



Helion



WYDANIE XIII

JAVA

Podstawy

ORACLE

Cay S. Horstmann

Tytuł oryginału: Core Java, Volume I: Fundamentals, 13th Edition

Tłumaczenie: Łukasz Piwko z wykorzystaniem fragmentów książki „Java. Techniki zaawansowane. Wydanie XI” w przekładzie Piotra Rajcy

ISBN: 978-83-289-3023-0

Ilustracja z okładki: Jon Chica/Shutterstock

Rysunek 1.1: Sourceforge

Rysunki 2.2, 3.2 - 3.5, 4.9, 5.4, 7.2, 10.5, 10.6, 11.1: Oracle Corporation

Rysunki 2.3 - 2.6, 12.2: Eclipse Foundation, Inc.

Rysunek 4.2: Violet UML Editor

Authorized translation from the English language edition, entitled CORE JAVA VOLUME I: FUNDAMENTALS, 13TH Edition 9780135328378 by Cay S. Horstmann published by Pearson Education, Inc, publishing as Oracle Press Copyright © 2025 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 © 2025.

Pearson uses AI, including to support the creation of content. We claim copyright to the fullest extent permissible by law.

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<https://ftp.helion.pl/przyklady/javp13.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

helion.pl/user/opinie/javp13

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: helion.pl (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp	15
Do Czytelnika	15
O książce	17
Konwencje typograficzne	19
Przykłady kodu	20
Podziękowania	21
Rozdział 1. Wprowadzenie do Javy	23
1.1. Java jako platforma programistyczna	23
1.2. Słowa klucze białej książki Javy	24
1.2.1. Prostota	25
1.2.2. Obiektywość	25
1.2.3. Sieciowość	26
1.2.4. Niezawodność	26
1.2.5. Bezpieczeństwo	26
1.2.6. Niezależność od architektury	27
1.2.7. Przenośność	28
1.2.8. Interpretacja	28
1.2.9. Wysoka wydajność	29
1.2.10. Wielowątkowość	29
1.2.11. Dynamiczność	30
1.3. Aplety Javy i internet	30
1.4. Krótka historia Javy	31
1.5. Główne nieporozumienia dotyczące Javy	35
Rozdział 2. Środowisko programistyczne Javy	38
2.1. Instalacja oprogramowania Java Development Kit	38
2.1.1. Pobieranie pakietu JDK	38
2.1.2. Instalacja pakietu JDK	39
2.1.3. Instalacja plików źródłowych i dokumentacji	41
2.2. Używanie narzędzi wiersza poleceń	42
2.3. Praca w zintegrowanym środowisku programistycznym	46
2.4. JShell	49

Rozdział 3. Podstawowe elementy języka Java	53
3.1. Prosty program w Javie	53
3.2. Komentarze	57
3.3. Typy danych	58
3.3.1. Typy całkowite	58
3.3.2. Typy zmiennoprzecinkowe	60
3.3.3. Typ char	61
3.3.4. Unicode i typ char	62
3.3.5. Typ boolean	64
3.4. Zmienne i stałe	65
3.4.1. Deklarowanie zmiennych	65
3.4.2. Inicjalizacja zmiennych	66
3.4.3. Stałe	67
3.4.4. Typ wyliczeniowy	68
3.5. Operatory	68
3.5.1. Operatory arytmetyczne	69
3.5.2. Funkcje i stałe matematyczne	70
3.5.3. Konwersja typów numerycznych	71
3.5.4. Rzutowanie	72
3.5.5. Przypisanie	73
3.5.6. Operatory inkrementacji i dekrementacji	74
3.5.7. Operatory relacyjne i logiczne	74
3.5.8. Operator warunkowy	75
3.5.9. Wyrażenia switch	75
3.5.10. Operatory bitowe	77
3.5.11. Nawiasy i priorytety operatorów	78
3.6. Łącuchy	79
3.6.1. Konkatenacja	79
3.6.2. Dzielenie łańcuchów	80
3.6.3. Indeksy i podłańcuchy	81
3.6.4. Łańcuchów nie można modyfikować	82
3.6.5. Porównywanie łańcuchów	83
3.6.6. Łańcuchy puste i łańcuchy null	84
3.6.7. API String	85
3.6.8. Dokumentacja API w internecie	87
3.6.9. Składanie łańcuchów	89
3.6.10. Bloki tekstowe	91
3.7. Wejście i wyjście	93
3.7.1. Odbieranie danych wejściowych	93
3.7.2. Formatowanie danych wyjściowych	96
3.7.3. Zapis i odczyt plików	99
3.8. Sterowanie wykonywaniem programu	101
3.8.1. Zasięg blokowy	102
3.8.2. Instrukcje warunkowe	102
3.8.3. Pętle	105
3.8.4. Pętle o określonej liczbie powtórzeń	109
3.8.5. Wybór wielokierunkowy — instrukcja switch	112
3.8.6. Instrukcje przerywające przepływ sterowania	116

3.9.	Wielkie liczby	119
3.10.	Tablice	122
3.10.1.	Deklarowanie tablic	122
3.10.2.	Dostęp do elementów tablicy	124
3.10.3.	Pętla typu for each	124
3.10.4.	Kopiowanie tablicy	125
3.10.5.	Argumenty wiersza poleceń	126
3.10.6.	Sortowanie tablicy	127
3.10.7.	Tablice wielowymiarowe	130
3.10.8.	Tablice postrzępione	133
Rozdział 4. Obiekty i klasy		136
4.1.	Wstęp do programowania obiektowego	136
4.1.1.	Klasy	137
4.1.2.	Obiekty	138
4.1.3.	Identyfikacja klas	139
4.1.4.	Relacje między klasami	140
4.2.	Używanie klas predefiniowanych	141
4.2.1.	Obiekty i zmienne obiektów	142
4.2.2.	Klasa LocalDate	144
4.2.3.	Metody udostępniające i zmieniające wartość elementu	146
4.3.	Definiowanie własnych klas	150
4.3.1.	Klasa Employee	150
4.3.2.	Używanie wielu plików źródłowych	153
4.3.3.	Analiza klasy Employee	153
4.3.4.	Pierwsze kroki w tworzeniu konstruktorów	154
4.3.5.	Deklarowanie zmiennych lokalnych za pomocą słowa kluczowego var	155
4.3.6.	Praca z referencjami null	156
4.3.7.	Parametry jawne i niejawne	157
4.3.8.	Korzyści z hermetyzacji	158
4.3.9.	Przywileje klasowe	161
4.3.10.	Metody prywatne	161
4.3.11.	Stałe jako pola klasy	162
4.4.	Pola i metody statyczne	162
4.4.1.	Pola statyczne	163
4.4.2.	Stałe statyczne	164
4.4.3.	Metody statyczne	164
4.4.4.	Metody fabryczne	166
4.4.5.	Metoda main	166
4.5.	Parametry metod	169
4.6.	Konstruowanie obiektów	175
4.6.1.	Przeciążanie	176
4.6.2.	Domyślna inicjalizacja pól	177
4.6.3.	Konstruktor bezargumentowy	177
4.6.4.	Jawna inicjalizacja pól	178
4.6.5.	Nazywanie parametrów	179
4.6.6.	Wywoływanie innego konstruktora	180
4.6.7.	Bloki inicjalizujące	180
4.6.8.	Niszczenie obiektów i metoda finalize	185

4.7.	Rekordy	185
4.7.1.	Koncepcja rekordu	186
4.7.2.	Konstruktory: kanoniczny, niestandardowy i kompaktowy	188
4.8.	Pakiety	190
4.8.1.	Nazwy pakietów	190
4.8.2.	Importowanie klas	191
4.8.3.	Importowanie statyczne	193
4.8.4.	Dodawanie klasy do pakietu	193
4.8.5.	Dostęp do pakietu	196
4.8.6.	Ścieżka klas	198
4.8.7.	Ustawianie ścieżki klas	200
4.9.	Pliki JAR	201
4.9.1.	Tworzenie plików JAR	201
4.9.2.	Manifest	202
4.9.3.	Wykonywalne pliki JAR	202
4.9.4.	Pliki JAR z wieloma wersjami klas	203
4.9.5.	Kilka uwag na temat opcji wiersza poleceń	205
4.10.	Komentarze dokumentacyjne	206
4.10.1.	Wstawianie komentarzy	206
4.10.2.	Komentarze do klas	207
4.10.3.	Komentarze do metod	207
4.10.4.	Komentarze do pól	208
4.10.5.	Komentarze do pakietów	209
4.10.6.	Kod HTML	209
4.10.7.	Łączy	210
4.10.8.	Komentarze ogólne	211
4.10.9.	Fragmenty kodu	212
4.10.10.	Pobieranie komentarzy	212
4.11.	Porady dotyczące projektowania klas	213

Rozdział 5. Dziedziczenie 217

5.1.	Klasy, nadklasy i podklasy	217
5.1.1.	Definiowanie podklas	218
5.1.2.	Przesłanianie metod	219
5.1.3.	Konstruktory podklas	221
5.1.4.	Hierarchia dziedziczenia	225
5.1.5.	Polimorfizm	225
5.1.6.	Zasady wywoływania metod	227
5.1.7.	Wyłączanie dziedziczenia — klasy i metody finalne	229
5.1.8.	Rzutowanie	232
5.1.9.	Operator instanceof i dopasowywanie wzorców	234
5.1.10.	Ograniczanie dostępu	236
5.2.	Kosmiczna klasa wszystkich klas — Object	238
5.2.1.	Zmienne typu Object	238
5.2.2.	Metoda equals	238
5.2.3.	Porównywanie a dziedziczenie	240
5.2.4.	Metoda hashCode	244
5.2.5.	Metoda toString	247

5.3.	Generyczne listy tablicowe	253
5.3.1.	Deklarowanie list tablicowych	253
5.3.2.	Dostęp do elementów listy tablicowej	256
5.3.3.	Zgodność pomiędzy typowanymi a surowymi listami tablicowymi	259
5.4.	Opakowania obiektów i automatyczne pakowanie	260
5.5.	Metody ze zmienną liczbą argumentów	264
5.6.	Klasy abstrakcyjne	265
5.7.	Klasy wyliczeniowe	270
5.8.	Klasy zapieczętowane	274
5.9.	Dopasowywanie do wzorca	280
5.9.1.	Obsługa wartości null	281
5.9.2.	Strażnicy	281
5.9.3.	Kompletność	282
5.9.4.	Dominacja	283
5.9.5.	Wzorce i stałe	284
5.9.6.	Zasięg zmiennych i przekazywanie sterowania w dół	284
5.10.	Refleksja	287
5.10.1.	Klasa Class	287
5.10.2.	Podstawy deklarowania wyjątków	290
5.10.3.	Zasoby	291
5.10.4.	Zastosowanie refleksji w analizie funkcjonalności klasy	294
5.10.5.	Refleksja w analizie obiektów w czasie działania programu	301
5.10.6.	Zastosowanie refleksji w generycznym kodzie tablicowym	306
5.10.7.	Wywoływanie dowolnych metod i konstruktorów	309
5.11.	Porady projektowe dotyczące dziedziczenia	313
Rozdział 6. Interfejsy, wyrażenia lambda i klasy wewnętrzne		316
6.1.	Interfejsy	317
6.1.1.	Koncepcja interfejsu	317
6.1.2.	Własności interfejsów	323
6.1.3.	Interfejsy a klasy abstrakcyjne	325
6.1.4.	Metody statyczne i prywatne	326
6.1.5.	Metody domyślne	326
6.1.6.	Wybieranie między metodami domyślnymi	328
6.1.7.	Interfejsy i wywołania zwrotne	330
6.1.8.	Interfejs Comparator	332
6.1.9.	Klonowanie obiektów	334
6.2.	Wyrażenia lambda	340
6.2.1.	Po co w ogóle są lambdy?	340
6.2.2.	Składnia wyrażeń lambda	342
6.2.3.	Interfejsy funkcyjne	344
6.2.4.	Typy funkcji	345
6.2.5.	Referencje do metod	347
6.2.6.	Referencje do konstruktorów	350
6.2.7.	Zakres dostępności zmiennych	351
6.2.8.	Przetwarzanie wyrażeń lambda	354
6.2.9.	Tworzenie komparatorów	357

6.3.	Klasy wewnętrzne	358
6.3.1.	Dostęp do stanu obiektu w klasie wewnętrznej	359
6.3.2.	Specjalne reguły składniowe dotyczące klas wewnętrznych	362
6.3.3.	Czy klasy wewnętrzne są potrzebne i bezpieczne?	363
6.3.4.	Lokalne klasy wewnętrzne	365
6.3.5.	Dostęp do zmiennych finalnych z metod zewnętrznych	365
6.3.6.	Anonimowe klasy wewnętrzne	367
6.3.7.	Klasy statyczne	370
6.4.	Moduły ładowania usług	374
6.5.	Klasy pośredniczące	376
6.5.1.	Kiedy używać klas pośredniczących?	377
6.5.2.	Tworzenie obiektów pośredniczących	377
6.5.3.	Właściwości klas pośredniczących	381
Rozdział 7. Wyjątki, asercje i dzienniki		384
7.1.	Obsługa błędów	385
7.1.1.	Klasyfikacja wyjątków	386
7.1.2.	Deklarowanie wyjątków kontrolowanych	388
7.1.3.	Zgłaszanie wyjątków	390
7.1.4.	Tworzenie klas wyjątków	391
7.2.	Przechwytywanie wyjątków	392
7.2.1.	Przechwytywanie wyjątku	393
7.2.2.	Przechwytywanie wielu typów wyjątków	395
7.2.3.	Powtórne generowanie wyjątków i budowanie łańcuchów wyjątków	397
7.2.4.	Klauzula finally	398
7.2.5.	Instrukcja try z zasobami	400
7.2.6.	Analiza danych ze stosu wywołań	402
7.3.	Wskazówki dotyczące stosowania wyjątków	406
7.4.	Asercje	410
7.4.1.	Koncepcja asercji	410
7.4.2.	Włączanie i wyłączanie asercji	411
7.4.3.	Zastosowanie asercji do sprawdzania parametrów	412
7.4.4.	Zastosowanie asercji do dokumentowania założeń	414
7.5.	Dzienniki	415
7.5.1.	Czy należy używać systemu dzienników Javy?	415
7.5.2.	Podstawy pracy z dziennikami	416
7.5.3.	API rejestrowania platformy	416
7.5.4.	Konfiguracja rejestrowania w dzienniku	418
7.5.5.	Handlerzy dzienników	419
7.5.6.	Filtry i formatery	421
7.5.7.	Przepis na dziennik	422
7.6.	Wskazówki dotyczące debugowania	429
Rozdział 8. Programowanie generyczne		435
8.1.	Dlaczego programowanie generyczne?	435
8.1.1.	Zalety parametrów typów	436
8.1.2.	Dla kogo programowanie generyczne?	437
8.2.	Definicja prostej klasy generycznej	438
8.3.	Metody generyczne	440

8.4.	Ograniczenia zmiennych typowych	441
8.5.	Kod generyczny a maszyna wirtualna	443
8.5.1.	Wymazywanie typów	444
8.5.2.	Translacja wyrażeń generycznych	445
8.5.3.	Translacja metod generycznych	445
8.5.4.	Używanie starego kodu	448
8.5.5.	Generyczne wzorce rekordów	449
8.6.	Zasady dziedziczenia dla typów generycznych	449
8.7.	Typy wieloznaczne	451
8.7.1.	Koncepcja typu wieloznacznego	451
8.7.2.	Ograniczenia nadtypów typów wieloznacznych	453
8.7.3.	Typy wieloznaczne bez ograniczeń	455
8.7.4.	Chwytnie typu wieloznacznego	456
8.8.	Ograniczenia i braki	458
8.8.1.	Nie można podawać typów prostych jako parametrów typowych	459
8.8.2.	Sprawdzanie typów w czasie działania programu jest możliwe tylko dla typów surowych	459
8.8.3.	Nie można tworzyć tablic typów generycznych	459
8.8.4.	Ostrzeżenia dotyczące zmiennej liczby argumentów	460
8.8.5.	Generyczne parametry zmienne nie rozkładają tablic typów podstawowych	461
8.8.6.	Nie wolno tworzyć egzemplarzy zmiennych typowych	462
8.8.7.	Nie można utworzyć egzemplarza generycznej tablicy	463
8.8.8.	Zmiennych typowych nie można używać w statycznych kontekstach klas generycznych	465
8.8.9.	Obiektów klasy generycznej nie można zgłaszać ani przechwytywać	465
8.8.10.	Można wyłączyć sprawdzanie wyjątków kontrolowanych	466
8.8.11.	Uważaj na konflikty, które mogą powstać po wymazaniu typów	467
8.8.12.	Dedukcja typów w generycznych wzorcach rekordów jest ograniczona	468
8.9.	Refleksja a typy generyczne	470
8.9.1.	Generyczna klasa Class	470
8.9.2.	Zastosowanie parametrów Class<T> do dopasowywania typów	471
8.9.3.	Informacje o typach generycznych w maszynie wirtualnej	472
8.9.4.	Literały typowe	475

Rozdział 9. Kolekcje 481

9.1.	Architektura kolekcji Javy	481
9.1.1.	Oddzielenie warstwy interfejsów od warstwy klas konkretnych	482
9.1.2.	Interfejs Collection	484
9.1.3.	Iteratory	484
9.1.4.	Generyczne metody użytkowe	487
9.2.	Interfejsy w systemie kolekcji Javy	491
9.3.	Konkretne klasy kolekcyjne	493
9.3.1.	Listy powiązane	494
9.3.2.	Listy tablicowe	504
9.3.3.	Zbiór HashSet	504
9.3.4.	Zbiór TreeSet	508
9.3.5.	Kolejki Queue i Deque	512
9.3.6.	Kolejki priorytetowe	514

9.4.	Słowniki	515
9.4.1.	Podstawowe operacje słownikowe	515
9.4.2.	Modyfikowanie wpisów w słowniku	519
9.4.3.	Widoki słowników	521
9.4.4.	Klasa WeakHashMap	523
9.4.5.	Klasy LinkedHashSet i LinkedHashMap	524
9.4.6.	Klasy EnumSet i EnumMap	525
9.4.7.	Klasa IdentityHashMap	526
9.5.	Kopie i widoki	528
9.5.1.	Małe kolekcje	529
9.5.2.	Niemodyfikowalne kopie i widoki	531
9.5.3.	Przedziały	532
9.5.4.	Zbiory ze słowników wartości logicznych	533
9.5.5.	Widoki odwrócone	534
9.5.6.	Widoki kontrolowane	534
9.5.7.	Widoki synchronizowane	535
9.5.8.	Uwagi dotyczące operacji opcjonalnych	535
9.6.	Algorytmy	540
9.6.1.	Dlaczego algorytmy generyczne?	540
9.6.2.	Sortowanie i tasowanie	542
9.6.3.	Wyszukiwanie binarne	544
9.6.4.	Proste algorytmy	546
9.6.5.	Operacje zbiorowe	548
9.6.6.	Konwersja pomiędzy kolekcjami a tablicami	549
9.6.7.	Pisanie własnych algorytmów	550
9.7.	Stare kolekcje	551
9.7.1.	Klasa Hashtable	552
9.7.2.	Wyliczenia	552
9.7.3.	Słowniki własności	553
9.7.4.	Właściwości systemowe	555
9.7.5.	Stosy	557
9.7.6.	Zbiory bitów	558
Rozdział 10. Współbieżność		562
10.1.	Uruchamianie wątków	563
10.2.	Stany wątków	567
10.2.1.	Wątki tworzone za pomocą operatora new	568
10.2.2.	Wątki wykonywalne	568
10.2.3.	Wątki BLOCKED i WAITING	569
10.2.4.	Zamykanie wątków	569
10.3.	Własności wątków	571
10.3.1.	Wątki wirtualne	571
10.3.2.	Przerywanie wątków	572
10.3.3.	Wątki demony	575
10.3.4.	Nazwy i identyfikatory wątków	576
10.3.5.	Procedury obsługi nieprzechwyconych wyjątków	576
10.3.6.	Priorytety wątków	578
10.3.7.	Fabryki i konstruktory wątków	579

10.4.	Koordinacja zadań	580
10.4.1.	Interfejsy Callable i Future	580
10.4.2.	Klasa Executors	583
10.4.3.	Wywoływanie grup zadań	587
10.4.4.	Zmienne lokalne wątków	593
10.4.5.	Metoda rozgałęzienie-złączenie	594
10.5.	Synchronizacja	597
10.5.1.	Przykład sytuacji powodującej wyścig	597
10.5.2.	Wyścigi	599
10.5.3.	Obiekty klasy Lock	601
10.5.4.	Warunki	604
10.5.5.	Zakleszczenia	609
10.5.6.	Słowo kluczowe synchronized	611
10.5.7.	Bloki synchronizowane	616
10.5.8.	Monitor	618
10.5.9.	Pola ulotne	619
10.5.10.	Pola finalne	620
10.5.11.	Zmienne atomowe	621
10.5.12.	Inicjalizacja na żądanie	623
10.5.13.	Bezpieczna publikacja	624
10.5.14.	Współdzielenie zmiennych lokalnych wątków	625
10.6.	Kolekcje bezpieczne wątkowo	626
10.6.1.	Kolejki blokujące	626
10.6.2.	Szybkie słowniki, zbiory i kolejki	632
10.6.3.	Atomowe modyfikowanie elementów słowników	634
10.6.4.	Operacje masowe na współbieżnych słownikach skrótów	637
10.6.5.	Współbieżne widoki zbiorów	639
10.6.6.	Tablice kopiowane przy zapisie	640
10.6.7.	Równoległe algorytmy tablicowe	640
10.6.8.	Starsze kolekcje bezpieczne wątkowo	641
10.7.	Obliczenia asynchroniczne	642
10.7.1.	Klasa CompletableFuture	643
10.7.2.	Tworzenie obiektów CompletableFuture	645
10.7.3.	Czasochłonne zadania w wywołaniach zwrotnych interfejsu użytkownika	651
10.8.	Procesy	657
10.8.1.	Budowanie procesu	658
10.8.2.	Uruchamianie procesu	660
10.8.3.	Uchwyty procesów	661
Rozdział 11.	Adnotacje	666
11.1.	Stosowanie adnotacji	667
11.1.1.	Elementy adnotacji	667
11.1.2.	Powielanie i mnożenie adnotacji	668
11.1.3.	Adnotacje deklaracji	668
11.1.4.	Adnotacje zastosowań typów	669
11.1.5.	Określanie adresatów wprost	671
11.2.	Definiowanie adnotacji	672
11.3.	Adnotacje w API Javy	675
11.3.1.	Adnotacje kompilacji	675
11.3.2.	Metaadnotacje	677

11.4.	Przetwarzanie adnotacji podczas działania programu	678
11.5.	Przetwarzanie adnotacji w kodzie źródłowym	682
11.5.1.	Procesory adnotacji	682
11.5.2.	Interfejs programowy modelu języka	683
11.5.3.	Stosowanie adnotacji do generacji kodu źródłowego	684
11.6.	Inżynieria kodu bajtowego	689
11.6.1.	Modyfikowanie plików klasowych	689
11.6.2.	Modyfikacja kodu bajtowego podczas ładowania	695
Rozdział 12. System modułów platformy Javy		698
12.1.	Pojęcie modułu	698
12.2.	Nadawanie nazw modułom	700
12.3.	Modularny program „Witaj, świecie!”	701
12.4.	Żądanie użycia modułów	702
12.5.	Eksportowanie pakietów	704
12.6.	Modularne pliki JAR	708
12.7.	Moduły a technika refleksji	709
12.8.	Moduły automatyczne	712
12.9.	Moduł nienazwany	714
12.10.	Flagi wiersza poleceń stosowane podczas migracji	715
12.11.	Wymagania przechodnie i statyczne	717
12.12.	Eksport kwalifikowany i otwieranie	718
12.13.	Wczytywanie usług	719
12.14.	Narzędzia do pracy z modułami	721
Dodatek A		725
Skorowidz		729

9

Kolekcje

Dobór struktur danych do użycia w programie może mieć niebagatelne znaczenie dla późniejszej implementacji metod i wydajności całej aplikacji. Decydując się na konkretne struktury, należy odpowiedzieć sobie na pytania typu: czy konieczne będzie przeszukiwanie tysięcy (może nawet milionów) posortowanych elementów? Czy konieczne będzie szybkie wstawianie i usuwanie elementów do i ze środka uporządkowanego szeregu elementów? Czy konieczne będzie powiązanie wartości z ich kluczami?

