione w późniejszym procesie optymalizacji, ale duże błędy strukturalne, ograniczające wydajność, mogą być niemal niemożliwe do usunięcia bez przepisywania całego systemu. Zmiany fundamentalnych aspektów projektu po jego zaimplementowaniu mogą spowodować powstanie systemu o niedobrej strukturze, który jest trudny w konserwacji i rozbudowie. Dlatego powinieneś pomyśleć o wydajności już na etapie procesu projektowania.

Staraj się unikać konstrukcji projektowych, które mogą ograniczać wydajność. Najtrudniejszymi do późniejszej modyfikacji składnikami projektu są elementy odpowiedzialne za komunikację między modułami i zewnętrznym środowiskiem. Najważniejszymi takimi składnikami są API, protokoły komunikacyjne i trwałe formaty danych. Nie tylko są one trudne lub niemożliwe do modyfikacji, ale mogą również poważnie ograniczać wydajność systemu.

Pomyśl o wpływie decyzji projektowych na wydajność API. Jeżeli typ publiczny będzie modyfikowalny, to kod będzie musiał wykonywać wiele niepotrzebnych operacji defensywnego kopiowania (temat 50.). Podobnie skorzystanie z dziedziczenia klas publicznych w przypadku, gdy można skorzystać z kompozycji, przywiązując na zawsze klasę do jej klasy bazowej, co może spowodować sztuczne ograniczenie wydajności klasy pochodnej (temat 18.). Ostatnim przykładem jest użycie w API typu implementacji zamiast interfejsu, co powoduje stałe związanie z wybraną implementacją bez możliwości zmiany, gdy zostaną napisane nowe wydajniejsze implementacje (temat 64.).

Wpływ projektu API na wydajność jest bardzo realny. Zwróć uwagę na metodę `getSize` z klasy `java.awt.Component`. Została podjęta decyzja, że ta metoda, krytyczna dla wydajności, zwraca obiekt `Dimension`, co — połączone z decyzją, że obiekty `Dimension` są modyfikowalne — powoduje, że wszystkie implementacje tej metody muszą zwracać przy każdym wywołaniu nowy obiekt. Nawet w nowoczesnych maszynach wirtualnych, gdzie tworzenie małych obiektów jest bardzo szybkie, utworzenie miliona niepotrzebnych obiektów może znacznie zmniejszyć szybkość działania programu.

W takim przypadku istnieje kilka możliwości poprawienia błędu. Idealnie, klasa `Dimension` powinna stać się niezmienna (temat 17.), można również zastąpić metodę `getSize` dwoma metodami, zwracającymi dwie wartości proste z obiektu `Dimension`. Metody takie zostały dodane do klasy `Component` w Javie 2, właśnie w celu poprawienia wydajności. Istniejący kod korzysta jednak nadal z metody `getSize` i nadal jego wydajność cierpi z powodu błędnej decyzji projektowej.

Na szczęście zazwyczaj dobry projekt API idzie w parze z dobrą wydajnością systemu. **Bardzo złym pomysłem jest komplikowanie API w celu osiągnięcia lepszej**

wydajności. Problem z wydajnością może zniknąć w kolejnej wersji platformy lub innego wykorzystywanego programu, natomiast wypaczone API i problemy z jego obsługą pozostaną do końca jego życia.

Po dokładnym zaprojektowaniu i napisaniu jasnego i spójnego kodu **można** rozważyć jego optymalizację, pod warunkiem, że program osiąga niezadowalającą wydajność.

Przypomnij sobie dwie zasady Jacksona na temat optymalizacji: „Nie optymalizować” i „(dla ekspertów) Jeszcze nie optymalizować”. Możemy dodać jeszcze jedną: **kontroluj wydajność przed i po każdej optymalizacji.** Możesz być zdziwiony otrzymanymi wynikami. Często optymalizacje nie przynoszą żadnego, dającego się zmierzyć, efektu — czasami nawet pogarszają sytuację. Najważniejszym problemem jest określenie miejsca, w którym program spędza najwięcej czasu. Fragment kodu, jaki według Ciebie może spowalniać program, może nie być tym właściwym i w takim przypadku stracisz jedynie czas na optymalizację. Znana prawda mówi, że programy przez 90% czasu wykonują 10% fragment kodu.

Narzędzia profilujące kod pozwalają odnaleźć miejsca, na które należy zwrócić szczególną uwagę. Programy te zwracają informacje na temat wykonywanej aplikacji, takie jak czas wykonania poszczególnych metod oraz liczba ich wywołań. Oprócz kierowania Twojej uwagi na odpowiednie fragmenty programu mogą one ostrzegać o konieczności wprowadzenia zmian algorytmicznych. Jeżeli program korzysta z algorytmu o koszcie kwadratowym (lub gorszym), żadna optymalizacja nie poprawi sytuacji radykalnie. Pomoże jedynie zastosowanie efektywniejszego algorytmu. Im większa jest objętość kodu, tym większe jest znaczenie profilera. Jest to podobne do szukania igły w stogu siana; im większy stóg, tym bardziej przydaje się wykrywacz metalu. Narzędziem zasługującym na dodatkową uwagę jest jmh. Narzędzie to nie jest profilerem, ale **frameworkiem do tworzenia miniaturowych testów wydajnościowych**, pozwalających w bardzo szczegółowy sposób poznać wydajność wybranego fragmentu kodu Javy [JMH].

Potrzeba mierzenia efektów optymalizacji jest o wiele większa dla platformy Java niż w przypadku tradycyjnych języków programowania, ponieważ Java nie posiada stabilnego **modelu wydajności**. Koszt wykonania różnych podstawowych operacji nie jest dobrze zdefiniowany. Tak zwany „odstęp składniowy” pomiędzy tym, co programista pisze, a tym, co wykonuje procesor, jest o wiele większy niż w przypadku tradycyjnie kompilowanych języków. Utrudnia to niezawodne przewidywanie konsekwencji wprowadzanych optymalizacji. Istnieje wiele mitów na temat wydajności, które okazują się często połowiczną prawdą lub zwykłymi kłamstwami.

Oprócz tego, że model wydajności Java jest bardzo słabo zdefiniowany, jest różny dla różnych implementacji JVM, a nawet dla różnych wersji tej samej platformy. Jeżeli Twój program będzie uruchamiany na różnych implementacjach JVM,

ważne jest, aby sprawdzić efekty optymalizacji na wszystkich tych maszynach wirtualnych. Czasami będziesz zmuszony do zaniechania niektórych zmian w celu utrzymania odpowiedniej wydajności w różnych implementacjach JVM.

Po niecałych dwóch dekadach od napisania pierwszej wersji tego tematu każdy element stosu programistycznego Javy stał się bardziej złożony — dotyczy to zarówno procesorów, jak i maszyn wirtualnych i innego sprzętu, na którym działają napisane w tym języku programy. Wszystko to powoduje, że dziś wydajność programów Javy jest jeszcze mniej przewidywalna niż w 2001 r., co jeszcze bardziej wzmacnia potrzebę wykonywania odpowiedniej analizy.

Podsumujmy. Nie należy usiłować pisać szybkich programów — należy pisać programy dobre. Należy myśleć o wydajności w czasie projektowania systemu, szczególnie, gdy projektujesz API, protokół komunikacyjny lub format danych. Po ukończeniu systemu należy zmierzyć wydajność. Jeżeli program jest wystarczająco szybki, praca jest zakończona. Jeżeli nie, należy zlokalizować źródło problemów za pomocą profilera i zoptymalizować odpowiednie elementy systemu. Pierwszym krokiem jest sprawdzenie wybranego algorytmu — żadna optymalizacja niskiego poziomu nie pomoże w przypadku mało wydajnego algorytmu. Jeżeli jest to niezbędne, należy powtórzyć proces, kontrolując wydajność po każdej modyfikacji, aż do osiągnięcia zadowalającego efektu.

Temat 68. Wykorzystanie ogólnie przyjętych konwencji nazewnictwa

Platforma języka Java posiada ugruntowany zbiór **konwencji nazewnictwa**, z których wiele jest zapisanych w dokumencie *The Java Language Specification* [JLS, 6.1]. Konwencje te można podzielić na dwie kategorie: typograficzne i gramatyczne.

Istnieje tylko kilka typograficznych konwencji nazewnictwa, obejmujących pakiety, klasy, interfejsy i pola. Nie powinienes zbyt często ich łamać i nigdy bez bardzo dobrze uzasadnionego powodu. Jeżeli API łamie te zasady, może być kłopotliwe w użyciu. Jeżeli narusza je implementacja, może być ona trudna do konserwacji. W obu przypadkach zmiany te mogą powodować pomyłki i irytację programistów, którzy będą wykorzystywać Twój kod i mogą przyjąć niewłaściwe założenia, prowadzące w rezultacie do błędów. Konwencje te zostały podsumowane w tym temacie.

Nazwy pakietów powinny mieć postać hierarchiczną, gdzie poszczególne części są rozdzielone kropką. Nazwy części powinny składać się z małych liter i, rzadko, cyfr. Nazwa pakietu, który będzie używany poza Twoją organizacją, powinna rozpoczynać się od nazwy domeny internetowej tej organizacji, rozpoczynając od domeny najwyższego poziomu, na przykład `edu.cmu`, `com.google`, `org.ietf`.

Biblioteki standardowe i pakiety opcjonalne, których nazwy rozpoczynają się od `java` lub `javax`, są wyjątkami od tej reguły. Użytkownicy nie mogą tworzyć pakietów, których nazwy rozpoczynają się od `java` i `javax`. Szczegółowe zasady zamiany domen internetowych na prefiksy nazw pakietów opisane są w dokumencie JLS [JLS, 6.1].

Pozostała część nazwy pakietu powinna składać się z jednego lub więcej składników opisujących pakiet. Części te powinny być krótkie, zwykle nie więcej niż osiem znaków. Dopuszczalne są zrozumiałe skróty, jak na przykład `util` zamiast `utilities`. Można stosować akronimy, np. `awt`. Części te powinny składać się zwykle z jednego słowa lub skrótu.

Wiele pakietów posiada nazwy składające się z tylko jednej części (oprócz nazwy domeny). Dodatkowe części są dopuszczalne w przypadku dużych jednostek, których wielkość wymagała podziału na mniejsze fragmenty, stanowiące nieformalną hierarchię. Na przykład pakiet `java.util` zawiera bogatą hierarchię pakietów o nazwach takich, jak `java.util.concurrent.atomic`. Pakiety te są często nazywane **podpakietami**, choć jest to jedynie założenie konwencji — nie ma mechanizmów językowych obsługujących hierarchię pakietów.

Nazwy klas i interfejsów powinny składać się z jednego lub kilku słów. Pierwsza litera każdego słowa winna być zapisana wielką literą, na przykład `List` i `FutureTask`. Powinno się unikać skrótów, poza akronimami i niektórymi często występującymi skrótami, na przykład `min` i `max`. Nie ma zgody, czy akronimy powinny być zapisane wielkimi literami, czy też wielka ma być jedynie pierwsza litera. Choć stosowanie tylko wielkich liter wydaje się być częściej spotykane, istnieje silny argument za zapisywaniem wielkiej tylko pierwszej litery. Nawet jeżeli kilka skrótów zostanie ze sobą sklejone, nadal będziesz mógł określić, w którym miejscu rozpoczyna się następne słowo. Jakie nazwy wolisz stosować, `HTTPURL` czy `HttpUrl`?

Nazwy metod i pól powinny stosować te same zasady, co klasy i interfejsy, z jednym wyjątkiem — pierwsza litera powinna być mała, na przykład: `remove` lub `ensureCapacity`. Jeżeli pierwszym słowem w nazwie pola lub metody jest akronim, powinien być zapisany małymi literami.

Jedynym wyjątkiem od przedstawionej tu reguły są „pola stałych”, których nazwy powinny składać się z jednego lub więcej słów zapisanych wielkimi literami, rozdzielonych podkreśleniami na przykład `VALUES` lub `NEGATIVE_INFINITY`. Pole stałe to statyczne pole `final`, którego wartość jest niezmienna. Jeżeli statyczne pole `final` posiada typ prosty lub niezmienny typ obiektowy (temat 17.), jest polem stałym. Na przykład stałe wyliczeniowe są polami stałymi. Jeżeli statyczne pole `final` ma zmienny typ referencyjny, może być polem stałym, jeżeli obiekt, do którego się odwołuje, jest niezmienny. Warto zwrócić uwagę, że stałe pola są **jedynym** zalecanym użyciem podkreśleń.

Nazwy zmiennych lokalnych korzystają z tych samych konwencji nazewnictwa, co nazwy składników, poza tym, że dopuszczalne są skróty, takie jak pojedyncze litery i krótkie sekwencje znaków, których znaczenie zależy jedynie od kontekstu, w jakim jest użyta zmienna, na przykład `i`, `denom`, `houseNumber`. Parametry wejściowe to szczególny rodzaj zmiennej lokalnej. Ich nazwy powinny być dobierane dużo uważniej niż w przypadku zwykłych zmiennych, ponieważ nazwy te stanowią integralną część dokumentacji.

Nazwy parametrów typu zwykle składają się z jednej litery. Najczęściej jest to jedna z następujących pięciu: `T` dla dowolnego typu, `E` dla typu elementów kolekcji, `K` oraz `V` dla typów kluczy i wartości w odwzorowaniach oraz `X` dla wyjątków. Typem zwracanym przez funkcję jest najczęściej `R`. Sekwencją dowolnych typów może być `T`, `U`, `V` lub `T1`, `T2`, `T3`.

W tabeli 9.1 zebrane są częste przykłady konwencji typograficznych.

Tabela 9.1. Przykłady konwencji typograficznych

Typ identyfikatora	Przykłady
Pakiet lub moduł	<code>org.junit.jupiter.api.com.google.common.collect</code>
Klasa lub interfejs	<code>Stream</code> , <code>FutureTask</code> , <code>LinkedHashMap</code> , <code>HttpClient</code>
Metoda lub pole	<code>remove</code> , <code>groupBy</code> , <code>getCrc</code>
Pole stałej	<code>MIN_VALUE</code> , <code>NEGATIVE_INFINITY</code>
Zmienna lokalna	<code>i</code> , <code>denom</code> , <code>houseNumber</code>
Parametr typu	<code>T</code> , <code>E</code> , <code>K</code> , <code>V</code> , <code>X</code> , <code>R</code> , <code>U</code> , <code>T1</code> , <code>T2</code>

Gramatyczne konwencje nazewnictwa są bardziej elastyczne i kontrowersyjne niż konwencje typograficzne. Nie ma żadnych konwencji gramatycznych dotyczących pakietów. Klasy są zwykle nazywane rzeczownikami lub związkami rzeczowników, na przykład `Thread`, `PriorityQueue` lub `ChessPiece`. Klasy użytkowe, których instancje nie są możliwe do utworzenia (temat 4.), są najczęściej nazywane rzeczownikami w liczbie mnogiej, na przykład `Collectors` lub `Collections`. Interfejsy są nazywane podobnie do klas, na przykład `Collection` lub `Comparator` lub przymiotnikiem z końcówką „-able” lub „-ible”, na przykład `Runnable`, `Iterable` lub `Accessible`. Ponieważ typy adnotacyjne mają bardzo wiele zastosowań, nie ma ustalonych zasad. Stosowane są rzeczowniki, czasowniki lub przymiotniki, na przykład `Binding` ↪ `Annotation`, `Inject`, `ImplementedBy` lub `Singleton`.

Metody wykonujące jakieś akcje są zwykle nazywane czasownikiem lub związkiem czasownikowym (zawierającym obiekt), na przykład `append` lub `drawImage`. Metody zwracające wartość boolean mają najczęściej nazwy rozpoczynające się od „is” (a rzadziej również od „has”), po którym następuje rzeczownik, na przykład `isDigit`, `isProbablePrime`, `isEmpty`, `isEnabled` lub `hasSiblings`.

Metody zwracające wartości innego typu niż `boolean` lub atrybuty obiektu są nazywane rzeczownikiem lub związkiem rzeczownikowym, rozpoczynającym się od słowa „`get`”, na przykład `size`, `hashCode` lub `getTime`. Istnieje tendencja do stosowania jedynie trzeciej formy, ale nie ma ona silnych podstaw. Pierwsze dwie postacie nazw pozwalają najczęściej uzyskać czytelniejszy kod:

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Uwaga na policję!");
```

Forma rozpoczynająca się od „`get`” ma swe korzenie w obecnie w większości przestarzałej specyfikacji `Java Beans`, która stanowiła podstawę pierwszej architektury dla komponentów wielokrotnego stosowania. Wiele nowoczesnych narzędzi nadal korzysta z tej konwencji, więc można jej używać, aby kod potrafił dobrze współpracować z tymi narzędziami. Zaleca się stosowanie tej zasady, jeżeli klasa zawiera zarówno metodę ustawiającą, jak i pobierającą ten sam atrybut. W takim przypadku te dwie metody powinny się nazywać `getAtrybut` i `setAtrybut`.

Kilka nazw metod wymaga specjalnego omówienia. Metody konwertujące typ obiektu, zwracające niezależny obiekt innego typu, są często nazywane *toTyp*, na przykład `toString`, `toArray`. Metody zwracające **widok** (temat 6.), którego typ różni się od otrzymywanego obiektu, często noszą nazwę *asTyp*, na przykład `asList`. Metody zwracające wartości proste o tej samej wartości, co obiekt, na rzecz którego są wywołane, mają nazwy *typValue*, na przykład `intValue`. Często spotykanymi nazwami statycznych metod fabrycznych są `from`, `of`, `valueOf`, `instance`, `getInstance`, `newInstance`, `getTyp` oraz `newTyp` (temat 1.).

Konwencje gramatyczne dla nazw pól są słabo przyjęte i mniej ważne niż te dotyczące klas, interfejsów i metod, ponieważ dobrze zaprojektowane API zawiera niewiele udostępnionych pól. Pola typu `boolean` są zwykle nazywane tak samo jak akcesor, ale bez początkowego „`is`”, na przykład `initialized` lub `composite`. Nazwy pól innych typów są najczęściej rzeczownikami lub związkami rzeczownikowymi, takimi jak `height`, `digits` lub `bodyStyle`. Konwencje gramatyczne w przypadku zmiennych lokalnych są podobne do konwencji dla pól, ale są jeszcze słabsze.

Podsumujmy. Należy skorzystać ze standardowych konwencji nazewnictwa i nauczyć się ich używania. Konwencje typograficzne są łatwe i w większości jednoznaczne, natomiast konwencje gramatyczne bardziej złożone, ale swobodne. I jeszcze cytat ze specyfikacji języka `Java` [JLS, 6.1]: „Konwencje te nie powinny być ślepo naśladowane, jeżeli długo stosowane własne konwencje stanowią inaczej”. Należy przede wszystkim zachować rozsądek.

Wyjątki

Korzystając z najlepszych cech wyjątków, można dzięki nim poprawić czytelność programu, jego solidność i łatwość konserwacji. Użyte niewłaściwie będą miały odwrotny efekt. W rozdziale tym zamieszczamy wskazówki, pomagające efektywnie wykorzystywać wyjątki.

Temat 69. Wykorzystanie wyjątków w sytuacjach nadzwyczajnych

Być może kiedyś spotkasz się z fragmentem kodu, który będzie wyglądał następująco:

```
//Nadużycie wyjątków. Nigdy tak nie rób!  
try {  
    int i = 0 ;  
    while (true)  
        range[i++].climb();  
} catch (ArrayIndexOutOfBoundsException e) {  
}
```

Co robi ten fragment? Działanie to nie jest oczywiste na pierwszy rzut oka i jest to wystarczający powód, aby odradzić stosowanie takiego stylu programowania (temat 67.). Jest to bardzo zły sposób na przeglądanie tablicy. Pętla nieskończona jest przerywana przez zgłoszenie, przechwycenie i zignorowanie wyjątku `ArrayIndexOutOfBoundsException`, gdy program będzie próbował odwołać się do pierwszego elementu spoza zakresu tablicy. Jest to koślawy odpowiednik standardowej metody przeglądania tablicy, natychmiast rozpoznawany przez każdego programistę:

```
for (Mountain m : range)  
    m.climb();
```

Skąd wziął się więc pomysł zastąpienia wypróbowanej metody przeglądania tablicy metodą wykorzystującą wyjątki? Jest to nieudolna próba poprawy wydajności, oparta o nieprawidłowe założenie, że maszyna wirtualna kontroluje dostęp do elementów spoza zakresu tablicy przy wszystkich odwołaniach, więc zwykły test zakończenia pętli — ukryty przez kompilator, ale nadal występujących w pętli `for-each` — jest nadmiarowy i powinien być usunięty. Założenie to powoduje wystąpienie trzech niekorzystnych zjawisk:

- Ponieważ wyjątki są zaprojektowane do obsługi sytuacji wyjątkowych, implementacje JVM słabo optymalizują kod obsługi wyjątku. Tworzenie, zgłaszanie i przechwytywanie wyjątków to operacje stosunkowo kosztowne.
- Umieszczenie kodu w bloku `try-catch` wyklucza niektóre optymalizacje, które nowoczesne maszyny wirtualne mogą zastosować.
- Standardowa metoda przeglądania elementów tablicy niekoniecznie musi wykonywać nadmiarowe sprawdzenia — nowoczesne implementacje JVM potrafią to zoptymalizować.

W rzeczywistości zaprezentowana metoda przeglądania tablicy z wykorzystaniem wyjątku na niemal wszystkich implementacjach JVM jest o wiele wolniejsza, niż metoda najczęściej wykorzystywana. Na moim komputerze kod, przeglądający stulementową tablicę metodą wykorzystującą wyjątki, wykonywał się dwa razy dłużej niż fragment korzystający ze standardowej metody.

Metoda wykorzystująca wyjątki nie tylko zaciemnia kod i spowalnia program, ale również nie gwarantuje poprawnej pracy. W przypadku wystąpienia całkowicie innego błędu metoda ta może zawieść bez żadnego ostrzeżenia, maskując błąd, co ogromnie skomplikuje proces uruchamiania programu. Załóżmy, że obliczenia wykonywane w pętli zawierają błąd przekroczenia zakresu całkowicie innej tablicy. Jeżeli zostałaby wykorzystana standardowa metoda przeglądania tablicy, błąd ten spowodowałby zgłoszenie wyjątku, co z kolei natychmiast zakończyłoby wątek wygenerowaniem odpowiedniego komunikatu o błędzie. Jeżeli jednak zostanie użyta metoda z przechwytywaniem wyjątku, to wyjątek związany z błędem zostanie mylnie zinterpretowany jako normalne zakończenie pętli.

Przykład ten ilustruje, że **wyjątki, tak jak ich nazwa wskazuje, powinny być wykorzystywane do obsługi sytuacji wyjątkowych — nie powinny być nigdy używane do sterowania normalnym przebiegiem kodu programu**. Mówiąc ogólniej, powinieneś stosować standardowe, rozpoznawalne konstrukcje programowe zamiast sprytnych, stosowanych jedynie w celu osiągnięcia lepszej wydajności. Nawet, gdy zysk wydajności jest znaczny, może szybko zostać zniwelowany w kolejnych wersjach szybko rozwijających się implementacji JVM. Pozostaną jednak kłopoty przy usuwaniu subtelnych błędów, występujące w przypadku zastosowania sprytnych metod.

Zasada ta powinna być również stosowana przy projektowaniu API. **Dobrze zaprojektowane API nie powinno wymuszać na klientach wykorzystania wyjątków do sterowania zwykłym przebiegiem programu.** Klasa z metodami „zależnymi od stanu”, które są wywoływane jedynie w określonych, nieprzewidywalnych warunkach, powinny posiadać zwykle oddzielne metody „testujące stan”, czyli sprawdzające, czy należy wywołać pierwszą metodę. Na przykład klasa `Iterator` posiada zależną od stanu metodę `next`, która zwraca następny element wyliczenia oraz skojarzoną z nią metodę `hasNext`, testującą stan. Pozwala to na skorzystanie ze standardowego idiomu, służącego do przeglądania kolekcji w standardowych pętlach `for` (jak również w pętlach `for-each`, w których metoda `hasNext` jest wykorzystywana wewnętrznie):

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
    ...
}
```

Jeżeli klasa `Iterator` nie posiadałaby metody `hasNext`, klient zostałby zmuszony do skorzystania z poniższej metody:

```
// Nie korzystaj z tej okropnej metody przeglądania kolekcji!
try {
    Iterator<Foo> i = collection.iterator();
    while (true) {
        Foo foo = i.next();
        ...
    }
} catch (NoSuchElementException e) {
}
```

Przedstawiony kod wygląda podobnie do przykładu rozpoczynającego ten temat. Oprócz tego, że przykład jest rozwlekły, działa dużo wolniej niż standardowy oraz może ukrywać błędy, pochodzące z innych części systemu.

Innym sposobem zapewnienia osobnej metody testującej stan jest zwracanie przez metodę zależną od stanu pustego obiektu opcjonalnego (temat 55.) lub specjalnej wartości, jak na przykład `null`, jeżeli obiekt jest w nieprawidłowym stanie.

Warto wymienić kilka sugestii, dotyczących wyboru pomiędzy metodą testującą stan i zwracaniem wartości specjalnej lub opcjonalnej. Jeżeli do obiektu można się odwoływać równolegle bez zewnętrznej synchronizacji lub jego stan zmienia się pod wpływem zewnętrznych czynników, wykorzystanie opcjonalnej lub specjalnej wartości może być kluczowe dla prawidłowego działania, ponieważ obiekt może zmienić swój stan pomiędzy wywołaniem metody testującej stan i odpowiadającej jej metody zależnej od stanu. Jeżeli ważna jest wydajność, należy skorzystać ze zwracania wartości opcjonalnej lub specjalnej (jeżeli wykonanie osobnej metody testującej stan wiąże się z wykonaniem tych samych operacji, co w odpowiadającej jej metodzie zależnej od stanu). Jeżeli nie zachodzą te przypadki, zalecane

jest tworzenie metody testującej stan. Pozwala ona osiągnąć lepszą czytelność kodu, a w przypadku jej niewłaściwego stosowania łatwiej można odszukać i poprawić błąd. Jeśli zapomnimy wywołać metodę testującą stan, metoda zależna od stanu zgłosi wyjątek, co uczyni błąd łatwym do odnalezienia; jeśli zapomnimy o sprawdzeniu wartości specjalnej, błąd będzie trudniejszy do odkrycia. Przedstawiony problem nie występuje w przypadku zwracania wartości typu opcjonalnego.

Podsumujmy. Wyjątki są zaprojektowane do wykorzystania w sytuacjach wyjątkowych. Nie należy ich używać do zwykłej kontroli przepływu sterowania i nie należy pisać API wymuszających takie działanie.

Temat 70. Stosowanie wyjątków przechwytywanych i wyjątków czasu wykonania

Język Java pozwala na stosowanie trzech rodzajów wyjątków: **wyjątków przechwytywanych**, **wyjątków czasu wykonania** i **błędów**. Często pojawiają się wątpliwości, dotyczące zastosowania odpowiedniego rodzaju wyjątku. Wybór nie zawsze jest jednoznaczny, ale istnieją ogólne zasady, ułatwiające podjęcie decyzji.

Podstawową zasadą jest określenie, czy wyjątek jest przechwytywany czy nieprzechwytywany. **Wyjątki przechwytywane należy stosować w przypadkach, w których spodziewamy się, że wywołujący może przywrócić normalny stan.** Zgłaszając wyjątek przechwytywany, zmuszasz wywołującego do obsłużenia tego wyjątku w klauzuli catch lub do jego propagacji na wyższy poziom. Każdy przechwytywany wyjątek deklarowany przez metodę jest wskazówką dla użytkownika API, że w trakcie wykonywania metody można spodziewać się, że taki będzie wynik wykonania danej metody.

Przekazując przechwytywany wyjątek użytkownikowi API, projektant nakazuje wykonanie operacji przywracających normalny stan systemu. Użytkownik może zgodzić się na ten nakaz, przechwytyując wyjątek, lub zignorować go, choć nie jest to najlepszym pomysłem (temat 77.).

Istnieją dwa rodzaje wyjątków nieprzechwytywanych — wyjątki czasu wykonania i błędy. Ich działanie jest identyczne — oba rodzaje nie mogą i zwykle nie powinny być przechwytywane. Jeżeli program zgłasza wyjątek nieprzechwytywany lub błąd, zwykle przywrócenie właściwego stanu systemu jest niemożliwe i kontynuowanie działania programu może spowodować więcej złego niż dobrego. Jeżeli program nie przechwyci takiego wyjątku, spowoduje to zakończenie bieżącego wątku z odpowiednim komunikatem o błędzie.

Do wskazywania błędów programistycznych należy używać wyjątków czasu wykonania. Większość wyjątków czasu wykonania wskazuje na **naruszenia założeń**. Naruszenie założenia występuje, gdy klient API nie może dotrzymać założeń, wymienionych w specyfikacji API. Dla przykładu, założenia dla dostępu do tablicy określają, że indeks tablicy musi być liczbą od zera do wielkości tablicy minus jeden. Wyjątek `ArrayIndexOutOfBoundsException` wskazuje na niedotrzymanie tego założenia.

Z przedstawioną powyżej radą jest pewien problem, bo nie zawsze w sposób oczywisty wiadomo, czy mamy do czynienia z błędem programisty, czy może sytuacją pozwalającą przywrócić normalny stan. Przykładem może tu być sytuacja wyczerpania zasobów, która mogła zostać spowodowana przez programistę próbującego utworzyć zbyt dużą tablicę lub rzeczywisty deficyt zasobów. Jeśli wyczerpanie zasobów wynika z tymczasowej sytuacji lub chwilowego zapotrzebowania na pamięć, zapewne łatwo będzie powrócić do stanu normalnego. To od oceny projektanta API zależy, czy będzie przekonany co do możliwości powrotu do stanu normalnego po zgłoszeniu tego rodzaju wyjątku. Jeśli wierzy, że stan normalny jest przywracalny, powinien użyć wyjątku przechwytywanego. W przeciwnym razie powinien użyć wyjątku czasu wykonywania. Jeśli decyzja jest zbyt trudna do podjęcia lub zbyt niepewna, lepiej zdecydować się na wyjątek nieprzechwytywany z powodów wymienionych w temacie 71.

Mimo że specyfikacja języka Java nie wymusza tego, istnieje bardzo silna konwencja, mówiąca, że **błędy** są zarezerwowane przez maszynę wirtualną do wskazywania braku zasobów, naruszenia niezmienników lub innych błędów, powodujących niemożliwość kontynuacji wykonania programu. Z powodu niemal całkowitej akceptacji tej konwencji najlepiej nie tworzyć nowych podklas dla klasy `Error`. **Wszystkie nieprzechwytywane wyjątki definiowane przez Ciebie powinny dziedziczyć po `RuntimeException`** (bezpośrednio bądź pośrednio). Nie tylko nie powinieneś tworzyć nowych podklas dla klasy `Error`, ale również nie powinieneś ich zgłaszać (jedynym wyjątkiem jest `AssertionError`).

Możliwe jest zdefiniowanie wyjątku, który nie dziedziczy po klasie `Exception`, `RuntimeException` lub `Error`. Specyfikacja języka nie zajmuje się bezpośrednio takimi wyjątkami, ale niejawnie określa, że mają one identyczne działanie, jak zwykle, przechwytywane wyjątki (które dziedziczą po `Exception`, a nie `RuntimeException`). Kiedy więc powinniśmy z nich skorzystać? Prawdopodobnie nigdy. Nie mają one żadnych zalet w porównaniu ze zwykłymi przechwytywanymi wyjątkami i prawdopodobnie będą jedynie myliły użytkowników Twojego API.