Ten rozdział traktuje o strukturach danych dostępnych w Java Collections Framework. Na studiach informatycznych przedmiot dotyczący *struktur danych* trwa z reguły jeden semestr, dzięki czemu ta ważna tematyka została wyczerpująco przedstawiona w bardzo licznych publikacjach i opracowaniach. W tej książce prezentujemy odmienne podejście w stosunku do innych publikacji z tego zakresu — pomijamy teorię, a koncentrujemy się na zastosowaniu kolekcji dostępnych w API Javy.

9.1. Architektura kolekcji Javy

Początkowo w Javie dostępnych było tylko kilka klas implementujących najbardziej przydatne struktury danych: `Vector`, `Stack`, `Hashtable`, `BitSet` oraz interfejs `Enumerable` odpowiadający za abstrakcyjny mechanizm dostępu do elementów w każdym z tych kontenerów. Było to z pewnością mądre posunięcie ze strony projektantów języka — opracowanie pełnej biblioteki klas kolekcyjnych wymaga czasu i doświadczenia.

Projektanci doszli do wniosku, że czas na zaprezentowanie pełnego zestawu struktur danych nadszedł w Java 1.2. Musieli dokonać wielu trudnych wyborów, ponieważ chcieli stworzyć bibliotekę mającą niewielkie rozmiary i łatwą do opanowania dla programistów. Starali się unikać poziomu złożoności charakteryzującego bibliotekę C++ STL (ang. *Standard Template Library*), jednocześnie próbując skorzystać z dobrodziejstw algorytmów generycznych, których pionierem była właśnie wymieniona biblioteka. Stare klasy musiały pasować do nowej architektury.

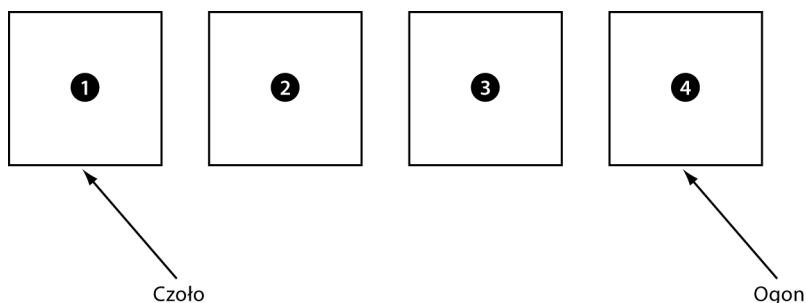
Jak to zwykle bywa przy projektowaniu bibliotek kolekcji, kilkakrotnie stawano przed trudnym wyborem, co zaowocowało kilkoma osobliwymi decyzjami projektowymi. W tym podrozdziale przedstawiamy podstawową strukturę architektury kolekcji Javy, pokazujemy sposoby jej wykorzystania oraz wyjaśniamy, czym kierowali się projektanci, podejmując niektóre bardziej kontrowersyjne decyzje.

9.1.1. Oddzielenie warstwy interfejsów od warstwy klas konkretnych

Podobnie jak większość nowoczesnych bibliotek struktur danych, Java Collections Framework dzieli się na warstwę **interfejsów** i warstwę **implementacji**. Przyjrzymy się temu podziałowi na przykładzie znanej struktury danych o nazwie **kolejka** (ang. *queue*).

Interfejs `Queue` pozwala na dodawanie elementów na końcu kolejki, usuwanie ich z początku struktury danych oraz sprawdzanie liczby elementów w kolejce. Ta struktura danych znajduje zastosowanie przy tworzeniu zbiorów obiektów, z których elementy są pobierane zgodnie z zasadą „pierwszy przyjdzie, pierwszy wyjdzie” (zobacz rysunek 9.1).

Rysunek 9.1.
Kolejka



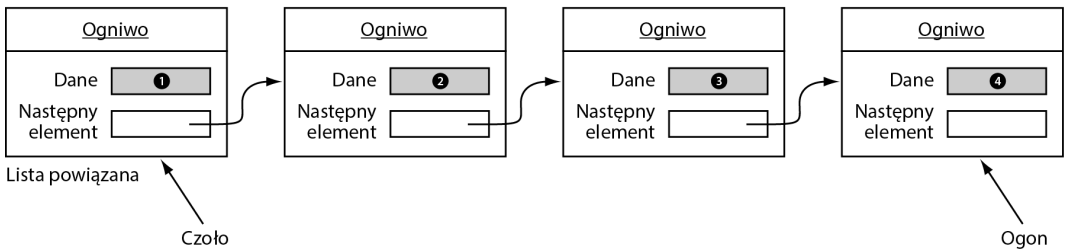
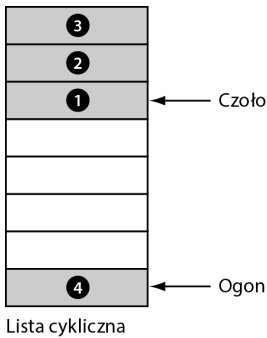
Interfejs `Queue` w swojej uproszczonej formie może wyglądać następująco:

```
public interface Queue<E> // uproszczona wersja interfejsu z API Javy
{
    void add(E e);
    E remove();
    int size();
}
```

Sam interfejs nie zawiera żadnych danych na temat implementacji kolejki. Z dwóch najczęściej spotykanych implementacji kolejki jedna działa na zasadzie listy cyklicznej, a druga listy powiązanej (rysunek 9.2).

Każdą z tych implementacji można zrealizować za pomocą klasy implementującej interfejs `Queue`. W API Javy klasy te noszą nazwy `ArrayDeque` i `LinkedList`.

```
public class ArrayDeque<E> implements Queue<E> // uproszczona wersja z API Javy
{
    private int first;
    private int last;
    private E[] elements;
```



Rysunek 9.2. Implementacje kolejki

```

    public ArrayDeque(int numElements) { . . . }
    public void add(E e) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    . . .
}

public class LinkedList<E> implements Queue<E> // uproszczona wersja z API Javy
{
    private Node<E> first;
    private Node<E> last;

    public LinkedList() { . . . }
    public void add(E e) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    . . .
}

```

Programista używający w programie kolejki po utworzeniu kolekcji nie musi wiedzieć, która jej implementacja została użyta. Dlatego dobrym rozwiązaniem jest używanie klas konkretnych *tylko* do konstruowania obiektów kolekcyjnych. **Typów interfejsowych** należy używać do przechowywania referencji do tych obiektów.

```

Queue<Customer> expressLane = new ArrayDeque<>(100);
expressLane.add(new Customer("Henryk"));

```

Dzięki takiemu podejściu w razie zmiany zdania można łatwo użyć innej implementacji. Wystarczy tylko jedna zmiana w programie — wywołanie konstruktora. Jeśli na przykład dojdziemy do wniosku, że lepszym wyborem byłaby lista powiązana, zmieniamy kod na następujący:

```
Queue<Customer> expressLane = new LinkedList<>();
expressLane.add(new Customer("Henryk"));
```

Jest jeszcze jeden powód, dla którego należy pozostać przy interfejsie. Zarówno klasa `ArrayDeque`, jak i `LinkedList` ma metody, które nie mają związku z kolejkowaniem. Przypadkowe użycie jednej z nich zostanie zgłoszone jako błąd kompilacji, ponieważ nie można ich wywoływać na zmiennych typu `Queue`.

Co sprawia, że wybieramy jedną implementację zamiast drugiej? Interfejs nie dostarcza żadnych informacji na temat wydajności. Lista cykliczna nieco lepiej wykorzystuje pamięć. Ma jednak ograniczoną pojemność i wymaga realokacji, gdy się zapełni. Pomiary pokazują, że `ArrayDeque` generalnie przewyższa wydajnością `LinkedList`, ponieważ ma mniejszy wpływ na system usuwania nieużytków. Kiedy struktura `ArrayDeque` została dodana do Javy 6, programiści korzystający z interfejsu `Queue` mogli na nią przejść przez zmianę jednego wiersza kodu.

9.1.2. Interfejs Collection

Interfejsem o kluczowym znaczeniu w Java Collections Framework jest interfejs `Collection`. Dwie z jego metod mają fundamentalne znaczenie:

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    . . .
}
```

Pozostałe metody tego interfejsu omawiamy dalej.

Metoda `add` dodaje elementy do kolekcji i zwraca wartość `true`, jeśli wykonana przez nią operacja powoduje zmianę stanu kolekcji, lub wartość `false`, jeśli kolekcja nie ulega zmianie. Jeśli na przykład do zbioru (ang. *set*) spróbujemy dodać obiekt, który już się tam znajduje, metoda `add` nie przyniesie żadnego skutku, ponieważ w zbiorach nie może być duplikatów.

Metoda `iterator` zwraca obiekt implementujący interfejs `Iterator`. Za pomocą tego obiektu można odwiedzić kolejno wszystkie znajdujące się w kolekcji elementy. Opis iteratorów znajduje się w następnej sekcji.

9.1.3. Iteratory

W interfejsie `Iterator` występują cztery metody:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

Wywołując wielokrotnie metodę `next`, można kolejno odwiedzić wszystkie elementy kolekcji. Jeśli metoda ta napotka koniec kolekcji, zgłosi wyjątek `NoSuchElementException`. Dlatego przed nią należy zawsze wywoływać metodę `hasNext`, która zwraca wartość `true`, jeśli iterator nie dotarł jeszcze do końca i jest przynajmniej jeden element do przejrzania. Aby przejrzeć wszystkie elementy kolekcji, należy utworzyć obiekt typu `Iterator` i wywoływać metodę `next` tak długo, jak metoda `hasNext` zwraca wartość `true`. Na przykład:

```
Collection<String> coll = . . .;
Iterator<String> iter = coll.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    obróbka elementu
}
```

Pętlę taką można wyrazić zwięźlej za pomocą konstrukcji typu `for each`:

```
for (String element : coll)
{
    obróbka elementu
}
```

Kompilator konwertuje pętlę w stylu `for each` na pętlę z iteratorem.

Pętlę typu `for each` można stosować do wszystkich obiektów implementujących interfejs `Iterable`, w którym znajduje się metoda abstrakcyjna:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
    ...
}
```

Interfejs `Collection` rozszerza interfejs `Iterable`. Dzięki temu pętli `for each` można używać do działań na wszystkich kolekcjach z `Java Collections Framework`.

Nie trzeba nawet pisać pętli. W zamian można wywołać metodę `Collection.forEach` lub `Iterator.forEachRemaining` z wyrażeniem lambda wykorzystującym `element`. Wyrażenie to jest wywoływane na wszystkich elementach kolekcji lub pozostałych elementach, które iterator może odwiedzić.

```
coll.forEach(element -> jakieś działania na elemencie);
iter.forEachRemaining(element -> operacje na elemencie);
```

Kolejność odwiedzania elementów zależy od rodzaju kolekcji. W przypadku listy `ArrayList` iterator zaczyna od indeksu o numerze 0 i zwiększa ten numer w każdym kolejnym powtórzeniu (iteracji). Natomiast dostęp do elementów w zbiorze `HashSet` odbywa się w zasadzie losowo. Pewne jest, że przemierzając tę kolekcję, uzyskamy dostęp do każdego jej elementu, ale nie wiadomo, w jakiej kolejności będzie się to odbywać. Nie sprawia to zazwyczaj problemu, ponieważ w działaniach typu obliczanie sumy lub zliczanie elementów pasujących do wzorca kolejność jest nieistotna.



Ci, którzy znają wcześniejsze wersje Javy, zauważą, że metody `next` i `hasNext` interfejsu `Iterator` spełniają tę samą funkcję co `nextElement` i `hasMoreElements` w interfejsie `Enumeration`. Projektanci Java Collections Framework mogli wykorzystać w swojej pracy interfejs `Enumeration`, ale nie podobały im się niezgrabne nazwy jego metod. Dlatego utworzono nowy interfejs z krótszymi nazwami metod.

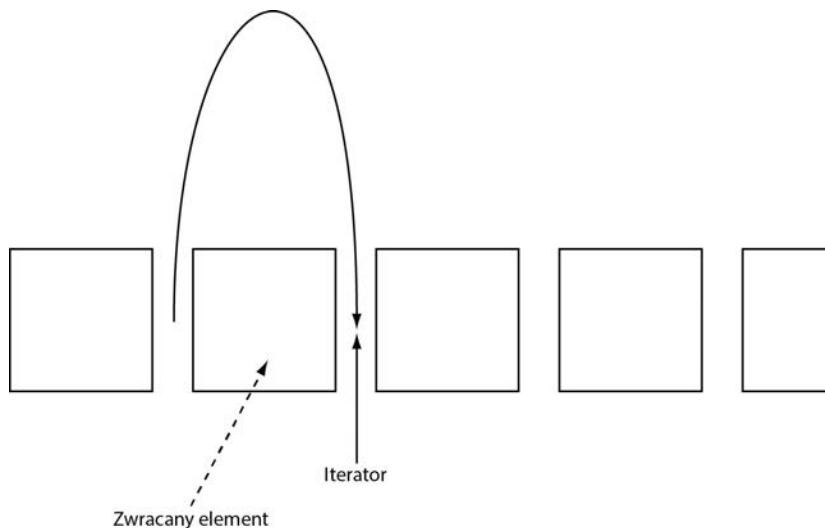
Pomiędzy iteratorami w bibliotece kolekcyjnej w Javie a iteratorami z innych bibliotek istnieje ważna różnica koncepcyjna. Modelem iteratorów w tradycyjnych bibliotekach, takich jak `Standard Template Library` w C++, są indeksy tablicowe. Za pomocą takiego tradycyjnego iteratora element znajdujący się w określonym miejscu można odszukać w podobny sposób jak element `a[i]` w tablicy, jeśli znany jest indeks `i`. Niezależnie od wyszukiwania, iterator taki można przesunąć do kolejnego elementu, co niczym się nie różni od operacji zwiększania indeksu za pomocą instrukcji `i++`, bez wyszukiwania. Iteratory w Javie działają nieco inaczej. Wyszukiwanie i zmiana położenia są ze sobą ściśle związane. Jedynym sposobem na znalezienie elementu jest wywołanie metody `next`, a to powoduje przejście do kolejnego elementu.

Iteratory w Javie należy wyobrażać sobie jako obiekty znajdujące się *pomiędzy elementami*. W chwili wywołania metody `next` iterator *przeskakuje* kolejny element i zwraca referencję do elementu, który właśnie przeskoczył (zobacz rysunek 9.3).



Istnieje jeszcze inna ciekawa analogia. Instrukcję `Iterator.next` można traktować jako odpowiednik instrukcji `InputStream.read`. Odczyt bajta ze strumienia automatycznie oznacza jego „połknięcie”. Kolejne wywołanie metody `read` powoduje „połknięcie” i zwrócenie następnego bajta z danych wejściowych. W podobny sposób seria wywołań metody `next` zwraca wszystkie elementy znajdujące się w kolekcji.

Rysunek 9.3.
Przesuwanie
iteratora



Metoda `remove` z interfejsu `Iterator` usuwa element zwrócony przez ostatnie wywołanie metody `next`. W większości sytuacji jest to rozsądne działanie — aby podjąć decyzję o usunięciu elementu, najczęściej trzeba go wpierw zobaczyć. Aby usunąć element znajdujący się w określonym miejscu, także trzeba za niego przejść. Na przykład poniższa procedura usuwa pierwszy element z kolekcji łańcuchów:

```
Iterator<String> iter = coll.iterator();
iter.next();      // przejście za pierwszy element
iter.remove();    // usunięcie pierwszego elementu
```

Między metodami `next` i `remove` istnieje pewna bardzo ważna zależność. Tej drugiej nie można wywołać, jeśli wcześniej nie wywołano pierwszej. Próba zrobienia tego zakończy się rzuceniem wyjątku `IllegalStateException`.

Aby usunąć dwa kolejne elementy, nie można zastosować dwóch kolejnych wywołań metody `remove`:

```
iter.remove();
iter.remove();    // Błąd!
```

Najpierw trzeba wywołać metodę `next`, aby przejść za kolejny element, który ma zostać usunięty.

```
iter.remove();
iter.next();
iter.remove();    // OK
```

9.1.4. Generyczne metody użytkowe

Dzięki temu, że interfejsy `Collection` i `Iterator` są generyczne, można pisać metody użytkowe operujące na dowolnych rodzajach kolekcji. Poniżej znajduje się przykładowa metoda generyczna sprawdzająca, czy dowolnego rodzaju kolekcja zawiera określony element:

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
    for (E element : c)
        if (element.equals(obj))
            return true;
    return false;
}
```

Twórcy Java Collections Framework doszli do wniosku, że niektóre z tych metod są tak przydatne, iż powinny być dostępne w bibliotece. Dzięki temu użytkownicy API nie muszą wielokrotnie wynajdywać koła. Jedna z tych metod nosi nazwę `contains`.

W rzeczywistości w interfejsie `Collection<E>` znajduje się spora liczba przydatnych metod, które muszą być udostępniane przez wszystkie implementujące go klasy:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean add(E obj)
boolean addAll(Collection<? extends E> from)
```

```

boolean remove(Object obj)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
Object[] toArray()
T[] toArray(T[] a)

```



Interfejs `Collection` deklaruje metody `equals` i `hashCode`, które są odziedziczone z klasy `Object`. Nie ma technicznego wymogu, aby dodać te deklaracje. Jest to po prostu dobre miejsce na umieszczenie dokumentacji API kolekcji. Podinterfejsy `Set` i `List` umożliwiają specyficzne zachowanie metody `equals`. Inne kolekcje mogą dziedziczyć metodę `Object.equals` lub mogą ją przesłaniać i porównywać elementy w jakiś sposób. W drugim przypadku konieczne jest przesłonięcie także metody `hashCode`.

Przeznaczenie wielu z tych metod łatwo odgadnąć po nazwie. Pełny ich opis znajduje się na końcu podrozdziału w wyciągach z API.



Java Collections Framework zaprojektowano przed dodaniem typów generycznych do Javy. W celu zachowania zgodności wstecznej metody `contains` i `remove` mają parametr typu `Object`, a nie `E`. Metody `containsAll`, `removeAll` i `retainAll` mają parametr typu `Collection<?>`, a nie `Collection<? extends E>`.

To znaczy, że błędy dotyczące typów mogą nie zostać wykryte w czasie kompilacji. Spójrz na przykład na poniższy kod, w którym przypadkowo usunięto `String` z kolekcji obiektów typu `Path`:

```

Collection<Path> paths = . . . ;
paths.remove("/tmp"); // Przechodzi kompilację, ale może nie powodować żadnego efektu

```

Oczywiście definiowanie tylu metod we wszystkich klasach implementujących interfejs `Collection` jest bardzo uciążliwe. Aby ułatwić życie programistom, utworzono klasę `AbstractCollection` implementującą wszystkie metody tego interfejsu w kategorii metod `size` i `iterator`, które jako jedyne pozostały w niej abstrakcyjne. Na przykład:

```

public abstract class AbstractCollection<E>
    implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();
    public abstract int size();

    public boolean isEmpty()
    {
        return size() == 0;
    }

    public boolean contains(Object obj)
    {
        Iterator<E> it = iterator();
        if (o == null)
        {
            while (it.hasNext()) if (it.next() == null) return true;
        }
    }
}

```

```

        else
        {
            while (it.hasNext()) if (o.equals(it.next())) return true;
        }
        return false;
    }
    . . .
}

```

Konkretna klasa kolekcyjna może być rozszerzeniem klasy `AbstractCollection`. W klasie konkretnej konieczne jest zdefiniowanie metod `iterator` i `size`. Kolekcje modyfikowalne także wymagają metody `add`. O pozostałe metody zadbało w nadklasie abstrakcyjnej `AbstractCollection`. Jeśli jednak w podklasie istnieje możliwość zdefiniowania efektywniejszej metody `contains`, nic nie stoi na przeszkodzie, aby to zrobić.

Takie podejście jest już przestarzałe. Byłoby lepiej, gdyby metody te były metodami domyślnymi interfejsu `Collection`, ale takie metody jeszcze nie istniały, kiedy projektowano Java Collections Framework. Jednak tak nie jest, choć dodano kilka metod domyślnych. Trzy z nich dotyczą strumieni (o których jest mowa w drugim tomie). Metoda `toArray` jest opisana w punkcie 9.6.6. Ponadto następująca bardzo przydatna metoda służy do usuwania elementów spełniających określony warunek:

```
default boolean removeIf(Predicate<? super E> filter)
```

`java.util.Collection<E>` **1.2**

■ `Iterator<E> iterator()`

Zwraca obiekt `Iterator`, za pomocą którego można odwiedzać elementy kolekcji.

■ `int size()`

Zwraca liczbę elementów przechowywanych w kolekcji.

■ `boolean isEmpty()`

Zwraca wartość `true`, jeśli kolekcja nie zawiera żadnych elementów.

■ `boolean contains(Object obj)`

Zwraca wartość `true`, jeśli kolekcja zawiera obiekt identyczny z obiektem `obj`.

■ `boolean containsAll(Collection<?> other)`

Zwraca wartość `true`, jeśli kolekcja zawiera wszystkie elementy znajdujące się w innej kolekcji.

■ `boolean add(E element)`

Próbuje dodać element do kolekcji. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

■ `boolean addAll(Collection<? extends E> other)`

Próbuje dodać do kolekcji wszystkie elementy z kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `boolean remove(Object obj)`

Usuwa obiekt `obj` z kolekcji. Zwraca wartość `true`, jeśli obiekt został znaleziony i usunięty.

- `boolean removeAll(Collection<?> other)`

Usuwa z kolekcji wszystkie elementy, które można znaleźć w kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `default boolean removeIf(Predicate<? super E> filter) 8`

Usuwa wszystkie elementy, dla których filtr zwróci wartość `true`. Zwraca `true`, jeżeli w wyniku wywołania tej metody nastąpiły zmiany w kolekcji.

- `void clear()`

Usuwa wszystkie elementy z kolekcji.

- `boolean retainAll(Collection<?> other)`

Usuwa z kolekcji wszystkie elementy, które nie są takie same jak jeden z obiektów w kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `Object[] toArray()`

Zwraca tablicę obiektów zapełnioną elementami z kolekcji.

- `<T> T[] toArray(IntFunction<T[]> generator) 11`

Zwraca tablicę obiektów z kolekcji. Tablica ta jest utworzona przez generator, który zazwyczaj jest wyrażeniem konstrukcyjnym `T[]::new`.

`java.util.Iterator<E> 1.2`

- `boolean hasNext()`

Zwraca wartość `true`, jeśli są jeszcze elementy do odwiedzenia.

- `E next()`

Zwraca następny obiekt do odwiedzenia. Wyrzuca wyjątek `NoSuchElementException`, jeśli osiągnięto koniec kolekcji.

- `void remove()`

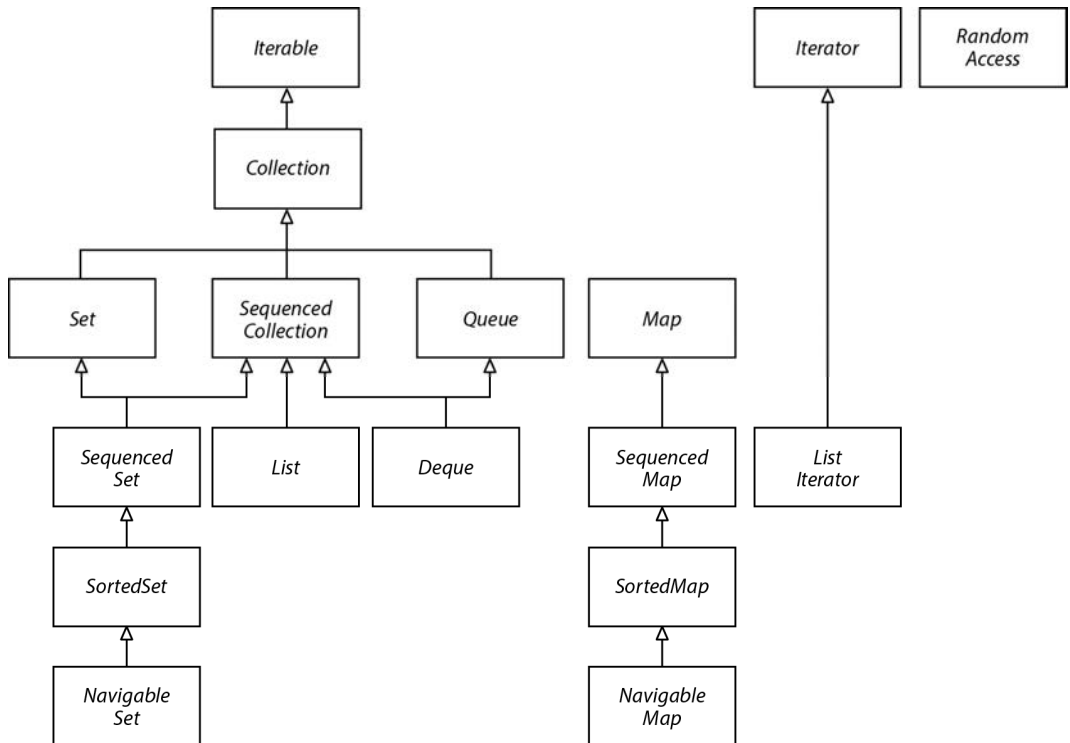
Usuwa ostatnio odwiedzony obiekt. Wywołanie tej metody musi następować bezpośrednio po odwiedzeniu elementu. Jeśli kolekcja zmieniła się od ostatniego odwiedzenia elementu, metoda ta wyrzuci wyjątek `IllegalStateException`. Iteratory nie muszą obsługiwać tej operacji. Domyślna implementacja zgłasza wyjątek `UnsupportedOperationException`.

- `default void forEachRemaining(Consumer<? super E> action) 8`

Przegląda elementy i przekazuje je do określonej akcji tak długo, aż przejrzy wszystkie elementy albo akcja zgłosi wyjątek.

9.2. Interfejsy w systemie kolekcji Javy

System kolekcji Javy (ang. *Java Collections Framework*) zawiera definicje kilku interfejsów dla różnych typów kolekcji. Na rysunku 9.4 są wymienione wszystkie z nich oprócz interfejsów do programowania współbieżnego, które są opisane w rozdziale 10.



Rysunek 9.4. Interfejsy Java Collections Framework

Dwa podstawowe interfejsy kolekcyjne to `Collection` i `Map`. Elementy do kolekcji dodaje się za pomocą metody `add`:

```
boolean add(E element)
```

Ponieważ słowniki przechowują pary klucz – wartość, elementy wstawia się do nich za pomocą metody `put`:

```
V put(K key, V value)
```

Do odczytu elementów z kolekcji służy iterator. Elementy ze słownika można także odczytać za pomocą metody `get`:

```
V get(Object key)
```

Lista to **kolekcja uporządkowana**. Elementy są dodawane w określonych miejscach kontenera, a dostęp do nich można uzyskać na dwa sposoby: za pomocą iteratora lub przy użyciu indeksu

całkowitoliczbowego. Drugi z wymienionych sposobów nazywa się **dostępem swobodnym** (ang. *random access*), ponieważ elementy można przeglądać w dowolnej kolejności. Natomiast iterator umożliwia tylko przeglądanie elementów po kolei.

Interfejs `List` zawiera definicje kilku metod dotyczących dostępu swobodnego:

```
void add(int index, E element)
E remove(int index)
E get(int index)
E set(int index, E element)
```



W `List<Integer>` są dwie metody `remove`:

```
boolean remove(int index) // Usuwa element o danym indeksie
boolean remove(Integer o) // Usuwa element równy o
```

Gdy wywołamy metodę `remove` z parametrem typu `int`, to wybrana zostanie pierwsza wersja. Nie będzie opakowywania ani nie zostanie wyświetlone żadne ostrzeżenie. Nie o to może Ci chodzić:

```
List<Integer> ids = . . .;
int id = . . .;
if (id == 0) ids.remove(id); // Usuwa indeks 0, nie element o wartości 0
```



Dokumentacja API interfejsu `List` definiuje zachowanie metody `equals` dla list. Aby lista została uznana za równą innemu obiektowi, obiekt ten także musi być jakiegoś rodzaju listą. Ponadto obie listy muszą być tego samego rozmiaru. Podczas iteracji przez każdą z nich korespondujące elementy muszą być równe.

Trzeba uczciwie przyznać, że ten składnik systemu kolekcji nie najlepiej udał się projektantom Javy. W praktyce istnieją dwa rodzaje kolekcji uporządkowanych, z których każda charakteryzuje się innymi rozwiązaniami w zakresie optymalizacji wydajności. Kolekcja uporządkowana zbudowana na bazie tablicy zapewnia szybki dostęp swobodny do elementów. Metod klasy `List` dobrze jest używać z indeksami całkowitoliczbowymi. Natomiast lista powiązana, choć też uporządkowana, zapewnia powolny dostęp swobodny i najlepiej ją przeglądać liniowo za pomocą iteratora. Stworzenie dwóch interfejsów byłoby lepszym rozwiązaniem.



Aby uniknąć wykonywania operacji dostępu swobodnego w listach powiązanych, w Java SE 1.4 wprowadzono interfejs znacznikowy o nazwie `RandomAccess`. Nie posiada on żadnych metod, ale można za jego pomocą sprawdzić, czy określona kolekcja umożliwia szybki dostęp swobodny do elementów:

```
if (c instanceof RandomAccess)
{
    przeglądanie swobodne
}
else
{
    przeglądanie liniowe
}
```

Interfejs `Set` ma taką samą metodę egzemplarza, jak interfejs `Collection`, ale jego metody są nieco bardziej restrykcyjne. Metoda `add` zbioru nie powinna dodawać duplikatów. Metoda `equals` powinna być zdefiniowana w taki sposób, aby dwa zbiory były identyczne, jeśli mają takie same elementy, ale niekoniecznie w takiej samej kolejności. Metoda `hashCode` dla dwóch zbiorów zawierających takie same elementy powinna zwracać tę samą wartość skrótu.

Po co tworzyć osobny interfejs, skoro sygnatury metod są takie same? Konceptyjnie nie wszystkie kolekcje są zbiorami. Dzięki istnieniu interfejsu `Set` programiści mogą pisać metody, które działają tylko na zbiorach.

Aby zrozumieć, dlaczego to rozróżnienie jest ważne, pomyśl o porównywaniu. Dwie listy są równe, gdy zawierają te same elementy ułożone w jakiejś kolejności. Metoda `equals` dla klasy implementującej interfejs `Set` musi sprawdzać, czy argument także jest typu `Set`, a nie jakąkolwiek inną kolekcją.

Interfejsy `SortedSet` i `SortedMap` udostępniają obiekt komparatora użyty do sortowania oraz definiują metody tworzące widoki podzbiorów kolekcji. Szerzej na ten temat piszemy w podrozdziale 9.5.3, „Kopie i widoki”.

Interfejsy `NavigableSet` i `NavigableMap` zawierają dodatkowe metody służące do znajdowania następnego lub poprzedniego elementu w posortowanych zbiorach i słownikach. Implementują je klasy `TreeSet` i `TreeMap`. Operacje nawigacji można efektywnie zaimplementować w drzewiastych strukturach danych.

W Javie 21 wprowadzono interfejs `SequencedCollection<E>`, który zapewnia jednolity dostęp do pierwszego i ostatniego elementu kolekcji oraz umożliwia przeglądanie wstecz.

```
E getFirst()
E getLast()
void addFirst(E e)
void addLast(E e)
E removeFirst()
E removeLast()
SequencedCollection<E> reversed()
```

Wcześniej te operacje były wykonywane przez różne metody list, zbiorów i kolejek dwukierunkowych. Pod interfejs precyzuje typ zwrotny metody `reversed` do `SequencedSet`. Interfejs `SequencedMap` ma analogiczne metody dla słowników.

9.3. Konkretnie klasy kolekcyjne

Tabela 9.1 przedstawia zestawienie kolekcji dostępnych w Java Collections Framework oraz ich krótkie opisy (dla uproszczenia pominęliśmy kolekcje bezpieczne dla wątków, które zostały opisane w rozdziale 10.).

Tabela 9.1. Kolekcje konkretne w bibliotece Javy

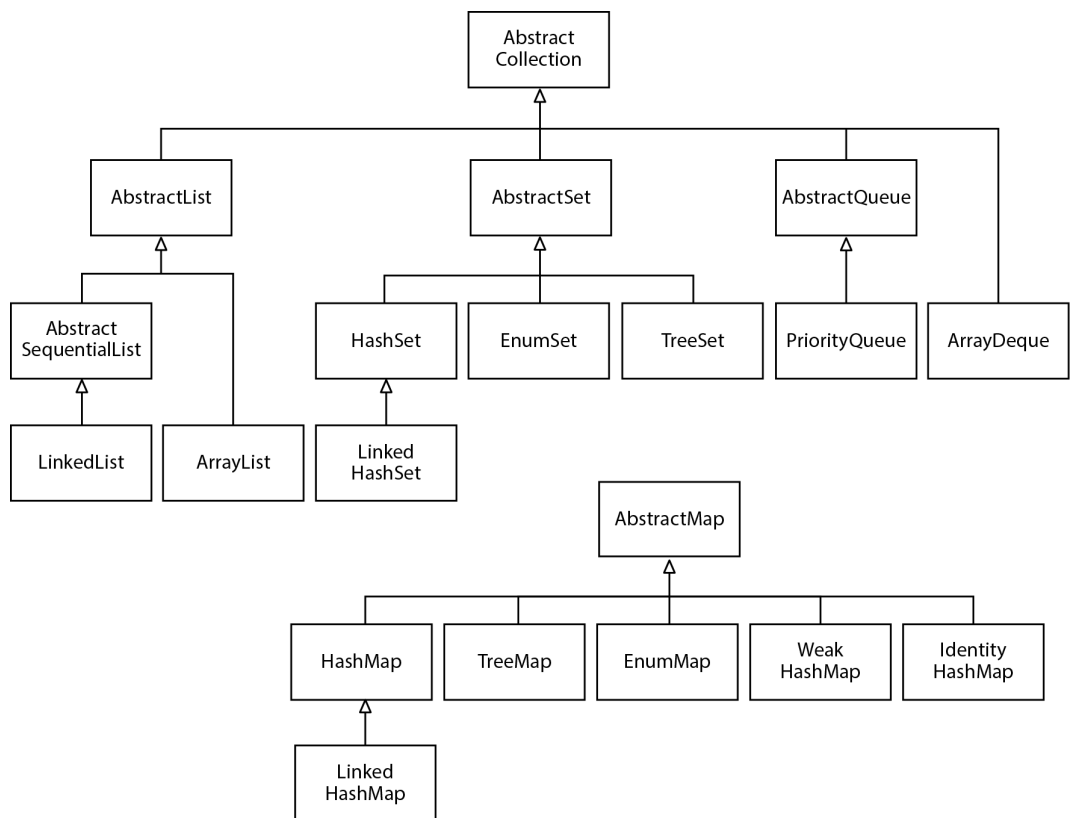
Typ kolekcji	Opis	Zobacz
ArrayList	Indeksowana lista o dynamicznie zmieniających się rozmiarach	Punkt 9.3.2
LinkedList	Uporządkowana lista pozwalająca na szybkie wstawianie i usuwanie elementów w dowolnej lokalizacji	Punkt 9.3.1
ArrayDeque	Nieposiadająca ani początku, ani końca lista cykliczna	Punkt 9.3.5
HashSet	Nieuporządkowana kolekcja, w której wszystkie obiekty muszą być unikatowe	Punkt 9.3.3
TreeSet	Uporządkowany zbiór	Punkt 9.4.5
EnumSet	Zbiór wartości typu wyliczeniowego	Punkt 9.4.6
LinkedHashSet	Zbiór pamiętający kolejność wstawianych do niego elementów	Punkt 9.4.5
PriorityQueue	Kolekcja pozwalająca na szybkie usunięcie najmniejszego elementu	Punkt 9.3.6
HashMap	Struktura danych przechowująca pary klucz – wartość	Punkt 9.4.1
TreeMap	Słownik sortujący klucze	Punkt 9.4.1
EnumMap	Słownik, w którym klucze są typami wyliczeniowymi	Punkt 9.4.6
LinkedHashMap	Słownik pamiętający kolejność wstawianych do niego elementów	Punkt 9.4.5
WeakHashMap	Słownik, którego wartości mogą zostać usunięte przez śmieciarkę, jeśli nie są używane gdzieś indziej	Punkt 9.4.4
IdentityHashMap	Słownik przechowujący klucze porównywane za pomocą operatora == zamiast metody equals	Punkt 9.4.7

Wszystkie klasy prezentowane w tabeli, z wyjątkiem tych, których nazwy kończą się słowem Map, implementują interfejs `Collection`. Słowniki implementują interfejs `Map` i zostały opisane w podrozdziale 9.4, „Słowniki”.

Na rysunku 9.5 pokazano relacje występujące między tymi klasami.

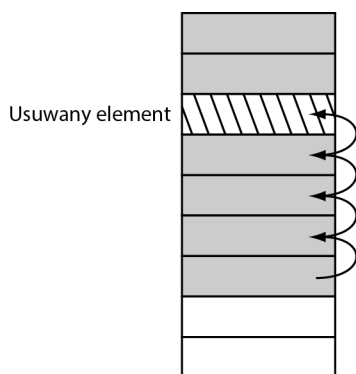
9.3.1. Listy powiązane

W wielu prezentowanych do tej pory przykładach kodu wykorzystywaliśmy tablice i ich dynamicznego krewniaka, czyli klasę `ArrayList`. Niestety tablice i listy tablicowe mają jedną poważną wadę — usuwanie elementów z ich środka jest mało efektywne, ponieważ czynność ta wymaga przesunięcia wszystkich elementów znajdujących się za tym usuwanym w stronę początku (zobacz rysunek 9.6). To samo dotyczy wstawiania elementów w środku struktury.

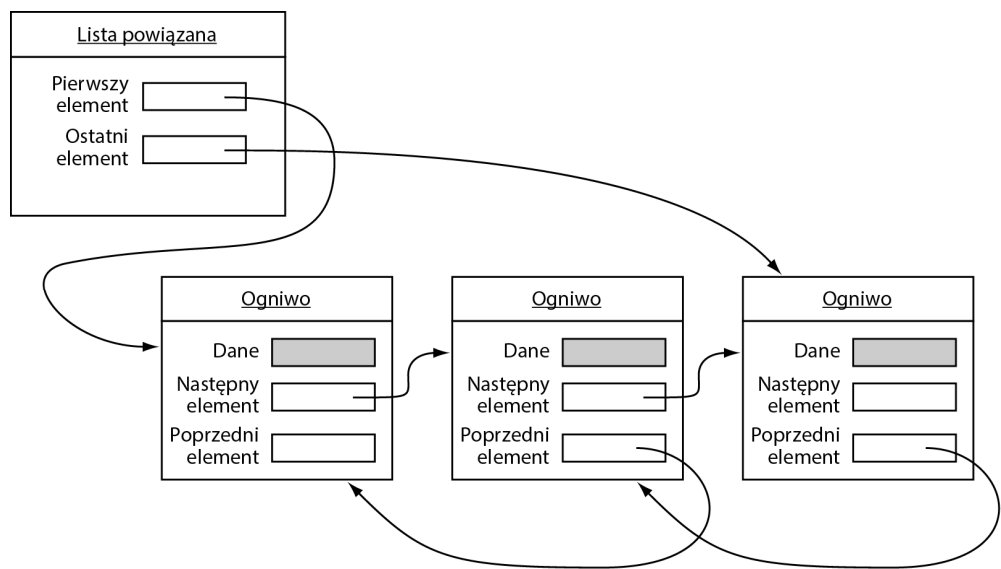


Rysunek 9.5. Klasy z systemu kolekcji Javy

Rysunek 9.6.
Usuwanie
elementu
z tablicy

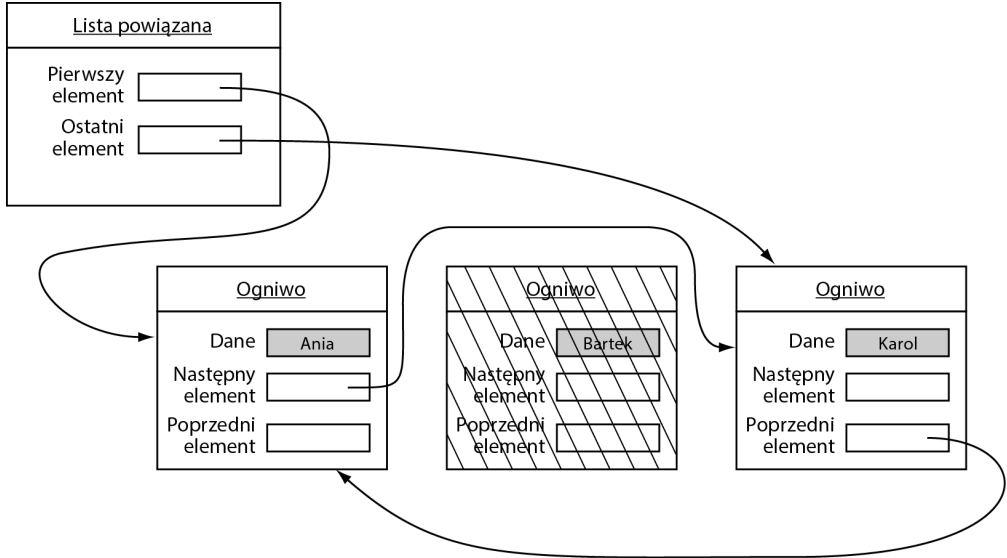


Problem ten pomaga rozwiązać inna powszechnie znana struktura danych, nazywana **listą powiązaną** (ang. *linked list*). Podczas gdy obiekty w tablicy są zapisywane w kolejnych komórkach pamięci, w liście powiązanej znajdują się one w osobnych **ogniwach** (ang. *link*). Każde ogniwo przechowuje także referencję do następnego ogniwa w szeregu. W Javie praktycznie wszystkie listy powiązane są **listami dwukierunkowymi** (ang. *doubly linked list*), co oznacza, że każde ogniwo przechowuje także referencję do ogniwa je poprzedzającego (zobacz rysunek 9.7).



Rysunek 9.7. Lista dwukierunkowa

Usuwanie elementów z środka listy powiązanej charakteryzuje się krótkim czasem wykonania, ponieważ aktualizowane są tylko elementy znajdujące się po obu stronach usuwanego obiektu (zobacz rysunek 9.8).



Rysunek 9.8. Usuwanie elementu z listy powiązanej

Wiele osób ukończyło kurs struktur danych, na którym uczy się implementacji list powiązanych. Niektórzy mogą mieć złe wspomnienia związane z plątaniną połączeń przy usuwaniu lub dodawaniu elementów do list powiązanych. Dla tych osób mamy dobrą wiadomość — w Java Collections Framework znajduje się gotowa do użycia klasa `LinkedList`.

Poniższy fragment programu dodaje do listy trzy elementy, a następnie usuwa drugi element:

```
var staff = new LinkedList<>(); // Klasa LinkedList implementuje interfejs List
staff.add("Ania");
staff.add("Bartek");
staff.add("Karol");
Iterator<String> iter = staff.iterator();
String first = iter.next(); // dojście do pierwszego elementu (Ania)
String second = iter.next(); // dojście do drugiego elementu (Bartek)
iter.remove(); // usunięcie ostatnio odwiedzonego elementu (Karol)
```

Pomiędzy listami powiązanymi a kolekcjami generycznymi istnieje pewna istotna różnica. Lista powiązana jest **kolekcją uporządkowaną**, w której położenie obiektów ma znaczenie. Metoda `LinkedList.add` dodaje obiekt na końcu listy. Często jednak elementy są wstawiane do środka listy. Metoda `add` związana z położeniem elementu znajduje się w zestawie obowiązków iteratora, ponieważ iteratory służą do opisu położenia elementów w kolekcjach. Zastosowanie iteratorów do dodawania elementów jest uzasadnione tylko w uporządkowanych kolekcjach. Na przykład omawiane w następnej kolejności *zbiory* są strukturami danych, które nie porządkują w żaden sposób swoich elementów. Dlatego Java Collections Framework dostarcza podinterfejs `List` ➤ `Iterator`, który zawiera metodę `add`.

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    . . .
}
```

W przeciwieństwie do metody `Collection.add`, ta metoda nie zwraca wartości logicznej — zakłada się, że metoda `add` zawsze modyfikuje listę.

Dodatkowo interfejs `ListIterator` zawiera dwie metody, za pomocą których można poruszać się po liście wstecz.

```
boolean hasPrevious()
E Previous()
```

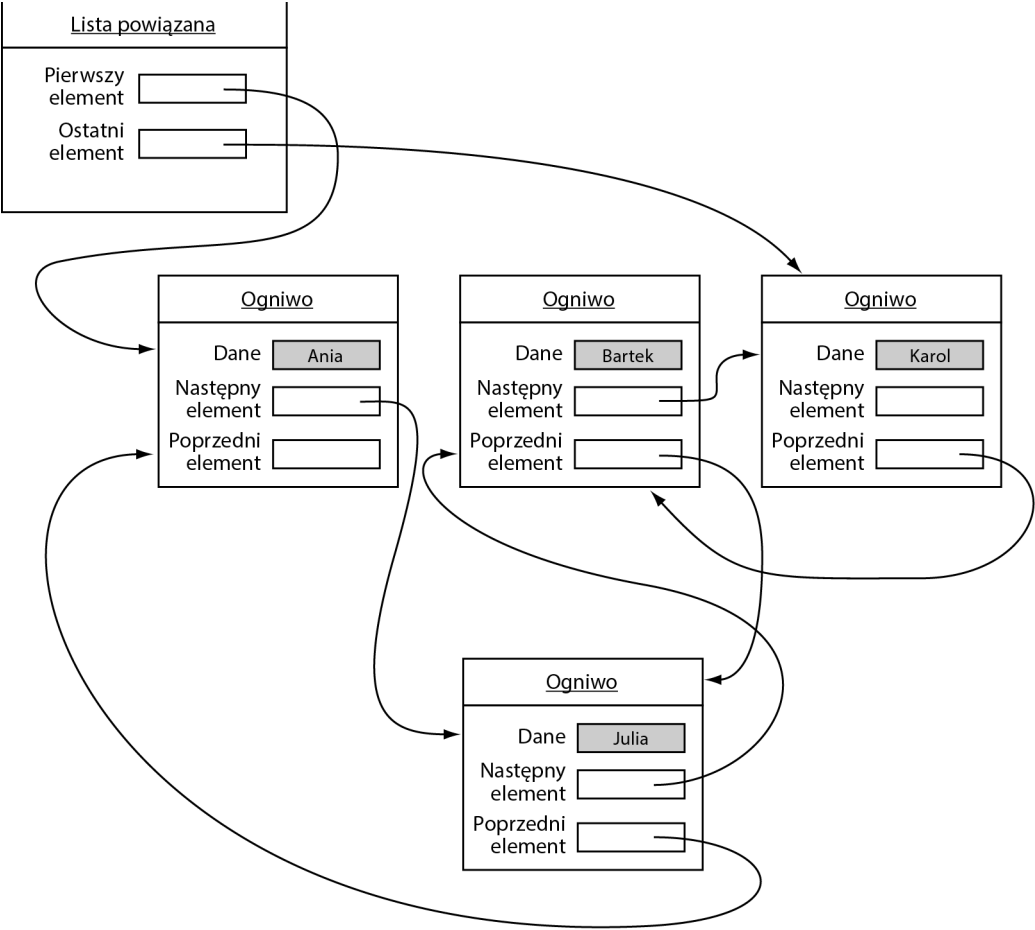
Metoda `previous`, podobnie jak `next`, zwraca obiekt, który właśnie przeskoczyła.

Metoda `listIterator` z klasy `LinkedList` zwraca obiekt iteratora implementujący interfejs `List` ➤ `Iterator`.