Projektanci API często zapominają, że wyjątki są w pełni funkcjonalnymi obiektami, w których mogą być definiowane normalne metody. Podstawowym zastosowaniem takich metod jest dostarczenie kodu przechwytyującego wyjątek i dodatkowych informacji na temat zdarzenia, które spowodowało zgłoszenie wyjątku.

W przypadku braku takich metod programiści muszą analizować ciąg zwracany przez wyjątek w celu wyluskania dodatkowych danych. Jest to bardzo zła praktyka (temat 12.). Klasy rzadko określają szczegóły ich reprezentacji jako ciąg znaków. Reprezentacja ta może różnić się w różnych implementacjach i wersjach. Dlatego kod analizujący ciąg reprezentujący wyjątek najprawdopodobniej będzie nieprzenośny i podatny na błędy.

Ponieważ wyjątki przechwytywane najczęściej wskazują sytuacje naprawialne, szczególnie ważne jest definiowanie w klasach wyjątków metod, które dostarczają danych pomagających przywrócić prawidłowy stan. Na przykład przechwytywany wyjątek może być zgłaszany w sytuacji, gdy wykonana została próba zakupu towaru kartą podarunkową z powodu niewystarczających środków. Wyjątek ten powinien zawierać metodę zwracającą brakującą kwotę, dzięki czemu niedobór może być wyświetlony kasjerowi, a kupujący będzie mógł w takiej sytuacji dopłacić brakującą kwotę. Więcej informacji zawiera temat 75.

Podsumujmy. Wyjątków przechwytywanych należy używać do obsługi przypadków, w których można naprawić błąd, a wyjątków nieprzechwytywanych należy używać do wykonania obsługi błędów programowych. Jeżeli nie jesteś pewien, lepiej skorzystaj z wyjątku nieprzechwytywanego. Nie definiuj innych klas dziedziczących po `Throwable`, które nie są wyjątkami przechwytywanymi lub czasu wykonywania. Zapewnij w klasie wyjątku przechwytywanego metody wspomagające powrót do normalnej sytuacji.

Temat 71. Unikanie niepotrzebnych wyjątków przechwytywanych

Wielu programistów języka Java nie lubi wyjątków przechwytywanych, ale ich właściwe użycie ulepsza API i programy. W przeciwieństwie do zwracanych wartości **wymuszają** one na programiście zajęcie się sytuacją wyjątkową, co niezmiernie zwiększa solidność aplikacji. Jednak nadużywanie wyjątków może spowodować, że API będzie mało wygodne. Jeżeli metoda zgłasza jeden lub więcej wyjątków, kod wywołujący ją musi obsługiwać te wyjątki w jednym lub większej liczby bloków `catch` albo musi zadeklarować, że będzie zgłaszał te wyjątki, co umożliwi ich propagację. Obie czynności powodują zwiększone obciążenie programistów. To zwiększenie obciążenia programistów wzrosło jeszcze bardziej w Javie 8, bo metody zwracające wyjątki przechwytywane nie mogą być stosowane bezpośrednio jako element strumienia (tematy od 45. do 48.).

Obciążenie to jest uzasadnione, gdy nie można uniknąć sytuacji wyjątkowych przy właściwym stosowaniu API **oraz** gdy programista wykorzystujący to API ma możliwość przedsięwziąć sensowne akcje po wystąpieniu takiego wyjątku. W sytuacji, gdy

nie są spełnione oba te warunki, bardziej właściwe jest zastosowanie wyjątku nieprzechwytywanego. Musimy odpowiedzieć sobie na pytanie, w jaki sposób programista ma obsłużyć wyjątek. Czy jest to najlepsze, co może być wykonane?

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Nie powinien się zdarzyć!
}
```

I drugi przypadek.

```
} catch (TheCheckedException e) {
    e.printStackTrace(); // To już koniec.
    System.exit(1);
}
```

Jeżeli programista nie może lepiej obsłużyć wyjątku, lepszy będzie wyjątek nieprzechwytywany.

Dodatkowo obciążenie programisty przez obsługę wyjątku jest nieco większe, jeżeli metoda zgłasza **jeden** wyjątek przechwytywany. Jeżeli zgłaszane jest więcej wyjątków, metoda musi być umieszczona w bloku try, a obsługa poszczególnych wyjątków jest umieszczona w kolejnych blokach catch. Jeżeli metoda zgłasza tylko jeden wyjątek, jest on odpowiedzialny za umieszczenie wywołania tej metody w bloku try i nie może zastosować metody w strumieniach. W tej sytuacji oplaća się pomyśleć o sposobach uniknięcia przechwytywania wyjątków.

Najprostszym sposobem wyeliminowania wyjątku przechwytywanego jest zwrócenie typu **opcjonalnego** ukrywającego rzeczywiście zwracany typ (temat 55.). Zamiast zgłaszać wyjątek, metoda po prostu zwraca pusty obiekt opcjonalny. Wadą tego rozwiązania jest to, że metoda nie jest w stanie zwrócić dodatkowych informacji ze szczegółami powodu niepowodzenia obliczeń. Wyjątki w przeciwieństwie do typów opcjonalnych mają konkretne podtypy i mogą udostępniać metody oferujące dodatkowe informacje (temat 70.).

Jedną z technik zmiany wyjątku przechwytywanego na nieprzechwytywany jest podział metody na dwie części — pierwsza zwraca wartość boolean, wskazującą na to, czy powinien być zgłoszony wyjątek. Stosując tę metodę, powodujemy, że poniższa sekwencja wywołań:

```
// Wywołanie ze sprawdzaniem wyjątków
try {
    obj.action(args);
} catch (TheCheckedException e) {
    ... // obsługa sytuacji wyjątkowej
}
```

zmienia się na:

```
// Wywołanie z użyciem metody testującej stan i wyjątku nieprzechwytywanego
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    ... // obsługa sytuacji wyjątkowej
}
```

Transformacja taka nie jest zawsze prawidłowa, ale tam, gdzie można ją zastosować, może ona ułatwić używanie API. Choć druga sekwencja wywołań nie jest ładniejsza od pierwszej, to jednak API jest bardziej elastyczne. W przypadkach, gdy programista wie, że wywołanie na pewno się uda lub pozwala na zakończenie programu w przypadku niepowodzenia, pokazana transformacja pozwala na zastosowanie najprostszej sekwencji wywołań:

```
obj.action(args);
```

Jeżeli podejrzewasz, że najprostsza sekwencja wywołania będzie powszechnie stosowana, to taka modyfikacja API jest właściwa. Po takiej modyfikacji API będzie niemal identyczne, jak w przypadku zastosowania metody testującej stan, przedstawionej w temacie 69. i należy tu przestrzegać tych samych zasad. Jeżeli do obiektu występują jednoczesne odwołania bez zewnętrznej synchronizacji lub jego stan zmienia się pod wpływem czynników zewnętrznych, przedstawiona modyfikacja jest nieprawidłowa, ponieważ stan obiektu może zmienić się pomiędzy wywołaniem metody `actionPermitted` i `action`. Jeżeli metoda `actionPermitted` będzie musiała powtórzyć operacje wykonywane w metodzie `action`, transformacja ta spowoduje obniżenie wydajności systemu.

Podsumujmy. Gdy wyjątki przechwytywane są używane oszczędnie, mogą poprawić czytelność programów; gdy są nadużywane, spowodują, że API będzie bardzo niewygodne w użyciu. Jeśli kod wywołujący nie jest w stanie powrócić do stanu normalnego, zgłoś wyjątek nieprzechwytywany. Jeśli wyjście z sytuacji wyjątkowej jest możliwe i chcemy zmusić programistę piszącego kod wywołujący do obsługi sytuacji nietypowej, najpierw rozważ zastosowanie typu opcjonalnego. Tylko jeśli nie zapewniłby on odpowiednich informacji o przyczynie porażki, zgłoś wyjątek przechwytywany.

Temat 72. Wykorzystanie wyjątków standardowych

Jednym z atrybutów najlepiej odróżniających ekspertów w dziedzinie programowania od mniej zaawansowanych programistów jest dbałość ekspertów o wysoki stopień ponownego wykorzystania kodu. Wyjątki nie są tutaj wyjątkiem od zasady

powtórznego wykorzystania kodu. Biblioteki platformy Java zawierają podstawowy zbiór wyjątków nieprzechwytywanych, który obejmuje większość sytuacji wyjątkowych spotykanych w API.

Ponowne wykorzystanie istniejących wyjątków niesie ze sobą wiele korzyści. Najważniejszą z nich jest ułatwienie nauki stosowania API, ponieważ korzysta ono z dobrze ugruntowanych konwencji, z którymi programista jest już zaznajomiony. Nie mniej ważną korzyścią jest czytelność programów korzystających z takiego API, ponieważ kod nie jest zabałaganiony nieznanymi wyjątkami. Na koniec, mniej klas wyjątków powoduje zmniejszenie ilości zużywanej pamięci i skraca czas ładowania klas.

Najczęściej wykorzystywanym wyjątkiem jest `IllegalArgumentException` (temat 49.). Najczęściej jest on zgłaszany, gdy wywołujący przekaże argument o niewłaściwej wartości. Na przykład można skorzystać z tego wyjątku, jeżeli jako parametr reprezentujący liczbę powtórzeń zostanie przekazana wartość ujemna.

Innym często stosowanym wyjątkiem jest `IllegalStateException`. Jest to wyjątek zgłaszany, jeżeli w aktualnym stanie obiektu wywołanie jest nielegalne, na przykład gdy wywołujący będzie próbował skorzystać z obiektu przed jego prawidłowym zainicjowaniem.

Można uznać, że wszystkie błędne wywołania metod sprowadzają się do nieprawidłowego argumentu i do niewłaściwego stanu obiektu, ale do sygnalizowania niektórych rodzajów nieprawidłowych stanów i argumentów wykorzystywane są najczęściej specjalizowane wyjątki. Jeżeli wartością parametru będzie `null` w miejscu, gdzie wartości `null` są niedozwolone, konwencja nakazuje zgłosić wyjątek `NullPointerException`, a nie `IllegalArgumentException`. Podobnie, jeżeli przekazana wartość parametru oznaczającego indeks wykracza poza dozwolony zakres, należy zgłosić wyjątek `IndexOutOfBoundsException`, a nie `IllegalArgumentException`.

Innym wyjątkiem ogólnego przeznaczenia, jaki powinniśmy znać, jest `ConcurrentModificationException`. Wyjątek ten powinien być zgłaszany, jeżeli obiekt zaprojektowany do wykorzystywania przez jeden wątek lub korzystający z zewnętrznej synchronizacji wykryje, że jest (lub był) równolegle modyfikowany. Wyjątek ten powinien być jednak tylko wskazówką, bo nie jesteśmy w stanie w sposób niebudzący wątpliwości wykrywać współbieżnych modyfikacji.

Ostatnim wyjątkiem ogólnego przeznaczenia wartym omówienia jest `UnsupportedOperationException`. Wyjątek ten jest zgłaszany, gdy obiekt nie obsługuje wywoływanej operacji. Jest on dość rzadko wykorzystywany, ponieważ większość obiektów obsługuje wszystkie implementowane przez siebie metody. Wykorzystywany jest w sytuacjach, gdy implementacja interfejsu nie posiada obsługi jakiejś **opcjonalnej operacji** definiowanej przez interfejs. Na przykład implementacja interfejsu `List` przeznaczona tylko do dołączania elementów zgłosi ten wyjątek, gdy ktoś będzie próbował usunąć element.

Nie używaj w sposób bezpośredni `Exception`, `RuntimeException`, `Throwable` lub `Error`. Traktuj te klasy tak, jakby były abstrakcyjne. Nie jest możliwe niezawodne sprawdzanie tych wyjątków, bo stanowią klasy bazowe dla wielu innych wyjątków zgłaszanych przez metody.

W tabeli 10.1 przedstawione jest zestawienie często wykorzystywanych wyjątków.

Tabela 10.1. Często wykorzystywane wyjątki

Wyjątek	Zastosowanie
<code>IllegalArgumentException</code>	Nieprawidłowa wartość parametru innego niż <code>null</code>
<code>IllegalStateException</code>	Nieprawidłowy stan obiektu dla wywoływanej metody
<code>NullPointerException</code>	Zabronione stosowanie wartości <code>null</code> w parametrze
<code>IndexOutOfBoundsException</code>	Wartość indeksu poza zakresem
<code>ConcurrentModificationException</code>	Wykryta równoległa modyfikacja stanu obiektu w przypadku, gdy jest ona zabroniona
<code>UnsupportedOperationException</code>	Obiekt nie obsługuje metody

Choć są to najczęściej wykorzystywane wyjątki w bibliotekach platformy Java, to w przypadkach, gdy sytuacja upoważnia do użycia innego wyjątku, należy go zastosować. Na przykład w przypadku tworzenia obiektów matematycznych, takich jak liczby zespolone lub macierze, bardziej właściwe będzie zastosowanie wyjątków `ArihtmeticException` i `NumberFormatException`. Jeżeli wyjątek spełnia Twoje potrzeby — wykorzystaj go, jednak jedynie wtedy, gdy warunki, w których zgłaszasz ten wyjątek, są zgodne z warunkami opisanymi w jego dokumentacji. Wykorzystanie wyjątków musi być uzasadnione semantyką, a nie jedynie nazwą. Gdy chcesz uzupełnić wyjątek o dodatkowe informacje, możesz również dziedziczyć po istniejących wyjątkach (temat 75.), ale pamiętaj, że wyjątki są serializowalne (rozdział 12.). To kolejny powód, dla którego nie warto tworzyć własnych wyjątków, jeśli nie ma ku temu wyraźnych przesłanek.

Musisz wiedzieć, że wybór właściwego wyjątku do wykorzystania nie jest czasami jednoznaczny, a „Zastosowania” z tabeli 10.1 nie zawsze wzajemnie się wykluczają. Pomyśl na przykład o przypadku obiektu reprezentującego talię kart. Załóżmy, że mamy metodę wydającą karty, która jako argument posiada liczbę kart. Załóżmy, że wywołujący tę metodę przekazał wartość parametru większą niż liczba kart pozostałych w talii. Może być to zinterpretowane jako `IllegalArgumentException` (wartość parametru jest zbyt duża) lub `IllegalStateException` (obiekt talia posiada za mało kart do zrealizowania żądania). W tym przypadku wydaje się, że właściwy będzie wyjątek `IllegalArgumentException`, gdyby mógł zadziałać brak wartości. **W przeciwnym razie zgłoś `IllegalStateException`.**

Temat 73. Zgłaszanie wyjątków właściwych dla abstrakcji

Niewłaściwe jest zgłaszanie przez metodę wyjątku, który nie ma widocznego związku z zadaniem, jakie ta metoda wykonuje. Często się to zdarza, gdy metoda propaguje wyjątek zgłoszony przez mechanizmy niskiego poziomu. Oprócz wprowadzania zamieszania powoduje to zaśmieszenie API wyższej warstwy szczegółami implementacji warstwy niższej. Jeżeli w przyszłej wersji implementacja wyższej warstwy zostanie zmieniona, zgłaszane wyjątki również mogą się zmienić, co może spowodować, że programy klientów przestaną działać.

Aby uniknąć tego problemu, **wyższe warstwy abstrakcji powinny przechwytywać wyjątki z warstwy niższej i w ich miejsce zgłaszać wyjątki, które są odpowiednie dla wyższej warstwy**. Idiom ten, nazywany **translacją wyjątków**, wygląda następująco:

```
// Translacja wyjątków
try {
    ... // wykorzystanie operacji niższej warstwy
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Poniżej przedstawiamy przykład translacji wyjątków, zaczerpnięty z klasy `AbstractSequentialList`, która jest **szkieletową implementacją** (temat 20.) interfejsu `List`. W przykładzie tym wykorzystanie translacji wyjątków jest wymuszone przez specyfikację metody `get` z interfejsu `List<E>`.

```
/**
 * Zwraca element z podanej pozycji w tej liście.
 * @throws IndexOutOfBoundsException jeżeli indeks jest poza zakresem
 * (index < 0 || index >= size()).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Indeks: " + index);
    }
}
```

Gdy wyjątki niższego poziomu mogą być przydatne w procesie szukania przyczyny wyjątku, można zastosować odmianę translacji wyjątków, nazywaną **łączeniem wyjątków**. W przypadku tego podejścia wyjątek niższego poziomu (**przyczyna**) jest zapamiętywany przez wyjątek wyższego poziomu, który zapewnia publiczną metodę (`Throwable.getCause`), umożliwiającą pobranie wyjątku niższego poziomu.

```
// Łączenie wyjątków
try {
    ... // wykorzystanie operacji niższej warstwy
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

Konstruktor wyjątku wyższego poziomu przekazuje przyczynę do konstruktora klasy bazowej **obsługującego łączenie**, więc jest ostatecznie przekazywany do jednego z konstruktorów Throwable obsługujących łączenie, takiego jak Throwable(Throwable):

```
// Wyjątki z konstruktorem obsługującym łączenie
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(t);
    }
}
```

Większość ze standardowych wyjątków posiada konstruktor obsługujący łączenie. Dla wyjątków, które ich nie mają, można ustawić przyczynę przy użyciu metody `initCause` z Throwable. Łączenie wyjątków nie tylko pozwala nam na programowy dostęp do przyczyny (za pomocą `getCause`), ale również integruje zapis śladu przyczyny z wyjątkiem wyższego poziomu.

Ponieważ translacja wyjątków doskonale nadaje się do bezmyślnej propagacji wyjątków między warstwami aplikacji, nie powinna być nadużywana. Tam, gdzie jest to możliwe, najlepszą metodą obsługi wyjątków pochodzących z niższych warstw jest ich unikanie, przy upewnieniu się przed wywołaniem metody niższej warstwy, że zostanie ona prawidłowo wykonana. Czasami można to zrealizować przez sprawdzenie poprawności argumentów metody wyższej warstwy przed ich przekazaniem do metody niższej warstwy.

Jeżeli niemożliwe jest zabezpieczenie przed zgłaszaniem wyjątków z niższej warstwy, kolejnym rozwiązaniem jest obejście tego wyjątku w wyższej warstwie, dzięki czemu izoluje się metody wyższej warstwy od problemów, pochodzących z warstwy niższej. W takich przypadkach właściwe jest zapisanie informacji o wystąpieniu wyjątku przy wykorzystaniu mechanizmu rejestrującego, na przykład `java.util.logging`. Pozwala to administratorowi przesłedzić problem bez zakłócania pracy kodu klienta i końcowego użytkownika.

Podsumujmy. Jeżeli zatem niemożliwe jest uniemożliwienie wyjątków z niższych warstw lub ich obsługa, należy użyć tłumaczenia wyjątków, o ile metoda niższego poziomu gwarantuje, że wszystkie jej wyjątki są propagowane do wyższego poziomu. Łączenie zapewnia najlepsze cechy obu metod — pozwala na zgłaszanie odpowiedniego wyjątku wyższego poziomu oraz przechwytywanie przyczyny w celu późniejszej analizy awarii (temat 75.).

Temat 74. Dokumentowanie wyjątków zgłaszanych przez metodę

Ważną częścią dokumentacji, wymaganą do właściwego wykorzystywania metod, jest opis wyjątków zgłaszanych przez metodę. Dlatego niezwykle ważne jest poświęcenie pewnego czasu w celu jasnego udokumentowania wszystkich wyjątków zgłaszanych przez metody (temat 56.).

Zawsze deklaruj (za pomocą znacznika Javadoc `@throws`) **poszczególne przechwytywane wyjątki i opisuj precyzyjnie warunki, w których są one zgłaszane**. Nie ułatwiał sobie zadania pisząc, że metoda zgłasza wyjątek pewnej klasy bazowej, która jest bazą dla wielu klas wyjątków. Jako ekstremalny przykład można przytoczyć deklarowanie, że metoda zgłasza wyjątek `Exception` lub, co gorsza, `Throwable` — jest to niedopuszczalne. Oprócz tego, że deklaracja taka nie daje żadnych wskazówek programiście wykorzystującemu metodę, to jeszcze utrudnia wykorzystanie tej metody, ponieważ zasłania wszystkie wyjątki, jakie mogą być zgłoszone w tym samym kontekście. Jedynym wyjątkiem od tej rady jest metoda `main`, która może być jawnie zadeklarowana jako zgłaszająca wyjątek `Exception`, bo wywołuje ją tylko maszyna wirtualna.

Choć język nie wymaga deklarowania nieprzechwytywanych wyjątków, jakie może zgłaszać metoda, warto udokumentować je tak samo dokładnie, jak wyjątki przechwytywane. Wyjątki nieprzechwytywane z reguły reprezentują błędy programowe (temat 70.) i zapoznanie programistów z nimi pomaga im unikać tych błędów. Dobrze opisana lista wyjątków nieprzechwytywanych, jakie mogą być zgłaszane przez metodę, efektywnie opisuje **warunki wstępne** jej prawidłowego wykonania. Niezmiernie ważne jest, aby każda dokumentacja metody zawierała warunki wstępne (temat 56.), a dokumentowanie wyjątków nieprzechwytywanych jest najlepszym sposobem spełnienia tego wymagania.

Szczególnie ważne jest udokumentowanie nieprzechwytywanych wyjątków, jakie mogą zgłaszać metody zadeklarowane w interfejsie. Dokumentacja ta staje się fragmentem **ogólnych założeń** interfejsu i pozwala na jednakowe działanie różnych implementacji tego interfejsu.

Należy skorzystać ze znacznika Javadoc `@throws` do udokumentowania wszystkich nieprzechwytywanych wyjątków zgłaszanych przez metodę, ale *nie* należy umieszczać tych wyjątków w deklaracji metody po słowie kluczowym `throws`. Ważne jest, aby programista korzystający z API wiedział, które wyjątki są przechwytywane, a które nie, ponieważ jego działania różnią się w tych dwóch przypadkach. Dokumentacja generowana przez znacznik Javadoc `@throws` w przypadku braku nagłówka metody wygenerowanego przez deklarację `throws` zapewnia silny wizualny sygnał pomagający programiście odróżnić przechwytywane wyjątki od nieprzechwytywanych.

Trzeba zauważyć, że dokumentowanie wszystkich nieprzechwytywanych wyjątków zgłaszanych przez metodę jest sytuacją idealną, ale nie zawsze osiągalną w praktyce. Gdy klasa jest modyfikowana, dodanie kolejnych nieprzechwytywanych wyjątków nie jest naruszeniem zgodności źródłowej ani binarnej. Załóżmy, że klasa wywołuje metodę z innej, niezależnie tworzonej klasy. Autorzy pierwszej klasy mogą dokładnie udokumentować wszystkie nieprzechwytywane wyjątki zgłaszane w swojej klasie, ale jeżeli druga klasa została zmodyfikowana i zgłasza nowe wyjątki, najprawdopodobniej pierwsza klasa (niepodlegająca modyfikacji) dokona propagacji nowego nieprzechwytywanego wyjątku, choć nie został on zadeklarowany.

Jeżeli wyjątek jest zgłaszany przez wiele metod klasy przy tych samych warunkach, dopuszczalne jest udokumentowanie takiego wyjątku w komentarzach do klasy zamiast umieszczania go przy każdej metodzie. Przykładem takiego wyjątku jest `NullPointerException`. Można w komentarzu dokumentującym klasę umieścić zdanie podobne do: „Wszystkie metody tej klasy zgłaszają wyjątek `NullPointerException` ↪ `Exception`, jeżeli do dowolnego parametru przekazana jest wartość `null`”.

Podsumujmy. Należy zatem dokumentować każdy wyjątek, który może być zgłaszany przez pisaną metodę. Odnosi się to zarówno do przechwytywanych, jak i nieprzechwytywanych wyjątków oraz zarówno do metod abstrakcyjnych, jak i konkretnych. Dokumentacja powinna przybrać formę znaczników `@throws` w komentarzach dokumentujących. Należy napisać osobne klauzule `throws` dla każdego przechwytywanego wyjątku i nie pisać klauzul `throws` dla wyjątków nieprzechwytywanych. Jeżeli nie udokumentujemy wyjątków zgłaszanych przez naszą metodę, dla innych programistów będzie trudno używać efektywnie naszych klas i interfejsów lub będzie to nawet niemożliwe.

Temat 75. Udostępnianie danych o błędzie

Gdy program zostanie zatrzymany przez nieprzechwycony wyjątek, system automatycznie drukuje informacje ze stosu wyjątków. Stos ten zawiera **reprezentację wyjątku w postaci ciągu znaków** — wynik metody `toString`. Dane te zwykle składają się z nazwy klasy, po której następują **informacje szczegółowe**. Często jest to jedyny fragment informacji, jakie otrzyma programista lub serwis oprogramowania w przypadku wystąpienia błędu w programie. Jeżeli trudno powtórzyć ten błąd, bardzo trudne może być uzyskanie dodatkowych danych. Dlatego niezwykle ważne jest, aby metoda `toString` dla wyjątku zwracała możliwie dużo informacji o przyczynie błędu. Inaczej mówiąc, ciąg reprezentujący wyjątek powinien **umożliwiać dalszą analizę błędu**.

Aby odszukać błąd, ciąg reprezentujący wyjątek powinien zawierać wartości wszystkich parametrów i pól „mających udział w przyczynie wyjątku”. Na przykład w przypadku wyjątku `IndexOutOfBoundsException` ciąg reprezentujący wyjątek powinien zawierać wartość dolnego i górnego zakresu oraz wartość indeksu, który nie zmieścił się w tych granicach. Informacje te niosą wiele informacji na temat przyczyny błędu. Każda z tych wartości może być nieprawidłowa. Aktualna wartość indeksu może być o jeden mniejsza od dolnego zakresu lub równa górnemu zakresowi lub może być to wartość całkowicie spoza zakresu w jedną lub drugą stronę. Dolna granica może być większa od górnej (poważne naruszenie niezmienników). Każda z tych sytuacji wskazuje na inny problem i informacja o tym niezmiennie ułatwia programiście poszukiwanie właściwego błędu.

Ważnym aspektem są tak zwane „dane wrażliwe”. Ponieważ ślady stosu wywołań mogą być widziane przez wiele osób na etapie ich diagnozowania w celu znalezienia i naprawienia oprogramowania, **nie umieszczaj w komunikatach szczegółowych takich informacji jak hasła lub klucze szyfrujące**.

Choć niezmiernie ważne jest dołączanie stosownych „twardych danych” do ciągu reprezentującego wyjątek, zwykle nie warto jest dodawać zbyt wielu informacji. Zrzut stosu przeznaczony jest do analizy kodu źródłowego i zazwyczaj zawiera dokładne nazwy plików i numery wierszy, w których rozpoczynają się wszystkie metody umieszczone na stosie. Długie opisy błędów są zwykle nadmiarowe — większość danych może być zebrana na podstawie analizy kodu źródłowego.

Ciąg reprezentujący wyjątek nie powinien być mylony z komunikatami o błędach zrozumiałymi dla użytkownika. W przeciwieństwie do komunikatów dla użytkowników ciągi te są przeznaczone dla programistów lub serwisu przy analizie awarii. Dlatego dołączone dane są ważniejsze niż czytelność komunikatu. Komunikaty dla użytkowników są często **tłumaczone** na wiele języków, co w przypadku ciągów reprezentujących wyjątek rzadko ma miejsce.

Jednym ze sposobów upewnienia się, że ciąg reprezentujący wyjątek zawiera odpowiednie informacje pomagające odszukać błąd, jest wymaganie tych danych zamiast ciągu reprezentującego wyjątek. Na podstawie podanych informacji komunikat może być generowany automatycznie. Na przykład zamiast konstruktora z parametrem `String` wyjątek `IndexOutOfBoundsException` mógłby mieć konstruktor wyglądający następująco:

```
/**
 * Tworzy IndexOutOfBoundsException.
 *
 * @param lowerBound - najmniejsza dopuszczalna wartość indeksu.
 * @param upperBound - największa dopuszczalna wartość indeksu plus jeden.
 * @param index      - aktualna wartość parametru.
 */
```

```
public IndexOutOfBoundsException(int lowerBound, int upperBound, int
index) {
    // generowanie komunikatu opisującego błąd
    super(String.format(
        "Dolny zakres: %d, Górny zakres: %d, Indeks: %d",
        lowerBound, upperBound, index));

    // zapisanie informacji i błędzie dla programowego dostępu
    this.lowerBound = lowerBound;
    this.upperBound = upperBound;
    this.index = index;
}
```

Od Javy 9 wyjątek `IndexOutOfBoundsException` nareszcie zyskał konstruktor przyjmujący parametr `index` typu `int`, ale niestety pominięto parametry `lowerBound` i `upperBound`. Biblioteki platformy Java nie korzystają z tego idiomu zbyt często, ale polecamy gorąco jego stosowanie. Ułatwia to programiście odszukanie błędu w przypadku zgłoszenia wyjątku, a właściwie trudno jest w takiej sytuacji nie znaleźć błędu. W efekcie stosowania tego idiomu generowanie wysokiej klasy ciągu reprezentującego wyjątek jest przeniesione do klasy wyjątku, nie powodując nadmiarowego generowania takiego ciągu przez wszystkich programistów, korzystających z klasy.

W temacie 70. sugerowaliśmy, że właściwe jest udostępnienie metod w klasie wyjątku, pozwalających na odczytanie danych zapamiętanych w czasie generowania wyjątku (`lowerBound`, `upperBound` i `index` z powyższego przykładu). Metody takie mają większy sens w przypadku wyjątków przechwytywanych niż w przypadku nieprzechwytywanych, ponieważ w trakcie obsługi wyjątku przechwytywanego dane te mogą zostać wykorzystane do przywrócenia normalnego działania. Rzadko zdarza się (choć nie jest to niemożliwe), że programista będzie potrzebował dostępu do szczegółów wyjątku nieprzechwytywanego. Nawet w przypadku wyjątków nieprzechwytywanych jako ogólną zasadę można jednak zalecić tworzenie takich metod (temat 12.).

Temat 76. Zachowanie atomowości w przypadku błędu

Po zgłoszeniu wyjątku przez obiekt zwykle pożądanym jest, aby obiekt cały czas był w dobrze zdefiniowanym stanie, nawet w przypadku wystąpienia błędu w czasie wykonywania operacji. Jest to szczególnie ważne w przypadku wyjątków przechwytywanych, które wywołujący może obsługiwać. **Nieudane wykonanie metody nie powinno powodować zmiany stanu obiektu przed wywołania tej metody.** Metoda o tej własności nazywana jest **atomową w przypadku błędu**.

Istnieje kilka sposobów na osiągnięcie tego efektu. Najprostszym jest projektowanie obiektów niezmiennych (temat 17.). Jeżeli obiekt jest niezmienny, atomowość w przypadku błędu jest zachowana bez żadnych nakładów. Jeżeli wykonanie operacji się nie powiedzie, może to spowodować, że nie zostanie utworzony wynikowy obiekt, lecz nigdy nie spowoduje powstania nieustalonego stanu istniejącego obiektu, ponieważ stan każdego z obiektów jest ustalany w trakcie tworzenia i nie może być później modyfikowany.

W przypadku metod operujących na obiektach modyfikowalnych najczęstszą metodą osiągnięcia atomowości w przypadku błędu jest sprawdzanie poprawności parametrów przed wykonaniem operacji (temat 49.). Dzięki temu wyjątki są zgłaszane przed rozpoczęciem modyfikacji obiektu. Dla przykładu przeanalizujmy metodę `Stack.pop` z tematu 7.