```
ListIterator<String> iter = staff.listIterator();
```

Metoda `add` dodaje element *przed* iteratorem. Na przykład poniższe instrukcje pomijają pierwszy element na liście powiązanej i dodają element `Julia` przed drugim elementem (zobacz rysunek 9.9):

```
var staff = new LinkedList<String>();
staff.add("Ania");
staff.add("Bartek");
staff.add("Karol");
ListIterator<String> iter = staff.listIterator();
iter.next(); // pominięcie pierwszego elementu
iter.add("Julia");
```



Rysunek 9.9. Dodawanie elementu do listy powiązanej

Jeśli metoda `add` zostanie wywołana kilka razy, elementy zostaną dodane do listy w kolejności podawania. Każdy z nich jest dodawany przed aktualnym miejscem pobytu iteratora.

Jeśli metoda `add` zostanie użyta z nieużywanym jeszcze iteratorem utworzonym przez metodę `listIterator`, wskazującym początek listy powiązanej, nowy element zostanie dodany na samym początku listy. Jeśli iterator przejdzie za ostatni element listy (czyli znajdzie się w miejscu, w którym metoda `hasNext` zwraca wartość `false`), dodawany element będzie stanowił nowy ogon listy. W liście o długości n istnieje $n+1$ miejsc, w których można wstawiać elementy. Ta liczba odpowiada liczbie położeń, w których może się znaleźć iterator. Jeśli na przykład lista powiązana zawiera trzy elementy A, B i C, to istnieją w niej cztery miejsca (oznaczone znakiem `|`), w których można wstawić nowy element:

```
|ABC
A|BC
AB|C
ABC|
```



Przy stosowaniu analogii do kursora trzeba zachować ostrożność. Metoda `remove` nie działa dokładnie tak jak klawisz *Backspace*. Rzeczywiście bezpośrednio po wywołaniu metody `next` usuwa ona element znajdujący się po lewej stronie iteratora, dokładnie jak klawisz *Backspace*. Jeśli jednak przed nią została wywołana metoda `previous`, zostanie usunięty element znajdujący się po prawej stronie iteratora. Ponadto metody `remove` nie można wywołać dwa razy z rzędu.

Metoda `remove`, w przeciwieństwie do `add`, polegającej wyłącznie na położeniu iteratora, bierze pod uwagę stan iteratora.

W końcu metoda `set` zastępuje ostatni element zwrócony przez metodę `next` lub `previous` nowym elementem. Na przykład poniższa procedura zastępuje pierwszy element listy nową wartością:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next();    // zwraca pierwszy element
iter.set(newValue);              // ustawia pierwszy element na newValue
```

Nietrudno sobie wyobrazić, że w sytuacji, w której jeden iterator przemierza kolekcję, a inny ją modyfikuje, mogą wystąpić nieprzyjemne zdarzenia. Wyobraźmy sobie, że jakiś iterator wskazuje miejsce przed elementem, który został usunięty przez inny iterator. Ten pierwszy iterator traci w takiej sytuacji rację bytu i nie powinien być używany. Iteratory list powiązanych zostały zaprojektowane w taki sposób, aby wykrywać tego typu modyfikacje. Jeśli iterator odkryje, że jego kolekcja została zmodyfikowana przez inny iterator lub metodę samej kolekcji, wyrzuca wyjątek `ConcurrentModificationException`. Przeanalizujmy poniższy przykładowy kod:

```
List<String> list = . . . ;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next();                // wyrzuca wyjątek ConcurrentModificationException
```

Wywołanie metody `iter2.next` powoduje wyjątek `ConcurrentModificationException`, ponieważ iterator `iter2` odkrył, że lista została zmodyfikowana przez jakiś czynnik zewnętrzny.

Aby uniknąć tego typu wyjątków, należy stosować się do prostej reguły: do kolekcji można wstawić dowolną liczbę iteratorów, pod warunkiem że służą one tylko do odczytu. Alternatywnie jeden z nich może być zarówno do odczytu, jak i zapisu.

Jednoczesne operacje modyfikujące są wykrywane w bardzo prosty sposób. Kolekcja pamięta liczbę operacji modyfikujących (takich jak dodawanie i usuwanie elementów). Każdy iterator pamięta, ile tego typu operacji *wykonał*. Przed przystąpieniem do działania iterator sprawdza, czy jego liczba zgadza się z liczbą kolekcji. Jeśli nie, zgłasza wyjątek `ConcurrentModificationException`.



Od operacji wykrywania jednoczesnych operacji modyfikujących istnieje jeden ciekawy wyjątek. Lista powiązana zapamiętuje tylko zmiany *struktury*, jak dodawanie i usuwanie ogniw. Metoda `set` *nie* liczy się jako modyfikacja strukturalna. W liście powiązanej może być kilka iteratorów zmieniających zawartość ogniw za pomocą metody `set`. To zachowanie jest potrzebne w kilku algorytmach w klasie `Collections`, które opisujemy dalej.

Poznaliśmy wszystkie podstawowe metody klasy `LinkedList`. Do przemierzania jej w dowolnym kierunku i dodawania oraz usuwania elementów służy obiekt klasy `ListIterator`.

Wiemy już z podrozdziału 9.2, że wiele przydatnych metod do działań na listach powiązanych znajduje się w interfejsie `Collection`. Większość z nich jest zdefiniowana w nadklasie `AbstractCollection` klasy `LinkedList`. Na przykład metoda `toString` wywołuje metodę `toString` na rzecz każdego elementu i tworzy jeden długi łańcuch w formacie `[A, B, C]`, co przydaje się podczas debugowania. Aby sprawdzić, czy na liście znajduje się określony element, należy użyć metody `contains`. Na przykład instrukcja `staff.contains("Henryk")` zwróci wartość `true`, jeśli lista powiązana zawiera łańcuch `Henryk`.

Java Collections Framework zawiera także kilka wątpliwych z teoretycznego punktu widzenia metod. Listy powiązane nie umożliwiają szybkiego dostępu do dowolnego elementu. Aby dotrzeć do n -tego elementu listy, trzeba zacząć wędrówkę od początku i ominąć pierwszych $n-1$ elementów. Nie da się nic skrócić. Z tego powodu listy powiązane rzadko są używane w sytuacjach, w których potrzebny jest dostęp do elementów za pomocą indeksu całkowitoliczbowego.

Pomimo tego klasa `LinkedList` udostępnia metodę `get`, która pozwala na dostęp do wybranego elementu:

```
LinkedList<String> list = . . . ;
String obj = list.get(n);
```

Oczywiście efektywność tej metody jest niska. Jeśli jej używasz, to zastanów się, czy nie należałoby zmienić zastosowanej struktury danych do rozwiązywanego problemu.

Nigdy nie należy przemierzać listy powiązanej za pomocą tej metody, dającej złudzenie swobodnego dostępu. Poniższy kod jest niebywale powolny:

```
for (int i = 0; i < list.size(); i++)
    operacje_na_elementie list.get(i);
```

Za każdym razem, kiedy potrzebujemy nowego elementu, poszukiwanie zaczyna się od początku listy. Obiekty `LinkedList` nie zapisują informacji o pozycji.



Metoda `get` posiada jedną niewielką optymalizację — jeśli wartość indeksu wynosi co najmniej `size()/2`, szukanie elementu zaczyna się od końca listy.

Interfejs `Iterator` listy zawiera także metodę informującą o aktualnym położeniu iteratora. W rzeczywistości, ze względu na to, że iteratory w Javie wskazują miejsce pomiędzy elementami, istnieją dwie takie metody. Metoda `nextIndex` zwraca indeks całkowitoliczbowy elementu, który zostałby zwrócony przez następne wywołanie metody `next`. Metoda `previousIndex` zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `previous`. Oczywiście wartość ta jest o jeden mniejsza od wartości `nextIndex`. Metody te są efektywne jako iteratory struktur `LinkedList`, ponieważ iteratory pamiętają swoją aktualną pozycję. W końcu, jeśli mamy indeks n , instrukcja `list.listIterator(n)` zwróci iterator wskazujący miejsce bezpośrednio przed elementem znajdującym się w polu o indeksie n . Oznacza to, że metoda `next` zwraca ten sam wynik co wywołanie `list.get(n)`.

Dysponując listą powiązaną zawierającą tylko kilka elementów, nie trzeba przesadnie obawiać się braku szybkości metod `get` i `set`. Po co więc w ogóle w takiej sytuacji używać listy powiązanej? Jedynym powodem do używania list powiązanych jest szybkość wstawiania i usuwania elementów do i ze środka kolekcji. Jeśli masz tylko kilka elementów, możesz użyć struktury `ArrayList`.

Zalecamy unikanie wszystkich metod określających położenie na liście za pomocą indeksów. Aby mieć wolny dostęp do wszystkich elementów kolekcji, należy używać obiektów `ArrayList` zamiast list powiązanych.

Program przedstawiony na listingu 9.1 demonstruje praktyczne zastosowanie list powiązanych. Tworzy on dwie listy, scala je, usuwa co drugi element z drugiej z nich, a na końcu testuje działanie metody `removeAll`. Zachęcamy do prześledzenia przepływu sterowania w tym programie i przyjrzenia się iteratorom. Pomocne mogą się okazać rysunki przedstawiające położenie iteratorów:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
...
```

Listing 9.1. `LinkedList/LinkedListTest.java`

```
package linkedList;

import java.util.*;

/**
 * Program demonstrujący działania na listach powiązanych
 * @version 1.13 2023-12-05
 * @author Cay Horstmann
 */
public class LinkedListTest
{
    public static void main(String[] args)
    {
        var a = new LinkedList<String>();
        a.add("Ania");
        a.add("Karol");
        a.add("Eryk");

        var b = new LinkedList<String>();
        b.add("Bartek");
        b.add("Daniel");
        b.add("Franek");
        b.add("Gosia");

        // Scalenie list a i b

        ListIterator<String> aIter = a.listIterator();
        Iterator<String> bIter = b.iterator();

        while (bIter.hasNext())
        {
            if (aIter.hasNext()) aIter.next();
            aIter.add(bIter.next());
        }
    }
}
```

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

// Usunięcie co drugiego słowa z listy b

bIter = b.iterator();
while (bIter.hasNext())
{
    bIter.next(); // Opuszczenie jednego elementu
    if (bIter.hasNext())
    {
        bIter.next(); // Opuszczenie następnego elementu
        bIter.remove(); // Usunięcie elementu
    }
}

System.out.println(b);

// Usunięcie wszystkich słów znajdujących się w liście b z listy a

a.removeAll(b);

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

Zwróć uwagę, że instrukcja `System.out.println(a)` drukuje wszystkie elementy listy powiązanej `a` za pomocą metody `toString` z klasy `AbstractCollection`.

`java.util.List<E>` **1.2**

■ `ListIterator<E> listIterator()`

Zwraca iterator listy.

■ `ListIterator<E> listIterator(int index)`

Zwraca iterator listy, którego pierwsze wywołanie metody `next` zwróci element znajdujący się pod podanym indeksem.

■ `void add(int i, E element)`

Dodaje element w określonym miejscu.

■ `boolean addAll(int i, Collection<? extends E> elements)`

Dodaje wszystkie elementy z kolekcji w określonym miejscu.

■ `E remove(int i)`

Usuwa i zwraca element znajdujący się w określonym miejscu.

■ `E get(int i)`

Zwraca element znajdujący się w określonym miejscu.

■ `E set(int i, E element)`

Zastępuje element znajdujący się w określonym miejscu nowym elementem i zwraca stary element.

■ `int indexOf(Object element)`

Zwraca położenie pierwszego wystąpienia elementu `element` lub wartość `-1`, jeśli `element` nie zostanie znaleziony.

■ `int lastIndexOf(Object element)`

Zwraca położenie ostatniego wystąpienia elementu `element` lub wartość `-1`, jeśli `element` nie zostanie znaleziony.

java.util.ListIterator<E> 1.2

■ `void add(E newElement)`

Dodaje element przed bieżącą pozycją.

■ `void set(E newElement)`

Zastępuje ostatni element odwiedzony przez metodę `next` lub `previous` nowym elementem. Zgłasza wyjątek `IllegalStateException`, jeśli lista została zmodyfikowana od czasu ostatniego wywołania metody `next` lub `previous`.

■ `boolean hasPrevious()`

W trakcie przemierzania listy wstecz zwraca wartość `true`, jeśli są jeszcze nieodwiedzone elementy.

■ `E previous()`

Zwraca poprzedni obiekt. W przypadku osiągnięcia początku listy zgłasza wyjątek `NoSuchElementException`.

■ `int nextIndex()`

Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `next`, lub zwraca rozmiar listy, jeśli iterator wyszedł już poza ostatni element.

■ `int previousIndex()`

Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `previous` lub `-1`, jeśli iterator znajduje się przed pierwszym elementem.

java.util.LinkedList<E> 1.2

■ `LinkedList()`

Tworzy pustą listę powiązaną.

■ `LinkedList(Collection<? extends E> elements)`

Tworzy listę powiązaną i dodaje do niej wszystkie elementy z określonej kolekcji.

■ `void addFirst(E element)`

■ `void addLast(E element)`

Dodaje element na początku lub końcu listy.

- `E getFirst()`

- `E getLast()`

Zwraca element znajdujący się na początku lub końcu listy.

- `E removeFirst()`

- `E removeLast()`

Usuwa element z początku lub końca listy.

9.3.2. Listy tablicowe

W poprzednim podrozdziale zapoznaliśmy się z interfejsem `List` i klasą `LinkedList`, która go implementuje. Interfejs ten opisuje uporządkowaną kolekcję, w której położenie elementów odgrywa rolę. Elementy można odwiedzać na dwa sposoby: za pomocą iteratora lub dowolnie przy użyciu metod `get` i `set`. Drugi z wymienionych sposobów nie nadaje się do list powiązanych, natomiast doskonale sprawdza się w tablicach. W bibliotece kolekcji znajduje się dobrze znana nam klasa `ArrayList`, która również implementuje interfejs `List`. Obiekt tej klasy zawiera dynamiczną tablicę obiektów.



W Javie 1.0 była inna kolekcja bazująca na listach — klasa `Vector`. Wszystkie metody klasy `Vector` są *synchronizowane*, dzięki czemu można bezpiecznie uzyskać dostęp do jednego obiektu tej klasy z dwóch wątków. Jeśli natomiast pracujemy tylko w jednym wątku — co zdarza się nieporównywalnie częściej — tracimy bardzo dużo czasu na synchronizację. Natomiast metody klasy `ArrayList` nie są synchronizowane. Jak zobaczysz w rozdziale 10., obecnie klasy metody są bezpieczne dla wątków. Nie ma już sensu używać klasy `Vector`, chyba że program musi współpracować z bardzo starym API.

9.3.3. Zbiór `HashSet`

W listach powiązanych i tablicach istnieje możliwość ustawienia przechowywanych w nich elementów w określonej kolejności. Aby jednak znaleźć jakiś element o nieznanym położeniu, trzeba przejść przez wszystkie elementy w jego poszukiwaniu. Jeśli kolekcja zawiera dużo elementów, operacja ta może zająć bardzo dużo czasu. Istnieją struktury danych, które pozwalają znacznie szybciej znajdować elementy, ale nie umożliwiają ich ustawiania w wybranej kolejności. Elementy w nich są ustawiane w takiej kolejności, która odpowiada ich własnemu celom.

Powszechnie znaną strukturą danych pozwalającą szybko wyszukiwać obiekty jest **tablica skrótów** (ang. *hash table*; nazywana też mieszającą). Tablica ta oblicza liczbę całkowitą, zwaną **skrót**em (ang. *hash code*), dla każdego obiektu. Skrót powstaje w jakiś sposób z pól obiektu, najlepiej w taki, aby obiekty zawierające różne dane dawały różne kody. Tabela 9.2 zawiera kilka przykładowych skrótów zwróconych przez metodę `hashCode` z klasy `String`.

Tabela 9.2. Skróty zwrócone przez metodę hashCode

Łańcuch	Skrót
Lee	76268
lee	107020
eel	100300

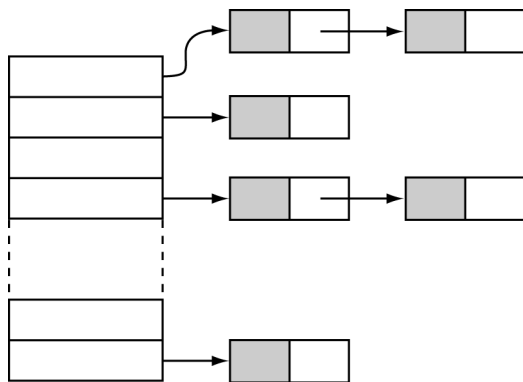
Definiując własne klasy, należy zaimplementować własną metodę hashCode — więcej na ten temat można znaleźć w rozdziale 5. Implementacja ta musi być zgodna z metodą equals — jeśli a jest równe b ($a.equals(b)$), to a i b muszą mieć ten sam skrót.

To, co jest ważne teraz, to fakt, że obliczanie skrótu przebiega szybko, a wynik tego działania jest uzależniony tylko od stanu obiektu, którego kod jest tworzony, a nie od innych obiektów w tablicy.

Tablice skrótów w Javie są zaimplementowane jako tablice list powiązanych. Listy w tej sytuacji noszą miano **komórek** lub **kubelków** (ang. *bucket*; zobacz rysunek 9.10). Aby określić miejsce do przechowywania obiektu w tablicy, należy obliczyć jego skrót i znaleźć resztę z dzielenia tej liczby przez liczbę wszystkich komórek. Jeśli na przykład obiekt ma skrót 76268, a wszystkich komórek jest 128, zostanie on umieszczony w komórce 108 (ponieważ reszta z dzielenia 76 268 przez 128 wynosi 108). Jeśli mamy szczęście i nie ma w tej komórce żadnego innego elementu, umieszczamy tam nasz obiekt. Oczywiście nie da się uniknąć trafienia komórki ze znajdującym się wewnątrz obiektem. Sytuację taką nazywamy **kolizją**. Wtedy porównujemy nowy obiekt z wszystkimi obiektami w tej komórce, aby sprawdzić, czy go tam jeszcze nie ma. Jeśli skróty są rozsądnie rozłożone losowo, a liczba komórek wystarczająco duża, powinno być konieczne przeprowadzenie tylko kilku porównań.

Rysunek 9.10.

Tablica skrótów



W implementacji HashMap kubelki po przekroczeniu pewnego progu zmieniają się z list powiązanych w zbalansowane drzewa binarne. Pozwala to podnieść wydajność w przypadku nieefektywnych funkcji obliczania skrótów, które powodują wiele kolizji, oraz stanowi zabezpieczenie, na wypadek gdyby złośliwy kod próbował zalać tablicę skrótów wieloma wartościami o identycznych skrótach.



Jeśli jest taka możliwość, klucze tablicy skrótów powinny należeć do klasy implementującej interfejs `Comparable`. Daje to gwarancję, że wydajność programu nie ucierpi z powodu niekorzystnej dystrybucji kodów skrótu.

Jeśli tablica zostanie przepełniona, konieczna jest jej **reorganizacja**. Polega to na utworzeniu nowej tablicy z większą liczbą komórek i przeniesieniu wszystkich danych do tej nowej tablicy. Stara tablica zostaje usunięta. O reorganizacji tablicy decyduje **współczynnik zapęłnienia** (ang. *load factor*). Jeśli na przykład współczynnik ten wynosi 0,75 (wartość domyślna), a tablica zostanie zapęłniona w ponad 75 procentach, następuje jej automatyczna reorganizacja, w wyniku której tworzona jest tablica o dwukrotnie większej liczbie komórek. W większości zastosowań najlepiej pozostawić współczynnik 0,75 bez zmian.

Aby zyskać większą kontrolę nad działaniem tablicy, można określić początkowy licznik komórek. Należy go ustawić na spodziewaną liczbę elementów podzieloną przez współczynnik zapęłnienia. Albo, od Javy 19, można użyć statycznej metody pomocniczej `newHashSet`, która wykonuje to obliczenie za nas:

```
HashSet<String> strings = HashSet.newHashSet(expectedElementCount);
```



Niektórzy badacze utrzymują, że dobrze jest ustawić ten licznik na liczbę pierwszą, co pozwoli uniknąć grupowania się kluczy, jednak nie ma na to przekonujących dowodów. Java Collections Framework stosuje liczniki komórek będące potęgami liczby 2, a domyślna liczba to 16 (każda wartość podana jako wielkość tablicy jest automatycznie zaokrąglana do najbliższej potęgi dwójki).

Za pomocą tablic skrótów można zaimplementować kilka bardzo ważnych struktur danych. Najprostszą z nich jest **zbiór** (ang. *set*). Zbiór to kolekcja unikatowych elementów. Metoda `add` zbioru najpierw sprawdza, czy w zbiorze nie ma wstawianego obiektu, i wstawia go, jeśli nic nie znajdzie.

W Java Collections Framework znajduje się klasa `HashSet`, która implementuje zbiór bazujący na tablicy skrótów. Dodawanie elementów do tego zbioru odbywa się za pomocą metody `add`. Metoda `contains` została zdefiniowana w taki sposób, że szybko sprawdza, czy dodawany element nie znajduje się już w zbiorze. Sprawdza tylko elementy w jednej komórce, a nie całej kolekcji.

Iterator zbioru `HashSet` odwiedza wszystkie komórki po kolei. Ponieważ elementy te są porzucane po tablicy, iterator odwiedza je w losowej kolejności. Dlatego zbioru `HashSet` należy używać wyłącznie wówczas, gdy kolejność elementów w kolekcji nie jest ważna.

Przykładowy program zaprezentowany na końcu tego podrozdziału (listing 9.2) wczytuje słowa z pliku, dodaje je do zbioru, a następnie drukuje je na ekranie. Można na przykład wprowadzić tekst *Alice in Wonderland* za pomocą poniższego polecenia:

```
java set.SetTest ../gutenberg/alice30.txt
```

Książka *Alice in Wonderland* zawiera 28 195 słów, w tym 5909 unikalnych, wliczając uwagę o prawach autorskich, która znajduje się na początku. Słowa są ułożone w losowej kolejności. Następnie zostają wstawione do zbioru drzewiastego, który przechowuje je w kolejności posortowanej, jak zobaczysz w następnym punkcie.

Listing 9.2. set/SetTest.java

```

package set;

import java.io.IOException;
import java.nio.file.Path;
import java.util.*;

/**
 * Program porównujący operację wstawiania do zbioru skrótów i zbioru drzewiastego.
 * Sposób uruchomienia:
 * java set.SetTest ../gutenberg/alice30.txt
 * java set.SetTest ../gutenberg/crsto10.txt 100
 * @version 1.2 2023.12.07
 * @author Cay Horstmann
 */
public class SetTest
{
    public static void time(Set<String> wordSet, List<String> wordList, int repetitions)
    {
        long totalTime = 0;
        for (int i = 1; i <= repetitions; i++)
        {
            for (String word : wordList)
            {
                long start = System.nanoTime();
                wordSet.add(word);
                long end = System.nanoTime();
                totalTime += end - start;
            }
        }
        Iterator<String> iter = wordSet.iterator();
        for (int i = 1; i <= 20 && iter.hasNext(); i++)
            System.out.print(iter.next() + " ");
        System.out.println("...");
        System.out.printf("%s: %d słów, %d unikalnych, %.3f sekund.%n",
            wordSet.getClass().getSimpleName(), wordList.size(), wordSet.size(),
            totalTime * 1E-9);
    }

    public static void main(String[] args) throws IOException
    {
        List<String> words = new ArrayList<>();
        String filename = args.length > 0 ? args[0] : "../gutenberg/crsto10.txt";
        int repetitions = args.length > 1 ? Integer.parseInt(args[1]) : 1;
        try (var in = new Scanner(Path.of(filename))) {
            while (in.hasNext()) {
                String word = in.next();
                words.add(word);
            }
        }
        time(new HashSet<>(), words, repetitions);
        time(new TreeSet<>(), words, repetitions);
    }
}

```



Uważaj, kiedy modyfikujesz elementy zbioru. Jeśli skrót elementu się zmieni, to element ten przestanie znajdować się na właściwej pozycji w strukturze danych.

```
var rects = new HashSet<Rectangle>();
var rect = new Rectangle(5, 10, 20, 30);
rects.add(rect);
rect.setLocation(0, 0);
// rects to teraz [java.awt.Rectangle[x=0,y=0,width=20,height=30]]
rects.remove(new Rectangle(0, 0, 20, 30)); // Zwraca false, nie usuwa
```

java.util.HashSet<E> 1.2

■ HashSet()

Tworzy pusty obiekt HashSet.

■ HashSet(Collection<? extends E> elements)

Tworzy obiekt HashSet i wstawia do niego wszystkie elementy określonej kolekcji.

■ HashSet(int initialCapacity)

■ HashSet(int initialCapacity, float loadFactor)

Tworzy obiekt HashSet o określonej pojemności i z określonym współczynnikiem zapełnienia lub współczynnikiem 0.75. Jeśli współczynnik rozmiaru do pojemności przekroczy współczynnik zapełnienia, tablica zostaje przeorganizowana i zamieniona na większą.

■ static <E> HashSet<E> newHashSet(int numMappings) 19

Tworzy pusty zbiór skrótów o pojemności wystarczającej do przechowywania spodziewanej liczby elementów (numMappings) bez reorganizacji.

java.lang.Object 1.0

■ int hashCode()

Zwraca skrót obiektu. Skrót może być dowolną liczbą całkowitą, także ujemną. Definicje metod equals i hashCode muszą być ze sobą zgodne — jeśli instrukcja `x.equals(y)` zwraca wartość `true`, to instrukcja `x.hashCode()` musi zwracać taką samą wartość jak `y.hashCode()`.

9.3.4. Zbiór TreeSet

Klasa `TreeSet` jest podobna do klasy `HashSet`, ale ma jedną zaletę w stosunku do niej. Zbiór `TreeSet` jest **zbiorem uporządkowanym**. Mimo że elementy dodaje się do niego w dowolnej kolejności w trakcie iteracji przez zbiór, są one automatycznie prezentowane w uporządkowanej kolejności. Wyobraźmy sobie na przykład, że dodajemy do zbioru trzy łańcuchy, a następnie odwiedzamy każdy z nich.

```
var sorter = new TreeSet<String>();
sorter.add("Bartek");
```

```
sorter.add("Ania");
sorter.add("Karol");
for (String s : sorter) System.out.println(s);
```

Wartości zostaną wydrukowane w kolejności alfabetycznej: Ania, Bartek, Karol. Jak wskazuje sama nazwa klasy, elementy w tym zbiorze są przechowywane w strukturze drzewiastej (obecnie używane są **drzewa czerwono-czarne**; ich opis można znaleźć na przykład w książce pod tytułem *Wprowadzenie do algorytmów*, której autorami są Thomas Cormen, Charles Leiserson, Ronald Rivest i Clifford Stein, WNT, Warszawa 2007). Każdy nowy element jest umieszczany w odpowiednim miejscu zgodnie z kolejnością. Dzięki temu elementy odwiedzane przez iterator są zawsze posortowane.

Operacja dodawania elementów do drzewa jest wolniejsza niż w przypadku tablicy skrótów, ale i tak znacznie szybsza niż dodawanie ich w odpowiednim miejscu tablicy lub listy powiązanej. Jeśli drzewo zawiera n elementów, znalezienie odpowiedniego położenia dla nowego elementu wymaga około $\log_2 n$ testów. Jeśli na przykład drzewo zawiera 1000 elementów, dodanie nowego elementu wiąże się z przeprowadzeniem około 10 porównań.



Ze zbioru drzewiastego można korzystać tylko wtedy, gdy jest możliwość porównywania elementów. Oznacza to, że elementy muszą implementować interfejs `Comparable` lub programista przy tworzeniu zbioru musi dostarczyć komparator (interfejsy `Comparable` i `Comparator` są opisane w rozdziale 6.).

O ile wolniejszy jest zbiór drzewiasty? Uruchom program na listingu 9.2 w następujący sposób:

```
java set.SetTest ../gutenberg/crsto10.txt 100
```

Książka *The Count of Monte Cristo* zawiera 466 300 słów, z których każde zostaje wstawione 100 razy. Na moim testowym komputerze `TreeSet` jest prawie 4 razy wolniejszy. Jeśli nie potrzebujesz sortowania danych, nie ma sensu marnować na to czasu. Co więcej, w niektórych przypadkach łatwiej jest napisać funkcję obliczającą skróty niż sortującą. Funkcja obliczająca skróty musi tylko dobrze mieszać obiekty, natomiast funkcja porównująca musi je precyzyjnie rozróżniać.

Przeanalizujmy na przykład przypadek zbioru prostokątów. Jeśli użyjemy zbioru `TreeSet`, musimy dostarczyć obiekt `Comparator<Rectangle>`. Powstaje pytanie, jakie kryteria zastosować do porównywania tych figur. Można na przykład porównywać pola powierzchni, ale to nie jest dobre rozwiązanie, ponieważ dwa inaczej wyglądające prostokąty o innych współrzędnych mogą mieć takie samo pole, a my chcemy zachować oba. Komparator musi być kompatybilny z metodą `equals`. To znaczy, że wynikiem porównywania może być zero tylko wtedy, gdy obiekty są równe. Istnieje porządek, który nadaje się do zastosowania z prostokątami (porządek leksyko-graficzny dla współrzędnych), ale jest on nienaturalny i sprawia trudności obliczeniowe. Funkcja obliczająca skróty jest już natomiast zdefiniowana w klasie `Rectangle`. Jej działanie polega na mieszaniu współrzędnych.



Jeśli komparator jest niekompatybilny z metodą `equals` typu elementów, porównywanie według wzoru może dawać niespójne wyniki. Oto typowy przykład:

```
var words = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
```

Intencją jest nie zbierać elementów, które różnią się tylko wielkością liter:

```
commands.add("ZAMKNIJ");
commands.add("zamknij"); // Nie zostanie dodany.
```

Teraz jednak metoda `equals` nie jest już symetryczna:

```
commands.equals(Set.of("zamknij")) // prawda
Set.of("zamknij").equals(commands) // fałsz
```

`TreeSet` uznaje dwa elementy za równe tylko wtedy, gdy komparator zwróci zero. Natomiast drugi zbiór używa metody `String.equals`.

Lepszym podejściem jest użycie porządku `String.compare` i wstawianie do zbioru tylko łańcuchów składających się z małych liter.



Klasa `TreeSet` implementuje interfejs `NavigableSet`. Znajduje się w nim kilka bardzo przydatnych metod służących do lokalizowania sąsiadujących elementów. Szczegółowe informacje na ten temat znajdują się w wyciągach z API.

Program przedstawiony na listingu 9.3 tworzy dwa zbiory `TreeSet` wypełnione obiektami `Item`. Pierwszy z nich jest sortowany według numerów części, czyli w domyślny sposób. Drugi natomiast posortowano według opisów za pomocą niestandardowego komparatora.

Listing 9.3. `treeSet/TreeSetTest.java`

```
package treeSet;

import java.util.*;

/**
 * Program sortujący zbiór obiektów typu Item przez porównywanie ich opisów.
 * @version 1.2 2024-12-05
 * @author Cay Horstmann
 */
public class TreeSetTest
{
    public static void main(String[] args)
    {
        var parts = new TreeSet<Item>();
        parts.add(new Item("Toster", 1234));
        parts.add(new Item("Wihajster", 4562));
        parts.add(new Item("Router", 9912));
        System.out.println(parts);

        var sortByDescription = new TreeSet<Item>(Comparator.comparing(Item::description));
        sortByDescription.addAll(parts);
        System.out.println(sortByDescription);
    }
}
```

Na listingu 9.4 przedstawiona jest klasa `Item`.

Listing 9.4. `treeSet/Item.java`

```
package treeSet;

/**
 * Pozycja z opisem i numerem katalogowym.
 */
public record Item(String description, int partNumber) implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        int diff = Integer.compare(partNumber, other.partNumber);
        return diff != 0 ? diff : description.compareTo(other.description);
    }
}
```

`java.util.TreeSet<E>` 1.2

- `TreeSet()`
- `TreeSet(Comparator<? super E> comparator)`
- `TreeSet(Collection<? extends E> elements)`
- `TreeSet(SortedSet<E> s)`

Tworzy pusty zbiór drzewiasty.

Tworzy zbiór drzewiasty i wstawia do niego wszystkie elementy z kolekcji lub posortowanego zbioru (w drugim przypadku stosuje niezmienny porządek).

`java.util.SortedSet<E>` 1.2

- `Comparator<? super E> comparator()`

Zwraca komparator, który został użyty do posortowania elementów, lub wartość `null`, jeśli elementy są porównywane za pomocą metody `compareTo` z interfejsu `Comparable`.

- `E first()`
- `E last()`

Zwraca najmniejszy lub największy element posortowanego zbioru.

`java.util.NavigableSet<E>` 6

- `E higher(E value)`
- `E lower(E value)`

Zwraca najmniejszy element większy od wartości `value` lub największy element mniejszy od wartości `value`, lub wartość `null`, jeśli nie ma takiego elementu.

- `E ceiling(E value)`

- `E floor (E value)`

Zwraca najmniejszy element większy od lub równy wartości `value` lub największy element mniejszy od lub równy wartości `value`, lub wartość `null`, jeśli nie ma takiego elementu.

- `E pollFirst()`

- `E pollLast()`

Usuwa i zwraca najmniejszy lub największy element zbioru lub wartość `null`, jeśli zbiór jest pusty.

9.3.5. Kolejki Queue i Deque

Z wcześniejszych podrozdziałów wiemy już, że kolejka jest strukturą danych pozwalającą na szybkie dodawanie elementów na końcu i usuwanie elementów z początku. Kolejka dwukierunkowa (**deque**) pozwala na szybkie dodawanie i usuwanie elementów po obu stronach. Nie można natomiast dodawać elementów w środku kolejki. W Java SE 6 zaprezentowano nowy interfejs o nazwie `Deque`. Implementują go klasy `ArrayDeque` i `LinkedList`. Służą one do tworzenia kolejek dwukierunkowych, które mogą zmieniać swój rozmiar w zależności od zapotrzebowania. W rozdziale 10. piszemy jeszcze o kolejkach `Queue` i `Deque` z ograniczeniami.

W Javie 21 metody

```
E getFirst()
E getLast()
void addFirst(E e)
void addLast(E e)
E removeFirst()
E removeLast()
```

zostały przeniesione z interfejsu `Deque` do `SequencedCollection`, który rozszerzają także interfejsy `List` i `Set`. Zapewnia to jednolity zestaw metod dostępu do pierwszego i ostatniego elementu kolekcji.

```
java.util.Queue<E> 5.0
```

- `boolean add(E e)`

- `boolean offer(E e)`

Wstawia element za ostatnim elementem kolejki i zwraca wartość `true`, jeśli kolejka nie jest pełna. Jeśli w kolejce nie ma miejsca, pierwsza z powyższych metod zgłasza wyjątek `IllegalStateException`, a druga wartość `false`.

- `E remove()`

- `E poll()`

Usuwa i zwraca pierwszy element kolejki, jeśli nie jest ona pusta. Jeśli kolejka jest pusta, pierwsza z powyższych metod zgłasza wyjątek `NoSuchElementException`, a druga wartość `null`.

- `E element()`

- `E peek()`

Zwraca element znajdujący się na czole kolejki (ale go nie usuwa), pod warunkiem że kolejka nie jest pusta. Jeśli kolejka jest pusta, pierwsza z powyższych metod zgłasza wyjątek `NoSuchElementException`, a druga wartość `null`.

`java.util.SequencedCollection<E>` 21

- `void addFirst(E element)`

- `void addLast(E element)`

Dodaje dany element przed pierwszym lub za ostatnim elementem kolekcji. Jeśli kolekcja jest pełna, zgłasza wyjątek `IllegalStateException`.

- `E removeFirst()`

- `E removeLast()`

Usuwa i zwraca pierwszy lub ostatni element kolekcji, pod warunkiem że kolekcja ta nie jest pusta. Jeśli kolekcja jest pusta, zgłasza wyjątek `NoSuchElementException`.

- `E getFirst()`

- `E getLast()`

Zwraca pierwszy lub ostatni element kolekcji bez jego usunięcia, pod warunkiem że kolekcja nie jest pusta. Jeśli kolekcja jest pusta, zgłasza wyjątek `NoSuchElementException`.

`java.util.Deque<E>` 6

- `boolean offerFirst(E element)`

- `boolean offerLast(E element)`

Dodaje element przed pierwszym lub za ostatnim elementem kolejki dwukierunkowej. Jeśli kolejka jest pełna, zwraca `false`.

- `E pollFirst()`

- `E pollLast()`

Usuwa i zwraca element znajdujący się przed pierwszym lub za ostatnim elementem kolejki dwukierunkowej, jeśli nie jest ona pusta. Jeśli kolejka jest pusta, zwraca `null`.

- `E peekFirst()`

- `E peekLast()`

Zwraca pierwszy lub ostatni element kolejki dwukierunkowej, nie usuwając go, pod warunkiem że kolejka nie jest pusta. Jeśli kolejka jest pusta, zwraca `null`.

```
java.util.ArrayDeque<E> 6
```

- `ArrayDeque()`
- `ArrayDeque(int initialCapacity)`

Tworzy nieograniczoną kolejkę dwukierunkową o początkowej pojemności 16 elementów lub odpowiadającej wartości `initialCapacity`.

9.3.6. Kolejki priorytetowe

Zasada działania tej struktury danych polega na przyjmowaniu elementów w losowej kolejności i oddawaniu ich w kolejności uporządkowanej. To znaczy, że metoda `remove` wywołana na rzecz kolejki priorytetowej zawsze zwraca najmniejszy element znajdujący się w kolekcji. Nie oznacza to jednak, że elementy w kolejce priorytetowej są przechowywane w kolejności uporządkowanej. Jeśli przejdziemy iteratorem po zawartości kolejki, znajdujące się w niej obiekty mogą nie być posortowane. Jednym ze sposobów realizacji kolejki priorytetowej jest bardzo elegancka i szybka struktura danych o nazwie **sterta** (ang. *heap*). Sterta jest pewnego rodzaju samoorganizującym drzewem binarnym, w którym metody `add` i `remove` powodują przemieszczanie najmniejszego elementu w stronę korzenia bez straty czasu na sortowanie.

Kolejka priorytetowa, podobnie jak struktura `TreeSet`, może przechowywać elementy klasy implementującej interfejs `Comparable` lub obiekt `Comparator` przekazywany do konstruktora.

Typowym zastosowaniem kolejek priorytetowych są harmonogramy zadań. Zadania są dodawane w losowej kolejności i każde z nich ma określony priorytet. Kiedy jakieś zadanie może zostać rozpoczęte, z kolejki usuwane jest zadanie o najwyższym priorytecie (ponieważ tradycyjnie najwyższy priorytet jest oznaczany numerem 1, metoda `remove` usuwa najmniejszy element).

Listing 9.5 demonstruje działanie kolejki priorytetowej. W przeciwieństwie do zbioru `TreeSet`, tutaj iterator nie odwiedza elementów w uporządkowanej kolejności. Natomiast usuwany jest zawsze najmniejszy dostępny element.

Listing 9.5. `priorityQueue/PriorityQueueTest.java`

```
package priorityQueue;

import java.util.*;
import java.time.*;

/**
 * Program demonstrujący zastosowanie kolejki priorytetowej
 * @version 1.02 2015-06-20
 * @author Cay Horstmann
 */
public class PriorityQueueTest
{
    public static void main(String[] args)
    {
        var pq = new PriorityQueue<LocalDate>();
        pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
```

```

pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse

System.out.println("Iteracja przez elementy...");
for (LocalDate date : pq)
    System.out.println(date);
System.out.println("Usuwanie elementów...");
while (!pq.isEmpty())
    System.out.println(pq.remove());
}

```

java.util.PriorityQueue 5

- PriorityQueue()
- PriorityQueue(int initialCapacity)

Tworzy kolejkę priorytetową do przechowywania obiektów implementujących interfejs Comparable.

- PriorityQueue(int initialCapacity, Comparator<? super E> c)

Tworzy kolejkę priorytetową i wykorzystuje do jej sortowania określony komparator.

9.4. Słowniki

Zbiór (ang. *set*) to rodzaj kolekcji pozwalający na szybkie wyszukiwanie elementów. Aby jednak znaleźć jakiś element, trzeba posiadać jego wierną kopię. Nie jest to zbyt często wykonywany rodzaj wyszukiwania — zazwyczaj do dyspozycji jest jakiś rodzaj informacji kluczowej i zadanie polega na znalezieniu związanego z nią elementu. Tego typu struktury realizowane są za pomocą **słowników**. Słowniki przechowują pary klucz – wartość. Aby znaleźć element w słowniku, trzeba znać jego klucz. Na przykład w słowniku można przechowywać tabelę danych o pracownikach, gdzie kluczami będą identyfikatory pracowników, a wartościami obiekty typu `Employee`. W kilku kolejnych sekcjach opisujemy sposoby posługiwania się słownikami.

9.4.1. Podstawowe operacje słownikowe

W Java Collections Framework znajdują się dwie implementacje słowników ogólnego przeznaczenia: `HashMap` i `TreeMap`. Obie te klasy implementują interfejs `Map`.

Struktura `HashMap` miesza klucze, natomiast w strukturze `TreeMap` klucze są zorganizowane w drzewie poszukiwań posortowanym w porządku liniowym. Funkcja obliczająca skróty lub porównująca jest wywoływana *wyłącznie na rzecz kluczy*. Wartości związane z tymi kluczami nie są mieszane ani porównywane.

Którą strukturę wybrać: `HashMap` czy `TreeMap`? Podobnie jak ze zbiorami, mieszanie jest nieco szybsze i należy je wybierać, gdy kolejność kluczy nie ma znaczenia.

Poniżej tworzona jest struktura `HashMap` do przechowywania danych pracowników:

```
var staff = new HashMap<String, Employee>(); // Klasa HashMap
// implementuje interfejs Map
var harry = new Employee("Henryk Kwiatek");
staff.put("987-98-9996", harry);
. . .
```

Dla każdego obiektu dodawanego do słownika `HashMap` należy podać jego klucz. W tym przypadku kluczem jest łańcuch, a odpowiadającą mu wartością obiekt typu `Employee`.

Aby pobrać (i zapisać w pamięci) obiekt ze słownika, należy posłużyć się jego kluczem.

```
var id = "987-98-9996";
Employee e = staff.get(s); // pobiera obiekt harry
```



Ze względów historycznych metoda `get` ma w deklaracji parametr typu `Object`, a nie typu klucza. Na przykład poniższy kod przechodzi kompilację:

```
long numericId = . . .
Employee e = staff.get(numericId); // Przechodzi kompilację, ale nigdy nie pobiera wartości
```

Jeśli z danym kluczem w słowniku nie jest skojarzona żadna wartość, metoda `get` zwraca wartość `null`.

Taka wartość zwrotna może być problematyczna. Czasami znana jest dobra inna wartość domyślna, której można by było użyć zamiast `null`. W takim przypadku można się posłużyć metodą `getOrDefault`:

```
Map<String, Integer> scores = . . . ;
int score = scores.get(id, 0); // jeśli nie ma id, zwraca 0
```

Klucze muszą być unikatowe, to znaczy, że nie można dwóm różnym wartościom przypisać takiego samego klucza. Jeśli metoda `put` zostanie wywołana dwa razy przy użyciu jednego klucza, wartość z drugiego wywołania zastąpi tę z pierwszego. W rzeczywistości metoda ta zwróci poprzednią wartość przechowywaną pod tym kluczem.

Do usuwania elementów z określonym kluczem ze słownika służy metoda `remove`. Natomiast metoda `size` zwraca liczbę elementów znajdujących się w słowniku.

Klucze i wartości słownika najłatwiej przejrzeć iteracyjnie za pomocą metody `forEach`, przekazując jej wyrażenie lambda odbierające kolejne klucze i wartości. Wyrażenie to jest wywołane po kolei dla każdej pozycji znajdującej się w słowniku.

```
scores.forEach((k, v) ->
    System.out.println("klucz=" + k + ", wartość=" + v));
```

Na listingu 9.6 pokazano praktyczne zastosowanie słownika. Na początku do słownika dodawane są pary klucz – wartość. Następnie program usuwa jeden klucz ze słownika, co pociąga za

sobą usunięcie skojarzonej z nim wartości. Dalej zostaje zmodyfikowana wartość skojarzona z pewnym kluczem i następuje wywołanie metody `get` wyszukującej wartość. Na zakończenie program iteruje przez zbiór `entrySet`.

Listing 9.6. `map/MapTest.java`

```
package map;

import java.util.*;

/**
 * Program demonstrujący użycie słownika z kluczami typu String i wartościami typu Employee
 * @version 1.12 2015-06-21
 * @author Cay Horstmann
 */
public class MapTest
{
    public static void main(String[] args)
    {
        var staff = new HashMap<String, Employee>();
        staff.put("144-25-5464", new Employee("Anna Kowalska"));
        staff.put("567-24-2546", new Employee("Henryk Kwiatek"));
        staff.put("157-62-7935", new Employee("Marcin Nowak"));
        staff.put("456-62-5527", new Employee("Franciszek Frankowski"));

        // wydruk wszystkich pozycji

        System.out.println(staff);

        // usunięcie wartości

        staff.remove("567-24-2546");

        // podmienienie pozycji

        staff.put("456-62-5527", new Employee("Weronika Kowalska"));
        // wyszukanie wartości

        System.out.println(staff.get("157-62-7935"));

        // iteracja przez wszystkie pozycje

        staff.forEach((k, v) ->
            System.out.println("klucz=" + k + ", wartość=" + v));
    }
}
```

`java.util.Map<K, V>` 1.2

■ `V get(Object key)`

Zwraca wartość skojarzoną z kluczem `key`. Jeśli nie znajdzie klucza w słowniku, zwraca wartość `null`. Klucz może mieć wartość `null`.

■ `default V getOrDefault(Object key, V defaultValue)` 8

Pobiera wartość skojarzoną z kluczem i ją zwraca. Jeśli nie znajdzie klucza, zwraca wartość domyślną `defaultValue`.

- `V put(K key, V value)`

Wstawia parę klucz – wartość do słownika. Jeśli taki klucz jest już w słowniku, skojarzony z nim obiekt jest zastępowany nowym. Metoda ta zwraca poprzednią wartość skojarzoną z kluczem lub wartość `null`, jeśli wcześniej takiego klucza nie było. Klasy implementujące mogą zabraniać stosowania kluczy lub wartości `null`.

- `void putAll(Map<? extends K, ? extends V> entries)`

Wstawia do słownika wszystkie pozycje z określonego słownika.

- `boolean containsKey(Object key)`

Zwraca wartość `true`, jeśli klucz znajduje się w słowniku.

- `boolean containsValue(Object value)`

Zwraca wartość `true`, jeśli wartość znajduje się w słowniku.

- `default void forEach(BiConsumer<? super K, ? super V> action)` **8**

Wykonuje działanie (`action`) na wszystkich parach klucz – wartość słownika.

`java.util.HashMap<K, V>` **1.2**

- `HashMap()`

- `HashMap(int initialCapacity)`

- `HashMap(int initialCapacity, float loadFactor)`

Tworzy pusty słownik `HashMap` o określonej pojemności i z określonym współczynnikiem zapęłnienia (liczba z zakresu od 0,0 do 1,0 określająca stopień zapęłnienia tablicy `HashTable`, po przekroczeniu którego następuje reorganizacja na większą jednostkę). Domyślny współczynnik zapęłnienia wynosi 0,75.

- `static <K, V> HashMap<K, V> newHashMap(int numMappings)` **19**

Tworzy pusty słownik skrótów o pojemności wystarczającej do pomieszczenia spodziewanej liczby elementów (`numMappings`) bez reorganizacji.

`java.util.TreeMap<K, V>` **1.2**

- `TreeMap()`

Tworzy pusty słownik drzewiasty dla wszystkich kluczy implementujących interfejs `Comparable`.

- `TreeMap(Comparator<? super K> c)`

Tworzy słownik drzewiasty i sortuje jego klucze za pomocą określonego komparatora.

- `TreeMap(Map<? extends K, ? extends V> entries)`

Tworzy słownik drzewiasty i dodaje do niego wszystkie elementy z innego słownika.

- `TreeMap(SortedMap<? extends K, ? extends V> entries)`

Tworzy słownik drzewiasty, dodaje do niego wszystkie pozycje z posortowanego słownika oraz wykorzystuje ten sam komparator co określony posortowany słownik.

```
java.util.SortedMap<K, V> 1.2
```

- `Comparator<? super K> comparator()`

Zwraca komparator użyty do sortowania kluczy lub wartość `null`, jeśli klucze są porównywane za pomocą metody `compareTo` z interfejsu `Comparable`.

- `K firstKey()`
- `K lastKey()`

Zwraca najmniejszy lub największy klucz dostępny w słowniku.

9.4.2. Modyfikowanie wpisów w słowniku

Jedną z trudności, jakie można napotkać w trakcie pracy ze słownikami, jest modyfikowanie wpisów. Normalnie pobiera się starą wartość związaną z kluczem, potem się ją zmienia, a na koniec wstawia się ją z powrotem do słownika. Trzeba jednak pamiętać o specjalnym przypadku, jakim jest pierwsze wystąpienie danego klucza. Powiedzmy, że chcemy wykorzystać słownik do policzenia, ile razy wybrane słowo występuje w pliku. Gdy je napotykamy, zwiększamy wartość licznika w następujący sposób:

```
counts.put(word, counts.get(word) + 1);
```

Technika ta działa prawidłowo w prawie każdym przypadku. Wyjątkiem jest pierwsze napotkanie szukanego słowa, kiedy to następuje zwrócenie przez `get` wartości `null` i wystąpienie wyjątku `NullPointerException`.

Prostym rozwiązaniem jest użycie metody `getOrDefault`:

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

Inna możliwość to wywołanie najpierw metody `putIfAbsent`, która wstawia do słownika wartość tylko wtedy, jeśli uprzednio jego klucz nie istniał (lub był mapowany na `null`).

```
counts.putIfAbsent(word, 0);
counts.put(word, counts.get(word) + 1); // teraz wiadomo, że metoda get zadziała
```

Ale jest jeszcze lepsze rozwiązanie. Tę często wykonywaną operację można uprościć za pomocą metody `merge`. Poniższe wywołanie wiąże `word` z `1`, jeśli takiego klucza wcześniej nie było, lub w przeciwnym przypadku łączy poprzednią wartość z `1` za pomocą funkcji `Integer::sum`.