```
public Object pop() {  
    if (size == 0)  
        throw new EmptyStackException();  
    Object result = elements[--size];  
    elements[size] = null; //usuwanie zbędnych referencji  
    return result;  
}
```

Jeżeli usuniemy początkowe sprawdzanie rozmiaru stosu, metoda nadal będzie zgłaszała wyjątek w przypadku próby pobrania elementu z pustego stosu. Jednak będzie pozostawiała pole `size` w niespójnym (ujemnym) stanie, powodując, że wszystkie kolejne wywołania metody zakończą się błędem. Dodatkowo wyjątek zgłaszany przez metodę `pop` może być niezgodny z abstrakcją (temat 73.).

Podobnym podejściem przy uzyskiwaniu atomowości w przypadku błędu jest porządkowanie obliczeń w taki sposób, aby wszystkie fragmenty, w których mogą wystąpić błędy, były wykonywane przed zmianą stanu obiektu. Podejście to jest naturalnym rozszerzeniem poprzedniego, gdy argumenty nie mogą być sprawdzone bez wykonania fragmentów operacji. Na przykład w przypadku `TreeMap` elementy są sortowane według określonego porządku. Aby dodać element do `TreeMap`, element ten musi mieć typ porównywalny przy użyciu porządkowania, wykorzystanego w `TreeMap`. Próba dodania elementu niewłaściwego typu spowoduje zgłoszenie wyjątku `ClassCastException` w wyniku operacji wyszukiwania elementu w drzewie, zanim drzewo zostanie w jakikolwiek sposób zmienione.

Trzecim podejściem przy uzyskiwaniu atomowości w przypadku błędu jest wykonywanie operacji na kopii obiektu i zamiana zawartości obiektu źródłowego zawartością kopii po udanym zakończeniu operacji. Podejście to jest naturalne w przypadku, gdy obliczenia na tymczasowych strukturach danych mogą być przeprowadzone z dużą szybkością. Na przykład niektóre funkcje sortujące kopiują wejściową listę do tablicy przed rozpoczęciem sortowania w celu zmniejszenia kosztu dostępu

do elementów w wewnętrznej pętli sortowania. Jest to zrobione w celu poprawienia wydajności, ale dodatkowym zyskiem jest to, że w przypadku wystąpienia błędu wejściowa lista będzie niezmienniona.

Ostatnim, najradszym podejściem jest tworzenie **kodu odtwarzającego**, który wykrywa błędy w czasie operacji i powoduje wycofanie obiektu do stanu sprzed rozpoczęcia operacji. Podejście to jest najczęściej używane w przypadku trwałych struktur danych.

Choć atomowość w przypadku błędu jest korzystna, nie zawsze może być osiągnięta. Na przykład jeżeli dwa wątki próbują równolegle zmodyfikować obiekt bez właściwej synchronizacji, może być on pozostawiony w niewłaściwym stanie. Dlatego niewłaściwe może być założenie, że obiekt będzie nadawał się do użytku po przechwyceniu wyjątku `ConcurrentModificationException`. W przypadku błędów (w przeciwieństwie do wyjątków) nie jest możliwe odtworzenie stanu obiektu, więc metody nie muszą próbować osiągnięcia atomowości w przypadku zgłoszenia błędu.

Nawet, gdy atomowość w przypadku błędu jest możliwa do osiągnięcia, nie zawsze jest pożądana. W przypadku niektórych operacji będzie ona znacznie zwiększała koszt i stopień skomplikowania kodu. Jednak zwracając uwagę na to zagadnienie, przekonasz się, że najczęściej jest ona łatwa do osiągnięcia.

Podsumujmy. Jako zasadę należy przyjąć, że wyjątki będące częścią specyfikacji metody nie powinny zmieniać stanu obiektu sprzed wywołania tej metody. Gdy zasada ta nie jest spełniona, dokumentacja API powinna jasno wskazywać, w jakim stanie są pozostawiane obiekty. Niestety, wiele istniejących dokumentacji API nie spełnia tego warunku.

Temat 77. Nie ignoruj wyjątków

Uwaga ta może się wydawać oczywista, jednak jest tak często naruszana, że wymaga ciągłego powtarzania. Gdy projektanci API deklarują, że metoda zgłasza wyjątek, chcą przez to coś przekazać. Nie należy tego ignorować! Łatwo zignorować wyjątek, umieszczając wywołanie metody wewnątrz instrukcji try z pustym blokiem catch:

```
// Pusty blok catch powoduje zignorowanie wyjątku - wysoce podejrzane!
try {
    ...
} catch (SomeException e) {
}
```

Pusty blok catch powoduje zignorowanie przeznaczenia wyjątków, którym jest wymuszenie obsługi sytuacji wyjątkowej. Zignorowanie wyjątku jest analogiczne do zignorowania alarmu przeciwpożarowego i jego wyłączanie, przez co nikt nie będzie miał szansy na zaalarmowanie o prawdziwym zagrożeniu. Gdy zobaczysz pusty blok catch, powinienes natychmiast podjąć środki zaradcze.

Istnieją sytuacje, gdy zignorowanie wyjątku będzie odpowiednie. Przykładowo, wyjątek może pojawić się w momencie zamykania pliku reprezentowanego przez `FileInputStream`. Ponieważ nie zmienialiśmy stanu pliku, nie musimy podejmować żadnych akcji naprawczych, a ponieważ odczytaliśmy już zawartość pliku, nie musimy niczego ponawiać. Warto byłoby jednak w takiej sytuacji zapamiętać wyjątek w dzienniku zdarzeń, aby móc bliżej przyjrzeć się sprawie, jeśli podobnych wyjątków jest dużo. **Jeśli zdecydujemy się wyjątek zignorować, blok catch powinien zawierać co najmniej komentarz, wyjaśniający, dlaczego wyjątek ten jest ignorowany.** Zmienna powinna nosić nazwę `ignored`:

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // domyślnie; gwarantowana odpowiedniość dla dowolnej mapy
try {
    numColors = f.get(1L, TimeUnit.SECONDS);
} catch (TimeoutException | ExecutionException ignored) {
    // użyj wartości domyślnej; minimalne kolorowanie jest pożądane, ale nie wymagane
}
```

Porada z tego tematu odnosi się zarówno do wyjątków przechwytywanych, jak i nieprzechwytywanych. Ponieważ wyjątki reprezentują przewidywalne sytuacje wyjątkowe lub błędy w programie, ich zignorowanie spowoduje, że program będzie kontynuował działanie bez żadnego komunikatu. Program ten może zatrzymać się w dowolnym czasie w przyszłości w takim miejscu kodu, które nie jest w żaden sposób związane ze źródłem problemu. Właściwa obsługa błędów może całkowicie zapobiec awarii. Zwykle pozostawienie nieprzechwytywanego wyjątku powoduje jego propagację i zatrzymanie jego działania, zachowując dane potrzebne do odszukiwania źródła błędu.

Współbieżność

Wątki umożliwiają równoległe wykonywanie różnych czynności w jednym programie. Programowanie wielowątkowe jest dużo trudniejsze od tradycyjnego programowania jednowątkowego, ponieważ znacznie więcej rzeczy może się nie udać, a błędy są trudniejsze do powtórzenia. Jednak nie można uniknąć współbieżności. Jest ona naturalna dla wielu wykonywanych operacji i jest wymagana, jeżeli chcemy uzyskać dobrą wydajność na procesorach wielordzeniowych, które są teraz powszechnie wykorzystywane. Rozdział ten zawiera porady, które pomogą Ci stworzyć jasne, prawidłowe i dobrze udokumentowane programy współbieżne.

Temat 78. Synchronizacja dostępu do wspólnych modyfikowalnych danych

Dzięki zastosowaniu słowa kluczowego `synchronized` tylko jeden wątek jednocześnie ma dostęp do instrukcji lub bloku. Wielu programistów uważa, że synchronizacja polega jedynie na **wzajemnym wykluczaniu** lub zabezpieczeniu obiektu przed odczytaniem niespójnego stanu przez jeden wątek w czasie jego modyfikacji przez drugi wątek. W tym podejściu obiekt jest tworzony w stanie spójnym (temat 17.) i blokowany przez metody, które mają do niego dostęp. Metody te pozwalają na odczytanie stanu i mogą powodować **przejście między stanami**, przenosząc obiekt z jednego spójnego stanu do drugiego. Właściwe wykorzystanie synchronizacji gwarantuje, że żadna z metod nie może odczytać wartości z obiektu będącego w stanie niespójnym.

Podejście to jest prawidłowe, ale nie wyczerpuje wszystkich możliwości. Bez synchronizacji zmiany z jednego wątku mogą być niewidoczne dla innych wątków. Synchronizacja nie tylko zabezpiecza przed odczytaniem niespójnego stanu obiektu,

ale również zapewnia, że każdy wątek wchodzący do bloku lub metody synchronizowanej „widzi” efekty wszystkich poprzednich modyfikacji chronionych przez tę samą blokadę.

Mechanizmy języka gwarantują, że operacja odczytania i zapisania zmiennej jest **atomowa**, chyba że zmienna jest typu `long` lub `double` [JLS, 17.4 i 17.7]. Oznacza to, że w przypadku odczytywania zmiennej typu innego niż `long` i `double` gwarantowane jest, że zwrócona wartość będzie równa wartości zapisanej przez jakiś wątek, nawet jeżeli wiele wątków jednocześnie modyfikuje tę zmienną bez stosowania żadnego mechanizmu synchronizacji.

Możesz spotkać się z opiniami, że w celu poprawienia wydajności powinienś unikać synchronizacji w czasie odczytywania lub zapisywania danych atomowych. Porada ta jest bardzo niebezpieczna. Choć atomowość gwarantuje, że wątek nie otrzyma przypadkowej wartości w czasie odczytu atomowej wartości, nie gwarantuje tego, że wartość zapisana przez jeden wątek będzie widoczna dla innego wątku. **Dla zapewnienia niezawodnej komunikacji pomiędzy wątkami niezbędna jest zarówno synchronizacja, jak i wzajemne wykluczanie.** Jest to konsekwencja stosowania przyjętych w języku Java technicznych rozwiązań, zwanych **modelem pamięci**, który definiuje, kiedy i jak zmiany z jednego wątku są widoczne dla innych [JLS, 17.4, Goetz06, 16].

Konsekwencje braku zapewnienia synchronizacji dostępu do zmiennej współdzielonej mogą być poważne, nawet jeżeli zmienna ta może być odczytywana i zapisywana w sposób atomowy. Weźmy pod uwagę zadanie zatrzymywania jednego wątku z drugiego. W bibliotece dostępna jest metoda `Thread.stop`, ale została ona już dawno uznana za przestarzałą, ponieważ jest z natury **mało bezpieczna** — jej zastosowanie może spowodować uszkodzenie danych. **Nie należy używać `Thread.stop`.** Zalecanym sposobem na zatrzymanie jednego wątku z innego jest udostępnienie pola `boolean` o wartości początkowej `false`, które jest regularnie kontrolowane przez wątek i które może być ustawione na `true` przez inny wątek, co oznacza, że pierwszy wątek sam zatrzyma swoje działanie. Ponieważ odczyt i zapis pola `boolean` jest atomowy, niektórzy programiści mogą pominąć synchronizację przy dostępie do tego pola:

```
// Żle! - Jak dugo będzie działał ten program?
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();
```

```

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

Można oczekiwać, że program będzie działał około sekundy, po czym główny wątek ustawi pole `stopRequest` na `true`, co spowoduje, że pętla w wątku `tła` zakończy się. Jednak na moim komputerze program **nigdy** się nie kończy — pętla w wątku `tła` działa bez przerwy!

Problem polega na tym, że w przypadku braku synchronizacji nie ma gwarancji tego, kiedy i czy w ogóle wątek `tła` zauważy zmiany wartości `stopRequest`, wprowadzone przez główny wątek. W przypadku braku synchronizacji można zaakceptować zmianę przez maszynę wirtualną tego kodu:

```

while (!done)
    i++;

```

w następujący:

```

if (!done)
    while (true)
        i++;

```

Taka optymalizacja jest nazywana **wyciąganiem** i jest realizowana przez maszynę wirtualną OpenJDK Server. Wynikiem jest **awaria żywotności** — program nie wykonuje żadnego postępu. Jedynym sposobem poprawienia tego błędu jest synchronizacja dostępu do pola `stopRequest`. Program, tak jak oczekiwaliśmy, zakończy działanie po mniej więcej sekundzie:

```

// Prawidłowo zsynchronizowany mechanizm zakończenia wątku
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested())
                i++;
        });
        backgroundThread.start();
    }
}

```

```

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}

```

Warto zauważyć, że zarówno metoda zapisująca (`requestStop`), jak i odczytująca (`stopRequested`) są synchronizowane. **Nie** wystarczy synchronizować tylko metody zapisującej! **W rzeczywistości synchronizacja nie daje efektu, jeżeli nie są jej poddawane zarówno operacje odczytu, jak i zapisu.** Zdarza się, że na niektórych komputerach program, który synchronizuje tylko zapisy (lub tylko odczyty), **wydaje się** działać poprawnie, ale nie należy na tym polegać.

Akcje w metodach synchronizowanych w `StopThread` byłyby atomowe nawet bez synchronizacji. Inaczej mówiąc, synchronizacja tych metod jest używana **wyłącznie** dla zapewnienia komunikacji, a nie wzajemnego wykluczania. Choć koszt synchronizacji w każdej iteracji pętli jest niewielki, istnieje prawidłowa alternatywa, która jest bardziej zwięzła i której wydajność jest zwykle lepsza. Blokowanie w drugiej wersji `StopThread` może być pominięte, jeżeli pole `stopRequest` zostanie zadeklarowane jako `volatile`. Modyfikator `volatile` nie zapewnia wzajemnego wykluczania, ale gwarantuje, że każdy wątek odczytujący to pole będzie „widział” ostatnio zapisaną wartość:

```

// Mechanizm kończenia wątku z polem volatile
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

Podczas korzystania z `volatile` należy zachować ostrożność. Spójrzmy na fragment realizujący generowanie numeru seryjnego:

```

// Błąd - wymaga synchronizacji!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}

```


Założeniem tego fragmentu programu jest zagwarantowanie, że każde wywołanie metody `generateSerialNumber` zwróci inny numer seryjny, dopóki nie zostanie ona wywołana 2^{32} razy. Do ochrony niezmienników tego generatora numerów seryjnych nie jest wymagana synchronizacja, ponieważ niezmienników tych nie ma — jego stan składa się z jednego pola, którego zapis jest operacją atomową (`nextSerialNumber`), a wszystkie wartości tego pola są dopuszczalne. Jednak metoda ta nie może działać bez synchronizacji.

Operator zwiększania (`++`) odczytuje i zapisuje pole `nextSerialNumber`, więc nie jest on atomowy. Wykonuje **dwie** operacje — odczytu i zapisu — które są niezależne od siebie i zachodzą jedna po drugiej. Kilka równoległych wątków może odczytać tę samą wartość pola `nextSerialNumber`, zwiększyć ją, zapisać i zwrócić ten sam numer seryjny, ponieważ operacje odczytu zajądą w tym samym czasie. Jest to **naruśnienie bezpieczeństwa** — program oblicza nieprawidłowe wyniki.

Poprawienie metody `generateSerialNumber` jest bardzo proste — wystarczy dodać do deklaracji metody modyfikator `synchronized`. Zapewni to, że równoległe wywołania nie będą na siebie nachodzić i każde wywołanie będzie widziało wyniki działania z poprzedniego wywołania. Następnie można i powinno się usunąć z `nextSerialNumber` modyfikator `volatile`. Aby dodatkowo zabezpieczyć tę metodę, należy zmienić typ pola na `long` lub zgłaszać wyjątek, gdy wartość pola `nextSerialNumber` osiągnie maksymalną wartość.

Jeszcze lepiej skorzystać z porady z tematu 59. i użyć klasy `AtomicLong`, wchodzącej w skład pakietu `java.util.concurrent.atomic`. Pakiet zawiera podstawowe mechanizmy do tworzenia kodu bezpiecznego wątkowo i bez blokad, który używa pojedynczych zmiennych. Choć zmienne `volatile` zapewniają informowanie o synchronizacji, ten pakiet zapewnia również atomowość. Klasa wykonuje dokładnie to, co chcemy, i najprawdopodobniej będzie pozwalała osiągnąć lepszą wydajność niż zsynchronizowana wersja `generateSerialNumber`:

```
// Synchronizacja bez blokad dzięki pakietowi java.util.concurrent.atomic
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

Najlepszym sposobem unikania problemów przedstawionych w tym temacie jest zapewnienie braku współdzielenia modyfikowalnych danych. Należy współdzielić dane niemodyfikowalne (temat 17.) lub ich nie współdzielić. Inaczej mówiąc, należy **zamknąć modyfikowalne dane w jednym wątku**. Jeżeli będziemy stosować tę zasadę, konieczne należy to udokumentować, aby była zachowana w czasie rozwoju aplikacji. Konieczne jest również dokładne zapoznanie się z używanym frameworkiem i bibliotekami, ponieważ mogą one wprowadzać wątki, o których nie będziemy wiedzieli.

Akceptowalne jest modyfikowanie danych obiektu przez pewien czas, a następnie ich udostępnianie innym wątkom, stosując synchronizację tylko momentu udostępnienia referencji obiektu. Inne wątki mogą odczytywać ten obiekt bez dalszej synchronizacji, do momentu jego powtórnej synchronizacji. Takie obiekty są określane jako **efektywnie niezmiennie** [Goetz06, 3.5.4]. Przeniesienie referencji takiego obiektu z jednego wątku do innego jest nazywane **bezpieczną publikacją** [Goetz06, 3.5.3]. Istnieje wiele sposobów na bezpieczną publikację referencji obiektu — można przechować ją w polu statycznym w czasie inicjalizacji klasy; można ją umieścić w polu `volatile`, `final` lub polu udostępnianym przy użyciu normalnego blokowania; można również umieścić ją w kolekcji współbieżnej (temat 81.).

Podsumujmy. **Jeżeli kilka wątków korzysta z modyfikowalnych danych, każdy wątek odczytujący i zapisujący dane musi korzystać z blokowania.** Bez synchronizacji nie ma gwarancji co do tego, który wątek zauważy zmiany wprowadzone przez inny wątek. W wyniku niezynchronizowanego dostępu do danych mogą wystąpić błędy w systemie bezpieczeństwa i zmniejszenie stabilności programu. Takie awarie mogą być niezwykle trudne do powtórzenia. Mogą one być warunkowane zależnościami czasowymi i szczegółami implementacji maszyny wirtualnej oraz typem komputera, na którym program jest uruchamiany. W przypadku konieczności zapewnienia wyłącznie komunikacji między wątkami, a nie wzajemnego wykluczania, wykorzystanie modyfikatora `volatile` w niektórych przypadkach stanowi alternatywę dla zwykłej synchronizacji, ale jest to zaawansowana technika.

Temat 79. Unikanie nadmiarowej synchronizacji

W temacie 78. ostrzegaliśmy przed niebezpieczeństwami niedostatecznej synchronizacji. W tym temacie omówimy przeciwny problem. W zależności od sytuacji nadmiarowa synchronizacja może spowodować obniżenie wydajności, zakleszczenie, a nawet niedeterministyczne działanie.

Aby uniknąć ryzyka awarii żywotności lub błędu bezpieczeństwa, nigdy nie oddawaj klientowi kontroli w synchronizowanym bloku lub metodzie. Inaczej mówiąc, wewnątrz synchronizowanego obszaru nie wywołuj publicznych ani zabezpieczonych metod, które są przeznaczone do przesłaniania, lub dostarczonych przez klienta metod umieszczonych w obiekcie funkcyjnym (temat 24.). Z perspektywy klasy zawierającej synchronizowany blok metoda taka jest obca. Klasa „nie wie”, co metoda robi, i nie ma nad nią kontroli. W zależności od tego, co wykonuje metoda obca, wywołanie jej z regionu synchronizowanego może spowodować wystąpienie wyjątku, zakleszczenia lub uszkodzenia danych.

Aby pokazać konkretny przykład, przeanalizujemy poniższą klasę, która implementuje klasę opakującą **obserwowany** zbiór. Pozwala ona klientom subskrybować powiadomienia wywoływane po dodaniu elementów do zbioru. Jest to wzorzec *Observer* (obserwator) [Gamma95]. Przykład jest skrócony — klasa nie zapewnia powiadomień w czasie usuwania elementów ze zbioru, ale ich dodanie jest bardzo proste. Klasa ta jest zaimplementowana na bazie *ForwardingSet* z tematu 18.:

// Źle - wywołanie metody obcej z bloku synchronizowanego!

```
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers = new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }

    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }

    @Override public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element); // wywołuje notifyElementAdded
        return result;
    }
}
```

Obserwatory subskrybują powiadomienia przez wywoływanie metody `addObserver` i usuwają subskrypcję przez wywołanie metody `removeObserver`. W obu przypadkach do metody jest przekazywany obiekt tego interfejsu **wywołania zwrotnego**:

```
@FunctionalInterface public interface SetObserver<E> {
    // wywoływana, gdy element jest dodawany do obserwowanego zbioru
    void added(ObservableSet<E> set, E element);
}
```

Ten interfejs jest strukturalnie identyczny z interfejsem `BiConsumer<ObservableSet<E>, E>`. Zdecydowaliśmy się utworzyć własny interfejs funkcyjny, ponieważ interfejs i nazwy metod czynią kod bardziej czytelnym, a także dlatego, że interfejs mógłby w przyszłości rozrosnąć się o wiele wywołań zwrotnych. Z drugiej strony można również przedstawić argumenty przemawiające za użyciem `BiConsumer` (temat 44.).

Po szybkim sprawdzeniu wydaje się, że klasa `ObservableSet` działa prawidłowo. Na przykład poniższy program wyświetla liczby od 0 do 99:

```
public static void main(String[] args) {
    ObservableSet<Integer> set =
        new ObservableSet<>(new HashSet<>());

    set.addObserver((s, e) -> System.out.println(e));

    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

Spróbujmy teraz czegoś bardziej skomplikowanego. Załóżmy, że chcemy zamienić wywołanie `addObserver` na takie, w którym przekazujemy obserwatora wyświetlającego wartość `Integer` dodaną do zbioru i usuwającego się, jeżeli przekazaną wartością jest 23:

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});
```

Zwróć uwagę, że w odróżnieniu od wcześniejszego wywołania, które korzystało z lambdy, tym razem pojawia się anonimowa instancja klasy. Wynika to z faktu, iż obiekt funkcyjny musi przekazać samego siebie do `s.removeObserver`, a lambda nie mają dostępu do samych siebie (temat 42.).

Można oczekiwać, że program wyświetli liczby od 0 do 23, po czym obserwator się wyrejestruje i program zakończy swoje działanie. W rzeczywistości program wyświetla te liczby, po czym zgłasza wyjątek `ConcurrentModificationException`. Problem polega na tym, że `notifyElementAdded` w czasie wywoływania metody `added` obserwatora realizuje proces przeglądania listy `observers`. Metoda `added` wywołuje metodę `removeObserver` obserwowanego zbioru, co z kolei powoduje

wywołanie `observers.remove`. Tu pojawiają się kłopoty. Próbujemy usunąć element z listy w czasie jej przeglądania, co jest nieprawidłowe. Przeglądanie w metodzie `notifyElementAdded` jest umieszczone w bloku synchronizowanym w celu uniemożliwienia współbieżnej modyfikacji, ale nie uniemożliwiło to wątkowi przeglądającemu wywołania zwrotnego i modyfikowania listy `observers`.

Spróbujmy teraz zrobić coś dziwnego. Napiszmy obserwatora, który próbuje się usunąć z subskrypcji, ale zamiast bezpośredniego wywołania `removeObserver` wykorzystuje do tego celu usługę z innego wątku. Ten obserwator wykorzystuje **usługę wykonawcy** (temat 80.):

```
// Obserwator wykorzystujący niepotrzebnie wątek tła
set.addObserver(new SetObserver<Integer>() {
    public void added(final ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService exec =
                Executors.newSingleThreadExecutor();
            try {
                exec.submit(() -> s.removeObserver(this)).get();
            } catch (ExecutionException | InterruptedException ex) {
                throw new AssertionError(ex);
            } finally {
                exec.shutdown();
            }
        }
    }
});
```

Zwróć uwagę, że program wyłapuje dwa różne rodzaje wyjątków w jednym bloku `catch`. Funkcjonalność ta, nazywana nieoficjalnie **wieloelementowym blokiem catch**, została dodana w Javie 7. Dzięki niej znacząco poprawia się przejrzystość i rozmiar programów, które muszą w ten sam sposób reagować na kilka różnych typów wyjątków.

Tym razem nie pojawi się wyjątek, ale zakleszczenie. Wątek tła wywołuje `s.removeObserver`, co powoduje próbę zablokowania kolekcji `observers`, jednak nie można uzyskać blokady, ponieważ jest ona utrzymywana przez główny wątek. W tym samym czasie główny wątek oczekuje, aby wątek tła zakończył usuwanie obserwatora, co powoduje zakleszczenie.

Przykład ten jest sztuczny, ponieważ nie ma powodu, aby obserwator korzystał z wątku tła, ale problem jest rzeczywisty. Wywoływanie metod obcych z regionów synchronizowanych powoduje w rzeczywistych systemach, takich jak systemy GUI, wiele zakleszczeń.

W obu poprzednich przykładach (wyjątek i zakleszczenie) mieliśmy szczęście. Zasób chroniony przez region synchronizowany (`observers`) był w stanie spójnym w momencie wywołania metody obcej (`added`). Załóżmy, że wywołujemy metodę

obcą z regionu synchronizowanego w czasie, gdy niezmiennik chroniony przez region synchronizowany tymczasowo nie był spełniony. Ponieważ blokady w języku Java są **współużywalne**, takie wywołania nie spowodują zakleszczenia. W pierwszym przykładzie, w którym był zgłaszany wyjątek, wątek wywołujący nałożył już blokadę, więc kolejny wątek również może ją nałożyć, nawet pomimo tego, że wykonywane są inne, koncepcyjnie niezwiązane operacje, przetwarzające dane chronione przez blokadę. Konsekwencje takiej awarii mogą być katastrofalne. W rzeczywistości blokada praktycznie nie zadziałała. Blokadę współużywalne upraszczają tworzenie wielowątkowych programów obiektowych, ale mogą one zmienić awarie żywotności w awarie bezpieczeństwa.

Na szczęście najczęściej łatwo poprawić takie błędy przez przeniesienie wywołania metody obcej poza blok synchronizowany. Dla metody `notifyElementAdded` wymaga to wykonania „migawki” listy `observers`, która może być bezpiecznie przeglądana bez konieczności nakładania blokady. Po tej zmianie oba poprzednie przykłady uruchamiają się bez wyjątków i zakleszczeń:

```
// Metoda obca przeniesiona poza blok synchronizowany - otwarte wywołanie
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<>(observers);
    }
    for (SetObserver<E> observer : snapshot)
        observer.added(this, element);
}
```

Istnieje jeszcze lepszy sposób na przeniesienie wywołania metody obcej poza blok synchronizowany. Biblioteka Java zawiera **kolekcję współbieżną** (temat 81.) o nazwie `CopyOnWriteArrayList`; kolekcja ta świetnie nadaje się do tych zastosowań. Jest to odmiana `ArrayList`, w której wszystkie operacje zapisu są implementowane przez wykonanie nowej kopii całej bazowej tablicy. Ponieważ wewnętrzna tablica nigdy nie jest modyfikowana, przeglądanie nie wymaga blokowania i jest bardzo szybkie. Dla większości zastosowań wydajność `CopyOnWriteList` będzie fatalna, ale doskonale nadaje się dla list obserwatorów, które są rzadko modyfikowane i często przeglądane.

Jeżeli zastosujemy `CopyOnWriteArrayList`, metody `add` oraz `addAll` z `ObservableSet` nie muszą być zmieniane. Odpowiedni fragment klasy wygląda teraz następująco (należy zauważyć, że nie jest tu jawnie używana jakakolwiek synchronizacja):

```
// Bezpieczny dla wątków zbiór obserwowany używający CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
    new CopyOnWriteArrayList<>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}
```

```
public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

Wywołanie metody obcej spoza bloku synchronizowanego nazywane jest **wywołaniem otwartym** [Goetz06, 10.1.4]. Oprócz zapobiegania zakleszczeniom wywołania otwarte znacznie zwiększają współbieżność. Metoda obca może wykonywać się przez dowolnie długi czas, a w przypadku zastosowania wywołania metody obcej z bloku synchronizowanego przez ten czas inne wątki niepotrzebnie miały blokowany dostęp do współdzielonego obiektu.

Jako zasadę należy przyjąć, że wewnątrz bloku synchronizowanego wykonuje się możliwie niewiele operacji. Nałożenie blokady, odczytanie współdzielonych danych, w razie potrzeby ich modyfikacja i zwolnienie blokady — tak powinny wyglądać operacje w bloku synchronizowanym. Jeżeli musisz wykonać czasochłonne obliczenia, spróbuj przenieść je poza blok synchronizowany bez naruszania zasad z tematu 78.

Pierwsza część tego tematu omawiała problemy ze współbieżnością. Teraz skupimy się na wydajności. Mimo że koszt synchronizacji zmniejszył się znacznie od pierwszych wersji platformy Java, szczególnie ważne jest unikanie nadmiernej synchronizacji. Przy stosowanych obecnie procesorach wielordzeniowych prawdziwym kosztem nie jest czas procesora przeznaczony na uzyskanie blokady — są nim stracone możliwości zapewnienia współbieżności i opóźnienia powodowane przez konieczność upewnienia się, że każdy rdzeń ma spójny obraz zawartości pamięci. Innym ukrytym kosztem nadmiernej synchronizacji jest brak możliwości optymalizacji wykonywanego kodu przez maszynę wirtualną.

Jeżeli klasa umożliwia modyfikowanie, mamy dwie opcje: pominąć synchronizację i pozwolić, aby klienci zapewniały synchronizację wszędzie tam, gdzie jest to właściwe, albo zapewnić klasie **bezpieczeństwo dla wątków** (temat 82.). Wybierz to drugie rozwiązanie, jeśli pozwala na osiągnięcie znacznie wyższej współbieżności przez wewnętrzne synchronizowanie, niż daje się ono osiągnąć przez blokowanie całych obiektów z zewnątrz przez klienty. Kolekcje z `java.util` (poza wycofanymi z użycia klasami `Vector` i `Hashtable`) stosują pierwsze podejście, a kolekcje z `java.util.concurrent` stosują drugie podejście (temat 81.).

W początkach rozwoju platformy Java wiele klas nie spełniało tych zasad. Na przykład obiekty `StringBuffer` niemal zawsze były używane przez jeden wątek, ale wewnętrznie korzystały z synchronizacji. Z tego powodu klasa `StringBuffer` została zastąpiona przez `StringBuilder`, która jest niesynchronizowaną wersją `StringBuffer`.

Z bardzo podobnych powodów synchronizowany generator liczb pseudolosowych `java.util.Random` został wyparty przez niesynchronizowaną implementację `java.util.concurrent.ThreadLocalRandom`. Jeżeli mamy wątpliwości, **nie** powinniśmy synchronizować naszych klas, ale udokumentować, że nie są bezpieczne dla wątków.

Jeżeli w klasie korzystamy z synchronizacji, możemy użyć wielu różnych technik osiągnięcia wysokiej współbieżności, takich jak rozdzielanie blokad, zdejmowanie blokad oraz sterowanie współbieżnością bez blokad. Techniki te wykraczają poza zakres tej książki, więc odsyłamy do innych źródeł [Goetz06, Herlihy08].

Jeżeli metoda modyfikuje pole statyczne, **konieczne** jest wewnętrzne synchronizowanie dostępu do tego pola, nawet gdy istnieje choć cień szansy na wywoływanie metody z wielu wątków (wyjątkiem jest sytuacja, w której klasa potrafi tolerować niedeterministyczne zachowanie). Niemożliwe jest wykonanie zewnętrznej synchronizacji takich metod, ponieważ nie można zagwarantować, że w niezwiązanym kodzie klienta również będzie ona wykonana. Pole jest w zasadzie globalną zmienną, nawet jeśli jest prywatne, bo może być odczytywane i zapisywane przez niepowiązane klienty. Przykładem takiej sytuacji jest pole `nextSerialNumber` używane przez metodę `generateSerialNumber` z tematu 78.

Podsumujmy. W celu uniknięcia zakleszczeń i uszkodzenia danych nigdy nie wywołuj metody obcej z bloku synchronizowanego. Mówiąc ogólniej, próbuj ograniczyć liczbę operacji w synchronizowanym bloku. Gdy projektujesz klasę modyfikowalną, pomyśl, czy będzie ona potrzebowała własnej synchronizacji. W erze procesorów wielordzeniowych ważne jest, aby nie nadużywać synchronizacji. Wewnętrzną synchronizację należy stosować tylko w przypadkach, w których istnieje ku temu powody, i należy jasno udokumentować swoją decyzję (temat 82.).

Temat 80. Stosowanie wykonawców, zadań i strumieni zamiast wątków

W pierwszym wydaniu tej książki zamieszczony był kod prostej **kolejki zadań** [Bloch01, temat 49.]. Klasa ta pozwalała klientom na kolejkovanie zadań do asynchronicznego wykonania przez wątek tła. Gdy kolejka robocza nie była już potrzebna, klient mógł wykonać metodę, przekazującą żądanie bezpiecznego zakończenia pracy przez wątek tła — po zakończeniu wszystkich zadań znajdujących się w kolejce. Implementacja ta była tylko nieco bardziej zaawansowana od zwykłego ćwiczenia, ale pomimo to wymagała zastosowania całej strony kodu, który był narażony na problemy z działaniem i bezpieczeństwem, jeżeli nie został odpowiednio wykonany. Na szczęście nie ma powodu, aby po raz kolejny pisać ten kod.

Przed drugim wydaniem książki do platformy Java został dodany pakiet `java.util.concurrent`. Zawiera on *Executor Framework*, zapewniający elastyczny, bazujący na interfejsach mechanizm wykonywania zadań. Tworzenie kolejek zadań jest lepsze niemal w każdym aspekcie od sposobów przedstawionych w pierwszym wydaniu tej książki i wymaga użycia tylko jednego wiersza kodu:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Przesłanie obiektu `Runnable` do wykonania odbywa się w następujący sposób:

```
executor.execute(runnable);
```

Bezpieczne zakończenie działania obiektu wykonującego (jeżeli tego nie wykonamy, maszyna wirtualna nie zakończy działania) jest realizowane w następujący sposób:

```
executor.shutdown();
```

Przy użyciu nowej usługi można wykonać znacznie **więcej** operacji. Można na przykład poczekać na zakończenie wykonywania konkretnego zadania (tak jak w metodzie `get` z tematu 79.), można poczekać na zakończenie dowolnego lub wszystkich zadań z kolekcji (przy użyciu metod `invokeAny` lub `invokeAll`), można poczekać na bezpieczne zakończenie usługi wykonawcy (przy użyciu metody `awaitTermination`), można pobrać wyniki kolejnych zadań po ich zakończeniu (za pomocą `ExecutorCompletionService`), można harmonogramować zadania do wykonania o określonym czasie lub okresowo (za pomocą `ScheduledThreadPoolExecutor`) i tak dalej.

Jeżeli chcemy użyć więcej niż jednego wątku do przetwarzania żądań z kolejki, wystarczy użyć innej statycznej metody fabrycznej, która tworzy inny rodzaj usługi wykonawczej, nazywanej pulą wątków. Można tworzyć pule wątków ze stałą lub zmienną liczbą wątków. Klasa `java.util.concurrent.Executors` zawiera statyczne metody fabryczne zapewniające większość potrzebnych wykonawców. Jeżeli jednak potrzebujemy wykonać niestandardowe operacje, można bezpośrednio użyć klasy `ThreadPoolExecutor`. Pozwala ona na bezpośrednie kontrolowanie niemal każdego aspektu operacji na puli wątków.

Wybranie usługi wykonawczej dla określonej aplikacji może być skomplikowane. Jeżeli piszemy mały program lub mało obciążony serwer, użycie `Executors.newCachedThreadPool` jest zwykle dobrym wyborem, ponieważ nie jest wymagana konfiguracja i zwykle wykonawca „wykonuje właściwie swoje zadania”. Jednak buforowana pula wątków nie jest dobrym wyborem dla mocno obciążonego serwera produkcyjnego! W buforowanej puli wątków wysłane zadanie nie jest kolejkwane, ale natychmiast przekazywane do wątku w celu wykonania. Jeżeli nie ma dostępnych wątków, tworzony jest nowy. Jeżeli serwer jest tak mocno obciążony, że cała moc procesora jest wykorzystana i pojawiają się nowe zadania, tworzone są kolejne wątki, co tylko pogarsza sytuację. Z tego powodu w mocno obciążonym serwerze produkcyjnym znacznie lepiej jest użyć metody `Executors.newFixedThreadPool`, gwarantującej stałą liczbę wątków lub bezpośrednie użycie klasy `ThreadPoolExecutor` dla zapewnienia maksymalnej kontroli.

Nie tylko należy powstrzymać się od pisania własnych kolejek roboczych, ale również powinno się powstrzymać przed bezpośrednimi operacjami na wątkach. Kluczową abstrakcją przestała być klasa `Thread`, służąca jako jednostka pracy, oraz mechanizm jej wykonywania. Obecnie jednostka pracy oraz mechanizm są rozdzielone. Kluczową abstrakcją jest jednostka pracy nazywana **zadaniem**. Występują dwa rodzaje zadań — `Runnable` oraz jego bliski kuzyn, `Callable` (który jest identyczny jak `Runnable`, ale zwraca wartość). Ogólny mechanizm wykonywania zadań to usługa wykonawcza. Jeżeli będziemy myśleć w kategoriach zadań i pozwolimy na ich wykonywanie przez usługę wykonawczą, uzyskamy doskonałą elastyczność przez możliwość wyboru odpowiednich zasad wykonania. W rzeczywistości `Executor Framework` jest tym dla wykonania, czym `Collections Framework` dla agregowania.

W Javie 7 `Executor Framework` został wzbogacony o obsługę zadań rozdzielania i złączenia, które są realizowane przez specjalną usługę wykonawczą nazywaną pulą rozdzielania i łączenia. Zadanie tego typu, reprezentowane przez instancję `ForkJoinTask`, może zostać rozbite na mniejsze podzadania, a wątki wchodzące w skład `ForkJoinPool` nie tylko realizują te zadania, ale mogą nawet „podkraść” je sobie, aby zapewnić pełne wykorzystanie wątków, a tym samym dużą przepływność i niskie opóźnienia. Tworzenie i dostrajanie zadań rozdzielania i łączenia jest trudne. Strumienie równoległe (temat 48.) zostały napisane na bazie puli rozdzielania i łączenia, co pozwala niewielkim nakładem środków skorzystać z oferowanej wydajności (o ile oczywiście wykonywane zadanie można rozsądnie zrównoleglić).

Kompletny opis wykorzystania `Executor Framework` wykracza poza zakres tej książki; wszyscy zainteresowani powinni się zapoznać z książką *Java. Współbieżność dla praktyków* [Goetz07].

Temat 81. Stosowanie narzędzi współbieżności zamiast wait i notify

W pierwszym wydaniu książki znajduje się temat dotyczący prawidłowego stosowania `wait` i `notify` [Bloch01, temat 50.]. Te porady są nadal aktualne i zostały podsumowane pod koniec tego tematu, ale mają znacznie mniejsze znaczenie niż kiedyś. Dzieje się tak, ponieważ mamy coraz mniej powodów używania `wait` i `notify`. W Javie 5 pojawiły się wysokiego poziomu narzędzia współbieżności, które realizują wiele zadań, które wcześniej trzeba było wykonywać ręcznie na bazie `wait` i `notify`. **Z powodu problemów związanych z prawidłowym zastosowaniem wait i notify powinniśmy używać narzędzi wysokiego poziomu.**

Narzędzia wysokiego poziomu dostępne w `java.util.concurrent` można podzielić na trzy kategorie — opisany krótko w temacie 80. Executor Framework, kolekcje współbieżne oraz synchronizatory. Kolekcje współbieżne oraz synchronizatory są krótko opisane w tym temacie.

Kolekcje współbieżne zapewniają wysokiej wydajności współbieżne implementacje standardowych interfejsów kolekcji, takich jak `List`, `Queue` oraz `Map`. W celu zapewnienia wysokiej współbieżności implementacje te zarządzają wewnętrznie własnymi mechanizmami synchronizacji (temat 79.). **Dlatego niemożliwe jest wyłączenie równoległego działania z kolekcji współbieżnych; blokowanie nie ma żadnego efektu i tylko spowalnia program.**

Oznacza to, że klienci nie mogą atomowo komponować wywołań metod w kolekcjach współbieżnych. Niektóre z interfejsów kolekcji zostały rozszerzone o **zależne od stanu operacje modyfikacji**, które łączą kilka operacji prostych w jedną atomową. Operacje te okazały się na tyle przydatne w kolekcjach współbieżnych, że zostały nawet dodane w Javie 8 do interfejsów kolekcji jako metody domyślne (temat 21.).

Na przykład metoda `putIfAbsent(key, value)` pozwala na wstawienie odwzorowania dla klucza, jeżeli wcześniej ono nie istniało, oraz zwraca poprzednią wartość skojarzoną z kluczem lub `null`, gdy takie odwzorowanie nie istniało. Ułatwia to implementację bezpiecznych dla wątków odwzorowań kanonizacyjnych. Na przykład poniższa metoda symuluje działanie `String.intern`:

```
// Współbieżne odwzorowanie kanonizujące na bazie
// ConcurrentHashMap - mało optymalne
private static final ConcurrentHashMap<String, String> map =
    new ConcurrentHashMap<>();

public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

Faktycznie możemy lepiej zrealizować to zadanie. Klasa `ConcurrentHashMap` jest zoptymalizowana pod kątem operacji pobierania, takich jak `get`. Z tego powodu warto początkowo wywołać `get` i korzystać z `putIfAbsent` tylko w przypadkach, gdy `get` wskaże, że jest to konieczne:

```
// Współbieżne odwzorowanie kanonizujące na bazie ConcurrentHashMap - szybsze!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

Oprócz zapewnienia świetnej współbieżności klasa `ConcurrentHashMap` jest również bardzo szybka. Na moim komputerze zoptymalizowana w przedstawiony powyżej sposób metoda `intern` jest ponad sześć razy szybsza niż `String.intern` (trzeba jednak pamiętać, że `String.intern` musi użyć jakiejś odmiany słabej referencji w celu uniknięcia wycieków pamięci). Kolekcje współbieżne czynią kolekcje synchronizowane w zasadzie przestarzałymi. **Używaj `ConcurrentHashMap` zamiast `Collections.synchronizedMap`.** Zwyczajna wymiana starych synchronizowanych odwzorowań na odwzorowania współbieżne może znacznie zwiększyć wydajność aplikacji współbieżnych.

Niektóre interfejsy kolekcji zostały rozszerzone o **operacje blokujące**, które oczekują na możliwość (lub ją **blokują**) wykonania operacji. Na przykład `BlockingQueue` rozszerza `Queue` i dodaje kilka metod, w tym `take`, która usuwa i zwraca główny element z kolejki, oczekując, gdy kolejka jest pusta. Pozwala to na wykorzystanie kolejek blokujących jako **kolejki zadań** (nazywanej również **kolejką producenta i konsumenta**), do której wysyła elementy co najmniej jeden **wątek producenta** kolejującego zadania oraz z której co najmniej jeden **wątek konsumenta** pobiera i przetwarza elementy natychmiast, gdy się pojawiają. Jak można było tego oczekiwać, większość implementacji `ExecutorService`, w tym `ThreadPoolExecutor`, korzysta z `BlockingQueue` (temat 80.).

Synchronizatory to obiekty pozwalające na realizację oczekiwania jednego wątku na drugi, dzięki czemu można skoordynować ich działania. Najczęściej używanymi synchronizatorami są `CountDownLatch` oraz `Semaphore`. Mniej powszechnie wykorzystywane są `CyclicBarrier` oraz `Exchanger`. Synchronizatorem o największych możliwościach jest `Phaser`.

Pierwszy synchronizator (zatrząsk odliczający) jest jednoużytkownikową barierą pozwalającą co najmniej jednemu wątkowi na oczekiwanie na wykonanie pewnej operacji przez co najmniej jeden inny wątek. Jedyny konstruktor klasy `CountDownLatch` oczekuje wartości `int` reprezentującej liczbę wykonań metody `countDown`, zanim wszystkie oczekujące wątki będą mogły kontynuować działanie.

Zaskakująco łatwo można budować użyteczne rzeczy na bazie tych prostych mechanizmów. Załóżmy, że chcemy zbudować prostą bibliotekę do taktowania współbieżnych wykonań akcji. Biblioteka ta zawiera jedną metodę, która oczekuje, że wykonawcy realizującemu akcję zostanie przekazany poziom współbieżności, reprezentujący liczbę akcji, jakie można wykonać równolegle, jak również obiekt `Runnable`, reprezentujący akcję. Wszystkie wątki robocze są gotowe do wykonania akcji przed wystartowaniem stopera przez jego wątek sterujący (jest to niezbędne do uzyskania dokładnego taktowania). Gdy ostatni wątek roboczy jest gotowy do uruchomienia akcji, wątek stopera „strzela pistoletem startowym”, pozwalając wątkom roboczym wykonać akcje. Gdy ostatni wątek roboczy zakończy wykonywanie akcji, wątek stopera zatrzymuje zegar. Implementacja takiego działania na bazie `wait` oraz `notify` jest co najmniej skomplikowana, ale przy wykorzystaniu `CountDownLatch` jest zaskakująco prosta:

// Prosta biblioteka do taktowania równoległym wykonaniem

```

public static long time(Executor executor, int concurrency,
    Runnable action) throws InterruptedException {
    CountdownLatch ready = new CountdownLatch(concurrency);
    CountdownLatch start = new CountdownLatch(1);
    CountdownLatch done = new CountdownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            ready.countDown(); // informujemy stopera o gotowości
            try {
                start.await(); // oczekujemy na gotowość sąsiadów
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown(); // informujemy stopera o zakończeniu
            }
        });
    }

    ready.await(); // oczekiwanie na gotowość wszystkich procesów roboczych
    long startNanos = System.nanoTime();
    start.countDown(); // procesy robocze zostały uruchomione
    done.await(); // oczekiwanie na ich zakończenie
    return System.nanoTime() - startNanos;
}

```

Warto zwrócić uwagę, że w metodzie tej wykorzystane zostały trzy zatraski. Pierwszy, `ready`, jest używany przez wątki robocze do przekazania wątkowi stopera informacji i gotowości. Następnie wątki robocze oczekują na drugi zatrask, czyli `start`. Gdy ostatni wątek roboczy wywoła `ready.countDown`, wątek stopera rejestruje czas rozpoczęcia i wywołuje `start.countDown`, pozwalając na uruchomienie wszystkich wątków roboczych. Następnie wątek stopera oczekuje na trzeci zatrask, `done`, do momentu gdy wszystkie wątki robocze zakończą działanie i wywołają `done.countDown`. Natychmiast po tym wątek stopera budzi się i rejestruje czas zakończenia.

Warto przedstawić kilka szczegółów tego mechanizmu. Wątek wykonawczy przekazywany do metody `time` musi pozwolić na utworzenie co najmniej tylu wątków, ile wynika z ustawionego poziomu współbieżności, ponieważ w przeciwnym razie program nigdy się nie zakończy. Jest to nazywane **zagłodzeniem wątku** [Goetz06, 8.1.1]. Jeżeli wątek stopera przechwyci `InterruptedException`, przekazuje przerwanie z użyciem idiomu `Thread.currentThread().interrupt()` i kończy wykonywanie metody `run`. Pozwala to wykonawcy na prawidłowe obsłużenie przerwania. Warto również zauważyć, że do zmierzenia aktywności stopera wykorzystana była metoda `System.nanoTime`, a nie `System.currentTimeMillis`. **Do pomiaru odstępów czasu zawsze należy korzystać z `System.nanoTime` zamiast `System.currentTimeMillis`.**

Metoda `System.nanoTime` jest dokładniejsza, precyzyjniejsza i nie wpływają na nią korekty zegara czasu systemowego. Warto w tym miejscu podkreślić, że przedstawiony przykład nie spowoduje wyświetlenia właściwych czasów wykonania, jeśli metoda `action` nie wykona pewnych dłuższych prac (trwających sekundę lub więcej). Tworzenie dokładnych mikrotestów wydajnościowych jest trudnym zadaniem, więc lepiej powierzyć je wyspecjalizowanym frameworkom, takim jak `jmh` [JMH].

W tym temacie pokazaliśmy tylko mały wycinek narzędzi obsługi współbieżności. Na przykład trzy zatrzaśki zliczające z poprzedniego rozdziału mogą być zastąpione jedną barierą cykliczną lub jedną instancją `Phaser`. Wynikowy kod byłby nawet bardziej spójny, ale również trudniejszy do zrozumienia.

Choć zawsze powinniśmy korzystać z narzędzi współbieżności zamiast `wait` i `notify`, być może zajdzie konieczność konserwacji istniejącego kodu korzystającego z tych metod. Metoda `wait` jest wykorzystywana do wstrzymania wątku do czasu spełnienia pewnego warunku. Musi być ona wywoływana wewnątrz regionu synchronizowanego, który blokuje wykorzystywane obiekty. Poniżej przedstawiony jest standardowy idiom korzystający z metody `wait`:

```
// Standardowy idiom wykorzystujący metodę wait
synchronized (obj) {
    while (<niespełniony warunek >)
        obj.wait(); // (zwolnienie blokady i ponowne nałożenie po wznowieniu)
    ... // wykonanie akcji odpowiedniej dla warunku
}
```

Zawsze należy korzystać z pętli przy wywoływaniu metody `wait`, nigdy nie należy robić tego poza pętlą. Pętla pozwala na testowanie warunku przed oczekiwaniem i po oczekiwaniu.

Testowanie warunku przed oczekiwaniem i pominięcie `wait`, gdy warunek jest już spełniony, jest niezbędne do zapewnienia żywotności. Jeżeli warunek jest już spełniony i metoda `notify` (lub `notifyAll`) została wywołana przed oczekiwaniem wątku, to nie ma gwarancji, że wątek **kiedykolwiek** się obudzi.

Testowanie warunku po oczekiwaniu i powtórne oczekiwanie, gdy warunek nie jest spełniony, jest niezbędne do zapewnienia bezpieczeństwa. Jeżeli wątek wykonuje akcję, gdy warunek nie jest spełniony, może to uszkodzić niezmiennik chroniony przez blokadę. Istnieje kilka powodów wybudzenia wątku pomimo niespełnienia warunku:

- Inny wątek może nałożyć blokadę i zmienić chroniony stan pomiędzy wywołaniem `notify` przez wątek stopera, a czasem oczekiwania na wybudzenie wątku.
- Inny wątek może przypadkowo lub złośliwie wywołać `notify`, gdy warunek nie jest spełniony. Klasy mają publicznie dostępne obiekty używane do skonstruowania warunku oczekiwania. Jakiegokolwiek

wywołania `wait` znajdujące się w synchronizowanych metodach publicznie dostępnych obiektów są narażone na takie problemy.

- Wątek powiadamiający może być zbyt „ogólny” przy budzeniu oczekujących wątków. Na przykład wątek powiadamiający może wywołać `notifyAll`, nawet gdy tylko część z oczekujących wątków ma spełnione warunki.
- Wątek oczekujący może (rzadko) wybudzić się pomimo braku powiadomienia. Jest to znane pod nazwą **przypadkowego wybudzenia** [Posix, 11.4.3.6.1; Java9-api].

Podobnym problemem jest to, czy powinniśmy używać `notify`, czy `notifyAll` do budzenia oczekujących wątków. (Jak pamiętamy, `notify` budzi jeden oczekujący wątek, zakładając, że on jeszcze istnieje, natomiast `notifyAll` budzi wszystkie oczekujące wątki). Często spotyka się opinię, że **zawsze** należy używać `notifyAll`. Jest to rozsądna konserwatywna porada. Zawsze uzyskamy prawidłowe wyniki, ponieważ gwarantowane jest, że obudzimy wątek wymagający obudzenia. Można przy tym obudzić kilka innych wątków, ale nie wpływa to na prawidłowość naszego programu. Wątki te sprawdzą warunek, na który oczekują, i ponieważ nie jest spełniony, będą kontynuować oczekiwanie.

Jako optymalizację można przyjąć, że używana będzie metoda `notify` zamiast `notifyAll`, jeżeli wszystkie oczekujące wątki mają ten sam warunek i tylko jeden wątek jednocześnie może skorzystać ze spełnionego warunku.

Nawet jeżeli te warunki wydają się spełnione, może istnieć powód użycia `notifyAll` zamiast `notify`. Tak samo jak umieszczenie wywołania `wait` w pętli chroni przed przypadkowym lub złośliwym powiadomieniem w dostępnym publicznie obiekcie, tak samo użycie `notifyAll` zamiast `notify` chroni przed przypadkowym lub złośliwym oczekiwaniem przez niezwiązany wątek. Takie oczekujące wątki mogą czasami „połknąć” ważne powiadomienie, pozostawiając ich właściwych odbiorców w nieskończonym zawieszeniu.

Podsumujmy. Użycie `wait` i `notify` bezpośrednio jest podobne do programowania w „języku asemblera współbieżności” — w przeciwieństwie do języka wysokiego poziomu, oferowanego przez `java.util.concurrent`. **Rzadko, o ile kiedykolwiek, zajdzie potrzeba używania `wait` i `notify` w nowym kodzie.** Jeżeli konserwujemy kod korzystający z `wait` oraz `notify`, należy pamiętać o wywoływaniu `wait` w pętli `while` przy użyciu standardowego idiomu. Metoda `notifyAll` powinna być generalnie używana zamiast `notify`. Jeżeli wykorzystywana jest metoda `notify`, należy włożyć więcej pracy w zapewnienie żywotności.

Temat 82. Dokumentowanie bezpieczeństwa dla wątków

Zachowanie się klasy, gdy jej obiekt lub metoda statyczna będą wykorzystywane współbieżnie, jest ważnym fragmentem warunków stosowania klasy przez jej klienty. Jeżeli ten element zachowania klasy nie zostanie udokumentowany, programista korzystający z klasy może przyjąć niewłaściwe założenia. Jeżeli założenia te są złe, program może nie stosować dostatecznej synchronizacji (temat 78.) lub stosować jej zbyt dużo (temat 79.). Każdy z tych przypadków może doprowadzić do powstania poważnych błędów.

Czasami użytkownicy określają bezpieczeństwo dla wątków na podstawie istnienia modyfikatora `synchronized` w dokumentacji wygenerowanej przez Javadoc. Jest to niewłaściwe z kilku powodów. Javadoc nie dodaje modyfikatora `synchronized` do dokumentacji, i to z ważnej przyczyny. **Istnienie modyfikatora `synchronized` w deklaracji metody jest szczegółem implementacji, a nie częścią udostępnianego API.** Jego występowanie nie może rzetelnie wskazywać na bezpieczeństwo dla wątków interesującej nas metody.

Dodatkowo założenie, że występowanie słowa kluczowego `synchronized` jest wystarczające do udokumentowania bezpieczeństwa dla wątków, wynika z częstego, niewłaściwego założenia, że bezpieczeństwo wątków oparte jest o zasadę „wszystko albo nic”. W rzeczywistości klasa może obsługiwać wiele poziomów bezpieczeństwa wątków. **Aby umożliwić wielowątkowe wykorzystanie klasy, należy jednoznacznie udokumentować obsługiwany poziom bezpieczeństwa dla wątków.** Poniższa lista omawia poziomy bezpieczeństwa wątków stosowane przez klasę. Lista ta nie jest wyczerpująca, omawia jedynie często spotykane przypadki:

- **Niezmienne** — obiekty tej klasy są dla swoich klientów stałe. Nie jest potrzebna zewnętrzna synchronizacja. Przykładami są klasy `String`, `Long` i `BigInteger` (temat 17.).
- **Bezpieczne dla wątków** — obiekty tej klasy są modyfikowalne, ale wszystkie metody zawierają wystarczającą wewnętrzną synchronizację, więc obiekty mogą być wykorzystywane równolegle bez potrzeby wykorzystania zewnętrznej synchronizacji. Przykładem takich klas są `AtomicLong` i `ConcurrentHashMap`.
- **Warunkowo bezpieczne dla wątków** — podobne do bezpiecznych dla wątków, poza tym że zawiera metody, które wymagają zewnętrznej synchronizacji dla zapewnienia bezpiecznego użycia współbieżnego. Przykładami są kolekcje zwracane przez klasy osłonne `Collections.synchronized`, których iteratory wymagają zewnętrznej synchronizacji.

- **Niezdadne z wątkami** — obiekty tej klasy są modyfikowalne. Aby mogły być bezpiecznie wykorzystywane równolegle, należy zamknąć każde wywołanie metody (a w niektórych przypadkach sekwencje wywołań metod) w zewnętrznie synchronizowany blok. Przykładami tego rodzaju klas są kolekcje ogólnego użytku, takie jak `ArrayList` czy `HashMap`.
- **Niebezpieczne dla wątków** — klasy takie nie mogą być wykorzystywane równolegle przez kilka wątków, nawet jeżeli wszystkie wywołania metod będą zewnętrznie synchronizowane. Najczęściej cecha ta wynika z modyfikowania danych statycznych, które wpływają na inne wątki. Nikt nie tworzy takich klas celowo, więc ich powstanie jest najczęściej wynikiem braku uwzględnienia współbieżności. Jeśli odkryta zostanie klasa i metoda wątkowo-niebezpieczna, jest ona najczęściej naprawiana lub uznawana za przestarzałą. Metoda `generateSerialNumber` z tematu 78. byłaby wątkowo-niebezpieczna, gdyby zabrakło wewnętrznej synchronizacji.

Kategorie te (oprócz niebezpiecznych dla wątków) odpowiadają z grubsza adnotacjom bezpieczeństwa dla wątków w *Java*. *Współbieżność dla praktyków*, którymi są `Immutable`, `ThreadSafe` oraz `NotThreadSafe` [Goetz07, Dodatek A]. Bezwarunkowe i warunkowe kategorie bezpieczeństwa dla wątków w powyższej taksonomii są ujęte przez adnotację `ThreadSafe`.

Dokumentowanie klasy warunkowo bezpiecznej dla wątków wymaga zachowania uwagi. Należy określić, która sekwencja wywołań wymaga zewnętrznej synchronizacji oraz kiedy musi być nałożona blokada (lub rzadko blokady) w celu wyłączenia możliwości równoległego dostępu. Jeżeli obiekt stanowi alternatywny widok na inny obiekt, klient musi wykonać blokadę na obiekcie źródłowym w celu zablokowania bezpośrednich modyfikacji tego obiektu. Na przykład, dokumentacja dla metody `Collections.synchronizedMap` powinna zawierać następujące informacje:

Niezmiernie ważne jest, aby użytkownik ręcznie synchronizował zwracane odwzorowanie przy przeglądaniu dowolnego widoku kolekcji:

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
Set<K> s = m.keySet(); // nie może znajdować się w bloku
                        // synchronizowanym
...
synchronized(m) { // Synchronizowanie m, a nie s!
    for (K key : s)
        key.f();
}
```

Jeżeli nie będziemy stosować się do tej porady, program może działać w sposób niedeterministyczny.

Opis bezpieczeństwa klas dla wątków należy generalnie do komentarzy dokumentujących, ale metody o specjalnych właściwościach bezpieczeństwa dla wątków powinny opisywać te własności w ich własnych komentarzach dokumentujących. Nie jest konieczne dokumentowanie niezmienności typów wyliczeniowych. O ile nie jest to oczywiste na podstawie zwracanego typu, statyczne metody fabryczne muszą mieć udokumentowane bezpieczeństwo dla typów zwracanego obiektu, jak jest to pokazane dla `Collections.synchronizedMap`.

Gdy klasa potwierdza użycie publicznie dostępnej blokady, pozwala klientom na atomowe wykonanie sekwencji metod, ale elastyczność ta ma swoją cenę. Jest to niezgodne z wewnętrzną kontrolą współbieżności o wysokiej wydajności, które są używane w kolekcjach współbieżnych takich jak `ConcurrentHashMap`. Dodatkowo klient może spowodować atak typu „odmowa usługi” przez przetrzymywanie publicznie dostępnej blokady przez dłuższy czas. Może to być wykonane przypadkowo lub rozmyślnie.

Aby zapobiec atakowi typu „odmowa usługi”, można używać **obiektu blokady prywatnej** zamiast metod synchronizowanych (co powoduje użycie publicznie dostępnej blokady):

// Idiom prywatnego blokowania obiektu - udaremnienia atak odmowa usługi

```
private final Object lock = new Object();
```

```
public void foo() {  
    synchronized(lock) {  
        ...  
    }  
}
```

Ponieważ prywatny obiekt blokady jest niedostępny dla klientów klasy, niemożliwe jest wpływanie na synchronizację obiektu. W efekcie stosujemy poradę z tematu 15. przez hermetyzację obiektu blokady w synchronizowanym obiekcie.

Warto zauważyć, że pole `lock` jest zadeklarowane jako `final`. Chroni to przed przypadkową zmianą jego zawartości, mogącą spowodować katastrofalny niesynchronizowany dostęp do znajdującego się w nim obiektu (temat 78.). Stosujemy tu poradę z tematu 17., minimalizując zmienność pola `lock`. **Pola blokad zawsze należy deklarować jako pola final.** Zasada ta jest w mocy niezależnie od tego, czy używamy zwykłej blokady, jak w kodzie powyżej, czy bardziej zaawansowanych wersji z pakietu `java.util.concurrent.locks`.

Przypomnijmy, że idiom prywatnego obiektu blokady może być używany tylko w klasach **bezwzględnie** bezpiecznych dla wątków. Klasy warunkowo bezpieczne dla wątków nie mogą wykorzystywać tego idiomu, ponieważ muszą mieć udokumentowane blokady nakładane przez klienty w czasie wykonywania konkretnych sekwencji metod.

Wykorzystanie wewnętrznego obiektu do blokowania jest szczególnie przydatne w przypadku klas zaprojektowanych jako klasa bazowa (temat 19.). Jeżeli klasa bazowa będzie wykorzystywała własny obiekt do blokowania, to klasy pochodne mogą nieświadomie zakłócać tę operację. Wykorzystywanie tych samych blokad do różnych celów przez klasę bazową i pochodną może skończyć się „następowaniem sobie na palce”. Nie jest to tylko problem teoretyczny. Na przykład zdarzyło się to w klasie Thread [Bloch05, Puzzle 77].

Podsumujmy. Każda klasa powinna mieć w sposób jednoznaczny udokumentowany sposób obsługi bezpieczeństwa dla wątków. Modyfikator `synchronized` nie ma żadnego znaczenia dla tworzenia tej dokumentacji. W przypadku klas warunkowo wątkowo-bezpiecznych ważne jest określenie obiektu, który należy zablokować w celu zapewnienia atomowości wykonania sekwencji metod. Jeżeli piszemy klasę bezwarunkowo bezpieczną dla wątków, warto rozważyć zastosowanie prywatnego obiektu blokady zamiast metod synchronizowanych. Pozwala to chronić obiekty przed interferencją synchronizacji przez klienty i klasy pochodne, jak również daje nam możliwość zastosowania bardziej skomplikowanego podejścia do kontroli współbieżności w kolejnych wersjach.

Temat 83. Rozsądne korzystanie z późnej inicjalizacji

Późna inicjalizacja polega na odkładaniu inicjalizacji pola do momentu, gdy jego wartość jest potrzebna. Jeżeli wartość nie jest nigdy potrzebna, pole nigdy nie jest inicjowane. Technika ta może być stosowana zarówno w odniesieniu do pól statycznych, jak i instancyjnych. Choć późna inicjalizacja jest przede wszystkim optymalizacją, może być jednak użyta do przerywania szkodliwych zależności cyklicznych w klasie oraz inicjalizacji instancji [Bloch05, Puzzle 51].

Tak jak w przypadku większości optymalizacji, najlepszą poradą jest „nie używaj, jeżeli faktycznie nie potrzebujesz” (temat 67.). Późna inicjalizacja jest jak obosieczny miecz. Zmniejsza koszt inicjalizacji klasy lub tworzenia obiektów, ale skutkuje wydłużeniem czasu potrzebnego na odwołanie do późno inicjalizowanego pola. W zależności od tego, jaka część późno inicjowanych pól w końcu będzie wymagała inicjalizacji, jak kosztowana jest ta inicjalizacja i jak często odwołujemy się do tego pola, późna inicjalizacja może (jak wiele „optymalizacji”) w rzeczywistości obniżyć wydajność.

Trzeba jednak powiedzieć, że późna inicjalizacja ma swoje zastosowania. Jeżeli pole jest wykorzystywane w części obiektów **oraz** jego inicjalizacja jest kosztowna, późna inicjalizacja może się opłacić. Jedynym sposobem, aby się tego dowiedzieć, jest zmierzenie wydajności klasy z użyciem późnej inicjalizacji oraz bez niej.

W przypadku stosowania wielu wątków późna inicjalizacja jest skomplikowana. Jeżeli co najmniej dwa wątki współdzielą późno inicjowane pole, niezwykle ważne jest, aby zastosować pewną postać synchronizacji, ponieważ w przeciwnym razie może to spowodować powstanie poważnych błędów (temat 78.). Wszystkie techniki inicjalizacji przedstawione w tym temacie są bezpieczne dla wątków.

W większości przypadków zamiast późnej inicjalizacji zalecana jest zwykła inicjalizacja. Poniżej przedstawiona jest typowa deklaracja normalnie inicjowanego pola instancyjnego. Należy zwrócić uwagę na zastosowanie modyfikatora `final` (temat 17.):

```
// Normalna inicjalizacja pola instancyjnego
private final FieldType field = computeFieldValue();
```

Jeżeli późna inicjalizacja jest stosowana do przerywania cyklicznej inicjalizacji, należy użyć synchronizowanego akcesora, ponieważ jest to najprostszy i najbardziej czytelny sposób:

```
// Późna inicjalizacja pola instancyjnego - synchronizowany akcesor
private FieldType field;

private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

Oba te idiomy (**normalna inicjalizacja** oraz **późna inicjalizacja z synchronizowanym akcesorem**) działają w niezmiennionej formie dla pól statycznych, poza tym że należy dodać modyfikator `static` do deklaracji pola i akcesora.