```
counts.merge(word, 1, Integer::sum);
```

Teraz wyobraź sobie inną często spotykaną sytuację. Chcemy powiązać zbiór z każdym kluczem. Na przykład w indeksie książki każdy termin ma przypisany zbiór numerów stron, na których występuje. Oto jak należy zmienić słownik:

```
var index = new TreeMap<String, TreeSet<Integer>>();
...
index.computeIfAbsent(term, k -> new TreeSet<>()).add(pageNumber);
```

Tutaj lepiej jest użyć metody `computeIfAbsent`, ponieważ struktura `TreeSet` jest tworzona, tylko jeśli z terminem nie jest powiązany wcześniej utworzony zbiór. Metoda `computeIfAbsent` zwraca nową wartość, dzięki czemu możemy łączyć wywołania metody `add` w łańcuchy. Z kolei metoda `putIfAbsent` zwraca poprzedni klucz lub `null`, więc nie nadaje się do łączenia w łańcuchy.



Kuszące jest zamienienie `k -> new TreeSet<>()` na `TreeSet::new`, ale to się nie uda. Funkcja dla danego klucza ma parametr `k`, który zostałby przekazany do konstruktora. Na szczęście nie ma konstruktora `TreeSet` z parametrem `String` i taki kod nie przejdzie kompilacji.

Mogło być gorzej. Wyobraź sobie, że masz słownik `Map<Integer, ArrayList<String>>` i wywołanie

```
map.computeIfAbsent(n, ArrayList::new)
```

Istnieje konstruktor `ArrayList`, który przyjmuje liczbę całkowitą jako określenie pojemności. Dla dużych wartości `n` zostałyby alokowane duże listy tablicowe, a z pewnością nie o to chodziło.



Metody słownika nie są zbyt konsekwentne w kwestii traktowania wartości `null`. Niektóre traktują ją jako prawidłową wartość, a inne — tak samo jak brakujący klucz.

Metoda `getOrDefault` należy do pierwszej grupy, a metoda `putIfAbsent` — do drugiej:

```
counts.put("C++", null);
counts.getOrDefault("C++", -1) // Zwraca null, nie używając wartości domyślnej
counts.putIfAbsent("C++", 1) // Wstawia 1, interpretując null jako brak
```

W dokumentacji API znajdują się opisy także innych, rzadziej używanych metod do modyfikowania wpisów w słownikach.

java.util.Map<K, V> 1.2

- `default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)` **8**

Jeżeli klucz jest związany z inną niż `null` wartością `v`, stosuje funkcję do `v` i `value`, a następnie wiąże klucz z wynikiem lub, jeśli ten wynosi `null`, usuwa klucz. W przeciwnym przypadku wiąże klucz z wartością. Zwraca `get(key)`.

- `default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` **8**

Stosuje funkcję do `key` i `get(key)`. Wiąże klucz z wynikiem albo, jeśli ten wynosi `null`, usuwa klucz. Zwraca `get(key)`.

- `default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` **8**

Jeżeli klucz jest związany z inną niż `null` wartością `v`, stosuje funkcję do `key` i `v`, a następnie wiąże klucz z wynikiem lub, jeśli ten wynosi `null`, usuwa klucz. Zwraca `get(key)`.

- `default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) 8`

Stosuje funkcję do klucza, chyba że jest on związany z wartością inną niż `null`. Wiąże klucz z wynikiem lub, jeśli ten wynosi `null`, usuwa klucz. Zwraca `get(key)`.

- `default void replaceAll(BiFunction<? super K,? super V,? extends V> function) 8`

Wywołuje funkcję na wszystkich pozycjach słownika. Z wynikami różnymi od `null` wiąże klucze, a dla wyników `null` usuwa klucze.

- `default V putIfAbsent(K key, V value) 8`

Jeśli klucza nie ma lub jest powiązany z `null`, kojarzy go z `value` i zwraca `null`. W przeciwnym przypadku zwraca powiązaną wartość.

9.4.3. Widoki słowników

W Java Collections Framework słowniki nie są traktowane jako kolekcje. (W innych systemach struktur danych słowniki są traktowane jako kolekcje par klucz – wartość lub kolekcje wartości indeksowanych kluczami). Istnieje jednak możliwość tworzenia **widoków** słowników, czyli obiektów implementujących interfejs `Collection` lub jeden z jego interfejsów podrzędnych.

Wyróżnia się trzy rodzaje widoków: zbiór kluczy, kolekcja wartości (która nie jest zbiorem) oraz zbiór par klucz – wartość. Klucze oraz pary klucz – wartość są reprezentowane jako zbiory, ponieważ w słowniku może się znajdować tylko jeden egzemplarz każdego klucza. Do tworzenia tych trzech widoków służą następujące metody:

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

Elementy zbioru tworzonego przez metodę `entrySet` są obiektami klasy implementującej interfejs `Map.Entry`.

Należy pamiętać, że `keySet` to *nie* `HashSet` ani `TreeSet`, lecz obiekt innej klasy implementującej interfejs `Set`, który jest rozszerzeniem interfejsu `Collection`. W związku z tym obiektów `keySet` można używać tak jak każdej kolekcji.

Można na przykład przejrzeć wszystkie klucze słownika:

```
Set<String> keys = map.keySet();
for (String key : keys)
{
    jakieś operacje na kluczu
}
```

Jeśli trzeba przejrzeć klucze i wartości, można uniknąć operacji wyszukiwania wartości przez enumerację *wpisów*. W tym celu można się posłużyć poniższym ramowym kodem:

```
for (Map.Entry<String, Integer> entry : counts.entrySet())
{
    String k = entry.getKey();
    Integer v = entry.getValue();
    jakieś operacje na k, v
}
```

Egzemplarze `Map.Entry` są połączone ze słownikiem. Przez obiekt pozycji można zmienić wartość w słowniku.

```
for (Map.Entry<String, Integer> entry : counts.entrySet())
{
    String k = entry.getKey();
    Integer v = entry.getValue();
    entry.setValue(v + 1); // to samo, co counts.put(k, v + 1);
}
```

Jeśli natomiast zmodyfikujesz wartość innym sposobem (na przykład za pomocą metody słownika `put`), to pozycja również zostanie zmieniona.

Jeśli chcesz przekazać pozycję do innej metody, to powinieneś je odłączyć od słownika przez wywołanie metody `Map.Entry.copyOf(entry)`. Możesz też tworzyć niepowiązane słowniki egzemplarze `Map.Entry`, wywołując metodę `Map.entry(klucz, wartość)`. Przydaje się to zawsze, gdy potrzebna jest para wartości.

Jeżeli na widoku `keySet` zostanie wywołana metoda `remove` iteratora, to ze słownika zostanie usunięty *klucz wraz z powiązaną wartością*. Nie można natomiast *dodać* elementu do widoku. Dodawanie klucza bez towarzyszącej mu wartości po prostu nie ma sensu. Próba wywołania metody `add` skończy się zgłoszeniem wyjątku `UnsupportedOperationException`. Takie same ograniczenia dotyczą widoku `entrySet`, choć teoretycznie dodanie nowej pary klucz – wartość ma sens.



Żmudnego wpisywania `Map.Entry` można uniknąć dzięki deklaracji `var`.

```
for (var entry : map.entrySet())
{
    jakieś operacje na entry.getKey(), entry.getValue()
}
```

Ewentualnie można skorzystać z metody `forEach`:

```
map.forEach((k, v) -> {
    jakieś operacje na k, v
});
```

`java.util.Map<K, V> 1.2`

■ `Set<Map.Entry<K, V>> entrySet()`

Zwraca widok w postaci zbioru obiektów `Map.Entry`, czyli par klucz – wartość znajdujących się w słowniku. Ze zbioru tego można usuwać elementy, a skutki tego działania będą uwzględnione w słowniku. Nie można natomiast niczego dodawać.

- `Set<K> keySet()`

Zwraca widok w postaci zbioru wszystkich kluczy znajdujących się w słowniku. Jeśli jakiś klucz z tego zbioru zostanie usunięty, ze słownika zostaną usunięte ten sam klucz i skojarzona z nim wartość. Nie można niczego dodawać.

- `Collection<V> values()`

Zwraca widok kolekcyjny wszystkich wartości znajdujących się w kolekcji. Jeśli jakaś wartość z tego zbioru zostanie usunięta, ze słownika zostaną usunięte ta sama wartość i skojarzony z nią klucz. Nie można niczego dodawać.

```
java.util.Map.Entry<K, V> 1.2
```

- `K getKey()`

- `V getValue()`

Zwraca klucz lub wartość określonego wpisu.

- `V setValue(V newValue)`

Zamienia wartość *znajdującą się w powiązanym słowniku* na nową wartość i zwraca starą wartość.

- `static <K, V> Map.Entry<K,V> copyOf(Map.Entry<? extends K,? extends V> map) 17`

Zwraca kopię podanego hasła słownika. W odróżnieniu od elementów zbioru haseł słownika kopia nie jest „żywa”. Wywołanie `setValue` nie powoduje aktualizacji słownika.

9.4.4. Klasa WeakHashMap

W Java Collections Framework znajduje się kilka klas słowników, które mają specjalne przeznaczenie. Omawiamy je krótko w kilku kolejnych podrozdziałach.

Klasa `WeakHashMap` rozwiązuje pewien bardzo interesujący problem. Spróbujmy sobie wyobrazić, co się dzieje z wartością, której klucz nie jest już używany nigdzie w programie. Załóżmy, że ostatnia referencja do tego klucza została utracona. W takiej sytuacji nie ma już żadnego sposobu dostania się do skojarzonego z tym kluczem obiektu. Ponieważ klucza już nigdzie nie ma, nie można usunąć ze słownika pary klucz – wartość. Nasuwa się myśl, że jest to zadanie dla śmieciarki.

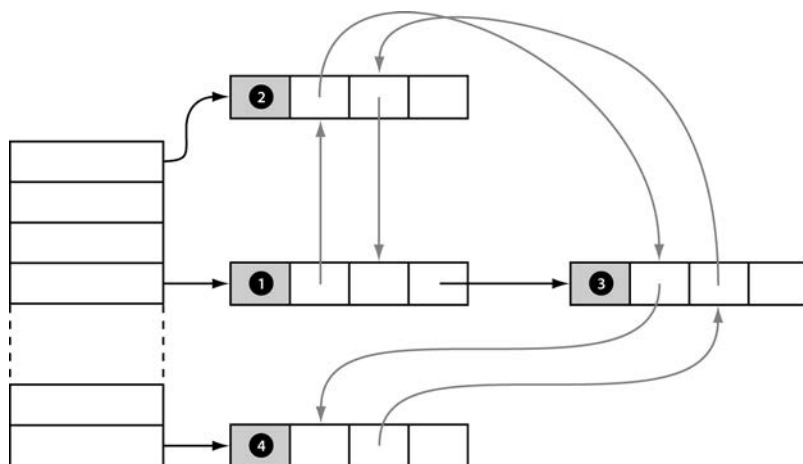
Niestety nie jest to takie proste. System ten przechowuje informacje o aktywnych obiektach. Dopóki słownik jest aktywny, *wszystkie* jego komórki również są aktywne, przez co nie można ich usunąć. Dlatego należy pamiętać, aby usuwać z programu nieużywane wartości z długo wykorzystywanych słowników. Można też wybrać inne rozwiązanie i użyć klasy `WeakHashMap`. Struktura ta współpracuje ze śmieciarką w zakresie usuwania par klucz – wartość, jeśli jedyne odwołanie do klucza pochodzi z pozycji w tablicy.

Oto zasada działania tego mechanizmu. Klucze w `WeakHashMap` są przechowywane w tak zwanych **słabych referencjach** (ang. *weak reference*). Obiekt typu `WeakReference` przechowuje referencję do innego obiektu, w tym przypadku do klucza w tablicy. Obiekty tego typu są traktowane przez moduł śmieciarki (ang. *Garbage Collector* — *GC*) w specjalny sposób. Standardowo, jeśli *GC* odkryje, że do jakiegoś obiektu nie ma żadnych referencji, usuwa go. Jeśli natomiast obiekt taki jest dostępny *wyłącznie* za pośrednictwem obiektu `WeakReference`, *GC* także go usuwa, ale słabą referencję odnoszącą się do niego wstawia do kolejki. Obiekt `WeakHashMap` sprawdza w równych odstępach czasu, czy w kolejce nie pojawiły się jakieś nowe słabe referencje. Pojawienie się takiej referencji w kolejce oznacza, że klucz nie był nigdzie używany i że został zabrany. Wtedy `WeakHashMap` usuwa odpowiednią pozycję.

9.4.5. Klasy `LinkedHashSet` i `LinkedHashMap`

Klasy `LinkedHashSet` i `LinkedHashMap`, które pamiętają kolejność wstawiania elementów. Tym samym rozwiązują problem losowej kolejności przechowywanych elementów. Elementy wstawiane do tablicy są łączone w listy dwustronne (zobacz rysunek 9.11).

Rysunek 9.11.
Powiązana
tablica skrótów



Przeanalizujmy na przykład poniższe instrukcje z listingu 9.6, wstawiające elementy do słownika:

```
var staff = new LinkedHashMap<String, Employee>();
staff.put("144-25-5464", new Employee("Anna Kowalska"));
staff.put("567-24-2546", new Employee("Henryk Kwiatek"));
staff.put("157-62-7935", new Employee("Marcin Nowak"));
staff.put("456-62-5527", new Employee("Franciszek Frankowski"));
```

Instrukcja `staff.keySet().iterator()` przegląda klucze w następującej kolejności:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

Następnie `staff.values().iterator()` przegląda wartości w takim porządku:

```
Anna Kowalska
Henryk Kwiatek
Marcin Nowak
Franciszek Frankowski
```

Po elementach struktur `LinkedHashMap` można także iterować według *kolejności dostępu* zamiast kolejności wstawiania. Każde wywołanie metody `get` lub `put` na rzecz jakiegoś elementu powoduje jego przeniesienie na *koniec* listy powiązanej (zmienia się tylko kolejność w liście powiązanej, nie w komórce w tablicy; pozycja pozostaje w komórce, która odpowiada skrótowi klucza). Aby utworzyć taką strukturę danych, można użyć następującej instrukcji:

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Porządek według kolejności dostępu znajduje zastosowanie w usuwaniu najdłużej nieużywanych elementów z pamięci podręcznej. Na przykład często używane obiekty mogą być przechowywane w pamięci, a rzadziej używane w bazie danych. Jeśli nie uda się znaleźć jakiegoś obiektu w tablicy i jest ona wypełniona, można do niej wpuścić iterator, który usunie kilka pierwszych elementów. Będą to te obiekty, które były używane w najdalszej przeszłości.

Proces ten można nawet zautomatyzować. W tym celu należy utworzyć podklasę klasy `LinkedHashMap` i przededefiniować poniższą metodę:

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Ta metoda jest wywoływana po dodaniu nowej pozycji. Jeśli zwróci `true`, to powoduje usunięcie najstarszego elementu (`eldest`). Poniższa przykładowa pamięć podręczna przechowuje maksymalnie sto elementów:

```
var cache = new LinkedHashMap<K, V>(128, 0.75F, true)
{
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
    {
        return size() > 100;
    }
};
```

Możesz też sprawdzić najstarszy element, aby podjąć decyzję, czy go usunąć. Można na przykład sprawdzić jego znacznik czasu i usunąć go tylko wtedy, gdy jest wystarczająco stary. Nie spowoduje to automatycznego usunięcia wszystkich starych elementów, ponieważ metoda `removeEldestEntry` jest wywoływana tylko raz dla każdej nowej pozycji. W metodzie `removeEldestEntry` możesz modyfikować słownik, na przykład przez usuwanie początkowych elementów, które są wystarczająco stare. W takim przypadku musisz zwracać `false`.

Jeśli potrzebujesz zbioru ostatnio dodanych elementów zamiast słownika, zobacz punkt 9.5.4.

9.4.6. Klasy `EnumSet` i `EnumMap`

Struktura danych realizowana przez klasę `EnumSet` przechowuje elementy typu wyliczeniowego. Ponieważ typ wyliczeniowy ma skończoną liczbę egzemplarzy, wewnętrzna implementacja struktury `EnumSet` jest szeregiem bitów. Jeśli odpowiednia wartość znajduje się w zbiorze, włączany jest jej bit.

Klasa `EnumSet` nie posiada konstruktora publicznego. Do utworzenia tego typu zbioru trzeba użyć statycznej metody fabrycznej:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Do modyfikacji zbioru `EnumSet` można używać zwykłych metod z interfejsu `Set`.

`EnumMap` to słownik, w którym klucze są typu wyliczeniowego. Jest on prostą i wydajną tablicą wartości. Typ klucza należy określić w konstruktorze:

```
var personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```



W dokumentacji API klasy `EnumSet` i `EnumMap` znajdują się dziwnie wyglądające parametry typowe w formie `E extends Enum<E>`. Zapis ten oznacza, że `E` jest typem wyliczeniowym. Wszystkie typy wyliczeniowe dziedziczą po generycznej klasie `Enum`. Na przykład klasa `Weekday` dziedziczy po klasie `Enum<Weekday>`.

9.4.7. Klasa `IdentityHashMap`

Klasa `IdentityHashMap` również jest przeznaczona do specjalnych celów. Wartości haszowe kluczy nie powinny być w niej obliczane przez metodę `hashCode`, ale przez `System.identityHashCode`. Metody tej używa metoda `Object.hashCode` do obliczania skrótu z adresu obiektu w pamięci. Ponadto klasa ta porównuje obiekty za pomocą operatora `==` zamiast metody `equals`.

Dzięki temu dwa obiekty są uznawane za różne, nawet jeśli mają taką samą zawartość. Klasa ta znajduje zastosowanie w implementacji algorytmów przemierzających obiekty (na przykład serializujących), gdzie konieczne jest pamiętanie, które obiekty zostały już przetworzone.



W Javie 20 naprawiono subtelny błąd z `IdentityHashMap`, który pojawił się w Javie 8 (<https://bugs.openjdk.org/browse/JDK-8284901>). Dwie nieczęsto używane metody błędnie używały metody `equals` zamiast operatora `==`:

```
void remove(Object key, Object value)
void replace(K key, V oldValue, V newValue)
```

Metody te są czasami używane w algorytmach współbieżnych, w przypadkach, w których słownik powinien być modyfikowany tylko wtedy, gdy nie został zmieniony przez inny wątek. Można sobie tylko wyobrazić frustrację pierwszego programisty, który natknął się na ten błąd.

Żaden system oprogramowania, który jest tak skomplikowany jak platforma Java, nie może być całkowicie wolny od błędów. Dlatego tak ważne jest regularne aktualizowanie do najnowszej wersji.

java.util.WeakHashMap<K, V> 1.2

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`

Tworzy pusty słownik `WeakHashMap` o określonej pojemności i określonym współczynniku zapęnlennia.

java.util.LinkedHashSet<E> 1.4

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`

Tworzy pusty zbiór `LinkedHashSet` o określonej pojemności i określonym współczynniku zapęnlennia.

java.util.LinkedHashMap<K, V> 1.4

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`

Tworzy pusty słownik `LinkedHashMap` o określonej pojemności oraz określonym współczynniku zapęnlennia i porządku. Jeśli parametr `accessOrder` ma wartość `true`, stosowany jest porządek według kolejności dostępu. Wartość `false` oznacza porządek w kolejności wstawiania.

- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`

Tę metodę należy przeddefiniować, aby zwracała wartość `true`, jeśli najstarsza pozycja ma być usuwana. Parametr `eldest` określa pozycję, której usunięcie jest rozważane. Metoda ta jest wywoływana po dodaniu pozycji do słownika. Domyślna implementacja zwraca wartość `false` — stare elementy nie są domyślnie usuwane. Można jednak ją przeddefiniować, aby selektywnie zwracała wartość `true`, na przykład gdy najstarsza pozycja spełnia określone warunki lub słownik przekracza określony rozmiar.

java.util.EnumSet<E extends Enum<E>> 5.0

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`
Zwraca zbiór zapęnlenny wartościami z określonego typu wyliczeniowego.
- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`
Zwraca zmienny zbiór, który początkowo jest pusty.
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`
Zwraca zmienny zbiór zapęnlenny wartościami z zakresu od `from` do `to` (włącznie).

- `static <E extends Enum<E>> EnumSet<E> of(E e)`

...

- `static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)`

- `static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)`

Zwraca zmienny zbiór zawierający określone elementy, które nie mogą być `null`.

- `public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)`

- `public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`

Tworzy zmienny zbiór, który początkowo zawiera podane elementy. W drugiej metodzie argument `c` musi być typu `EnumSet` lub musi coś zawierać (aby można było określić typ elementów).

`java.util.EnumMap<K extends Enum<K>, V>` **5.0**

- `EnumMap(Class<K> keyType)`

Tworzy słownik, którego klucze są określonego typu.

`java.util.IdentityHashMap<K, V>` **1.4**

- `IdentityHashMap()`

- `IdentityHashMap(int expectedMaxSize)`

Tworzy pusty zmienny słownik `IdentityHashMap` o pojemności równej najmniejszej potęgze cyfry 2 większej niż `1,5*expectedMaxSize` (domyślna wartość parametru `expectedMaxSize` to 21).

`java.lang.System` **1.0**

- `static int identityHashCode(Object obj)` **1.1**

Zwraca ten sam skrót (obliczony na podstawie adresu w pamięci obiektu) co metoda `Object.hashCode`, nawet jeśli w klasie, do której należy obiekt `obj`, zdefiniowano metodę `hashCode`.

9.5. Kopie i widoki

Z rysunków 9.4 i 9.5 można wyciągnąć wniosek, że projektanci Javy wpadli w lekką przesadę, tworząc tak wiele interfejsów i klas abstrakcyjnych do implementacji raczej umiarkowanej liczby konkretnych klas kolekcyjnych. Jednak rysunki te nie pokazują wszystkiego. Za pomocą **widoków** (ang. *view*) można tworzyć inne obiekty implementujące interfejsy `Collection` i `Map`. Przykład tego przedstawiliśmy na podstawie metody `keySet` z klas słownikowych. Na pierwszy rzut oka wydaje się, że to ta metoda tworzy nowy zbiór, zapewnia go wszystkimi kluczami ze słownika i zwraca go. Nie jest to jednak prawda. Metoda `keySet` zwraca obiekt klasy implementującej interfejs `Set`, którego metody operują na oryginalnym słowniku. Taka kolekcja nazywana jest **widokiem**.

Widoki mają kilka ważnych zastosowań w Java Collections Framework. Opisujemy je w poniższych podrozdziałach.

9.5.1. Małe kolekcje

Dostępne są metody statyczne zwracające zbiory lub listy określonych elementów oraz słowniki zawierające dane pary klucz-wartość.

Na przykład:

```
List<String> names = List.of("Piotr", "Paweł", "Maria");
Set<Integer> numbers = Set.of(2, 3, 5);
```

Powyższy kod zwraca listę i zbiór zawierające po trzy elementy.

Można też przekazać tablicę obiektów:

```
String[] namesArray = { "Piotr", "Paweł", "Maria" };
List<String> names = List.of(namesArray); // Lista zawierająca trzy obiekty
```



Jeśli przekażesz tablicę wartości typu podstawowego, wynikiem będzie lista zawierająca jeden element — tablicę:

```
int[] numbersArray = { 2, 3, 5 };
List<int[]> arrays = List.of(numbersArray);
// Lista zawierająca jeden obiekt — tablicę int[]
```

Nie ma metody pomocniczej, która zamienia tablicę wartości typu podstawowego na listę elementów typu opakowaniowego.

Aby utworzyć słownik, należy przekazać klucze i wartości, na przykład:

```
Map<String, Integer> scores = Map.of("Piotr", 2, "Paweł", 3, "Maria", 5);
```

Elementy, klucze i wartości nie mogą być null. Klucze w zbiorach i słownikach nie mogą się powtarzać:

```
numbers = Set.of(13, null); // Błąd -- element null
scores = Map.of("Piotr", 4, "Piotr", 2); // Błąd -- powtórzony klucz
```



Nie ma żadnej gwarancji co do kolejności iteracji tych zbiorów i słowników. Nawet celowo jest ona pomieszana za pomocą ziarna, które jest randomizowane przy każdym uruchomieniu maszyny wirtualnej. Spójrz na dwa poniższe wykonania polecenia `jshell`:

```
$ jshell -q
jshell> Set.of("Piotr", "Paweł", "Maria")
$1 ==> [Piotr, Maria, Paweł]
jshell> /exit
$ jshell -q
jshell> Set.of("Piotr", "Paweł", "Maria")
$1 ==> [Paweł, Maria, Piotr]
```

Niektórzy programiści Javy piszą programy, których poprawność opiera się na założeniu, że szczegóły implementacyjne nigdy się nie zmieniają. To bardzo utrudnia implementatorom bibliotek wprowadzanie praktycznych zmian. W tym przypadku przesłanie jest jasne — nie zakładaj w programach jakiegokolwiek kolejności elementów.