Jeżeli potrzebna jest późna inicjalizacja w celu poprawienia wydajności pola statycznego, należy skorzystać z idiomu klasy przechowującej dla późnej inicjalizacji. Idiom ten wykorzystuje gwarancję, że klasa nie będzie zainicjowana do momentu użycia [JLS, 12.4.1]. Wygląda ona następująco:

```
// Idiom klasy przechowującej dla późnej inicjalizacji dla pól statycznych
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}

private static FieldType getField() { return FieldHolder.field; }
```

Gdy metoda `getField` zostanie wywołana po raz pierwszy, odczytuje `FieldHolder`. ↪ `field`, powodując, że klasa `FieldHolder` zostanie zainicjowana. Elegancja tego idiomu polega na tym, że metoda `getField` nie jest synchronizowana i jest wykonywana tylko przy dostępie do pola, więc późna inicjalizacja nie daje praktycznie żadnego kosztu przy dostępie. Nowoczesne maszyny wirtualne synchronizują dostęp do pól tylko przy inicjalizacji klasy. Po jej zainicjowaniu VM zmienia kod, aby kolejne dostępy do pola nie wymagały testowania ani synchronizacji.

Jeżeli potrzebna jest późna inicjalizacja w celu poprawienia wydajności pola instancyjnego, należy skorzystać z *idiomu dwukrotnego sprawdzenia*. Idiom ten pozwala uniknąć blokowania przy dostępie do pola, gdy jest ono już zainicjowane (temat 79.). Mechanizmem wykorzystanym w tym idiomie jest dwukrotne sprawdzenie wartości pola (stąd nazwa **dwukrotne sprawdzenie**) — raz bez blokowania i, jeżeli pole wydaje się niezainicjowane, drugi raz z blokowaniem. Inicjalizacja pola jest wywoływana wyłącznie w przypadku, gdy drugie sprawdzenie wskaże, że pole nie jest zainicjowane. Ponieważ zainicjowane już pole nie jest blokowane, **konieczne** jest zadeklarowanie pola jako `volatile` (temat 78.). Idiom ten wygląda następująco:

```
// Idiom podwójnego sprawdzenia dla późnej inicjalizacji pola instancyjnego
private volatile FieldType field;
FieldType getField() {
    FieldType result = field;
    if (result == null) { // pierwsze sprawdzenie (bez blokowania)
        synchronized(this) {
            result = field;
            if (result == null) // drugie sprawdzenie (z blokowaniem)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```

Kod ten może wydawać się nieco zawiły. W szczególności potrzeba zastosowania zmiennej lokalnej `result` może być niejasna. Zmienna ta zapewnia, że pole `field` jest odczytywane raz, w przypadku gdy jest już zainicjowane. Choć nie jest to niezbędne, może poprawić wydajność i jest bardziej eleganckie z punktu widzenia standardów stosowanych w programowaniu współbieżnym niskiego poziomu. Na moim komputerze powyższa metoda jest o mniej więcej 1,4 razy szybsza niż oczywista wersja bez zmiennej lokalnej.

Choć można go zastosować również do pól statycznych, nie ma powodu, aby to robić — idiom klasy przechowującej dla późnej inicjalizacji jest lepszym wyborem.

Warto przedstawić dwa warianty idiomu podwójnego sprawdzenia. Czasami może być potrzebna późna inicjalizacja pola instancyjnego, która może tolerować wielokrotną inicjalizację. Jeżeli znajdziemy się w takiej sytuacji, można skorzystać z odmiany idiomu podwójnego sprawdzenia, który nie zawiera drugiego sprawdzenia. Jest on nazywany **idiomem pojedynczego sprawdzenia**. Wygląda on w sposób przedstawiony poniżej. Należy zwrócić uwagę, że pole `field` jest nadal zadeklarowane jako `volatile`:

```
// Idiom pojedynczego sprawdzenia - może powodować wielokrotną inicjalizację!
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
```

```
if (result == null)
    field = result = computeFieldValue();
return result;
}
```

Wszystkie techniki inicjalizacji przedstawione w tym temacie odnoszą się do pól prostych, jak również do pól z referencjami obiektów. Gdy idiom podwójnego lub pojedynczego sprawdzenia jest stosowany w odniesieniu do numerycznego pola prostego, wartość pola jest porównywana z 0 (domyślna wartość prostych zmiennych numerycznych), a nie z null.

Jeżeli nie musimy się przejmować, że **każdy** wątek wyliczy wartość pola, a pole to ma typ prosty inny niż long lub double, można również usunąć modyfikator `volatile` z deklaracji pola w idiomie z pojedynczym sprawdzaniem. Ten wariant jest znany pod nazwą **idiomu pojedynczego sprawdzenia z wyścigiem**. W niektórych architekturach przyspiesza to dostęp do pól kosztem dodatkowych inicjalizacji (do jednej na wątek odwołujący się do pola). Jest to zdecydowanie egzotyczna technika, nieprzeznaczona do codziennego stosowania.

Podsumujmy. Większość pól powinniśmy inicjować w normalny sposób, a nie opóźniony. Jeżeli konieczne jest późne inicjowanie pól w celu osiągnięcia zakładanej wydajności lub przerwania szkodliwej inicjalizacji cyklicznej, to należy zastosować odpowiednią technikę późnej inicjalizacji. Dla pól instancyjnych jest to idiom podwójnego sprawdzenia, a dla pól statycznych idiom klasy przechowującej dla późnej inicjalizacji. Dla pól instancyjnych tolerujących wielokrotną inicjalizację można również rozważyć użycie idiomu pojedynczego sprawdzenia.

Temat 84. Nie polegaj na harmonogramie wątków

Gdy może być uruchomionych wiele wątków, harmonogram wybiera, które wątki mają być wykonywane i na jak długo. Wszystkie rozsądne systemy operacyjne będą próbowały wykonać ten wybór sprawiedliwie, ale zasady mogą się różnić. Dlatego dobrze napisane programy nie powinny polegać na szczegółach tych zasad. **Każdy program, którego prawidłowe działanie lub odpowiednia wydajność zależy od harmonogramu wątków, najprawdopodobniej nie będzie przenośny.**

Najlepszym sposobem na pisanie solidnych, szybko reagujących i przenośnych programów jest upewnienie się, że średnia liczba **możliwych do wykonania** wątków będzie niewiele większa od liczby procesorów. Dzięki temu nie pozostawia się harmonogramowi zadań wielkiego wyboru — po prostu będzie on uruchamiał wątki do momentu, gdy nie będzie dostępny żaden gotowy do uruchomienia.

Działanie programu nie będzie różniło się zbyt wiele, nawet w przypadku zastosowania różnych zasad harmonogramu wątków. Należy zwrócić uwagę, że liczba gotowych do uruchomienia wątków nie jest taka sama jak całkowita liczba wątków, których może być znacznie więcej. Wątki, które oczekują, nie są gotowe do uruchomienia.

Podstawową techniką pozwalającą ograniczać liczbę wątków gotowych do wykonania jest przekazanie wątkowi pewnych użytecznych operacji do wykonania, a następnie oczekiwanie na więcej. **Wątki nie powinny działać, jeżeli nie mają do wykonania użytecznych operacji.** W odniesieniu do biblioteki Executor Framework (temat 80.) oznacza to, że należy odpowiednio wyskalować pulę wątków [Goetz06, 8.2] i tworzyć rozsądnie małe i niezależne od siebie zadania. Zadania nie powinny być **zbyt** małe, ponieważ narzut na ich rozsyłanie może obniżyć wydajność.

Wątki nie powinny być stale **uruchamiane i zawieszane** przy wielokrotnym sprawdzaniu za pomocą współdzielonego obiektu, czy coś się wydarzyło. Oprócz tego, że program będzie narażony na kaprysy harmonogramu, wielokrotne zatrzymywanie i uruchamianie zwiększa obciążenie procesora, redukując liczbę użytecznych operacji, które mogłyby być wykonane. Jako ekstremalny przykład, jak nie należy programować, przedstawimy tę kuriozalną implementację CountdownLatch:

// Fatalna implementacja CountdownLatch - niepotrzebne

// zatrzymywanie i uruchamianie!

```
public class SlowCountDownLatch {
    private int count;

    public SlowCountDownLatch(int count) {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }

    public void await() {
        while (true) {
            synchronized(this) {
                if (count == 0)
                    return;
            }
        }
    }

    public synchronized void countDown() {
        if (count != 0)
            count--;
    }
}
```

Na moim komputerze klasa `SlowCountDownLatch` jest około 10 razy wolniejsza niż `CountDownLatch` przy 1000 oczekujących wątkach. Choć przykład ten może wydawać się zbyt przesadzony, niezbyt trudno spotkać się z systemami, w których co najmniej jeden wątek jest niepotrzebnie wykonywany. Wydajność i przenośność z pewnością na tym ucierpią.

Gdy spotkamy program, który ledwie działa, ponieważ niektóre wątki nie otrzymują wystarczająco dużo czasu procesora w stosunku do innych, **nie należy „poprawiać” takiego programu przez dodanie wywołania `Thread.yield`**. Być może uda się nam uruchomić program w taki sposób, ale nie będzie on przenośny. Ta samo wywołanie `yield`, poprawiające wydajność na jednej implementacji JVM, może pogarszać ją w drugiej i nie mieć żadnego efektu w trzeciej. **`Thread.yield` nie posiada testowalnego działania**. Lepszym rozwiązaniem jest zmiana struktury aplikacji w celu zredukowania liczby wątków jednocześnie gotowych do wykonania.

Podobną techniką z tego typu pułapkami jest zmiana priorytetów wątków. **Priorytety wątków są jedną z najmniej przenośnych funkcji platformy Java**. Dostrajanie czasów odpowiedzi aplikacji przez zmianę priorytetów kilku wątków nie jest nierozsądne, ale rzadko jest to potrzebne i nie jest przenośne. Nierozsądne jest rozwiązywanie poważnych problemów z żywotnością przez modyfikowanie priorytetów. Problem najprawdopodobniej będzie powracał do momentu znalezienia i poprawienia jego faktycznej przyczyny.

Podsumujmy. Nie należy polegać na harmonogramie wątków w celu zapewnienia prawidłowego działania naszego programu. Wynikowy program nie będzie ani solidny, ani przenośny. Zgodnie z tą zasadą nie należy polegać na `Thread.yield` ani na priorytetach wątków. Mechanizmy te są wyłącznie podpowiedziami dla harmonogramu. Priorytety wątków mogą być używane w rzadkich przypadkach do poprawienia jakości sprawnie działającego programu, ale nigdy nie mogą być traktowane jako „poprawka” w ledwie działającym programie.

Serializacja

W rozdziale tym omawiamy **serializację obiektów**, która w Javie odpowiada za przekształcenie obiektu do postaci strumienia bajtów (**serializacja**), jak i za rekonstrukcję obiektu z jego wersji bajtowej (**deserializacja**). Po serializacji obiektu jego postać bitowa może być przesyłana pomiędzy różnymi maszynami wirtualnymi lub zapisana na dysku w celu późniejszej deserializacji. Rozdział ten skupia się na zagrożeniach serializacji i podpowiada, jak je zminimalizować.

Temat 85. Stosuj rozwiązania alternatywne wobec serializacji Javy

Gdy dodawano do Javy serializację w roku 1997, wiedziano, że może nieść to ze sobą ryzyka. Sposób działania serializacji został wypróbowany na języku badawczym (Modula-3), ale nigdy nie był jeszcze stosowany w języku produkcyjnym. Choć zaoferowanie rozproszonych obiektów przy minimalnym udziale programisty było bardzo kuszące, ceną za tę wygodę były niewidzialne konstruktory oraz rozmycie podziału między API i implementację, a także narażenie się na problemy z poprawnością, wydajnością, bezpieczeństwem czy konserwacją kodu. Propagatorzy tego rozwiązania wierzyli, że zalety przewyższają ryzyka, ale historia pokazała coś odwrotnego.

Problemy z bezpieczeństwem opisywane we wcześniejszych wydaniach książki okazały się tak poważne, jak niektórzy sądzili. Podatności omawiane na początku XXI wieku zostały w następnej dekadzie zamienione na poważne exploity, w tym na poważny atak elektroniczny dla okupu na koleje SFMTA Muni (*San Francisco Metropolitan Transit Agency Municipal Railway*), który spowodował w listopadzie 2016 zamknięcie całego systemu płatności za przejazd na dwa dni.

Fundamentalnym problemem synchronizacji jest to, że **powierzchnia ataku** jest zbyt duża, by ją skutecznie chronić, a na dodatek cały czas rośnie — grafy obiektów są deserializowane wywołaniami metody `readObject` dla obiektu `ObjectInputStream`. Metoda ta to w praktyce magiczny konstruktor, który potrafi utworzyć obiekty w zasadzie dowolnego typu, o ile ich klasa znajduje się w ścieżce klas i implementuje interfejs `Serializable`. W procesie deserializacji strumienia bajtów metoda może wykonać kod znajdujący się w dowolnym z tych typów, więc potencjalną powierzchnią ataku jest kod **wszystkich** tych typów.

Powierzchnia ataku zawiera więc klasy znajdujące się w bibliotekach standardowych Javy, w bibliotekach firm trzecich (np. w Apache Commons Collections), a także w samej aplikacji. Nawet jeśli będziesz się stosował do wszystkich najlepszych praktyk i napiszesz serializowalne klasy niepodatne na atak, aplikacja nadal pozostanie podatna na atak. Zacytujmy Roberta Seacorda, menedżera ds. technicznych w CERT Coordination Center:

Deserializacja w Javie to wyraźne i aktualne zagrożenie, bo jest stosowane bezpośrednio przez aplikacje i pośrednio przez wiele podsystemów Javy, takich jak **RMI** (ang. *Remote Method Invocation*), **JMX** (ang. *Java Management Extension*) i **JMS** (ang. *Java Messaging System*). Deserializacja niezaufanych strumieni może prowadzić do zdalnego wykonania kodu, ataku typu DoS lub przeprowadzenia innych działań wykorzystujących znane podatności. Aplikacje mogą być podatne na te ataki nawet wtedy, gdy w niczym nie zawiniły [Seacord17].

Atakujący i badacze ds. bezpieczeństwa analizują serializowane typy w bibliotekach Javy i innych popularnych bibliotekach w poszukiwaniu metod wykonywanych w trakcie deserializacji, które mogą realizować potencjalnie niebezpieczne działania. Metody takie nazywa się **gadżetami**. Wiele gadżetów można wykorzystać w tandemie, tworząc **łańcuch gadżetów**. Od czasu do czasu odkrywane są łańcuchy gadżetów na tyle potężne, że pozwalają atakującemu na wykonanie dowolnego kodu natywnego na atakowanym sprzęcie po przesłaniu do ofiary jedynie precyzyjnie przygotowanego strumienia bajtów. Właśnie takiego ataku doświadczyło SFMTA Muni. Atak ten nie był odizolowanym przypadkiem. Były też inne ataki i będzie ich więcej.

Nawet bez uciekania się do gadżetów, możemy łatwo wygenerować atak typu DoS, wymuszając deserializację krótkiego strumienia, który wymaga wiele czasu na wykonanie. Takie strumienie nazywa się **bombami deserializacyjnymi** [Svoboda16]. Oto przykład przygotowany przez Wouteru Coekaerts, który używa tylko zbiorów mieszających i ciągów znaków [Coekaerts15]:

```
// Bomba deserializacyjna - deserializacja tego strumienia zajmie wieczność
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
```

```

Set<Object> s2 = new HashSet<>();
for (int i = 0; i < 100; i++) {
    Set<Object> t1 = new HashSet<>();
    Set<Object> t2 = new HashSet<>();
    t1.add("coś"); // uczyni t1 różnym od t2
    s1.add(t1); s1.add(t2);
    s2.add(t1); s2.add(t2);
    s1 = t1;
    s2 = t2;
}
return serialize(root); // metoda pominięta, aby zapewnić przejrzystość kodu
}

```

Graf obiektu składa się z 201 instancji `HashSet`, z których każda zawiera 3 lub mniej referencji do obiektów. Cały strumień ma tylko 5744 bajty, ale Słońce zdąży się wypalić, zanim uda się go zdeserializować. Problem polega na tym, że deserializacja instancji `HashSet` wymaga obliczenia funkcji skrótów dla wszystkich elementów. Dwa elementy korzenia zbioru również są zbiorami, które znowu zabierają dwa zbiory i tak dalej, aż do 100 poziomów w głąb. Oznacza to, że proces deserializacji musi wywołać metodę `hashCode` 2^{100} razy. Poza tym, że deserializacja trwa wieczność, mechanizm deserializacji nie znajdzie w danych bajtowych niczego niezwykłego. Powstaje relatywnie niewiele obiektów, a głębokość stosu jest ograniczona.

W jaki sposób bronić się przed tego rodzaju problemami? Otwieramy się na atak za każdym razem, gdy dopuszczamy deserializację strumienia bajtów, do którego nie mamy zaufania. **Najlepszym sposobem uniknięcia ataku poprzez deserializację jest brak deserializacji.** Można tu przytoczyć słowa komputera o imieniu Joshua z filmu *Gry wojenne* z 1983 roku: „Jedyną zwycięską strategią jest nie grać”. **Nie ma powodów, aby stosować serializację Javy w jakimkolwiek z nowo tworzonych projektów.** Istnieje wiele innych mechanizmów przekształcania obiektów na strumienie bajtów i odwrotnie, które pozwalają uniknąć niebezpieczeństw serializacji Javy, a przy okazji mają wiele innych zalet, takich jak obsługa wielu platform, wysoka wydajność, bogactwo narzędzi i większa społeczność użytkowników. W tej książce mechanizmy te nazywamy **wieloplatformowymi reprezentacjami danych strukturalnych**. Choć w wielu innych materiałach nazywa się je często systemami serializacji, w tej książce unikamy tego wyrażenia, aby nie wprowadzać skojarzeń z serializacją Javy.

Reprezentacje te mają pewną cechę wspólną — są **znacznie** prostsze od serializacji Javy. Nie obsługują automatycznej serializacji i deserializacji dowolnych grafów obiektów. Zamiast tego obsługują proste, strukturalne obiekty danych składające się ze zbiorów par atrybut-wartość. Obsługiwanych jest tylko kilka typów prostych oraz tablice. Ta prosta abstrakcja okazuje się wystarczająca do tworzenia bardzo rozbudowanych i elastycznych systemów rozproszonych, a jednocześnie jest na tyle prosta, że pozwala uniknąć poważnych problemów trapiących serializację Javy od momentu jej powstania.

Obecnie wiodącymi, wieloplatformowymi reprezentacjami danych strukturalnych są JSON [JSON] i Protocol Buffers nazywane skrótowo protobuf [Protobuf]. JSON został zaprojektowany przez Douglasa Crockforda do komunikacji między serwerem i przeglądarką, a Protocol Buffers zostały zaprojektowane przez firmę Google do przechowywania i wymiany danych strukturalnych między serwerami. Choć wymienione reprezentacje są czasem nazywane neutralnymi językowo, JSON powstał początkowo na potrzeby języka JavaScript, a protobuf na potrzeby języka C++; w obu reprezentacjach dostrzegalne są ślady ich początków.

Największą różnicą między JSON i protobuf jest to, że JSON to reprezentacja tekstowa i czytelna dla człowieka, a protobuf to reprezentacja binarna o znacznie większej wydajności. Gdy JSON to tylko i wyłącznie reprezentacja danych, protobuf reprezentuje również **schematy** (typy) dokumentów i wymusza ich poprawne użycie. Choć protobuf jest wydajniejszy niż JSON, JSON jest bardzo wydajny, jak na reprezentację tekstową. Choć protobuf to reprezentacja binarna, istnieje jego alternatywna reprezentacja tekstowa czytelna dla ludzi (pbtxt).

Jeśli nie można całkowicie uniknąć serializacji Javy, bo na przykład korzysta się ze starszych systemów, które jej wymagają, najlepszym rozwiązaniem jest **wykluczenie deserializacji niezaufanych danych**. W szczególności nigdy nie przyjmuj ruchu RMI z niezaufanych źródeł. Oficjalne zalecenia dotyczące pisania bezpiecznego kodu w Javie informują: „Deserializacja niezaufanych danych jest z natury niebezpieczna i należy jej unikać”. To zdanie jest zapisane dużą, pogrubioną czcionką z zastosowaniem kursywy i koloru czerwonego; to jedyny tekst w dokumencie, który został oznaczony w ten sposób [Java-secure].

Jeśli nie jesteś w stanie uniknąć serializacji i nie jesteś do końca pewien bezpieczeństwa deserializowanych danych, skorzystaj z filtrowania deserializowanych obiektów wprowadzonego w Javie 9 i przeniesionego również do wcześniejszych wydań (`java.io.ObjectInputFilter`). Strumień ten pozwala na wskazanie filtra stosowanego dla strumieni danych przed przeprowadzeniem deserializacji. Działa on na poziomie klasy, co pozwala dopuścić lub odrzucić poszczególne klasy. Domyślne akceptowanie wszystkich klas i odrzucanie tylko tych uznanych za potencjalnie niebezpieczne nazywamy korzystaniem z **czarnej listy**. Domyślnie odrzucanie wszystkich klas i akceptowanie jedynie wybranych, uznanych za bezpieczne, nazywamy korzystaniem z **białej listy**. **Preferuj korzystanie z białych list zamiast czarnych list**, bo czarna lista chroni jedynie przed znanym zagrożeniem. Narzędzie o nazwie SWAT (*Serial Whitelist Application Trainer*) pozwala na automatyczne przygotowanie białej listy dla aplikacji [Schneider16]. Mechanizm filtracji chroni również przed nadmiernym użyciem pamięci i wyjątkowo głębokimi grafami obiektów, ale nie chroni przed bombami deserializacyjnymi podobnymi do tej przedstawionej powyżej.

Niestety, serializacja nadal pozostaje wszechobecna w ekosystemie Javy. Jeśli odpowiadasz za konserwację systemu bazującego na serializacji Javy, poważnie rozważ migrację do wieloplatformowej i strukturalnej reprezentacji danych, mimo że będzie to zadanie kosztowne czasowo. Realistycznie rzecz biorąc, zapewne znajdziesz się w sytuacji, w której będziesz musiał napisać lub zmodyfikować klasę korzystającą z serializacji. Napisanie klasy serializowanej, która jest bezpieczna, poprawna i wydajna, wymaga dużej uwagi. Pozostała część rozdziału opisuje, kiedy i jak należy tego dokonać.

Podsumujmy. Serializacja jest niebezpieczna i należy jej unikać. Jeśli projektujesz system od podstaw, skorzystaj z wieloplatformowej i strukturalnej reprezentacji danych, takiej jak JSON lub protobuf. Nie deserializuj niezaufanych danych. Jeśli jednak musisz to zrobić, skorzystaj z obiektu filtrującego deserializację, ale pamiętaj, że nie gwarantuje to odporności na wszystkie ataki. Unikaj pisania klas używających serializacji. Jeśli już musisz to robić, postępuj bardzo ostrożnie.

Temat 86. Rozsądne implementowanie interfejsu Serializable

Aby umożliwić serializację obiektów danej klasy, wystarczy do jej deklaracji dodać `implements Serializable`. Ponieważ jest to tak łatwe do zrealizowania, często uważa się, że serializacja nie wymaga od programisty zbyt wiele pracy. W rzeczywistości jest to jednak bardziej skomplikowane. Mimo że krótkoterminowy koszt uaktywnienia serializacji klasy może być nieznaczny, to koszty długoterminowe często są znaczne.

Głównym problemem przy implementacji interfejsu Serializable jest zmniejszenie możliwości zmiany implementacji klasy po jej rozpowszechnieniu. Jeżeli klasa implementuje interfejs `Serializable`, reprezentujący ją strumień bajtów (**postać serializowana**) staje się częścią udostępnianego API. Gdy klasa zostanie rozpowszechniona, powinno się traktować jej serializowaną postać w identyczny sposób, jak inne części udostępnionego API. Jeżeli nie skorzystasz z możliwości zaprojektowania **własnej postaci serializowanej**, lecz jedynie przyjmiesz ustawienia domyślne, postać serializowana będzie na stałe związana z oryginalną, wewnętrzną reprezentacją obiektu. Inaczej mówiąc, jeżeli zaakceptujesz domyślną postać serializowaną, prywatne i prywatne w ramach pakietu pola obiektu staną się częścią udostępnianego API i praktyka minimalizacji dostępu do pól (temat 15.) utraci swoją efektywność jako sposób na hermetyzację.

Jeżeli zastosujesz domyślną postać serializowaną, a następnie zmienisz wewnętrzną budowę klasy, może to spowodować, że w postaci serializowanej znajdą zmiany powodujące jej niekompatybilność z istniejącą wcześniej postacią. Klienci próbujące

serializować obiekty za pomocą starej wersji klasy i deserializować je za pomocą nowej, nie będą mogły wykonać tej operacji. Możliwa jest zmiana wewnętrznej reprezentacji obiektu bez zmiany oryginalnej postaci serializowanej (przy wykorzystaniu metod `ObjectOutputStream.putFields` i `ObjectInputStream.readFields`), ale może być to trudne i pozostawiać w kodzie źródłowym widoczne niedoskonałości. Dlatego powinienś uważnie projektować wysokiej jakości postacie serializowane, które będą mogły być długo wykorzystywane (tematy 87. i 90.). Tworzenie tych postaci zwiększa koszty produkcji oprogramowania, ale warto jest zainwestowanych środków. Nawet dobrze zaprojektowana postać serializowana ogranicza rozbudowę klasy — źle zaprojektowana niemal ją uniemożliwia.

Prostym przykładem ograniczeń rozwoju, związanych z serializacją, jest problem **unikalnych identyfikatorów strumienia**, znanych pod nazwą **UID wersji**. Każda klasa, którą można serializować, posiada skojarzony z nią unikalny numer identyfikacyjny. Jeżeli jawnie nie określisz numeru identyfikacyjnego przez zadeklarowanie prywatnego pola typu `final long` o nazwie `serialVersionUID`, system automatycznie wygeneruje wartość, wykonując na klasie skomplikowaną funkcję deterministyczną (SHA-1). Automatycznie wygenerowana wartość zależy od nazwy klasy, nazw implementowanych interfejsów oraz wszystkich pól publicznych i zabezpieczonych. Jeżeli zmienisz dowolny z tych składników, na przykład dodając prostą funkcję pomocniczą, automatycznie generowany UID wersji zmieni się. Jeżeli jawnie nie zadeklarujesz numeru UID, nowa klasa przestanie być zgodna z poprzednimi wersjami, co będzie powodowało zgłaszanie w czasie działania wyjątków `InvalidClassException`.

Drugim ograniczeniem implementacji interfejsu `Serializable` jest zwiększenie prawdopodobieństwa wystąpienia błędów i luk w systemie bezpieczeństwa (temat 85.). Normalnie, obiekty są tworzone przy wykorzystaniu konstruktorów. Serializacja jest **pozajęzykowym mechanizmem** tworzenia obiektów. Niezależnie od tego, czy zachowasz domyślne metody, czy je przeddefiniujesz, deserializacja jest ukrytym konstruktorem. Obowiązują ją zatem wszystkie reguły, którym podlegają inne konstruktory. Ponieważ mamy tu do czynienia z jawnym konstruktorem, łatwo zapomnieć, że należy zapewnić utworzenie wszystkich niezmienników tworzonych przez konstruktory. Korzystanie z domyślnego mechanizmu deserializacji często umożliwia uszkodzenie niezmienników i nieautoryzowany dostęp do obiektu (temat 88.).

Trzecim ograniczeniem przy implementacji interfejsu `Serializable` jest zwiększanie kosztów testowania podczas tworzenia nowej wersji klasy. Gdy klasa jest modyfikowana, należy sprawdzić, czy możliwa jest serializacja obiektu w nowej wersji i deserializacja w wersji poprzedniej i odwrotnie. Liczba niezbędnych testów jest więc proporcjonalna do iloczynu liczby serializowanych klas oraz liczby wersji, co w wyniku może dać dużą liczbę. Testy te nie mogą być wygenerowane automatycznie, ponieważ oprócz zgodności binarnej należy przetestować zgodność składniową. Inaczej mówiąc, musisz sprawdzić, czy proces serializacji i deserializacji

przebiegły prawidłowo, jak również czy utworzona została wierna replika oryginalnego obiektu. Im większe zmiany w serializowanej klasie, tym większa jest potrzeba przeprowadzenia testów. Potrzeba ta jest mniejsza, jeżeli w czasie tworzenia klasy została odpowiednio zaprojektowana własna postać serializowana (tematy 87. i 90.).

Decyzja o implementowaniu interfejsu Serializable nie powinna być podejmowana pochopnie. Jest niezbędna, jeżeli klasa wchodzi w skład szkieletu programowego wykorzystującego serializację do przesyłania bądź zapamiętywania obiektów. Dodatkowo ułatwia wykorzystanie klasy jako komponentu w innej klasie, która musi implementować interfejs Serializable. Serializacja ma jednak wiele ograniczeń. Gdy projektujesz klasę, porównaj wady i zalety. Z powodów historycznych klasy posiadające wartość, takie jak `Instant` czy `BigInteger`, implementują `Serializable`, podobnie jak większość klas kolekcji. Klasy reprezentujące jednostki aktywne, takie jak pule wątków, rzadko powinny implementować `Serializable`.

Klasy zaprojektowane do dziedziczenia (temat 19.) rzadko powinny implementować interfejs Serializable, a interfejsy rzadko powinny go rozszerzać. Naruszenie tej zasady powoduje znaczne kłopoty przy rozszerzaniu klasy bądź implementacji interfejsu. Istnieją jednak przypadki, w których należy złamać tę zasadę. Na przykład jeżeli klasa lub interfejs są częścią szkieletu programowego wymagającego od wszystkich swoich klas implementacji interfejsu `Serializable`, to klasa może implementować, a interfejs rozszerzać interfejs `Serializable`.

Klasami zaprojektowanymi do dziedziczenia, które jednak implementują `Serializable`, są: `Throwable`, i `Component`. `Throwable` implementuje `Serializable`, aby możliwe było przesłanie wyjątku pomiędzy serwerem i klientem. `Component` implementuje `Serializable`, aby można było przysyłać, zapisywać i odtwarzać elementy GUI, ale nawet w latach świetności interfejsów `Swing` i `AWT` rozwiązanie to było rzadko stosowane.

Jeżeli implementujemy klasę z polami instancyjnymi, które są serializowane i rozszerzalne, trzeba pamiętać o kilku ryzykach. Jeśli istnieją jakiekolwiek niezmienniki w wartościach pól instancji, niezwykle ważne jest zapobiegnięcie przesłonięcia metody `finalize` w podklasach poprzez przesłonięcie metody `finalize` w klasie bazowej i zadeklarowanie jej jako `final`. W przeciwnym razie klasa będzie narażona na **ataki finalizacji** (temat 8.). Co więcej, jeżeli klasa ma niezmienniki, które są naruszone w przypadku zainicjowania pól wartościami domyślnymi (zero dla typów całkowitych, `false` dla `boolean` i `null` dla typów referencyjnych), koniecznie należy dodać do tej klasy poniższą metodę `readObjectNoData`:

```
// Metoda readObjectNoData dla rozszerzalnych serializowanych klas ze stanem
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Wymagane dane strumienia");
}
```

Metoda `readObjectNoData` została dodana w Javie 4 w celu zabezpieczenia przed warunkiem brzegowym, polegającym na dodaniu serializowanej klasy bazowej do istniejącej klasy serializowanej [Serialization, 3.5].

Istnieje jedna wada związana z podjęciem decyzji o **braku** implementacji `Serializable`. Jeśli klasa zaprojektowana na potrzeby dziedziczenia nie jest serializowana, zapewnienie serializacji podklasy będzie prawdopodobnie wymagało większych nakładów pracy. Standardowa deserializacja takiej klasy będzie wymagała, aby klasa bazowa posiadała dostępny konstruktor bezparametrowy [Serialization, 1.10]. Jeżeli go nie będzie, podklasa będzie zmuszona do skorzystania z wzorca pośrednika (temat 90.).

Klasy zagnieżdżone (temat 24.) nie powinny implementować interfejsu `Serializable`. Korzystają one z **pól sztucznych**, generowanych przez kompilator w celu zapamiętania referencji do **zawierających je obiektów** i do zapamiętywania wartości zmiennych lokalnych. Sposób współdziałania tych pól z definicją klasy jest nieokreślony. Dlatego domyślna postać serializowana dla klasy zagnieżdżonej jest nieodpowiednio zdefiniowana. Jednak **statyczne klasy składowe** mogą implementować interfejs `Serializable`.

Podsumujmy. Łatwość implementacji interfejsu `Serializable` jest złudna. Jeżeli klasa nie będzie stosowana tylko w środowisku chronionym, gdzie nie będzie problemów z różnymi wersjami, a serwery będą komunikowały się tylko za pomocą zaufanych danych, implementacja interfejsu `Serializable` jest poważnym zobowiązaniem, nad którym należy się poważnie zastanowić. Trzeba zachować dodatkową ostrożność, jeśli klasa dopuszcza dziedziczenie.

Temat 87. Wykorzystanie własnej postaci serializowanej

Gdy musisz bardzo szybko napisać klasę, zwykle ważne jest, aby skoncentrować się na zaprojektowaniu najlepszego API. Czasami związane jest z tym publikowanie tymczasowych implementacji, o których wiemy, że będą musiały być w przyszłości zastąpione lepszym rozwiązaniem. Normalnie nie stanowi to problemu, ale jeżeli klasa implementuje interfejs `Serializable` i korzysta ze standardowej postaci serializowanej, nigdy nie będziesz mógł całkowicie usunąć tymczasowej implementacji. Jej postać serializowana będzie musiała pozostać niezmieniona przez cały czas życia tej klasy. Problem nie jest tylko teoretyczny. Zdarzyło się to w przypadku kilku klas bibliotek platformy Java, na przykład `BigInteger`.

Zawsze sprawdzaj, czy domyślna postać serializowana jest odpowiednia dla Twojej klasy. Zaakceptowanie domyślnej postaci serializowanej powinno być konsekwencją analizy elastyczności, wydajności i prawidłowości domyślnego sposobu kodowania. Powinieneś wykorzystywać domyślną postać jedynie wtedy, gdy jest ona zgodna z kodowaniem, które byś wybrał, projektując własną postać serializowaną.

Domyślna serializowana postać obiektu jest umiarkowanie wydajną, zakodowaną postacią **fizycznej** reprezentacji grafu obiektów o początku w bieżącym obiekcie. Inaczej mówiąc, opisuje dane zawarte w obiekcie oraz wszystkie obiekty dostępne z tego obiektu. Opisuje również topologię połączeń tych obiektów. Idealna serializowana postać obiektu zawiera jedynie **logiczne** dane reprezentowane przez obiekt. Reprezentacja taka jest niezależna od jego fizycznej reprezentacji.

Domyślna postać serializowana jest odpowiednia jedynie, gdy reprezentacja fizyczna obiektu jest taka sama jak jej reprezentacja logiczna. Na przykład domyślna postać serializowana może być odpowiednia dla klasy reprezentującej nazwisko osoby.

// Dobry kandydat do wykorzystania domyślnej postaci serializowanej

```
public class Name implements Serializable {  
    /**  
     * Nazwisko. Musi być różne od null.  
     * @serial  
     */  
    private String lastName;  
  
    /**  
     * Imię. Musi być różne od null.  
     * @serial  
     */  
    private String firstName;  
  
    /**  
     * Drugie imię lub null, jeżeli nie został podany.  
     * @serial  
     */  
    private String middleName;  
  
    ... // pozostała część pominięta  
}
```

Pełne dane osoby składają się z trzech ciągów reprezentujących imię i nazwisko oraz dodatkowo w razie potrzeby drugie imię. Pola w klasie Name dokładnie odwierciedlają logiczną zawartość klasy.

Nawet jeżeli zdecydujesz się na zastosowanie domyślnej postaci serializowanej, często będziesz musiał napisać metodę readObject, zapewniającą zachowanie niezmienników i bezpieczeństwa. W przypadku klasy Name metoda readObject może sprawdzać, czy pola lastName i firstName mają wartości różne od null. Zagadnienie to zostanie omówione w tematach 88. i 90.

Zauważ, że pola `lastName`, `firstName` i `middleName` posiadają komentarze dokumentujące, mimo że są to pola prywatne. Ponieważ pola te definiują publiczne API — serializowaną postać klasy — to choć są one prywatne, muszą być udokumentowane. Zastosowanie znacznika `@serial` powoduje, że program Javadoc umieści tę dokumentację na specjalnej stronie, opisującej postacie serializowane.

Niemal całkowitym przeciwieństwem klasy `Name` jest poniższa klasa, reprezentująca listę ciągów (zignorujmy na moment fakt, że lepiej jest skorzystać ze standardowej klasy `List`).

```
// Fatalny kandydat do wykorzystania domyślnej postaci serializowanej
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }

    ... // pozostała część pominięta
}
```

Klasa ta reprezentuje sekwencję ciągów znaków. Fizycznie jest to lista dwukierunkowa. Jeżeli skorzystasz z domyślnej postaci serializowanej, to zostanie w niej umieszczona każda pozycja listy oraz wszystkie połączenia w obu kierunkach pomiędzy poszczególnymi elementami.

Korzystając z domyślnej postaci serializowanej, gdy różni się ona znacznie od logicznej zawartości obiektu, możemy spotkać się z czterema problemami:

- **Udostępnione API na stałe zostaje przywiązane do wewnętrznej reprezentacji danych.** W powyższym przykładzie prywatna klasa `StringList.Entry` staje się częścią publicznego API. Jeżeli w przyszłej wersji reprezentacja ta zmieni się, klasa `StringList` nadal będzie musiała pobierać na wejściu reprezentację danych w postaci listy i listę tę generować na wyjściu. Klasa nigdy nie będzie mogła pozbyć się kodu operującego na liście, nawet jeżeli nie będzie go do niczego innego potrzebowała.
- **Zajmowane jest zbyt dużo miejsca.** W powyższym przykładzie postać serializowana niepotrzebnie reprezentuje każdy element listy wraz z jego wszystkimi połączeniami. Te elementy i połączenia są jedynie szczegółami implementacji, które nie są warte umieszczenia w postaci serializowanej. Ponieważ postać serializowana jest niepotrzebnie duża, jej zapis na dysk lub przesłanie przez sieć będzie zajmowało więcej czasu.

- **Program serializujący długo się wykonuje.** Nie ma on dostępu do topologii grafu obiektów, więc musi wykonywać kosztowne przeglądanie grafu. W powyższym przykładzie wystarczy korzystać z referencji next.
- **Może dojść do przepełnienia stosu.** Domyślna procedura serializacji wykonuje rekurencyjne przeglądanie grafu obiektów, co może spowodować przepełnienie stosu, nawet w przypadku średnich grafów obiektów. Serializacja obiektu `StringList`, zawierającego od tysiąca do ośmiu tysięcy elementów, na moim komputerze powoduje przepełnienie stosu (wyjątek `StackOverflowError`). Co ciekawe, liczba elementów potrzebnych do spowodowania tego problemu zmienia się między uruchomieniami programu (przynajmniej na moim komputerze). Minimalna liczba elementów powodująca ten problem może zależeć od implementacji platformy lub opcji wiersza poleceń; w przypadku niektórych implementacji problem ten może nigdy nie wystąpić.

Wystarczającą postacią serializowaną w przypadku klasy `StringList` jest liczba, oznaczająca liczbę ciągów w liście, po której następują same ciągi. Są to tylko dane logiczne, zawarte w klasie `StringList`, niezawierające szczegółów fizycznej reprezentacji tych danych. Poniżej przedstawiamy poprawioną klasę `StringList`, zawierającą metody `writeObject` i `readObject`, implementujące tę postać serializowaną. Przypominamy, że modyfikator `transient` wskazuje, iż pole obiektu nie zostanie dołączone do domyślnej postaci serializowanej.

```
// Klasa StringList z własną postacią serializowaną
public class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;

    // Nie jest już serializowalne!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // dołącza podany ciąg do listy
    public final void add(String s) { ... }

    /**
     * Serializacja tego obiektu {@code StringList}.
     *
     * @serialData Rozmiar przesyłanej listy (liczba zawartych w niej ciągów)
     * ({@code int}), po którym następują wszystkie jej elementy
     * (każdy to {@code String}) w odpowiedniej kolejności.
     */
    private void writeObject(ObjectOutputStream s)
        throws IOException {
```

```

        s.defaultWriteObject();
        s.writeInt(size);

        // zapisanie wszystkich elementów w odpowiedniej kolejności
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        int size = s.readInt();

        // odczytanie wszystkich elementów i wstawienie ich do listy
        for (int i = 0; i < size; i++)
            add((String)s.readObject());
    }

    ... // pozostała część pominięta
}

```

Zwróć uwagę, że metoda `writeObject` wywołuje domyślną metodę `defaultWriteObject`, a metoda `readObject` wywołuje domyślną `defaultReadObject` nawet wtedy, gdy wszystkie pola klasy `StringList` są nietrwałe. Jeżeli wszystkie pola obiektu są nietrwałe, technicznie dopuszczalne jest pominięcie wywoływania metod `defaultWriteObject` i `defaultReadObject`, ale specyfikacja serializacji nadal zaleca ich wywołanie. Nawet jeżeli wszystkie pola obiektu są nietrwałe, wywołanie metody `defaultWriteObject` wpływa na postać serializowaną, co powoduje znaczne zwiększenie elastyczności. Taka postać serializowana umożliwia dodanie w późniejszych wersjach pól trwałych, przy zachowaniu zgodności z poprzednimi i kolejnymi wersjami klasy. Jeżeli obiekt został serializowany z wykorzystaniem późniejszej wersji klasy, a deserializowany przez klasę wcześniejszą, dodatkowe pola zostaną zignorowane. Jeżeli w metodzie `readObject` nie zostanie wywołana metoda `defaultReadObject`, deserializacja ta nie uda się i zostanie zgłoszony wyjątek `StreamCorruptedException`.

Należy zwrócić uwagę na komentarze dokumentujące w metodzie `writeObject`, mimo że jest ona prywatna. Zastosowanie tych komentarzy jest analogiczne do komentarzy dokumentujących w klasie `Name`. Metoda prywatna definiuje postać serializowaną, która jest elementem publicznego API, a to wymaga udokumentowania. Podobnie do znacznika `@serial` dla pól, znacznik `@serialData`, przeznaczony dla metod, nakazuje umieszczenie tych danych na osobnej stronie dokumentacji zawierającej postacie serializowane.

Przybliżmy skalę wspomnianych problemów związanych z wydajnością. Jeżeli średnia długość ciągu wynosi 10 znaków, to poprawiona postać serializowana dla klasy `StringList` zajmuje o połowę mniej miejsca niż postać domyślna. Na moim komputerze serializacja poprawionej wersji `StringList` jest ponad dwa razy

szybsza niż serializacja oryginalnej wersji. Przy zastosowaniu poprawionej wersji nie występują problemy z przepełnieniem stosu, dzięki czemu nie ma też ograniczenia wielkości serializowanych obiektów `StringList`.

Choć domyślna postać serializowana w przypadku klasy `StringList` nie byłaby zła, w przypadku innych klas skala problemów jest o wiele większa. W przypadku klasy `StringList` domyślna postać serializowana jest mało elastyczna i ma słabą wydajność, ale jest **prawidłowa** w takim sensie, że serializacja i deserializacja obiektu `StringList` powodują powstanie wiernej kopii, z zachowanymi wszystkimi niezmiennikami. Nie zachodzi to w przypadku klas, w których niezmienniki są powiązane ze szczegółami implementacji.

Rozważmy przykład tablicy mieszającej. Fizyczną reprezentacją jest sekwencja kubelków, zawierających pozycje klucz-wartość. To, do którego kubelka trafi element, jest funkcją kodu mieszającego klucza. W przypadku zastosowania różnych implementacji JVM nie ma gwarancji, że kod ten będzie identyczny. Nie ma takiej gwarancji nawet w przypadku kolejnych uruchomień programu na tej samej maszynie wirtualnej. Dlatego zastosowanie domyślnej postaci serializowanej w przypadku tablic mieszających może spowodować powstanie poważnych błędów. Serializacja i deserializacja obiektów może spowodować powstanie obiektów ze zniszczonymi niezmiennikami.

Niezależnie od tego, czy została zastosowana domyślna postać serializowana, czy nie, wywołanie metody `defaultWriteObject` spowoduje serializację każdego pola niestatycznego obiektu, nieoznaczonego jako `transient`. Dlatego w miarę możliwości każde pole obiektu powinno być oznaczone jako `transient`. Dotyczy to również nadmiarowych pól, na przykład zapamiętanych wartości kodu mieszającego, których wartości mogą być wyliczone na podstawie wartości podstawowych pól danych. Dotyczy to również pól, których wartości są zależne od konkretnego uruchomienia JVM, na przykład wartości pól `long`, reprezentujących wskaźniki do wewnętrznych pól danych. **Przed podjęciem decyzji o niedeklarowaniu pola jako `transient` należy przeanalizować, czy jest ono częścią logicznego stanu obiektu.** Jeżeli korzystasz z własnej postaci serializowanej, większość lub wszystkie niestaticzne pola obiektu powinny zostać zadeklarowane jako `transient`, tak jak w przedstawionej klasie `StringList`.

Korzystając z domyślnej postaci serializowanej i deklarując jedno lub więcej pól jako `transient`, należy pamiętać, że pola te zostaną zainicjowane podczas deserializacji **wartościami domyślnymi**: `null` dla referencji obiektów, `zero` dla prostych pól numerycznych i `false` dla pól `boolean` [JLS, 4.12.5]. Jeżeli wartości te nie są odpowiednie, należy stworzyć metodę `readObject` wywołującą metodę `defaultReadObject`, a następnie ustawiającą odpowiednie wartości pól `transient` (temat 88.). Można również zastosować opóźnioną inicjalizację tych pól (temat 83.).

Niezależnie od tego, czy stosujemy postać domyślną serializacji, czy nie, **konieczne jest wymuszenie synchronizacji w czasie serializacji obiektu, która zapewniłaby, że każdy inny obiekt odczyta pełny stan obiektu**. Jeżeli mamy na przykład obiekt bezpieczny dla wątków (temat 82.), który osiąga to bezpieczeństwo przez synchronizację każdej metody, i wybierzemy zastosowanie domyślnej postaci serializacji, należy skorzystać z następującej metody `writeObject`:

```
// Metoda writeObject dla klasy synchronizowanej z domyślną postacią
// synchronizacji
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
    s.defaultWriteObject();
}
```

Jeżeli dodamy synchronizację do metody `writeObject`, musimy również upewnić się, że jest ona zgodna z tymi samymi ograniczeniami kolejności blokad, co w innych funkcjach, ponieważ w przeciwnym razie ryzykujemy powstaniem zakleszczenia [Goetz06, 10.1.5].

W przypadku zastosowania zarówno domyślnej, jak i własnej postaci serializowanej, w każdej klasie podlegającej serializacji należy jawnie deklarować UID wersji. Eliminuje to potencjalne niezgodności związane z tym identyfikatorem (temat 86.). Dodatkowo uzyskuje się niewielki wzrost wydajności. Jeżeli nie zostanie zadeklarowany UID wersji, wymagane jest wykonanie czasochłonnych obliczeń.

Aby zadeklarować ID wersji, należy do klasy dodać następujący wiersz:

```
private static final long serialVersionUID = losowa_wartość_long;
```

Jeśli stworzysz nową klasę, wartość przypisana do tego pola nie ma znaczenia. Często stosowaną praktyką jest generowanie tej wartości poprzez uruchomienie programu `serialver` dla klasy, ale równie dobrze możesz wymyślić sobie dowolną liczbę. **Nie** jest wymagane, aby wartość ta była unikatowa. Jeżeli modyfikujemy istniejącą klasę, która nie ma UID wersji, a chcemy skorzystać z nowej wersji w celu przyjęcia istniejących serializowanych obiektów, konieczne jest użycie wartości automatycznie wygenerowanej dla starej wersji. Można uzyskać ten numer przez uruchomienie programu `serialver` na starej wersji klasy — tej, dla której istnieją serializowane instancje.

Jeżeli kiedykolwiek chcielibyśmy utworzyć nową wersję klasy, która jest **niezgodna** z istniejącą wersją, wystarczy zmienić wartość UID w deklaracji. Spowoduje to, że próby deserializacji obiektów poprzedniej wersji zostaną przerwane przez wyjątek `InvalidClassException`. **Nie zmieniaj wartości UID, chyba że naprawdę chcesz złamać zgodność wsteczną ze wszystkimi istniejącymi zserializowanymi instancjami klasy.**

Podsumujmy. Gdy uznasz, że klasa powinna być serializowana (temat 86.), musisz określić, w jakiej postaci będzie ona serializowana. Domyślną postać serializowaną można wykorzystać **jedynie** w przypadku, w którym stanowi ona opis logicznego stanu obiektu. W przeciwnym wypadku trzeba zaprojektować domyślną postać serializowaną, która odpowiednio opisuje obiekt. Na zaprojektowanie postaci serializowanej powinieneś poświęcić tyle samo czasu, co na zaprojektowanie udostępnianych metod (temat 51.). Tak samo jak nie możesz usuwać udostępnianych metod z kolejnych wersji klasy, tak samo nie będziesz mógł usuwać pól z postaci serializowanej. Nie mogą być one zmienione w celu zachowania kompatybilności serializacji. Wybór nieprawidłowej postaci serializowanej negatywnie wpływa na złożoność i wydajność klasy.

Temat 88. Defensywne tworzenie metody readObject

W temacie 50. omawialiśmy niezmienną klasę reprezentującą zakres dat, która zawierała modyfikowalne pola prywatne typu Date. Klasa aktywnie chroniła niezmienniki i swoją niezmienność, stosując defensywne kopiowanie obiektów Date, zarówno w konstruktorach, jak i w akcesorach. Poniżej przedstawiamy tę klasę:

```
//Niezmienna klasa wykorzystująca defensywne kopiowanie
public final class Period implements Serializable {
    private final Date start;
    private final Date end;

    /**
     * @param start początek odcinka czasu
     * @param end koniec odcinka czasu, nie może być wcześniejszy niż start
     * @throws IllegalArgumentException jeżeli start jest większy niż end
     * @throws NullPointerException jeżeli start lub end ma wartość null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start + " po " + end);
    }

    public Date start () { return new Date(start.getTime()); }

    public Date end () { return new Date(end.getTime()); }

    public String toString() { return start + " - " + end; }

    ... //pozostała część pominięta
}
```

Załóżmy, że chcemy serializować tę klasę. Ponieważ logiczna reprezentacja obiektu `Period` jest identyczna z reprezentacją fizyczną, można zastosować domyślną postać serializowaną (temat 87.). Dlatego może się wydawać, że wystarczy do deklaracji klasy dodać słowa `implements Serializable`. Jeżeli jednak to zrobimy, klasa nie będzie gwarantowała utrzymania swoich krytycznych niezmienników.

Problem wynika z tego, że metoda `readObject` efektywnie jest kolejnym publicznym konstruktorem i wymaga takiej samej uwagi, jak każdy inny konstruktor. Tak samo, jak konstruktor musi ona sprawdzać poprawność argumentów (temat 49.) oraz, w niezbędnych przypadkach, defensywnie tworzyć ich kopie (temat 50.), więc jej istnienie jest niezbędne. Jeżeli metoda `readObject` nie wykona tych operacji, dosyć łatwo można naruszyć niezmienniki klasy.

Metoda `readObject` efektywnie jest konstruktorem, który jako jedyne argumentu oczekuje strumienia bajtów. Zazwyczaj taki strumień jest generowany przez normalnie utworzony obiekt. Problem pojawia się, jeżeli do metody `readObject` przekazany zostanie ciąg bajtów, który został sztucznie wygenerowany w celu naruszenia niezmienników klasy. Taki ciąg bajtów pozwoli wtedy na utworzenie **obiekту niemożliwego**, czyli takiego, którego nie udałoby się wykonać przy użyciu standardowego konstruktora.

Załóżmy, że do deklaracji klasy `Period` dodaliśmy jedynie `implements Serializable`. Poniższy program generuje obiekt `Period`, w którym punkt końcowy jest wcześniejszy niż punkt początkowy. Rzutowania na `byte` dla wartości, w których ustawiony jest najwyższy bit, to konsekwencja braku w Javie literalów `byte` i podjęcia niefortunnej decyzji, że typ `byte` jest typem ze znakiem.

```
public class BogusPeriod {
    // Ciąg bajtów, który nie może być wygenerowany przez prawdziwy obiekt Period
    private static final byte[] serializedForm = {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
        0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
        0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
        (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
        0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
        0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
        0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
        0x00, 0x78
    };

    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
    }
}
```



```

        System.out.println(p);
    }

    //zwraca obiekt o podanej postaci serializowanej
    static Object deserialize(byte[] sf) {
        try {
            return new ObjectInputStream(
                new ByteArrayInputStream(sf)).readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new IllegalArgumentException(e);
        }
    }
}

```

Tablica bajtów wykorzystana do inicjalizacji pola `serializedForm` została wygenerowana przez serializację normalnego obiektu `Period` i edycję otrzymanego w ten sposób pliku. Szczegóły budowy tego strumienia nie są istotne dla tego przykładu, ale dociekliwi Czytelnicy mogą znaleźć opis tego formatu w publikacji *Java Object Serialization Specification* [Serialization, 6]. Jeżeli uruchomisz ten program, na ekranie wydrukuje on: `Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984`. Umożliwiając serializację klasy `Period`, umożliwiliśmy również tworzenie obiektów, które nie spełniają niezmienników klasy.

Aby poprawić ten błąd, w klasie `Period` należy utworzyć metodę `readObject`, która wywołuje metodę `defaultReadObject`, a następnie sprawdza poprawność deserializowanego obiektu. Jeżeli kontrola wykaze naruszenie niezmienników, metoda `readObject` zgłasza wyjątek `InvalidObjectException`, uniemożliwiając zakończenie deserializacji.

```

// Metoda readObject ze sprawdzaniem poprawności
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // sprawdzenie poprawności niezmienników
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " po " + end);
}

```

Choć w ten sposób uniemożliwiliśmy tworzenie nieprawidłowych obiektów `Period`, to w implementacji tej istnieje jeszcze jeden subtelny błąd. Możliwe jest utworzenie modyfikowalnego obiektu `Period` poprzez stworzenie strumienia bajtów, rozpoczynającego się reprezentacją prawidłowego obiektu `Period`, a następnie dodanie dodatkowych referencji do prywatnych pól `Date` z obiektu klasy `Period`. Z `Object` ↪ `InputStream` atakujący odczytuje obiekt `Period`, a następnie dodatkowe referencje dołączone do strumienia. Referencje te pozwalają na dostęp do prywatnych pól `Date` w obiekcie `Period`. Modyfikując obiekty `Date`, atakujący modyfikuje również obiekt `Period`. Poniższa klasa prezentuje ten rodzaj ataku:

```

public class MutablePeriod {
    // obiekt Period
    public final Period period;

    // pole początkowe obiektu, do którego nie możemy mieć dostępu
    public final Date start;

    // pole końcowe obiektu, do którego nie możemy mieć dostępu
    public final Date end;

    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);

            // serializacja prawidłowego obiektu Period
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Dołączenie wrogich "poprzednich referencji obiektów" do wewnętrznych
             * pól Date w obiekcie Period. Więcej informacji w "Java
             * Object Serialization Specification", część 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // pole start
            ref[4] = 4; // referencja nr 4
            bos.write(ref); // pole end

            // deserializacja obiektu Period i "skradzionych" referencji Date
            ObjectInputStream in = new ObjectInputStream(
                new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start = (Date) in.readObject();
            end = (Date) in.readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new AssertionError(e);
        }
    }
}

```

Aby zobaczyć, jak działa ten rodzaj ataku, wykonajmy następujący program:

```

public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;

    // cofamy zegar
    pEnd.setYear(78);
    System.out.println(p);
}

```

```
// Wracamy do lat sześćdziesiątych!
pEnd.setYear(69);
System.out.println(p);
}
```

Po uruchomieniu tego programu otrzymamy następujące wyniki:

```
Wed Nov 22 00:21:29 PST 2017 - Wed Nov 22 00:21:29 PST 1978
Wed Nov 22 00:21:29 PST 2017 - Sat Nov 22 00:21:29 PST 1969
```

Choć obiekt `Period` został utworzony z prawidłowymi niezmiennikami, możliwe jest dowolna zmiana składowych obiektu. Po przejściu kontroli nad obiektem `Period` atakujący może przekazać go do obiektu odpowiedzialnego za bezpieczeństwo systemu, który korzysta z naruszonej niezmienności obiektu `Period`. Nie jest to nieprawdopodobna sytuacja — istnieją klasy, które opierają się na niezmienności klasy `String` przy tworzeniu systemu bezpieczeństwa.

Źródłem problemów jest brak defensywnego kopiowania w metodzie `readObject`. **Gdy obiekt jest deserializowany, niezwykle ważne jest zastosowanie defensywnego kopiowania wszystkich pól, zawierających referencje do obiektów, do których klienci nie mogą mieć dostępu.** Dlatego w przypadku każdej serializowanej niezmiennej klasy, zawierającej prywatne modyfikowalne obiekty, w metodzie `readObject` należy defensywnie kopiować te obiekty. Poniższa metoda `readObject` zapewnia ochronę niezmienników klasy `Period` i zapewnia jej niezmiennność:

```
// Metoda readObject z defensywnym kopiowaniem i kontrolą poprawności
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // defensywne kopiowanie modyfikowalnych komponentów
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // sprawdzenie poprawności niezmienników
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " po " + end);
}
```

Zwróć uwagę, że defensywne kopiowanie jest wykonywane przed kontrolą poprawności. Do kopiowania obiektów `Date` wykorzystano metodę `clone`. Operacje te są niezbędne do zabezpieczenia obiektu `Period` przed atakami (temat 50.). Zwróć również uwagę, że nie jest możliwe defensywne kopiowanie pól typu `final`. Aby skorzystać z metody `readObject`, należy usunąć modyfikator `final` z deklaracji pól `start` i `end`. Niestety, jest to wybór mniejszego zła. Gdy zastosujemy nową metodę `readObject` i usuniemy modyfikator `final` z deklaracji pól `start` i `end`, klasa `MutablePeriod` nie będzie mogła nic zdziałać. Otrzymamy następujący wynik:

```
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
```

Aby sprawdzić, czy dopuszczalna jest domyślna metoda `readObject`, możesz odpowiedzieć na następujące pytanie. Czy właściwe jest dodanie publicznego konstruktora, który jako parametrów oczekuje wartości wszystkich trwałych pól obiektu i zapisuje te wartości bez żadnej kontroli poprawności? Jeżeli nie możesz twierdząco odpowiedzieć na takie pytanie, musisz utworzyć metodę `readObject`, która będzie kontrolowała poprawność danych i defensywnie kopiowała pola wymagane przez konstruktor. Można również użyć **wzorca pośrednika serializacji** (temat 90.). Wzorec ten jest polecany, bo pozwala uniknąć mnóstwa wysiłku związanego standardowo z bezpieczną deserializacją.

Istnieje jeszcze jedno podobieństwo pomiędzy metodą `readObject` i konstruktorami, które ma zastosowanie dla niefinalnych, serializowanych klas. Metoda `readObject` nie może pośrednio ani bezpośrednio wywoływać metod, które mogą zostać zdefiniowane (temat 19.). Jeżeli zasada ta zostanie naruszona i metoda zostanie zdefiniowana, to zdefiniowana metoda będzie uruchomiona przed deserializacją stanu klasy pochodnej. W wyniku tego program przestanie działać [Bloch05, Puzzle91].

Podsumujmy. Tworząc metodę `readObject`, należy stosować wszystkie zasady dotyczące konstruktorów. Należy utworzyć prawidłowy obiekt niezależnie od zawartości przekazanego strumienia bajtów. Nie należy zakładać, że przekazany strumień bajtów jest reprezentacją prawidłowego, serializowanego obiektu. Choć przykłady w tym temacie przedstawiały klasy korzystające z domyślnej postaci serializowanej, te same zasady należy stosować w przypadku korzystania z własnej postaci serializowanej. Poniżej jeszcze raz zamieszczamy zasady tworzenia bezpiecznej metody `readObject`.

- W przypadku klas z polami zawierającymi referencje do obiektów, które muszą pozostać prywatne, należy defensywnie kopiować każdy obiekt wskazywany przez takie pole. Do tej kategorii należą modyfikowalne komponenty klas niezmiennych.
- W przypadku klas z niezmiennikami należy sprawdzać ich poprawność i, w sytuacji naruszenia niezmienników, zgłaszać wyjątek `InvalidObjectException`. Sprawdzenie to powinno być przeprowadzane po defensywnym kopiowaniu.
- Jeżeli cały graf obiektów musi zostać skontrolowany po jego deserializacji, powinien zostać wykorzystany interfejs `ObjectInputValidation` (nie jest omawiany w tej książce).
- Nie wywołuj metod, które mogą zostać zdefiniowane, zarówno bezpośrednio, jak i pośrednio.

Temat 89. Stosowanie typów wyliczeniowych zamiast readResolve do kontroli obiektów

W temacie 3. opisaliśmy wzorzec **singleton**, posiłkując się przedstawioną poniżej klasą typu singleton. Klasa ta blokuje dostęp do swojego konstruktora, dzięki czemu można zapewnić utworzenie tylko jednego obiektu tej klasy.

```
public class Elvis {  
    public static final Elvis INSTANCE = new Elvis();  
    private Elvis() {...}  
  
    public void leaveTheBuilding() {...}  
}
```

W temacie 3. wspomniano, że jeżeli do deklaracji zostanie dodana fraza `implements Serializable`, klasa przestanie być typu singleton. Nie ma znaczenia, czy wykorzystamy domyślną postać serializacji, czy też stworzymy własną (temat 87.); nie ma również znaczenia, czy klasa posiada własną klasę `readObject` (temat 88.). Dowlolna metoda `readObject`, zarówno własna, jak i domyślna, zwraca nowy obiekt, który nie będzie tym samym obiektem, utworzonym podczas inicjalizacji klasy.

Metoda `readResolve` pozwala na zastąpienie jednego obiektu innym, utworzonym przez `readObject` [Serialization, 3.7]. Jeżeli w klasie zostanie zdefiniowana metoda `readResolve`, to w czasie deserializacji jest ona wywoływana dla nowo utworzonego obiektu po zakończeniu deserializacji. Referencja zwracana przez tę metodę jest traktowana jako referencja do obiektu powstałego w procesie deserializacji. W większości zastosowań tej metody nie jest zwracana referencja do nowo utworzonego obiektu, lecz do obiektu już istniejącego, więc nowo utworzony obiekt od razu jest niszczone przez mechanizm usuwania nieużytków.

Jeżeli klasa `Elvis` implementowałaby interfejs `Serializable`, poniższa metoda `readResolve` gwarantuje, że będzie istniał tylko jeden obiekt klasy.

```
// Metoda readResolve przy kontroli obiektów - można to zrobić lepiej!  
private Object readResolve() {  
    // zwraca jedyny prawdziwy obiekt Elvis, a jego naśladowcami  
    // zajmie się proces usuwania nieużytków  
    return INSTANCE;  
}
```

Metoda ta po prostu ignoruje deserializowany obiekt, zwracając obiekt `Elvis` utworzony podczas inicjalizacji klasy. Dlatego serializowana postać klasy `Elvis` nie musi zawierać żadnych danych — wszystkie pola obiektu mogą być zadeklarowane jako `transient`. Dotyczy to nie tylko klasy `Elvis`, ale wszystkich klas typu singleton.

Jeżeli korzystamy z `readResolve` do kontroli obiektów, wszystkie pola instancyjne z referencjami do obiektów *muszą* być zadeklarowane jako **transient**. W przeciwnym razie możliwe będzie uzyskanie przez napastnika referencji do deserializowanego obiektu, jeszcze przed wywołaniem metody `readResolve`, co można osiągnąć przez zastosowanie techniki bardzo podobnej do ataku `MutablePeriod` z tematu 88.

Atak jest nieco skomplikowany, ale wykorzystano w nim prosty pomysł. Jeżeli singleton zawiera inne niż **transient** pola z referencjami obiektów, zawartość tych pól będzie deserializowana przed uruchomieniem metody `readResolve` obiektu singleton. Pozwala to na użycie odpowiednio spreparowanego strumienia do „wykradzenia” referencji do początkowo deserializowanego singletonu w momencie deserializacji pola z referencją obiektu.

Jak to działa w praktyce? Należy napisać klasę „złodziejkę”, która posiada metodę `readResolve`, a także pole instancyjne odwołujące się do serializowanego singletonu, w którym „ukrywa się” napastnik. W strumieniu serializowanym, pole singletonu inne niż **transient** jest zastępowane instancją złodzieja. Mamy teraz cykliczność: singleton zawiera złodzieja, a złodziej odwołuje się do singletonu.

Ponieważ singleton zawiera złodzieja, metoda `readResolve` ze złodzieja będzie wykonana jako pierwsza w czasie deserializacji singletonu. W wyniku tego po uruchomieniu metody `readResolve` jego pole instancyjne odwołuje się do częściowo deserializowanego (i przez to nierozwiązanego) singletonu.

Metoda `readResolve` ze złodzieja kopiuje referencję z jego pola instancyjnego do statycznego, przez co można się do niej odwołać po zakończeniu działania `readResolve`. Następnie metoda ta zwraca wartość typu prawidłowego dla pola, w którym się ukrywa. Jeżeli nie zrobiłaby tego, maszyna wirtualna zgłosiłaby wyjątek `ClassCastException` w momencie próby zapisu referencji złodzieja w tym polu.

Aby przedstawić ten problem, przeanalizujmy nieprawidłowy singleton:

```
// Nieprawidłowy singleton - posiada referencje do obiektów w polach innych
// niż transient!
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }

    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() throws ObjectStreamException {
        return INSTANCE;
    }
}
```

Poniżej przedstawiona jest klasa „złodzieja”, skonstruowana zgodnie z powyższym opisem:

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private Object readResolve() {
        // zapisanie referencji do "nierozwiązanej" instancji Elvis
        impersonator = payload;

        // zwrócenie obiektu o prawidłowym typie pola favorites
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
```

Jako ostatni przedstawimy brzydki program, który deserializuje ręcznie przygotowany strumień, tworząc dwie osobne instancje niewłaściwego singletonu. Metoda deserializacji jest w tym programie pominięta, ponieważ jest identyczna z tą z tematu 88.:

```
public class ElvisImpersonator {
    // Strumień bajtów, jaki nie może być wynikiem serializacji obiektu Elvis!
    private static final byte[] serializedForm = {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
        0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
        (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
        0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
        0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
        0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
        0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
        0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
        0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
        0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
        0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
        0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
    };

    public static void main(String[] args) {
        // inicjuje ElvisStealer.implicitator i zwraca
        // prawdziwy obiekt Elvis (który jest Elvis.INSTANCE)
        Elvis elvis = (Elvis) deserialize(serializedForm);
        Elvis impersonator = ElvisStealer.implicitator;

        elvis.printFavorites();
        impersonator.printFavorites();
    }
}
```

Uruchomienie tego programu daje wynik zamieszczony poniżej, co udowadnia, że jest możliwe utworzenie dwóch osobnych obiektów Elvis (o innych gustach muzycznych).

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

Można naprawić ten problem przez zadeklarowanie pola `favoriteSongs` jako `transient`, ale znacznie lepiej jest utworzyć jednoelementowy typ wyliczeniowy z obiektem `Elvis` (temat 3.). Jak pokazaliśmy na przykładzie ataku `ElvisStealer`, technika użycia metod `readResolve` do uniemożliwienia atakującemu dostępu do „tymczasowej” instancji po deserializacji jest krucha i wymaga dużo uwagi.

Jeżeli zapiszemy naszą klasę serializowaną z kontrolą instancji jako typ wyliczeniowy, otrzymamy solidną gwarancję, że nie będą istniały żadne obiekty poza zadeklarowanymi stałymi, chyba że atakujący wykorzysta metody zabezpieczone takie jak `AccessibleObject.setAccessible`. Jeśli jednak atakujący ma uprawnienia, aby to zrobić, może równie dobrze wykonać dowolny kod natywny. Poniżej przedstawiona jest klasa `Elvis` jako typ wyliczeniowy:

```
// Singleton wyliczeniowy - preferowane podejście
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

Stosowanie `readResolve` do kontroli obiektów jest teraz niepotrzebne. Jeżeli mamy napisać klasę serializowaną z kontrolą instancji, ale te instancje nie są znane na etapie kompilacji, nie można ich reprezentować jako typu wyliczeniowego.

Dostępność `readResolve` jest znacząca. Jeżeli umieścimy metodę `readResolve` w klasie finalnej, powinna być ona prywatna. Jeżeli umieścimy ją w klasie niefinalnej, należy uważnie przeanalizować widoczność. Jeżeli jest prywatna, nie będzie stosowana do żadnej klasy pochodnej. Jeżeli będzie prywatna dla pakietu, będzie stosowana tylko w klasach pochodnych z tego samego pakietu. Jeżeli będzie publiczna lub zabezpieczona, będzie stosowana do wszystkich klas pochodnych, które jej nie przesłonią. Jeżeli metoda `readResolve` jest zabezpieczona lub publiczna, a klasa pochodna jej nie przesłania, deserializacja obiektu klasy pochodnej spowoduje utworzenie obiektu klasy bazowej, co najprawdopodobniej spowoduje wygenerowanie wyjątku `ClassCastException`.

Podsumujmy. Wszędzie tam gdzie, jest to możliwe, powinniśmy stosować typy wyliczeniowe do wymuszenia kontroli niezmienników obiektu. Jeżeli nie jest to możliwe, a potrzebujemy klasy serializowanej oraz kontrolującej instancje, konieczne jest napisanie metody `readResolve` i upewnienie się, że wszystkie pola instancyjne klasy są albo typu prostego, albo są zadeklarowane jako `transient`.

Temat 90. Użycie pośrednika serializacji zamiast serializowanych obiektów

Jak wspomnieliśmy w tematach 85. i 86., a także przedstawiamy w całym rozdziale, decyzja o implementowaniu interfejsu `Serializable` zwiększa prawdopodobieństwo powstania błędów i problemów z bezpieczeństwem, ponieważ powoduje tworzenie obiektów z użyciem mechanizmów pozajęzykowych, a nie zwykłych konstruktorów. Istnieje jednak technika, która znacznie ogranicza te problemy. Jest ona nazywana **wzorcem pośrednika serializacji**.

Wzorec ten jest dosyć prosty. Przede wszystkim należy zaprojektować prywatną, statyczną klasę zagnieżdżoną w klasie serializowanej, która w spójny sposób reprezentuje stan klasy, w której jest zagnieżdżona. Ta klasa zagnieżdżona, znana pod nazwą **pośrednika serializacji**, powinna mieć jeden konstruktor, którego parametrem jest klasa gospodarza. Konstruktor ten tylko kopiuje dane z argumentu — nie zawiera żadnego sprawdzania spójności ani kopiowania defensywnego. Zgodnie z projektem domyślna serializowana postać pośrednika serializacji jest doskonałą serializowaną postacią klasy gospodarza. Zarówno klasa gospodarza, jak i jej pośrednik serializacji muszą implementować `Serializable`.

Dla przykładu przeanalizujmy niezmienną klasę `Period` przedstawioną w temacie 50., która była serializowana w temacie 88. Tym razem przedstawimy pośrednika serializacji tej klasy. Klasa `Period` jest tak prosta, że jej pośrednik serializacji ma dokładnie te same pola co klasa:

```
// Pośrednik serializacji dla klasy Period
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;

    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }

    private static final long serialVersionUID =
        234098243823485285L; // dowolna liczba (temat 87.)
}
```

Następnie do klasy gospodarza dodajemy poniższą metodę `writeReplace`. Metoda ta może zostać skopiowana bez zmian do dowolnej klasy z pośrednikiem serializacji:

```
// Metoda writeReplace dla wzorca pośrednika serializacji
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

Metoda ta powoduje, że system serializacji wyśle obiekt `SerializationProxy` zamiast obiektu klasy gospodarza. Inaczej mówiąc, metoda `writeReplace` przekształca obiekt klasy gospodarza na jej pośrednika serializacji.

Gdy mamy już napisaną metodę `writeReplace`, system serializacji nigdy już nie wygeneruje serializowanej instancji klasy gospodarza, ale napastnik może spreparować taką reprezentację w celu próby naruszenia niezmienników klasy. Aby zagwarantować, że taki atak się nie uda, wystarczy dodać do klasy gospodarza metodę `readObject`:

```
// Metoda readObject dla wzorca pośrednika serializacji
private void readObject(ObjectInputStream stream)
    throws InvalidObjectException {
    throw new InvalidObjectException("Wymagany pośrednik");
}
```

Na koniec należy napisać metodę `readResolve` dla klasy `SerializationProxy`, która zwraca logiczny odpowiednik obiektu klasy gospodarza. Metoda ta powoduje, że system serializacji przekształca pośrednika serializacji z powrotem na obiekt klasy gospodarza.

Taka metoda `readResolve` tworzy obiekt klasy gospodarza z użyciem wyłącznie publicznego API, co stanowi o elegancji wzorca. W większości eliminuje on pozajęzykowy charakter serializacji, ponieważ deserializowane obiekty są tworzone z użyciem tych samych konstruktorów, statycznych metod fabrycznych i metod, co inne obiekty. Uwalnia to nas od konieczności osobnego zapewniania, że deserializowane obiekty spełniają niezmienniki klasy. Jeżeli statyczne metody fabryczne lub konstruktory klasy zapewniają zainicjowanie tych niezmienników, a metody instancyjne ich zachowanie, jesteśmy pewni, że niezmienniki będą spełnione również dla serializacji.

Poniżej przedstawiona jest metoda `readResolve` dla `Period.SerializationProxy`:

```
// Metoda readResolve method dla Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end); // używa publicznego konstruktora
}
```

Podobnie jak mechanizm kopiowania defensywnego, pośrednik serializacji nie pozwala na atak fałszywym strumieniem bajtów oraz atak na wewnętrzne pole (temat 88.). W przeciwieństwie do dwóch poprzednich podejść pozwala to na oznaczenie pól `Period` jako `final`, co jest wymagane, aby klasa `Period` była naprawdę niezmienna (temat 17.). W przeciwieństwie do dwóch poprzednich metod ta nie wymaga zbyt wiele myślenia. Nie musimy sprawdzać, czy któreś pole może być wykorzystane przez atak na mechanizm serializacji, ani nie trzeba dodawać testów spójności do mechanizmów deserializacji.

Istnieje również inny aspekt, z powodu którego wzorec pośrednika serializacji jest lepszy od kopiowania defensywnego. Wzorec pośrednika serializacji pozwala na to, aby obiekty deserializowane były innej klasy niż oryginalne serializowane obiekty. Może się wydawać, że nie jest to w praktyce przydatne, ale w rzeczywistości jest.

Weźmy pod uwagę EnumSet (temat 36.). Klasa ta nie posiada konstruktorów publicznych, a jedynie statyczne metody fabryczne. Z perspektywy klienta zwracają one obiekty EnumSet, ale w rzeczywistości w aktualnej implementacji OpenJDK zwracają jedną z dwóch klas pochodnych, w zależności od wielkości bazowego typu wyliczeniowego. Jeżeli bazowy typ wyliczeniowy posiada nie więcej niż sześćdziesiąt cztery elementy, statyczne metody fabryczne zwracają RegularEnumSet, a w przeciwnym wypadku zwracają JumboEnumSet.

Teraz sprawdźmy, co się zdarzy, jeżeli będziemy serializować zbiór wyliczeniowy posiadający sześćdziesiąt elementów, a następnie dodamy kolejnych pięć elementów i zdeserializujemy ten zbiór. W czasie serializacji był to RegularEnumSet, ale w czasie deserializacji powinien być to obiekt JumboEnumSet. Faktycznie tak się dzieje, ponieważ EnumSet korzysta z wzorca pośrednika serializacji. Dla zainteresowanych zamieszczamy pośrednika serializacji klasy EnumSet. Jest on bardzo prosty:

// Pośrednik serializacji EnumSet

```
private static class SerializationProxy <E extends Enum<E>>
    implements Serializable {
    // typ elementu tego zbioru wyliczeniowego
    private final Class<E> elementType;

    // elementy znajdujące się w zbiorze wyliczeniowym
    private final Enum<?>[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(new Enum<?>[0]);
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum<?> e : elements)
            result.add((E)e);
        return result;
    }
    private static final long serialVersionUID =
        362491234563181265L;
}
```

Wzorec pośrednika serializacji ma dwa ograniczenia. Nie jest zgodny z klasami, które są rozszerzalne przez swoje klienty (temat 19.). Dodatkowo nie jest zgodny z niektórymi klasami, których grafy obiektów zawierają cykle — jeżeli będziemy

próbować metodę na obiekcie na podstawie metody `readResolve` pośrednika serializacji, otrzymamy `ClassCastException`, ponieważ nie mamy jeszcze obiektu, a tylko jego pośrednika serializacji.

Na koniec dodatkowe możliwości i bezpieczeństwo wzorca pośrednika serializacji nie są darmowe. Na moim komputerze serializacja i deserializacja obiektów `Period` jest o 14 procent kosztowniejsza w przypadku użycia pośrednika serializacji niż w przypadku kopiowania defensywnego.

Podsumujmy. Warto wziąć pod uwagę wzorzec pośrednika serializacji, jeżeli będziemy musieli napisać metody `readObject` lub `writeObject` w klasach, które nie będą rozszerzane przez klienty. Wzorzec ten jest najprawdopodobniej najbardziej solidną metodą serializacji obiektów ze złożonymi niezmiennikami.

Tematy odpowiadające drugiemu wydaniu

Numer tematu w drugim wydaniu	Numer tematu w trzecim wydaniu, tytuł
1.	1. Tworzenie statycznych metod fabrycznych zamiast konstruktorów
2.	2. Zastosowanie budowniczego do obsługi wielu parametrów konstruktora
3.	3. Wymuszanie właściwości singleton za pomocą prywatnego konstruktora
4.	4. Wykorzystanie konstruktora prywatnego w celu uniemożliwienia utworzenia obiektu
5.	6. Unikanie powielania obiektów
6.	7. Usuwanie niepotrzebnych referencji do obiektów
7.	8. Unikanie finalizatorów i oczyszczaczy
8.	10. Zachowanie założeń w trakcie przeddefiniowywania metody equals
9.	11. Przeddefiniowywanie metody hashCode wraz z equals
10.	12. Przeddefiniowywanie metody toString
11.	13. Rozsądne przeddefiniowywanie metody clone
12.	14. Implementacja interfejsu Comparable
13.	15. Ograniczanie dostępności klas i ich składników
14.	16. Stosowanie metod akcesorów zamiast pól publicznych w klasach publicznych

Numer tematu w drugim wydaniu	Numer tematu w trzecim wydaniu, tytuł
15.	17. Zapewnianie niezmienności obiektu
16.	18. Zastępowanie dziedziczenia kompozycją
17.	19. Projektowanie i dokumentowanie klas przeznaczonych do dziedziczenia
18.	20. Stosowanie interfejsów zamiast klas abstrakcyjnych
19.	22. Wykorzystanie interfejsów jedynie do definiowania typów
20.	23. Zastępowanie oznaczania klas hierarchią
21.	42. Stosuj lambdy zamiast klas anonimowych
22.	24. Zalety stosowania statycznych klas składowych
23.	26. Nie korzystaj z typów surowych
24.	27. Eliminowanie ostrzeżeń o braku kontroli
25.	28. Korzystanie z list zamiast tablic
26.	29. Stosowanie typów ogólnych
27.	30. Stosowanie metod ogólnych
28.	31. Zastosowanie związanych szablonów do zwiększania elastyczności API
29.	33. Wykorzystanie heterogenicznych kontenerów bezpiecznych dla typów
30.	34. Użycie typów wyliczeniowych zamiast stałych int
31.	35. Użycie pól instancyjnych zamiast kolejności
32.	36. Użycie EnumSet zamiast pól bitowych
33.	37. Użycie EnumMap zamiast indeksowania kolejnością
34.	38. Emulowanie rozszerzalnych typów wyliczeniowych za pomocą interfejsów
35.	39. Korzystanie z adnotacji zamiast wzorców nazw
36.	40. Spójne użycie adnotacji Override
37.	41. Użycie interfejsów znacznikowych do definiowania typów
38.	49. Sprawdzanie poprawności parametrów
39.	50. Defensywne kopiowanie
40.	51. Projektowanie sygnatur metod
41.	52. Rozsądne korzystanie z przeciążania
42.	53. Rozsądne korzystanie z metod varargs
43.	54. Zwracanie pustych tablic lub kolekcji zamiast wartości null
44.	56. Tworzenie komentarzy dokumentujących dla udostępnianych elementów API
45.	57. Ograniczanie zasięgu zmiennych lokalnych
46.	58. Stosowanie pętli for-each zamiast tradycyjnych pętli for
47.	59. Poznanie i wykorzystywanie bibliotek

Numer tematu w drugim wydaniu	Numer tematu w trzecim wydaniu, tytuł
48.	60. Unikanie typów float i double, gdy potrzebne są dokładne wyniki
49.	61. Stosowanie typów prostych zamiast opakowanych typów prostych
50.	62. Unikanie typu String, gdy istnieją bardziej odpowiednie typy
51.	63. Problemy z wydajnością przy łączeniu ciągów znaków
52.	64. Odwoływanie się do obiektów poprzez interfejsy
53.	65. Stosowanie interfejsów zamiast refleksyjności
54.	66. Rozważne wykorzystywanie metod natywnych
55.	67. Unikanie przesadnej optymalizacji
56.	68. Wykorzystanie ogólnie przyjętych konwencji nazewnictwa
57.	69. Wykorzystanie wyjątków w sytuacjach nadzwyczajnych
58.	70. Stosowanie wyjątków przechwytywanych i wyjątków czasu wykonania
59.	71. Unikanie niepotrzebnych wyjątków przechwytywanych
60.	72. Wykorzystanie wyjątków standardowych
61.	73. Zgłaszanie wyjątków właściwych dla abstrakcji
62.	74. Dokumentowanie wyjątków zgłaszanych przez metodę
63.	75. Udostępnianie danych o błędzie
64.	76. Zachowanie atomowości w przypadku błędu
65.	77. Nie ignoruj wyjątków
66.	78. Synchronizacja dostępu do wspólnych modyfikowalnych danych
67.	79. Unikanie nadmiarowej synchronizacji
68.	80. Stosowanie wykonawców, zadań i strumieni zamiast wątków
69.	81. Stosowanie narzędzi współbieżności zamiast wait i notify
70.	82. Dokumentowanie bezpieczeństwa dla wątków
71.	83. Rozsądne korzystanie z późnej inicjalizacji
72.	84. Nie polegaj na harmonogramie wątków
73.	(usunięty)
74.	85. Stosuj rozwiązania alternatywne wobec serializacji Javy
	86. Rozsądne implementowanie interfejsu Serializable
75.	85. Stosuj rozwiązania alternatywne wobec serializacji Javy
	87. Wykorzystanie własnej postaci serializowanej
76.	85. Stosuj rozwiązania alternatywne wobec serializacji Javy
	88. Defensywne tworzenie metody readObject

Numer tematu w drugim wydaniu	Numer tematu w trzecim wydaniu, tytuł
77.	85. Stosuj rozwiązania alternatywne wobec serializacji Javy 89. Stosowanie typów wyliczeniowych zamiast readResolve do kontroli obiektów
78.	85. Stosuj rozwiązania alternatywne wobec serializacji Javy 90. Użycie pośrednika serializacji zamiast serializowanych obiektów

B

Zasoby

- [Asserts] *Programming with Assertions*, Sun Microsystems, 2002,
<http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>.
- [Beck04] Beck Kent, *JUnit Pocket Guide*, O'Reilly Media, Inc., Sebastopol CA, 2004, ISBN: 0596007434.
- [Bloch01] Bloch Joshua, *Effective Java™ Programming Language Guide*, Addison-Wesley, Boston 2001, ISBN: 0201310058.
- [Bloch05] Bloch Joshua i Gafter Neal, *Java™ Puzzlers: Traps, Pitfalls, and Corner Cases*, Addison-Wesley, Boston 2005, ISBN: 032133678X.
- [Blum14] Blum Scott, *Faster RSA in Java with GMP*, „The Square Corner” (blog), 14 lutego 2014, <https://medium.com/square-corner-blog/faster-rsa-in-java-with-gmp-8b13c51c6ec4>.
- [Bracha04] Bracha Gilad, *Lesson: Generics*, dodatek online do „The Java Tutorial: A Short Course on the Basics”, 6th ed., Addison-Wesley, Upper Saddle River NY, 2014,
<https://docs.oracle.com/javase/tutorial/extra/generics/>.
- [Burn01] Oliver Burn, „Checkstyle”, 2001 – 2007,
<http://checkstyle.sourceforge.net>.
- [Coekaerts15] Coekaerts Wouter (@WouterCoekaerts), *Billion-laughstyle DoS for Java serialization*
[https://gist.github.com/coekie/a27cc406fc9f3dc7a70d ...](https://gist.github.com/coekie/a27cc406fc9f3dc7a70d...)
WONTFIX, Twitter, 9 listopada 2015, 9:46,
<https://twitter.com/woutercoekaerts/status/663774695381078016>.

- [CompSci17] *Brief of Computer Scientists as Amici Curiae for the United States Court of Appeals for the Federal Circuit*, Case No. 17-1118, Oracle America, Inc. kontra Google, Inc., 2017.
- [Dagger] *Dagger*, Square, Inc., 2013, <http://square.github.io/dagger/>.
- [Gallagher16] Gallagher Sean, *Muni system hacker hit others by scanning for year-old Java vulnerability*, „Ars Technica”, 29 listopada 2016, <https://arstechnica.com/information-technology/2016/11/san-francisco-transit-ransomware-attacker-likely-used-year-old-java-exploit/>.
- [Gamma95] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995, ISBN: 0201633612.
- [Goetz07] Goetz Brian, Peierls Tim i inni, *Java. Współbieżność dla praktyków*, Helion, Gliwice, 2007, ISBN: 8324609210.
- [Gosling97] Gosling James, „The Feel of Java”, *Computer* 30, nr 6, czerwiec 1997, s. 53 – 57, <http://dx.doi.org/10.1109/2.587548>.
- [Guava] Guava, Google Inc., 2017, <https://github.com/google/guava>.
- [Guice] Guice, Google Inc., 2016, <https://github.com/google/guice>.
- [Herlihy12] Herlihy Maurice i Nir Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, Morgan Kaufmann Publishers, Waltham, MA, 2012, ISBN: 0123973376.
- [Jackson75] Jackson M.A., *Principles of Program Design*, Academic Press, London 1975, ISBN: 0123790506.
- [Java-secure] *Secure Coding Guidelines for Java SE*, Oracle, 2017, <http://www.oracle.com/technetwork/java/seccodeguide-139067.html>.
- [Java8-feat] *What's New in JDK 8*, Oracle, 2014, <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>.
- [Java9-feat] *Java Platform, Standard Edition What's New in Oracle JDK 9*, Oracle, 2017, <https://docs.oracle.com/javase/9/whatsnew/toc.htm>.

- [Java9-api] Java Platform, Standard Edition & Java Development Kit Version 9 API Specification, Oracle, 2017, <https://docs.oracle.com/javase/9/docs/api/overview-summary.html>.
- [Javadoc-guide] *How to Write Doc Comments for the Javadoc Tool*, Sun Microsystems 2000 – 2004, <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>.
- [Javadoc-ref] *Javadoc Reference Guide*, Oracle, 2014 – 2016, <https://docs.oracle.com/javase/9/javadoc/javadoc.htm>.
- [JLS] Gosling James, Joy Bill, Steele Guy, Bracha Gilad, *The Java™ Language Specification, JAVa SE 8 Edition*, Addison-Wesley, Boston, 2014, ISBN: 013390069X.
- [JMH] *Code Tools: jmh*, Oracle, 2014, <http://openjdk.java.net/projects/code-tools/jmh/>.
- [JSON] *Introducing JSON*, Ecma International, 2013, <https://www.json.org>.
- [Kahan91] Kahan William, Thomas J.W., *Augmenting a Programming Language with Complex Arithmetic*, UCB/CSD-91-667, University of California, Berkeley, 1991.
- [Knuth74] Knuth Donald, *Structured Programming with go to Statements*, „Computing Surveys” 6 (1974), s. 261 – 301.
- [Lea14] Lea Doug, *When to use parallel streams*, 2014, <http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html>.
- [Lieberman86] Lieberman Henry, *Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems*, „Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications”, s. 214 – 223, ACM Press, Portland, wrzesień 1986.
- [Liskov87] Liskov B., *Data Abstraction and Hierarchy*, „Addendum to the Proceedings of OOPSLA '87 and SIGPLAN Notices”, Vol. 23, No. 5: 17 – 34, maj 1988.
- [Naftalin07] Naftalin Maurice i Philip Wadler, *Java Generics and Collections*, O'Reilly Media, Inc., Sebastopol, CA, 2007, ISBN: 0596527756.

- [Parnas72] Parnas D.L., *On the Criteria to Be Used in Decomposing Systems into Modules*, „Communications of the ACM” 15, s. 1053 – 1058, 1972.
- [POSIX] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application: Program Interface (API) [C Language]* (ANSI), IEEE Standards Press, ISBN: 1559375736.
- [Protobuf] Protocol Buffers, Google Inc., 2017, <https://developers.google.com/protocol-buffers>.
- [Schneider16] Schneider Christian, *SWAT (Serial Whitelist Application Trainer)*, 2016, <https://github.com/cschneider4711/SWAT/>.
- [Seacord17] Seacord Robert, *Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS)*, NCC Group Whitepaper, San Francisco, 2017, https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2017/june/ncc_group_combating_java_deserialization_vulnerabilities_with_look-ahead_object_input_streams1.pdf.
- [Serialization] *Java™ Object Serialization Specification*, Sun Microsystems, marzec 2005, <http://docs.oracle.com/javase/9/docs/specs/serialization/index.html>.
- [Sestoft16] Sestoft Peter, *Java Precisiely, Third Edition*, The MIT Press, Cambridge, MA, 2016, ISBN 0262529076.
- [Shipilëv16] Aleksey Shipilëv, *Arrays of Wisdom of the Ancients*, 2016, <https://shipilev.net/blog/2016/arrays-wisdom-ancients/>.
- [Smith62] Smith Robert, *Algorithm 116 Complex Division*, „Communications of the ACM”, 5.8 (sierpień 1962), s. 435.
- [Snyder86] Synder Alan, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, „Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings”, 38 – 45, ACM Press, 1986.
- [Spring] Spring Framework, Pivotal Software, Inc., 2017, <https://projects.spring.io/spring-framework/>.