Interfejsy `List` i `Set` mają po jedenaście metod przyjmujących od zera do dziesięciu argumentów oraz metodę `of` przyjmującą zmienną liczbę argumentów. Specjalizacje utworzono w celu optymalizacji wydajności.

W interfejsie `Map` nie może być wersji ze zmienną liczbą argumentów, ponieważ typy argumentów dotyczących kluczy i wartości przeplatają się po kolei. Istnieje statyczna metoda `ofEntries` przyjmująca dowolną liczbę obiektów typu `Map.Entry <K, V>`, które można utworzyć za pomocą statycznej metody `entry`. Na przykład:

```
import static java.util.Map.*;
...
Map<String, Integer> scores = ofEntries(
    entry("Piotr", 2),
    entry("Paweł", 3),
    entry("Maria", 5));
```

Metody `of` i `ofEntries` tworzą obiekty klas, które mają zmienną egzemplarzową dla każdego elementu lub opierają się na tablicy.

Takie obiekty kolekcji są **niemodyfikowalne**. Jakkolwiek próba zmiany ich zawartości skutkuje wyjątkiem `UnsupportedOperationException`.

Jeśli potrzebujesz zmiennej kolekcji, możesz przekazać niezmienną kolekcję do konstruktora:

```
var names = new ArrayList<>(List.of("Piotr", "Paweł", "Maria")); // Zmienna lista imion
```

Poniższa instrukcja zwraca niemodyfikowalny obiekt implementujący interfejs `List` oraz tworzy złudzenie istnienia `n` elementów typu `anObject`.

```
Collections.nCopies(n, anObject)
```

Na przykład poniższa instrukcja tworzy listę zawierającą 100 łańcuchów DOMYŚLNY:

```
List<String> settings = Collections.nCopies(100, "DOMYŚLNY");
```

Struktura ta zajmuje bardzo mało miejsca w pamięci — obiekt jest zapisany tylko w jednym miejscu. Jest to sprytne zastosowanie techniki widoków.



Klasa `Collections` zawiera kilka metod, których parametry lub wartości zwrótne są kolekcjami. Nie należy jej mylić z interfejsem `Collection`.



W Javie nie ma klasy `Pair`, więc niektórzy programiści z braku lepszego rozwiązania używają `Map.Entry`.

9.5.2. Niemodyfikowalne kopie i widoki

Aby utworzyć **niemodyfikowalną kopię** kolekcji, należy użyć metody `copyOf` z typu kolekcji:

```
ArrayList<String> names = . . . ;
Set<String> nameSet = Set.copyOf(names); // Imiona jako niemodyfikowalny zbiór
List<String> nameList = List.copyOf(names); // Imiona jako niemodyfikowalna lista
```

Tak jak w przypadku metod `of`, metody `theCopyOf` nie tworzą kolekcji elementów `null`, lecz zgłaszają wyjątek `NullPointerException`.

Każda metoda `copyOf` tworzy kopię kolekcji. Jeśli oryginalna kolekcja zostanie zmodyfikowana, nie ma to wpływu na kopię.

Jeśli oryginalna kolekcja jest niemodyfikowalna i prawidłowego typu, wówczas metoda `copyOf` po prostu ją zwraca:

```
Set<String> names = Set.of("Piotr", "Paweł", "Maria");
Set<String> nameSet = Set.copyOf(names); // Nie trzeba wykonywać kopii: names == nameSet
```

Klasa `Collections` zawiera metody, które tworzą **niemodyfikowalne widoki** kolekcji. Stanowią one dodatkowy element kontrolny w istniejącej kolekcji w czasie działania programu. Jeśli zostaje wykryta próba modyfikacji niemodyfikowalnej kolekcji, następuje zgłoszenie wyjątku.

Jeśli jednak oryginalna kolekcja się zmieni, to widok odzwierciedli te zmiany. To odróżnia widoki od kopii.

Niemodyfikowalne widoki można tworzyć za pomocą ośmiu metod:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableNavigableSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
Collections.unmodifiableNavigableMap
```

Każda z nich jest zdefiniowana tak, aby działała na podstawie interfejsu. Na przykład `Collections.unmodifiableList` działa z `ArrayList`, `LinkedList` i każdą inną klasą implementującą interfejs `List`.

Powiedzmy na przykład, że chcesz, aby pewna część Twojego programu miała dostęp do zawartości kolekcji, ale jej nie modyfikowała. Oto co możesz zrobić:

```
var staff = new LinkedList<String>();
. . .
lookAt(Collections.unmodifiableList(staff));
```

Metoda `Collections.unmodifiableList` zwraca obiekt klasy implementującej interfejs `List`. Jej metody dostępne pobierają wartości z kolekcji `staff`. Oczywiście metoda `lookAt` może wywoływać wszystkie metody z interfejsu `List`, nie tylko dostępne. Jednak wszystkie metody modyfikujące (takie jak `add`) zostały tak zmodyfikowane, aby zgłaszały wyjątek `UnsupportedOperationException`, zamiast przekazywać wywołanie do kolekcji.

Niemodyfikowalny widok nie czyni samej kolekcji niemodyfikowalną. Ją nadal można modyfikować przez oryginalną referencję (w tym przypadku `staff`). I nadal można wywoływać metody modyfikujące na jej elementach.

Widoki opakowują *interfejsy*, nie rzeczywisty obiekt kolekcji, więc programista ma dostęp tylko do tych metod, które są zdefiniowane w interfejsie. Na przykład klasa `LinkedList` zawiera metody pomocnicze `addFirst` i `addLast`, które nie należą do interfejsu `List`. Nie ma więc do nich dostępu przez niemodyfikowalny widok.



Metoda `unmodifiableCollection` (jak również metody `synchronizedCollection` i `checkedCollection`, opisane dalej) zwraca kolekcję, której metoda `equals` nie wywołuje metody `equals` podstawowej kolekcji. Zamiast tego dziedziczy metodę `equals` klasy `Object`, sprawdzającą tylko, czy obiekty są identyczne. Jeśli zamienisz zbiór lub listę w kolekcję, stracisz możliwość porównywania treści. Widok działa w ten sposób, ponieważ sprawdzanie równości na tym poziomie hierarchii nie jest dobrze zdefiniowane. Widoki traktują metodę `hashCode` w taki sam sposób.

Preferuj metody opakowujące `unmodifiableSet` i `unmodifiableList`, których metody `equals` i `hashCode` są odpowiednie do zbiorów i list.

```
var names = Set.of("Piotr", "Paweł", "Maria");
Collections.unmodifiableCollection(names).equals(names) // fałsz
Collections.unmodifiableSet(names).equals(names) // prawda
```

9.5.3. Przedziały

Przedziały można tworzyć z kilku rodzajów kolekcji. Wyobraźmy sobie na przykład, że mamy listę o nazwie `staff` i chcemy z niej wyodrębnić elementy od 10 do 19. Można utworzyć widok tego przedziału za pomocą metody `subList`.

```
List<Employee> group2 = staff.subList(10, 20);
```

Pierwszy indeks jest wliczany, drugi nie — podobnie jak z parametrami metody `substring` z klasy `String`.

Na przedziale można wykonywać dowolne działania, a ich rezultat będzie automatycznie widoczny w całej liście. Można na przykład usunąć cały przedział:

```
group2.clear(); // redukcja personelu
```

Przedział `group2` jest teraz pusty, a z listy `staff` zostały usunięte te elementy, które znajdowały się w tym przedziale.

Przedziały uporządkowanych zbiorów i słowników tworzy się przy użyciu kolejności sortowania, a nie pozycji elementów w kolekcji. W interfejsie `SortedSet` znajdują się trzy metody:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

Zwracają one podzbiory wszystkich elementów, które są większe niż lub równe `from` i mniejsze od `to`. Podobne metody dla uporządkowanych słowników to:

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

Zwracają one widoki słowników zawierające pozycje, których *klucze* mieszczą się w określonych zakresach.

Interfejs `NavigableSet` daje większe możliwości kontroli działań na przedziałach. Można określić, czy wartości graniczne mają być wliczane (ang. *inclusive*):

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

9.5.4. Zbiory ze słowników wartości logicznych

Czasami API dostarcza przydatnych słowników o praktycznych cechach, jakie chcielibyśmy mieć w zbiorze. W punkcie 9.4.5 pokazałem, jak za pomocą klasy `LinkedHashMap` utworzyć pamięć podręczną usuwającą starsze elementy. Klasa `LinkedHashSet` nie ma takiej funkcjonalności.

Jeśli chcesz utworzyć zbiór 100 ostatnio wstawionych łańcuchów, to utwórz `LinkedHashMap` `↪<String, Boolean>`, po czym wywołaj metodę `Collections.newSetFromMap`:

```
Set<String> cache = Collections.newSetFromMap(new LinkedHashMap<String, Boolean>()
{
    protected boolean removeEldestEntry(Map.Entry<String, Boolean> eldest)
    {
        return size() > 100;
    }
});
```

Widok zbioru bazuje na słowniku. Kiedy dodajemy element `e`, to widok dodaje do słownika pozycję o kluczu `e` i wartości `Boolean.TRUE`. Jeśli w słowniku pojawi się więcej niż 100 pozycji, najstarsza zostanie usunięta.

Metodę `newSetFromMap` należy wywołać z pustym słownikiem. Potem nie należy modyfikować słownika, tylko pozwolić, aby wszystkie modyfikacje były wykonywane przez metody widoku zbioru. W tym celu najlepiej jest przekazać słownik do metody `newSetFromMap`, nie zachowując do niego referencji, tak jak pokazałem w poprzednim przykładzie.

Metoda `newSetFromMap` jest przydatna także podczas pracy ze strukturami `WeakHashMap` (punkt 9.4.4). W przypadku struktury `ConcurrentHashMap` użyj w zamian metody `newKeySet` (rozdział 10.).

9.5.5. Widoki odwrócone

Metody `reversed` interfejsów `SequencedCollection` i `SequencedSet` zwracają widoki, które pokazują elementy w odwróconej kolejności. Na przykład poniżej pokazuję, jak odwrotnie przeglądać listę lub zbiór drzewiasty:

```
for (String element : collection.reversed())
{
    działania na elemencie
}
```

W przypadku `SequencedMap` metoda `reversed` pokazuje słownik z kluczami w odwróconej kolejności.

9.5.6. Widoki kontrolowane

Widoki kontrolowane mają na celu wspieranie programisty w procesie usuwania błędów związanych z typami generycznymi. Wiemy już z rozdziału 8., że do generycznej kolekcji da się prze-mycić elementy niewłaściwego typu. Na przykład:

```
var strings = new ArrayList<String>();
ArrayList rawList = strings; // Zostanie zgłoszone tylko ostrzeżenie (nie błąd) dotyczące
                             // zgodności ze starszym kodem.
rawList.add(new Date());     // Teraz referencja strings wskazuje na obiekt typu Date!
```

Błąd w instrukcji `add` nie zostanie wykryty w czasie działania programu. W zamian wystąpi wyjątek rzutowania, kiedy w innej części kodu zostanie wywołana metoda `get` i zostanie wykonane rzutowanie jej wyniku na typ `String`.

Widok kontrolowany wykryje taki błąd. Zdefiniujmy następującą bezpieczną listę:

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

Metoda `add` widoku sprawdza, czy wstawiany obiekt należy do określonej klasy; jeśli nie, generuje wyjątek `ClassCastException`. Jest to korzystne, ponieważ błąd zostaje zgłoszony w odpowiednim miejscu:

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Lista kontrolowana generuje wyjątek ClassCastException.
```



Widoki kontrolowane są ograniczone zestawem testów, które może przeprowadzić maszyna wirtualna w czasie działania programu. Na przykład struktury `Array` `List<Pair<String>>` nie można ochronić przed wstawieniem do niej typu `Pair<Date>`, ponieważ maszyna wirtualna dysponuje tylko jedną surową klasą `Pair`.

9.5.7. Widoki synchronizowane

Jeśli do kolekcji mają dostęp procedury z kilku wątków, należy zadbać o to, aby nie została ona przypadkowo zniszczona. Na przykład sytuacja, w której jeden wątek próbuje dodać elementy do kolekcji `HashMap`, podczas gdy inny ją odświeża, mogłaby być fatalna w skutkach.

Projektanci języka zdecydowali, że zamiast implementować kolekcje bezpieczne dla wątków, uczynią zwykle kolekcje bezpiecznymi dla wątków za pomocą mechanizmu widoków. Na przykład statyczna metoda `synchronizedMap` z klasy `Collections` zamienia zwykle słowniki na słowniki z synchronizowanymi metodami dostępowymi:

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String,  
↳ Employee>());
```

Teraz można dostawać się do obiektu `map` z różnych wątków. Metody takie jak `get` i `put` są synchronizowane, to znaczy każde wywołanie metody musi zostać w pełni ukończone, zanim inny wątek będzie mógł wywołać kolejną metodę. Kwestię synchronizowanego dostępu do struktur danych szczegółowo omawiamy w rozdziale 10.

9.5.8. Uwagi dotyczące operacji opcjonalnych

Widoki z reguły mają jakieś ograniczenia — mogą być tylko do odczytu, mogą nie mieć możliwości zmiany rozmiaru lub pozwalać na usuwanie elementów, ale nie na dodawanie, jak na przykład widok kluczy słownika. Próba wykonania niedozwolonej operacji na ograniczonym widoku kończy się zgłoszeniem wyjątku `UnsupportedOperationException`.

W dokumentacji API wiele metod klas i interfejsów kolekcyjnych oznaczono jako operacje opcjonalne (ang. *optional operations*). Wydaje się to sprzeczne z ideą interfejsu, który przecież — jak wiadomo — określa zestaw metod, które klasa *musi* implementować. Rzeczywiście ten stan rzeczy nie jest satysfakcjonujący z teoretycznego punktu widzenia. Lepszym rozwiązaniem byłoby utworzenie osobnych interfejsów dla widoków tylko do odczytu i widoków, które nie mogą zmienić rozmiaru kolekcji. Wtedy jednak liczba interfejsów potroiłaby się, a to dla projektantów biblioteki byłoby nie do zaakceptowania.

Czy powinno się rozszerzać technikę opcjonalnych operacji na własne projekty? Naszym zdaniem nie. Mimo iż kolekcje są używane często, styl programowania związany z ich implementacją nie jest typowy dla innych dziedzin. Projektanci biblioteki kolekcyjnej stoją przed niezwykle trudnym zadaniem zaspokojenia sprzecznych wymagań. Z jednej strony użytkownicy wymagają, aby biblioteka była łatwa do nauki, wygodna w użyciu, w pełni generyczna i niepodatna na błędy, a z drugiej strony ma być tak samo wydajna jak ręcznie optymalizowane algorytmy. Spełnienie tych wszystkich żądań naraz (czy choćby zbliżenie się do nich) jest najzwyczajniej niemożliwe. Jednak we własnej praktyce programistycznej nie będziesz często rozwiązywać tak ekstremalnych problemów. Powinno być możliwe znalezienie rozwiązań, które nie polegają na wykorzystaniu ostatecznego środka w postaci opcjonalnych operacji interfejsu.

java.util.List 1.2

- `static <E> List<E> of()` **9**
- `static <E> List<E> of(E e1)` **9**
- ...
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)` **9**
- `static <E> List<E> of(E... elements)` **9**

Zwraca niezmienną listą danych elementów, które nie mogą być null.

- `static <E> List<E> copyOf(Collection<? extends E> coll)` **10**

Zwraca niezmienną kopię danej kolekcji.

java.util.Set 1.2

- `static <E> Set<E> of()` **9**
- `static <E> Set<E> of(E e1)` **9**
- ...
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)` **9**
- `static <E> Set<E> of(E... elements)` **9**

Zwraca niezmienny zbiór danych elementów, które nie mogą być null.

- `static <E> Set<E> copyOf(Collection<? extends E> coll)` **10**

Zwraca niezmienną kopię danej kolekcji.

java.util.Map 1.2

- `static <K, V> Map<K, V> of()` **9**
- `static <K, V> Map<K, V> of(K k1, V v1)` **9**
- ...
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)` **9**

Zwraca niezmienny słownik danych kluczy i wartości, które nie mogą być null.

- `static <K,V> Map.Entry<K,V> entry(K k, V v)` **9**

Zwraca niezmienny element słownika o określonych kluczu i wartości, które nie mogą być null.

- `static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)` **9**

Zwraca niezmienny słownik danych elementów.

- `static <K, V> Map<K,V> copyOf(Map<? extends K,? extends V> map)` 10

Zwraca niezmienną kopię danego słownika.

java.util.Collections 1.2

- `static <E> Set<E> newSetFromMap(Map<E, Boolean> map)` 6

Zwraca zbiór bazujący na określonym słowniku, który początkowo powinien być pusty, a potem modyfikowany tylko przez zwrócony widok.

- `static <E> Collection unmodifiableCollection(Collection<E> c)`

- `static <E> SequencedCollection<E>
unmodifiableSequencedCollection(SequencedCollection<E> c)` 21

- `static <E> List<E> unmodifiableList(List<E> c)`

- `static <E> Set<E> unmodifiableSet(Set<E> c)`

- `static <E> SequencedSet<E>
unmodifiableSequencedSet(SequencedCollection<E> c)` 21

- `static <E> SortedSet<E> unmodifiableSortedSet(SortedSet<E> c)`

- `static <E> SortedSet<E> unmodifiableNavigableSet(NavigableSet<E> c)` 8

- `static <K, V> Map<K, V> unmodifiableMap(Map<K, V> c)`

- `static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> c)`

- `static <K, V> SequencedMap<K, V> unmodifiableSequencedMap(SequencedMap<K, V> c)` 21

- `static <K, V> SortedMap unmodifiableNavigableMap(NavigableMap<K, V> c)` 8

Tworzy widoki kolekcji, których metody modyfikujące generują wyjątek `UnsupportedOperationException`.

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`

- `static <E> List synchronizedList(List<E> c)`

- `static <E> Set synchronizedSet(Set<E> c)`

- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`

- `static <E> NavigableSet synchronizedNavigableSet(NavigableSet<E> c)` 8

- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`

- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

- `static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> c)` 8

Tworzy widoki kolekcji, których metody są zsynchronizowane.

- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`

- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <E> NavigableSet checkedNavigableSet(NavigableSet<E> c, Class<E> elementType)` **8**
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> NavigableMap checkedNavigableMap(NavigableMap<K, V> c, Class<K> keyType, Class<V> valueType)` **8**
- `static <E> Queue<E> checkedQueue(Queue<E> queue, Class<E> elementType)` **8**

Tworzy widoki kolekcji, których metody zgłaszają wyjątek `ClassCastException`, jeśli wstawiany element jest złego typu.

- `static <E> List<E> nCopies(int n, E value)`
Zwraca niemodyfikowalną listę zawierającą *n* identycznych kopii.
- `static <E> List<E> singletonList(E value)`
- `static <E> Set<E> singleton(E value)`
- `static <E> List<E> emptyList()`
- `static <T> Set<T> emptySet()`
- `static <E> SortedSet<E> emptySortedSet()`
- `static NavigableSet<E> emptyNavigableSet()`
- `static <K, V> Map<K, V> emptyMap()`
- `static <K, V> SortedMap<K, V> emptySortedMap()`
- `static <K, V> NavigableMap<K, V> emptyNavigableMap()`
- `static <T> Enumeration<T> emptyEnumeration()`
- `static <T> Iterator<T> emptyIterator()`
- `static <T> ListIterator<T> emptyListIterator()`
Zwraca pustą kolekcję, słownik lub iterator.

`java.util.Arrays` **1.2**

- `static <E> List<E> asList(E... array)`
Zwraca widok listowy elementów tablicy, który można modyfikować, ale nie można zmieniać jego rozmiaru.

java.util.List<E> 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`

Zwraca widok listowy elementów składający się z pozycji należących do określonego przedziału.

java.util.SortedSet<E> 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

Zwraca widok elementów z określonego przedziału.

API java.util.NavigableSet<E> 6

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`

Zwraca widok elementów z określonego przedziału. Parametr logiczny określa, czy element brzegowy ma być włączany do widoku.

java.util.SortedMap<K, V> 1.2

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`

Zwraca widok pozycji słownika, których klucze mieszczą się w określonym przedziale.

java.util.NavigableMap<K, V> 6

- `NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)`
- `NavigableMap<K, V> headMap(K from, boolean fromIncluded)`
- `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`

Zwraca widok pozycji słownika, których klucze mieszczą się w określonym przedziale. Parametr logiczny określa, czy element brzegowy ma być włączany do widoku.

java.util.SequencedCollection<E> 21

- `SequencedCollection<E> reversed()`

Zwraca widok sekwencyjnej kolekcji z elementami w odwróconej kolejności.

```
java.util.SequencedSet<E> 21
```

```
■ SequencedSet<E> reversed()
```

Zwraca widok sekwencyjnego zbioru z elementami w odwróconej kolejności.

```
java.util.SequencedMap<E> 21
```

```
■ SequencedMap<E> reversed()
```

Zwraca widok sekwencyjnego słownika z kluczami w odwróconej kolejności.

9.6. Algorytmy

System kolekcji Javy, oprócz implementacji klas kolekcyjnych, zawiera także kilka praktycznych algorytmów. W tym podrozdziale dowiesz się, jak korzystać z tych algorytmów i jak pisać własne, które będą dobrze współpracowały z systemem kolekcji.

9.6.1. Dlaczego algorytmy generyczne?

Generyczne interfejsy kolekcyjne mają pewną dużą zaletę — algorytmy wystarczy zaimplementować tylko jeden raz. Weźmy na przykład prosty algorytm znajdujący największy element w kolekcji. Standardowo jego implementacja polegałaby na użyciu pętli. Poniższa procedura znajduje największy element tablicy:

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];
```

Oczywiście w przypadku listy tablicowej kod wyglądałby nieco inaczej:

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

Jak wygląda sytuacja w listach powiązanych? Nie istnieje w nich szybki mechanizm dostępu swobodnego, ale można użyć iteratora:

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

Pisanie tych pętli jest uciążliwe, a ponadto są one podatne na błędy. Łatwo popełnić błąd pomyłki o jeden (ang. *off-by-one error*), nie wiadomo, czy pętla zadziała prawidłowo w przypadku pustego kontenera lub zawierającego tylko jeden element. Perspektywa ciągłego testowania i debugowania całego tego kodu nie jest zachęcająca, ale implementacja całej masy metod także nie napawa radością, na przykład:

```
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

W takiej sytuacji z pomocą przychodzą interfejsy kolekcyjne. Pomyślmy, jak powinien wyglądać *minimalny* interfejs potrzebny do realizacji tego algorytmu. Dostęp swobodny za pomocą metod `get` i `set` jest operacją bardziej złożoną niż zwykły dostęp za pomocą iteratora. Jak przekonał się przy okazji szukania największego elementu listy powiązanej, do wykonania tego zadania nie jest potrzebny dostęp swobodny. Największy element można znaleźć za pomocą prostej iteracji po elementach. Dlatego metodę `max` można zaimplementować w taki sposób, aby przyjmowała *każdy* obiekt implementujący interfejs `Collection`.

```
public static <T extends Comparable> T max(Collection<T> elements)
{
    if (elements.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = elements.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

Oczywiście, jako że tablice nie są kolekcjami, musimy napisać dla nich osobną metodę:

```
static <T extends Comparable> T max(T[] a)
```

Algorytmy generyczne dają bardzo duże możliwości. W bibliotece standardowej C++ znajduje się mnóstwo przydatnych algorytmów, z których każdy operuje na kolekcjach generycznych. Biblioteka Java Collections Framework nie jest tak bogata, ale klasy `Collections` i `Arrays` zapewniają podstawowe funkcje: sortowanie, wyszukiwanie binarne i kilka prostych algorytmów użytkowych.



W Java Collections Framework brakuje niektórych przydatnych algorytmów, ale można je znaleźć w bibliotece strumieni opisanej w tomie drugim. Aby na przykład znaleźć wartość największą lub najmniejszą, zamień kolekcję lub tablicę w strumień:

```
largest = coll.stream().max(Comparator.naturalOrder()).get();
```

9.6.2. Sortowanie i tasowanie

Weterani komputerowi pamiętają jeszcze czasy, kiedy algorytmy sortujące programowali ręcznie za pomocą kart dziurkowanych. Obecnie algorytmy sortujące wchodzą w skład biblioteki standardowej większości języków programowania. Java nie jest tu wyjątkiem.