- [Stroustrup] Stroustrup Bjarne, *Is Java the language you would have designed if you didn't have to be compatible with C?*, „Bjarne Stroustrup's FAQ”, około 2000, aktualizacja, 1 października 2017, http://www.stroustrup.com/bs_faq.html#Java.
- [Stroustrup95] Stroustrup Bjarne, *Why C++ is not just an object-oriented programming language*, „Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications”, redakcja: Steven Craig Bilow i Patricia S. Bilow, ACM, Nowy Jork, NY, 1995, <http://dx.doi.org/10.1145/260094.260207>.
- [Svoboda16] Svoboda David, *Exploiting Java Serialization for Fun and Profit*, Software Engineering Institute, Carnegie Mellon University, 2016, <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484347>.
- [Thomas94] Thomas Jim, Coonen Jerome T., *Issues Regarding Imaginary Types for C and C++*, „The Journal of C Language Translation”, 5.3 (marzec 1994), s. 134 – 138.
- [ThreadStop] *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*, Sun Microsystems, 1999, <https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>.
- [Viega01] Viega John i Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, Boston, 2001, ISBN: 020172152X.
- [W3C-validator] *W3C Markup Validation Service*, World Wide Web Consortium, 2007, <http://validator.w3.org>.
- [Wulf72] Wulf W., „A Case Against the GOTO”, *Proceedings of the 25th ACM National Conference 2* (1972): s. 791 – 797.

Skorowidz

A

abstrakcja, 323, 346
adapter, 43, 122, 133
adnotacja, 21, 200
 @FunctionalInterface, 223
 @Nullable, 248
 @Override, 207, *Patrz*
 @SuppressWarnings, 145, 146, 147
Override, 209
powtarzalna, 205, 206, 207
procesor, *Patrz* procesor
 adnotacji
Retention, 201
SafeVarargs, 149, 167, 169
Target, 201
znacznikowa, 210, 211
antywzorzec, 20
API, 22
 dokumentacja, 272, 273, 275, 276, 278
 dostępu do usług, 26
 klient, 22
 rejestracji dostawcy, 26
 użytkownik, 22
asercja, 249
atak
 DoS, 362
 finalizacji, 50
 poprzez deserializację, 363
atomowość działań, 104
autoboxing, 44, 293, 295
auto-unboxing, 293, 294

B

baza danych, 44
bezpieczeństwo, 50, 361
biblioteka
 Apache Commons, 125
 AutoValue, 68, 70, 72, 75, 78
 Collections, 25, 121
 Dagger, 41
 dostawcy usług, 26, 303
 Guava, 112, 290
 Guice, 41
 Java Database Connectivity
 API, *Patrz*: biblioteka
 JDBC
 java.awt.Point, 106
 java.io, 21, 289, 301
 java.lang, 289
 java.util, 21, 289, 310, 343
 java.util.Collections, 174
 java.util.concurrent, 242, 289, 290, 343, 345, 347
 java.util.concurrent.
 ↳atomic, 310
 java.util.Date, 106
 java.util.function, 216, 220
JDBC, 26, 27
korzystająca z
 interfejsów, 25
nazwa, 309
Spring, 41

standardowa, 287, 288, 289
wstrzykiwania zależności, 41
blok
 catch, 318, 331
 wieloelementowy, 341
 try z zasobami, 49, 55, 56
 try-catch, 80, 282, 314
 try-finally, 49, 54, 55
błąd, 316, 317, 327, 328
bomba deserializacyjna, 362, 364
bufor, 47, 48

C

ciąg znaków, 296,
 Patrz też: typ String
 łączenie, 298, 299
covariant return typing,
 Patrz: kowariancja
 zwracanych typów
czas kontroli/czas użycia,
 Patrz: TOCTOU

D

dane
 lista biała/czarna, 364
 niezaufane, 364
 strukturalne reprezentacja
 wieloplatformowa, 363, 365

deklaracja eksportu, 97
 dekorator, 112
 delegacja, 112
 deserializacja, 117, 361, 362,
 363, 366, 381
 niezaufanych danych, 364
 destruktor, 48
 dostęp, 22
 dostępność, 94
 pola, 96
 poziom
 niejawny, 97
 prywatny, 94, 95, 99
 prywatny w ramach
 pakietu, 22, 95
 publiczny, 22, 94, 95,
 97, 99
 zabezpieczony, 95, 97
 zmiana na czas
 testowania, 95
 tablicy, 96
 dziedziczenie, 22, 27, 107,
 109, 110, 112, 113, 114,
 118, 275, 367
 implementacji, 107
 interfejsu, 107
 jednobazowe, 119
 wielobazowe, 123

E

efekt uboczny, 273
 eksplozja typów, 121
 Executor Framework, 345,
 346, 359
 exploit, 361

F

fabryka, 41
 kopiująca, 85, 86
 singletonów uogólnionych,
 157, 158
 Factory Method, *Patrz:*
 metoda fabryczna
 finalizator, 48, 49, 50, 51
 funkcja
 czysta, 231
 Function.identity, 157, 218

mieszająca, 74
 skrótów, 363
 tożsamościowa, 157
 wydobywająca klucz, 91

G

gadżet, 362
 Gosling James, 11

H

hermetyzacja, 93, 98, 107, 114
 hierarchia
 klas, 120, 130, 131
 napęczniała, 120
 typów, 120

I

IDE, 70
 idiom pojedynczego
 sprawdzenia, 357, 358
 iloczyn kartezjański, 230
 interfejs, 21, 22, 119, 126, 299
 Autocloseable, 120
 AutoCloseable, 55
 BiConsumer, 221, 340
 BiFunction, 221
 BinaryOperator, 221, 233
 BiPredicate, 221
 BooleanSupplier, 221
 CharSequence, 262
 Cloneable, 78, 79, 85, 86,
 117
 Collection, 125, 236, 238
 Comparable, 86, 87, 89, 90,
 92, 120, 158
 Comparator, 213, 222
 Consumer, 221
 dostawcy usług, 26
 dostępność, *Patrz:*
 dostępność
 DoubleBinaryOperator,
 216
 EldestEntryRemovalFunc-
 tion, 223
 Function, 218, 221

funkcyjny, 214, 220, 222
 implementacja, 22, 119,
 122, 127
 przez klasę, 95
 szkieletowa, 118, 323
 szkieletowa abstrakcyjna,
 121
 Iterable, 120, 236
 jako typ, 300
 java.io.ObjectStreamConsta-
 nts, 128
 java.util.Collections, 38
 java.util.List, 255
 JNI, 304
 List, 139, 236, 323
 Map, 43
 mieszany, 78, 120
 nazwa, 310, 311
 ogólny, 139
 Predicate, 220, 221
 Serializable, 95, 106, 117,
 118, 210, 362, 365, 366,
 367, 368, 381, 385
 Set, 190, 211, 236, 299,
 302
 składowy, 21
 stałych, 127, 128
 Stream, 237
 Supplier, 221
 ToIntBiFunction, 222
 UnaryOperator, 221
 usługi, 26
 znacznikowy, 210

J

Java wersja, 21
 Java Native Interface, *Patrz:*
 interfejs JNI
 Javadoc, 114, 272, 274, 278,
 325, 352
 konwencje, 273
 JavaDoc, 121
 język bezpieczny, 250
 JSON, 364

K

- klasa, 21, 22
 - AbstractSet, 58
 - abstrakcyjna, 33, 118, 119, 121, 216
 - anonimowa, 122, 132, 134, 213, 216
 - Arrays, 88
 - AtomicLong, 337
 - BigDecimal, 76, 100, 105
 - BigInteger, 76, 100, 103, 104, 105, 248, 300, 305, 368
 - Boolean, 23
 - CaseInsensitiveString, 67
 - Collections, 88, 156
 - Collectors, 232, 233, 236
 - Comparator, 91
 - Complex, 107
 - ConcurrentHashMap, 347, 348
 - CountDownLatch, 107, 348
 - CyclicBarrier, 348
 - czas kontroli/czas
 - użycia, *Patrz:* TOCTOU
 - Date, 252
 - Dimension, 99, 307
 - dokumentacja, 113, 114, 115, 116, 118, 124
 - dostępność, *Patrz:*
 - dostępność
 - EnumMap, 192, 193, 194, 196
 - EnumSet, 26, 190
 - Error, 317, 322
 - Exception, 317, 322
 - Exchanger, 348
 - Favorites, 171, 172, 173
 - File, 69
 - FileInputStream, 51
 - FileOutputStream, 51
 - HashMap, 57
 - HashSet, 57, 108, 109
 - Hashtable, 109, 112, 343
 - hierarchia, 65,
 - Patrz:* hierarchia klas
 - implementacja
 - domyślna, 125
 - prosta, 124
 - szkieletowa abstrakcyjna, 121, 122, 123, 124
 - InputStream, 54
 - instancja, 21
 - Instant, 252
 - InstrumentedHashSet, 109
 - Iterator, 315
 - java.awt.Component, 307
 - java.lang.ref, 48
 - java.sql.Connection, 51, 54
 - java.sql.Timestamp, 65
 - java.util.AbstractList, 115
 - java.util.concurrent.Executors, 345
 - java.util.Date, 65
 - java.util.LinkedHashMap, 48
 - java.util.logging, 324
 - java.util.regex.Pattern, 58
 - kontenerowa, 59
 - kontrolowana przez
 - instancje, 24, 59
 - LinkedHashMap, 219, 220
 - LinkedHashSet, 299
 - literal, 143
 - LocalDateTime, 252
 - lokalna, 132, 134
 - mieszana, 120
 - narzędziowa, 38
 - nazwa, 310, 311
 - nieostateczna, 50
 - nietworząca obiektów, 25
 - niezmienna, 24, 31, 100, 103
 - alternatywa, 104, 105
 - niezwiązana, 262
 - Object, 57, 75, 121
 - ObjectOutputStream, 259
 - Objects, 73
 - ogólna, 139
 - Optional, 271
 - OptionalDouble, 271
 - OptionalInt, 271
 - OptionalLong, 271
 - osłonowa, 23, 112, 118, 121, 123, 125
 - ostateczna, 50
 - OutputStream, 54, 301
 - oznaczana, 130
 - Phaser, 348, 350
 - podrzędna, 22
 - tworzenie, *Patrz:*
 - dziedziczenie
 - Point, 99
 - pomocnicza, 256
 - ProcessHandle, 270, 271
 - projektowanie, 113, 114, 115, 116, 118
 - Properties, 112
 - przekazująca, 110
 - Random, 288
 - RegularEnumSet, 26
 - równoważności, 60
 - RuntimeException, 317, 322
 - RunTimeException, 317
 - Semaphore, 348
 - serializowana, 365
 - postać domyślna, 365, 369, 370, 373, 374, 376
 - postać własna, 365, 367, 369, 374
 - singleton, 25
 - składowa, 21, 25
 - niestatyczna, 132, 133, 134
 - statyczna, 132, 134, 368
 - SplittableRandom, 288
 - Stack, 47, 80, 81, 112
 - String, 104, 154, 262, 263, 300
 - StringBuffer, 104, 343
 - StringBuilder, 104
 - StringList, 373
 - SynchronizedCollection, 125, 126
 - Thread, 58, 346
 - ThreadLocalRandom, 246, 288
 - ThreadPoolExecutor, 51, 345

klasa
 Throwable, 318, 322, 367
 TreeMap, 88
 TreeSet, 88
 Types, 25
 uogólnianie, 152
 użytkowa, 128
 Vector, 109, 112, 343
 WeakHashMap, 47, 48
 wewnętrzna, 132
 wymagania
 implementacyjne, 114, 115
 zagnieżdżona, 132, 133, 135, 368
 zdefiniowanych
 wartościach, 58
 ZonedDateTime, 252
 klasaLOptional, 268
 klucz, 297
 mapowania, 103
 mieszający, 71, 72, 73, 106
 sparametryzowany, 171
 kolejka zadań, 344, 345, 348
 kolekcja, 271
 checkedList, 174
 checkedMap, 174
 checkedSet, 174
 Collection, 174
 CopyOnWriteArrayList, 342
 deklaracja, 140, 151
 filtrowanie destrukcyjne, 286
 HashMap, 70, 71
 HashSet, 70
 iteracja równoległa, 286
 Map, 171
 Set, 171
 synchronizowana, 348
 współbieżna, 342, 347, 348
 kolekcjoner, 232, 233, 234, 236
 wartości, 235
 komentarz
 doc, 272
 dokumentujący, 272, 273, 275, 276, 278
 komparator, 90, 293

komponent, 20
 kopiowanie defensywne, 253, 254
 kompozycja, 110, 112, 113
 komunikat
 ClassCastException, 140
 konstruktor, 22, 24, 28
 bezparametrowy, 30
 domyślny, 38
 konwertujący, 86
 kopiujący, 85, 86, 103
 obsługujący łączenie, 324
 publiczny, 23
 teleskopowy, 28, 30
 kontener heterogeniczny, 172
 kopiowanie defensywne, 44, 253, 254
 kowariancja zwracanych typów, 34

L

lambda, 91, 125, 134, 214, 215, 216, 217, 227, 231, 262, 340
 serializacja, 217
 liczba
 całkowita przepełnienie, 92
 losowa, 287, 288
 Mersenne'a, 229, 243
 pierwsza, 229
 pseudolosowa, 344
 zespolona, 102
 lista
 transformacja, 286
 właściwości, 112
 literal klasy, 143

M

mapowanie, 103
 mechanizm
 kontroli dostępu, 94
 zacierania, *Patrz:*
 zacieranie
 metaadnotacja, 201
 @Repeatable, 205

metoda, 21
 action, 320
 actionPermitted, 320
 add, 108, 109
 addAll, 108, 109
 addObserver, 339
 addTopping, 35
 agregująca, 27
 akcesora, 98, 99, 102
 apply, 215
 argument, 263, 264, 265
 arguments, 271
 Arrays.asList, 238
 asSubclass, 175
 atomowa w przypadku błędu, 328, 329
 averaging, 235
 awaitTermination, 345
 binarySearch, 156
 build, 31, 32, 34
 cast, 173
 checkFromIndexSize, 249
 checkFromToIndex, 249
 checkIndex, 249
 classify, 257
 clear, 115
 clone, 57, 78, 79, 81, 82, 85, 86, 103, 117, 252
 close, 51, 54, 55
 collect, 244
 collectingAndThen, 236
 Collections.emptyList, 266
 Collections.emptyMap, 266
 Collections.emptySet, 266
 Collections.synchronizedMap, 353, 354
 Collections.sort, 250
 Comparable.compareTo, 57
 compare, 92, 293, 294
 compareTo, 86, 87, 88, 89, 92
 tworzenie, 89
 comparing, 91, 232
 comparingInt, 91
 computeIfAbsent, 225
 contentEquals, 262
 counting, 235

- create, 28
- defaultReadObject, 377
- defaultWriteObject, 372, 373
- dokumentacja, 273, 275, 276, 277, 372
- domyślna, 121, 122, 124, 125, 126
- Double.compare, 67
- equals, 57, 58, 59, 62, 66, 89
 - generowanie, 70
 - przesłanianie, 69, 70, 71, 75, 208
 - relacja równoważności, 59, 60, 63
 - wydajność, 68
- Executors.newCached
 - ↳ ThreadPool, 345
- Executors.newFixed
 - ↳ ThreadPool, 345
- fabryczna, 41, 86
 - konwertująca, 86
 - statyczna, 23, 24, 25, 26, 27, 28, 36, 42, 134, 157
 - uogólniona, 37
- filter, 270
- filtering, 235
- finalize, 57, 367
- flatMap, 270, 271
- flatMapMapping, 235
- Float.compare, 67
- from, 27
- fromString, 184, 185
- get, 98
- getAnnotationsByType, 206
- getFavorite, 173
- getFavorites, 173
- getSize, 307
- getType, 28
- groupingBy, 234, 235, 236
- groupingByConcurrent, 235
- hashCode, 57, 69, 70, 71, 106
 - generowanie, 70
 - opóźniona inicjalizacja, 74, 106
 - przesłanianie, 69, 70, 71, 75, 208
- hasNext, 315
- ifPresent, 270
- ifPresentOrElse, 270
- instance, 27
- instanceOf, 27
- instancyjna, 218
- invokeAll, 345
- invokeAny, 345
- isAnnotationPresent, 202, 206
- isPresent, 270
- joining, 236
- keySet, 43
- konstruuująca komparator, 90, 214
- konwersji typu, 27
- List.of, 170
- łańcuch, 33
- macierzysta, 51
- map, 270
- mapping, 235
- maxBy, 234, 236
- minBy, 236
- mutatora, 98, 100
- natywna, 250, 304, 305
- nazwa, 255, 310, 311, 312
- newInstance, 28
- newType, 28
- next, 315
- nextInt, 288
- notify, 346, 348, 350, 351
- notifyAll, 350, 351
- ObjectInputStream.
 - ↳ readFields, 366
- ObjectInputStream.
 - ↳ readUnshared, 106
- ObjectOutputStream.
 - ↳ putFields, 366
- ObjectOutputStream.write, 210
- ObjectOutputStream.
 - ↳ writeUnshared, 106
- Objects.requireNonNull, 248
- of, 27
- ogólna, 143, 144, 156, 214
- Optional.empty, 269
- Optional.of, 269
- Optional.ofNullable, 269
- or, 270
- ordinal, 188, 191
- orElseGet, 270
- parallel, 242, 243
- parametr, 255
 - boolean, 256
 - kopiowanie defensywne, 252, 253, 254
 - sprawdzanie poprawności, 247, 249, 252
- partitioningBy, 235
- pomocnicza, 255
- printf, 75
- println, 75, 117
- przeciążanie, 69, 257, 258, 259, 260, 261, 262
- przekazująca, 110, 112
- przesłanianie, 69, 95, 113, 258, 259
- putFavorite, 173
- putIfAbsent, 347
- random, 288
- readObject, 50, 106, 117, 362, 380, 381
 - defensywna, 375, 379, 380
- readObjectNoData, 368
- readResolve, 37, 50, 106, 118, 381, 382, 384
- reducing, 235
- referencja, 217, 218
 - niezwiązana, 218, 219
 - statyczna, 219
 - związana, 218, 219
- remove, 114, 260, 282
- removeEldestEntry, 48, 219
- removeIf, 121, 125, 126
- removeObserver, 339
- removeRange, 115
- reversed, 233
- Runtime.runFinalizers
 - ↳ OnExit, 50
- self, 33
- set, 98

metoda

sort, 156
 specyficzna dla stałych,
 183, 185, 186
 spliterator, 244
 stream, 271
 Stream.of, 239
 summarizing, 235
 summing, 235
 super.clone, 84, 85
 Supplier, 270
 sygnatura, 21
 System.currentTimeMillis,
 349
 System.gc, 50
 System.nanoTime, 349
 System.runFinalization, 50
 System.runFinalizersOnExit,
 50
 szablonowa, 121, 219
 take, 348
 thenComparingInt, 91
 Thread.stop, 334
 toArray, 146
 toList, 236
 toMap, 233, 234, 236
 toSet, 236
 toString, 57, 75, 76, 181,
 184
 toUpperCase, 154
 transferTo, 289
 type, 28
 union, 157
 valueOf, 27, 184, 263
 values, 181
 varargs, 149, 263, 264, 265
 wait, 346, 348, 350
 writeReplace, 118
 wywołanie
 otwarte, 343
 zwrotne, 48
 migracja zgodność, *Patrz:*
 zgodność migracji
 moduł, 97
 nazwa, 311
 modyfikator
 public, 94
 static, 133

synchronized, 333, 337,
 352
 transient, 371, 382
 volatile, 336, 337

N

naruszenie założenia, 317
 nasłuch, 48
 notacja znak-moduł, 103
 null, 46, 47, 66, 248, 266, 267,
 293

O

obiekt, 21
 bezstanowy, 36
 blokady prywatnej, 354,
 355
 budowniczego, 31, 32
 Class, 192, 301
 Constructor, 301
 defensywne tworzenie
 kopii, 103
 efektywnie niezmienny, 338
 Favorites, 172, 173
 Field, 301
 funkcyjny, 134, 213, 215,
 217
 JumboEnumSet, 26
 macierzysty zewnętrzny,
 51
 Map, 43
 Method, 301
 niemożliwy, 376
 niespójny, 31
 niezmienny, 100, 102, 103,
 329
 alternatywa, 104, 105
 współdzielenie, 103
 ObjectInputStream, 362
 opcjonalny, 267, 269, 271,
 272, 315
 postać kanoniczna, *Patrz:*
 postać kanoniczna
 powielanie, 41
 procesowy, 134

przetrzymywanie
 nieświadome, 46
 RegularEnumSet, 26
 Runnable, 345
 Stack, 155
 tworzenie, 23, 103
 w stanie niespójnym, 31
 zamrożenie, 31
 oczyszczacz, 48, 49, 51, 53
 odwzorowanie, 192, 271
 OpenJDK Server, 335
 operacja
 blokująca, 348
 forEach, 231, 232, 244
 forEachOrdered, 244
 kończąca, 224
 pośrednia, 224
 operator
 +, 298, 299
 ==, 67, 294
 instanceof, 66, 67, 143
 rombu, 145
 optymalizacja, 306, 308

P

pakiet, 97
 java.awt, 99
 nazwa, 309
 pamięć
 podręczna, 47
 wyciek, 46, 47, 48
 zarządzanie, *Patrz:*
 zarządzanie pamięcią
 parametr
 typu, 144, 156, 164
 jawni, 163
 nazwa, 311
 rekurencyjny, 33
 związany, 155
 varargs, 166, 167, 168
 PECS, 162, 164
 pętla
 for, 282, 283
 for-each, 282, 283, 285
 ograniczenia, 286
 zagnieżdżona, 285
 while, 282, 283

plik

dekryptor, 49
 module-info.java, 97, 278
 package-info.java, 278
 źródłowy, 135

pole, 21

bitowe, 189, 190, 191
 inicjalizacja, 356, 358
 późna, 355, 356, 357
 instancyjne
 dostępność, 96
 final, 96
 nazwa, 310, 311, 312
 stałej, 310, 311
 statyczne, 25
 dostępność, 96
 final, 96, 310
 znacznika, 129

porządek

leksykograficzny, 91
 naturalny, 87

postać kanoniczna, 68

pośrednik serializacji, 385,
 386, 387

problem SELF, 112

proces zbierania nieużytków,
 48, 49, 50

procesor adnotacji, 201

profiler stogu, 48

programowanie

funkcyjne, 102
 imperatywne,
 Patrz: programowanie
 proceduralne
 proceduralne, 102
 wielowątkowe,
 Patrz: współbieżność

protobuf, 364

Protocol Buffer, *Patrz:*

protobuf

przekazywanie, 110

R

refleksja, 36, 78

refleksyjność, 301, 302, 303

relacja równoważności, 59,
 60, 63

rzutowanie, 33

S

serializacja, 38, 50, 106, 210,
 278, 361, 365, 366, 373
 pośrednik, *Patrz:*
 pośrednik serializacji
 singletona, 37
 singletonu, 381, 382, 383
 siatka zabezpieczająca, 50, 51,
 53
 singleton, 36, 37, 39, 381, 382
 serializacja, *Patrz:*
 serializacja singletona
 tworzenie, 36
 uogólniony, 157, 158
 słownik, 39, 40
 stała
 import statyczny, 128
 MAX_VALUE, 128
 MIN_VALUE, 128
 nazwa, 128
 wyliczeniowa
 String, 178
 int, 178
 zmienna, 134, 178
 sterta skażenie, 154, 166
 stos, 45, 46, 112, 160
 wyjątków, 326
 strategia, 214
 Stroustrup Bjarne, 11
 strumień, 192, 224, 226, 227,
 228, 231, 232, 235, 237, 238,
 271, 284
 identyfikator unikalny,
 Patrz: UID
 potok, 224, 226, 227, 228,
 243
 redukcja, 232
 zrównoleglony, 242, 243,
 244, 245, 246, 288
 SWAT, 364
 symulowanie dziedziczenia
 wielobazowego, *Patrz:*
 dziedziczenie wielobazowe
 symulowanie
 synchronizator, 348
 system modułów, 22
 szablon typu, 163, 164

niezwiązany, 142, 143, 144
 związany, 143, 144, 157,
 159, 160, 199

Ś

środowisko programistyczne
 zintegrowane, *Patrz:* IDE

T

tablica, 21, 271, 313
 dostępność, 96
 indeks, 191, 196
 kowariancyjność, 147
 mieszająca, 112, 373
 ogólna, 153, 154
 pusta, 266
 tablic, 193
 transformacja, 286
 typu
 ogólnego, 148
 parametru typu, 148
 parametryzowanego,
 148
 uściślona, 147
 wielkość różna od zera, 96
 test wydajnościowy, 308
 TOCTOU, 252
 token typu, *Patrz:* typ token
 typ
 adnotacyjny zawierający,
 205
 ArrayList, 155
 BigDecimal, 291, 292
 BigInteger, 296
 boolean, 23, 292
 double, 290, 291, 292, 334
 eksplozja, *Patrz:* eksplozja
 typów
 enum, 38, 178
 float, 290, 296
 funkcyjny, 213
 HashMap, 155
 hierarchia, *Patrz:*
 hierarchia typów
 int, 291, 292, 296
 kowariancyjny zwracany,
 80

typ

List, 292
 long, 291, 292, 334
 nieuściślony, 148
 numeryczny, 296
 ogólny, 46, 139, 140, 143,
 144, 149, 155, 214
 niezmienniczość, 147
 tworzenie, 151
 zacieranie, 148
 parametr, *Patrz:* parametr
 typu
 związany, 155
 parametryzowany, 139,
 141, 143, 144, 295
 prosty, 44, 292, 293, 294,
 295
 opakowany, 44, 292, 293,
 294, 295
 referencyjny, 21, 177, 292
 rzutowanie, 139, 141
 dynamiczne, 173
 Stack, 155
 String, 292, 296,
Patrz też: ciąg znaków
 surowy, 139, 141, 143, 144,
 214
 szablon, 163, 164
 niezwiązany, 142, 143,
 144
 związany, 143, 144, 157,
 159, 160, 199
 tablicowy, 262
 token, 144, 171
 niezwiązany, 174
 związany, 174, 175
 uogólniony
 z rekurencyjnym
 parametrem typu, 33
 varargs, 35
 wiązanie rekurencyjne,
 158, 159
 wieloznaczny ograniczony,
 41
 własny, 33
 wnioskowanie, *Patrz:*
 wnioskowanie typów

wyliczeniowy, 59, 128,
 177, 178, 179, 182, 188,
 296
 instrukcja switch, 185,
 186, 187
 metoda, 179, 181
 Planet, 180, 181
 pole, 179
 rozszerzalny, 196, 197,
 199
 złożony, 296
 związany, 144
 token, 192, 198, 203

U

UID, 366, 374
 ukrywanie informacji,
Patrz: hermetyzacja
 uprawnienie, 297,
Patrz też: klucz

W

walidator HTML, 279
 wartość
 null, *Patrz:* null
 prosta, 21
 wątek
 atomowość, 334, 336
 bezpieczeństwo, 343, 352,
 353, 354
 bezpieczna publikacja, 338
 harmonogram, 358, 359
 odwzorowanie
 kanonizacyjne, 347
 priorytet, 360
 synchronizacja, 333
 nadmiarowa, 338
 niedostateczna, 334, 335
 wybudzenie, 351
 wzajemne wykluczanie,
 333
 zagłodzenie, 349
 zmienna współdzielona,
 334
 wektor, 112
 bitowy, 239

widok, *Patrz:* adapter
 wieloplatformowa
 reprezentacja danych
 strukturalnych, 363, 365
 właściwość, 112
 wnioskowanie typów, 214
 współbieżność, 333, 346, 348
 wstrzykiwanie zależności, 40,
 41, 302
 wyciąganie, 335
 wyciek pamięci, *Patrz:*
 pamięć wyciek
 wydajność, 20, 46, 50, 93, 243,
 289, 299, 305, 307, 314
 model, 308
 testowanie, 308
 wyjątek, 313
 ArithmeticException, 322
 ArrayIndexOutOfBoundsException
 ↳ Exception, 192, 194,
 313, 317
 AssertionError, 39, 249,
 317
 ClassCastException, 148
 ClassCastException, 87,
 88, 142, 145, 168, 173,
 250, 304, 329
 CloneNotSupportedException
 ↳ Exception, 80, 84
 ConcurrentModification
 ↳ Exception, 321, 322,
 330
 czasu wykonania, 316, 317
 dokumentowanie, 325,
 327
 ignorowanie, 330, 331
 IllegalArgumentException,
 159, 248, 268, 321, 322
 IllegalStateException, 32,
 51, 233, 321, 322
 IndexOutOfBoundsException
 ↳ Exception, 328
 IndexOutOfBoundsException, 327
 IndexOutOfBoundsException
 ↳ Exception, 248, 321,
 322
 InvalidClassException,
 366

InvocationTarget
 ↳ Exception, 202
 konstruktor obsługujący
 łączenie, 324
 metoda, 317
 nieobsłużony, 50
 nieprzechwytywany, 325, 326
 nieprzechwytywany, 316, 319, 321, 325
 NullPointerException, 47, 66, 67, 89, 117, 194, 248, 249, 268, 294, 295, 321, 322, 326
 NumberFormatException, 322
 propagacja, 318, 323, 324
 przechwytywany, 80, 316, 317, 318, 319
 ReflectiveOperationException, 304
 serializacja, 322
 StackOverflowError, 371
 stłumiony, 56
 stos, *Patrz*: stos wyjątków
 testowanie, 204
 tłumaczenie, 250
 translacja, 323, 324
 UnsupportedOperationException
 ↳ Exception, 114, 321, 322

wyliczenie, 21
 wyrażenie
 lambda, *Patrz*: lambda
 regularne, 42, 43
 wywołanie zwrotne, 112
 wzorzec
 Bridge, 27
 Builder, 31, 32, 33, 35, 256
 dekorator, *Patrz*:
 dekorator
 Flyweight, 24
 JavaBeans, 30, 31
 konstruktora
 teleskopowego, 30
 nazw, 200
 Observer, 339
 pośrednika serializacji,
 380, 385
 wstrzykiwania zależności,
 40
 wyliczeniowy
 int, 178, 189
 String, 178

Z

zacieranie, 141
 zadanie
 Callable, 346
 Runnable, 346
 zakleszczenie, 341

zarządzanie pamięcią, 45
 zasada
 podstawiania Liskov, 64, 95
 ukrywania informacji, 306
 zbiór potęgowy, 239
 zgodność migracji, 141
 zmienna
 definiowanie, 47
 lokalna, 281, 282
 nazwa, 311
 współdzielona, 334
 znacznik, 129
 @code, 273, 274
 @implSpec, 114, 121, 273, 275
 @inheritDoc, 278
 @literal, 273, 275
 @param, 273, 274
 @return, 273, 274
 @serial, 372
 @serialData, 372
 @throws, 248, 273, 274, 325
 HTML, 274

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 