Metoda `sort` z klasy `Collections` sortuje kolekcje implementujące interfejs `List`.

```
var staff = new ArrayList<String>();
//Wstawienie elementów do kolekcji
Collections.sort(staff);
```

Ta metoda zakłada, że elementy listy implementują interfejs `Comparable`. Aby posortować listę w inny sposób, można użyć metody `sort` z interfejsu `List` i przekazać obiekt typu `Comparator`. Oto sposób posortowania pracowników według wysokości wynagrodzenia:

```
Collections.sort(staff, sort(Comparator.comparingDouble(Employee::getSalary)));
```

Aby posortować listę w kolejności *malejącej*, należy użyć metody `Comparator.reverseOrder()`. Tworzy ona komparator, który zwraca wynik operacji `b.compareTo(a)`. Na przykład poniższa instrukcja sortuje elementy listy `staff` w kolejności malejącej według porządku określonego przez metodę `compareTo` należącą do typu elementu.

```
Collections.sort(staff, sort(Comparator.reverseOrder()))
```

Natomiast poniższe wyrażenie sortuje pracowników w kolejności od najwyższego wynagrodzenia:

```
Collections.sort(staff, sort(Comparator.comparingDouble(Employee::getSalary).reversed()))
```

W tym miejscu może się rodzić pytanie, jak metoda `sort` sortuje listy. Typowe algorytmy sortujące prezentowane w książkach o algorytmach działają na tablicach i korzystają z dostępu swobodnego. W listach ten typ dostępu może być bardzo wolny. Można je jednak szybko sortować przy użyciu algorytmu sortowania przez scalanie. W Javie jednak nie wykorzystano tej metody. W zamian wszystkie elementy listy są wrzucane do tablicy, tam sortowane przy użyciu innej wersji algorytmu sortowania przez scalanie, a następnie kopiowane w uporządkowanej kolejności z powrotem do listy.

Algorytm sortowania stosowany w Java Collections Framework ustępuje nieco prędkości algorytmowi **quick sort**, który jest standardowo używany do implementacji algorytmów sortujących ogólnego przeznaczenia. Ma on jednak pewną ważną zaletę — jest **stabilny**, to znaczy nie zamienia miejscami takich samych elementów. Po co przejmować się kolejnością identycznych elementów? Oto typowa sytuacja. Mamy listę pracowników posortowaną według nazwisk. Teraz sortujemy według wysokości zarobków. Stabilny algorytm sortujący zachowa kolejność nazwisk. Mówiąc inaczej, lista będzie posortowana najpierw według zarobków, a potem według nazwisk.

Ponieważ kolekcje nie muszą implementować wszystkich swoich metod opcjonalnych, wszystkie metody przyjmujące parametry kolekcyjne muszą informować, kiedy dana kolekcja może być bezpiecznie przekazana do algorytmu. Na przykład z pewnością nie można przekazać listy `unmodifiableList` do algorytmu `sort`. Jakiego rodzaju listę *można* przekazać? Zgodnie z dokumentacją lista musi być modyfikowalna, ale nie musi zezwalać na zmianę rozmiaru.

Właściwości list można przedstawić następująco:

- Lista jest **modyfikowalna** (ang. *modifiable*), jeśli obsługuje metodę `set`.
- Lista **zezwała na zmianę jej rozmiaru** (ang. *resizable*), jeśli obsługuje metody `add` i `remove`.



W rozdziale 6. pokazałem metodę `Arrays.sort`. Są dwie wersje – jedna do sortowania tablic obiektów implementujących interfejs `Comparable`, a druga z komparatorem:

```
String[] staff = . . .;
Arrays.sort(staff);
Arrays.sort(staff, Comparator.comparingDouble(Employee::getSalary));
```

W klasie `Collections` dostępna jest też metoda `shuffle`, która działa odwrotnie do sortowania — tasuje elementy listy. Na przykład:

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards, RandomGenerator.getDefault());
```

Jeśli metodzie `shuffle` zostanie przekazana lista nieimplementująca interfejsu `RandomAccess`, jej zawartość zostanie skopiowana do tablicy, przetasowana i wstawiona z powrotem w odmiennym porządku.

Program przedstawiony na listingu 9.7 wstawia do listy tablicowej 49 obiektów typu `Integer` zawierających liczby od 1 do 49. Następnie tasuje zawartość tej listy i wybiera sześć pierwszych elementów. Na końcu sortuje pobrane wartości i drukuje je.

Listing 9.7. `shuffle/ShuffleTest.java`

```
package shuffle;

import java.util.*;
import java.util.random.*;

/**
 * Program demonstrujący algorytmy tasowania i sortowania
 * @version 1.13 2023-09-30
 * @author Cay Horstmann
 */
public class ShuffleTest
{
    public static void main(String[] args)
    {
        var numbers = new ArrayList<Integer>();
        for (int i = 1; i <= 49; i++)
            numbers.add(i);
        Collections.shuffle(numbers, RandomGenerator.getDef());
        List<Integer> winningCombination = numbers.subList(0, 6);
        Collections.sort(winningCombination);
        System.out.println(winningCombination);
    }
}
```

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`

Sortuje elementy listy przy użyciu stabilnego algorytmu sortującego. Gwarantowana wydajność tego algorytmu wynosi $O(n \log n)$, gdzie n oznacza długość listy.

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
- `static void shuffle(List<?> elements, RandomGenerator r)` **21**

Tasuje element listy. Wydajność tego algorytmu to $O(n \cdot a(n))$, gdzie n to długość listy, a $a(n)$ to średni czas dostępu do elementu.

java.util.List<E> 1.2

- `default void sort(Comparator<? super T> comparator)` **8**

Sortuje listę przy użyciu podanego komparatora.

java.util.Comparator<E> 1.2

- `static <T extends Comparable<? super T>> Comparator<T> reverseOrder()` **8**
- `default Comparator<T> reversed()` **8**

Zwraca komparator odwracający kolejność określoną przez interfejs `Comparable`.

Zwraca komparator odwracający kolejność określoną przez ten komparator.

9.6.3. Wyszukiwanie binarne

Typowy proces poszukiwania obiektu w tablicy polega na odwiedzaniu po kolei wszystkich elementów aż do natrafienia na ten właściwy. Jeśli tablica jest posortowana, można poszukiwany element porównać ze środkowym elementem tej tablicy. Jeśli szukany element jest mniejszy, szukanie kontynuuje się w pierwszej połowie elementów, w przeciwnym przypadku w drugiej. Dzięki temu skala zadania zmniejsza się o połowę. Ten sam proces jest powtarzany na wybranej połowie itd. Jeśli na przykład tablica zawiera 1024 elementy, szukany element można znaleźć (lub stwierdzić, że go nie ma) w dziesięciu krokach, podczas gdy wyszukiwanie liniowe wymagałoby średnio 512 operacji, aby znaleźć element, i 1024, aby potwierdzić, że go nie ma.

Algorytm ten implementuje metoda `binarySearch` z klasy `Collections`. Należy pamiętać, że warunkiem do poprawnego działania tego algorytmu jest wcześniejsze posortowanie kolekcji. Na wejściu należy podać kolekcję, która ma zostać przeszukana (musi implementować interfejs `List`), i element, który ma zostać znaleziony. Jeśli kolekcja nie jest posortowana przez metodę `compareTo` z interfejsu `Comparable`, konieczne jest dostarczenie dodatkowo komparatora.

```
i = Collections.binarySearch(elements, target);
i = Collections.binarySearch(elements, target, comparator);
```

Wartość zwrotna metody `binarySearch` większa bądź równa zero określa indeks pasującego obiektu. To znaczy `elements.get(i)` równa się `target` w porządku porównywania. Wartość ujemna oznacza, że element nie został znaleziony. Można jednak wykorzystać ją do określenia miejsca, w którym *należałoby* `target` wstawić do kolekcji, aby zachować porządek sortowania. Miejsce to jest następujące:

```
insertionPoint = -i - 1;
```

Nie można jednak napisać po prostu `-i`, ponieważ wartość zero nie byłaby jednoznaczna. Innymi słowy, element w odpowiednim miejscu umieszcza poniższa procedura:

```
if (i < 0)
    elements.add(-i - 1, target);
```

Aby było warte uwagi, wyszukiwanie binarne musi obsługiwać dostęp swobodny. Gdyby trzeba było w poszukiwaniu środkowego elementu przemierzyć połowę listy powiązanej, to stracilibyśmy wszystkie jego zalety. Jeśli algorytmowi `binarySearch` przekażesz listę, która nie implementuje interfejsu `RandomAccess`, to przestawia się on na algorytm iteracyjny o wydajności wyszukiwania liniowego.

Klasa `Arrays` ma implementacje wyszukiwania binarnego dla tablic typów podstawowych i obiektów.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

Szuka klucza w liście posortowanej przy użyciu algorytmu wyszukiwania binarnego, jeśli elementy implementują interfejs `RandomAccess`, albo wyszukiwania liniowego we wszystkich pozostałych przypadkach. Długość czasu działania tych metod to $O(a(n) \log n)$, gdzie n określa średni czas dostępu do elementu. Zwracają indeks klucza w liście lub wartość ujemną i , jeśli takiego klucza nie ma w liście. W takim przypadku taki klucz powinien zostać wstawiony w miejscu obliczanym za pomocą wzoru `-i - 1`, aby zachować porządek sortowania.

java.util.Arrays 1.2

- `static int binarySearch(T[] a, T key)`
- `static int binarySearch(T[] a, int start, int end, T key)` 6

Używa algorytmu wyszukiwania binarnego w celu znalezienia klucza `key` w posortowanej tablicy `a`. Jeśli znajdzie klucz, zwraca jego indeks. W przeciwnym razie zwraca ujemną wartość `r`; `-r - 1` to miejsce, w którym należy wstawić klucz, aby tablica `a` pozostała posortowana. Typem komponentu `T` tablicy może być `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` lub `double`.

- `static <T> T binarySearch(T[] a, T key, Comparator<? super T> c)`
- `static <T> T binarySearch(T[] a, int start, int end, T key, Comparator<? super T> c)` 6

Używa algorytmu wyszukiwania binarnego w celu znalezienia klucza `key` w tablicy `a`, która została posortowana przy użyciu określonego komparatora.

9.6.4. Proste algorytmy

W klasie `Collections` znajduje się jeszcze kilka prostych, ale przydatnych algorytmów. Wśród nich można znaleźć prezentowany na początku tego rozdziału algorytm wyszukiwania największej wartości w kolekcji. Z pozostałych można wymienić algorytm kopiujący elementy z jednej listy do innej, zapętlający kontener stałą wartością czy odwracający listę.

Po co w bibliotece umieszczać takie proste algorytmy? Przecież każdy programista mógłby je zaimplementować za pomocą pojedynczych pętli. Lubimy te algorytmy, ponieważ ułatwiają życie programistom *czytającym* cudzy kod. Czytając kod napisany przez kogoś innego, zawsze musimy rozszyfrować intencje innego programisty. Spójrz na przykład na poniższą pętlę:

```
for (int i = 0; i < words.size(); i++)
    if (words.get(i).equals("C++")) words.set(i, "Java");
```

Teraz porównaj ją z poniższym wywołaniem metody:

```
Collections.replaceAll(words, "C++", "Java");
```

Kiedy widzimy wywołanie metody, od razu wiemy, jakie jest przeznaczenie danego fragmentu kodu.

Znajdujące się niżej wyciągi z API opisują proste algorytmy dostępne w klasie `Collections`.

Domyślne metody `Collection.removeIf` i `List.replaceAll` są tylko odrobinę bardziej skomplikowane. Do testowania lub przekształcania elementów należy przekazać wyrażenie lambda. Poniżej na przykład usuwamy wszystkie krótkie słowa, a w pozostałych wszystkie litery zmieniamy na małe:

```
words.removeIf(w -> w.length() <= 3);
words.replaceAll(String::toLowerCase);
```



Predykat `removeIf` powinien sprawdzać tylko ten element, którego usunięcie jest rozważane, a nie kolekcję. Jeśli wykonasz tę drugą czynność, to jej efekt będzie zależny od implementacji.

Wersja `ArrayList` metody `removeIf` przegląda elementy dwa razy. Najpierw wyszukuje elementy, które należy usunąć, a następnie wszystkie je usuwa. W innych kolekcjach znalezione elementy są usuwane od razu.

To ma znaczenie, gdy predykat odczytuje kolekcję. Spójrz na poniższy przykład, w którym `words` jest kolekcją typu `ArrayList` o długości 3:

```
var words = new ArrayList<String>(List.of("Ada", "C++", "Java"))
words.removeIf(w -> w.length() == words.size()); // Teraz words zawiera ["Java"]
```

W pierwszym przebiegu do usunięcia zostały oznaczone wszystkie elementy o długości 3, a w drugim przebiegu zostały one usunięte.

Gdyby jednak zmienna `words` była kolekcją typu `LinkedList` albo `HashSet`, to takie samo wywołanie metody `removeIf` spowodowałoby usunięcie tylko jednego z trzyliterowych słów. Po pierwszym usunięciu predykat sprawdza, czy kolekcja ma rozmiar 2.

Skąd ta różnica? Naiwna, jednoprzebiegowa implementacja byłaby nieefektywna na kolekcjach bazujących na tablicy. Programiści nie od razu się zorientowali, że istnieje efektywny jednoprzebiegowy algorytm. A kiedy już się dowiedzieli, to postanowili go nie używać ze względu na zgodność — zobacz <https://bugs.openjdk.org/browse/JDK-8143577>.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`

Zwraca najmniejszy lub największy element kolekcji (wartości brzegowe parametrów uproszczone dla zachowania przejrzystości).

- `static <T> void copy(List<? super T> to, List<T> from)`

Kopiuje wszystkie elementy z listy źródłowej do tych samych miejsc w liście docelowej. Lista docelowa musi mieć przynajmniej taką samą długość jak lista źródłowa.

- `static <T> void fill(List<? super T> l, T value)`

Wstawia na wszystkich pozycjach w liście tę samą wartość.

- `static <T> boolean addAll(Collection<? super T> c, T... values) 5`

Dodaje wszystkie wartości do określonej kolekcji i zwraca wartość `true`, jeśli kolekcja w wyniku tego działania zmieniła się.

- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4`

Zastępuje wszystkie elementy identyczne z `oldValue` wartościami `newValue`.

- `static int indexOfSubList(List<?> l, List<?> s) 1.4`

- `static int lastIndexOfSubList(List<?> l, List<?> s) 1.4`

Zwraca indeks pierwszej lub ostatniej podlisty listy `l` równej `s` lub `-1`, jeśli żadna podlista `l` nie jest równa `s`. Jeśli na przykład lista `l` to `[s, t, a, r]`, a `s` to `[t, a, r]`, obie metody zwrócą indeks 1.

- `static void swap(List<?> l, int i, int j) 1.4`

Zamienia miejscami elementy znajdujące się w określonych miejscach.

- `static void reverse(List<?> l)`

Odwraca kolejność elementów listy. Na przykład lista `[t, a, r]` po odwróceniu to `[r, a, t]`. Długość czasu działania tej metody to $O(n)$, gdzie n określa długość listy.

- `static void rotate(List<?> l, int d)` **1.4**

Przeprowadza rotację elementów listy, przenosząc obiekt z indeksem i do miejsca $(i + d) \% l.size()$. Na przykład rotacja listy `[t, a, r]` o dwa pola daje w wyniku listę `[a, r, t]`. Długość czasu działania tej metody to $O(n)$, gdzie n określa długość listy

- `static int frequency(Collection<?> c, Object o)` **5.0**

Zwraca liczbę elementów w `c`, które są równe obiektowi `o`.

- `boolean disjoint(Collection<?> c1, Collection<?> c2)` **5.0**

Zwraca wartość `true`, jeśli kolekcje nie mają więcej elementów wspólnych.

`java.util.Collection<T>` **1.2**

- `default boolean removeIf(Predicate<? super E> filter)` **8**

Usuwa wszystkie elementy spełniające warunek.

`java.util.List<E>` **1.2**

- `default void replaceAll(UnaryOperator<E> op)` **8**

Wykonuje operację na wszystkich elementach listy.

9.6.5. Operacje zbiorowe

Istnieje kilka metod do zbiorowego kopiowania i usuwania elementów. Na przykład poniższe wyrażenie usuwa z `coll1` wszystkie elementy obecne w `coll2`.

```
coll1.removeAll(coll2);
```

Natomiast poniższe wyrażenie usuwa z `coll1` wszystkie elementy, których *nie* ma w `coll2`.

```
coll1.retainAll(coll2);
```

Zobaczmy typowy przykład zastosowania tej operacji.

Przyjmijmy, że chcemy znaleźć *część wspólną* dwóch zbiorów. Zaczniemy od utworzenia nowego zbioru do przechowywania znalezionej części.

```
var result = new HashSet<String>(firstSet);
```

Wykorzystujemy tutaj fakt, że każda kolekcja posiada konstruktor, którego parametr jest inną kolekcją przechowującą wartości początkowe.

Teraz użyjemy metody `retainAll`:

```
result.retainAll(secondSet);
```

Zachowuje ona wszystkie elementy, które znajdują się w obu zbiorach. W ten sposób znaleźliśmy część wspólną dwóch zbiorów bez tworzenia pętli.

Można pójść nawet dalej i zastosować operację zbiorczą do *widoku*. Wyobraźmy sobie, że dysponujemy słownikiem przechowującym obiekty typu `Employee` z kluczami w postaci identyfikatorów ID oraz zbiorem identyfikatorów pracowników, którzy mają zostać zwolnieni.

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

Wystarczy utworzyć zbiór kluczy i usunąć wszystkie identyfikatory zwolnionych pracowników.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Ponieważ zbiór kluczy jest widokiem słownika, klucze i skojarzeni z nimi pracownicy są automatycznie usuwani z tego słownika.

Przy użyciu widoku podprzedziału można ograniczyć operacje zbiorcze do podlist i podzbiorów. Załóżmy, że chcemy dodać 10 elementów z listy do innego kontenera. Pobieramy dziesięć pierwszych elementów i umieszczamy je w podliście:

```
relocated.addAll(staff.subList(0, 10));
```

Na podprzedziale tym można także wykonywać operacje modyfikujące.

```
staff.subList(0, 10).clear();
```

9.6.6. Konwersja pomiędzy kolekcjami a tablicami

Ponieważ znaczna część API Javy powstała przed pojawieniem się systemu Java Collections Framework, czasami konieczna jest konwersja tradycyjnych tablic na bardziej nowoczesne kolekcje.

Jeśli dysponujemy tablicą, musimy przekonwertować ją na kolekcję. Można do tego celu użyć metody `List.of`. Na przykład:

```
String[] names = . . .;
List<String> staff = List.of(names);
```

Utworzenie tablicy z kolekcji nie jest już takie łatwe. Można oczywiście użyć do tego metody `toArray`:

```
Object[] names = staff.toArray();
```

Jednak wynik będzie tablicą obiektów typu `Object`. Nie można zastosować rzutowania, nawet jeśli wiadomo, że kolekcja zawierała obiekty określonego typu:

```
String[] names = (String[]) staff.toArray(); // Błąd!
```

Metoda `toArray` zwraca tablicę typu `Object[]`, którego nie można zmienić. Zamiast tego należy do metody `toArray` przekazać wyrażenie konstrukcyjne. Zostanie użyty konstruktor, który utworzy tablicę odpowiedniego typu:

```
String[] values = staff.toArray(String[]::new);
```



Zanim pojawił się pakiet JDK 11, programista musiał używać innej formy metody `toArray`, przekazując tablicę odpowiedniego typu:

```
String[] values = staff.toArray(new String[0]);
```

Ta metoda `toArray` tworzy inną tablicę tego samego typu. Albo, jeśli tablica ma wystarczającą długość, zostaje wykorzystana ta tablica:

```
staff.toArray(new String[staff.size()]);
```

W tym przypadku nie zostaje utworzona żadna nowa tablica.

9.6.7. Pisanie własnych algorytmów

Pisząc własny algorytm (a raczej każdą metodę przyjmującą jako parametr kolekcję), należy, kiedy to tylko możliwe, wykorzystywać *interfejsy*, a nie konkretne implementacje. Wyobraźmy sobie na przykład, że chcemy przetworzyć pewne elementy. Oczywiście możemy zaimplementować taką metodę:

```
public void processItems(ArrayList<Item> items)
{
    for (Item item : items)
        operacje na elementach
}
```

Jednak w ten sposób ograniczamy zakres ruchu wywołującego metodę — elementy muszą być podane w postaci obiektu `ArrayList`. Jeśli zdarzy się, że elementy te będą w jakiejś innej kolekcji, konieczne będzie ich przepakowanie. Znacznie lepiej byłoby wykorzystać bardziej ogólną kolekcję.

Należy sobie zadać następujące pytanie: jaka jest najbardziej ogólna metoda, która przyda się w tym zastosowaniu? Czy zależy nam na kolejności? W takim razie należy użyć listy. Jeśli natomiast kolejność jest nieistotna, można użyć dowolnej kolekcji:

```
public void processItems(Collection<Item> items)
{
    for (Item item : items)
        operacje na elementach
}
```

Teraz metodę tę można wywołać przy użyciu struktury `ArrayList`, `LinkedList` bądź tablicy opakowanej w wywołanie metody `List.of`.



W tym przypadku lepszym rozwiązaniem jest akceptowanie `Iterable<Item>`. Interfejs `Iterable` zawiera jedną abstrakcyjną metodę `iterator`, której ulepszona pętla `for` używa w swoich wewnętrznych mechanizmach. Interfejs `Collection` rozszerza `Iterable`.

Z drugiej strony, jeśli metoda zwraca wiele elementów, nie należy ograniczać swoich możliwości w odniesieniu do przyszłych ulepszeń. Spójrz na poniższy przykład kodu:

```

public ArrayList<Item> lookupItems(. . .)
{
    var result = new ArrayList<Item>();
    . . .
    return result;
}

```

Ta metoda informuje, że zwraca `ArrayList`, mimo że dla wywołującego rodzaj listy prawie na pewno będzie bez znaczenia. Jeśli zamiast tego zwrócisz `List`, zawsze możesz dodać gałąź zwracającą pustą lub singletonową listę za pomocą wywołania metody `List.of`.



Skoro używanie interfejsów kolecyjnych jako parametrów i typów zwrotnych jest takim dobrym rozwiązaniem, czemu nie jest ono stosowane konsekwentnie w API Javy? Na przykład klasa `JComboBox` ma dwa konstruktory:

```

JComboBox(Object[] items)
JComboBox(Vector<?> items)

```

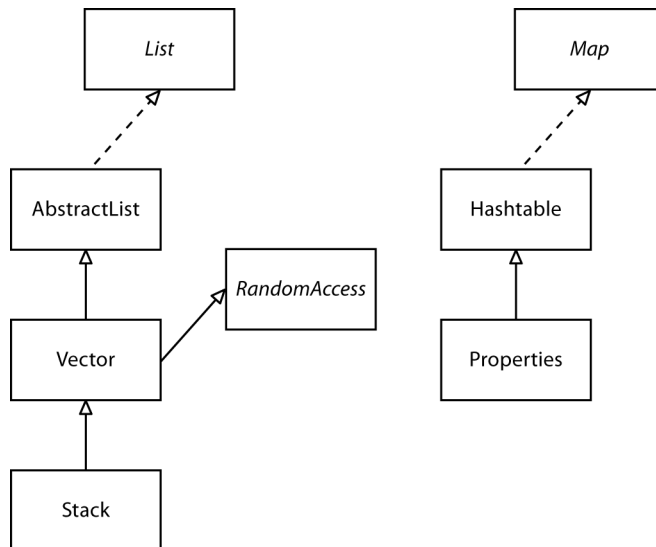
Powodem jest po prostu czas. API Swing powstał przed Java Collections Framework.

9.7. Stare kolekcje

Od początku istnienia Javy w języku istnieje kilka „przestarzałych” klas kontenerowych, które zostały dodane jeszcze przed powstaniem Java Collections Framework.

Wszystkie je wcielono do tego systemu (zobacz rysunek 9.12), więc poniżej przedstawiamy ich zwięzły opis.

Rysunek 9.12.
Stare klasy
w ramowym
systemie kolekcji



9.7.1. Klasa Hashtable

Standardowa metoda `Hashtable` pełni takie samo zadanie co `HashMap` i ma zasadniczo taki sam jak ona interfejs. Metody tej klasy, podobnie jak klasy `Vector`, są synchronizowane. Jeśli nie potrzebujesz zgodności ze starym kodem, używaj klasy `HashMap`. Jeśli potrzebna jest możliwość dostępu współbieżnego, należy skorzystać ze struktury `ConcurrentHashMap` — zobacz rozdział 10.

9.7.2. Wyliczenia

Stare kolekcje do przemierzania elementów ułożonych liniowo wykorzystują interfejs `Enumeration`. Znajdują się w nim dwie metody: `hasMoreElements` i `nextElement`. Są one wiernymi odpowiednikami metod `hasNext` i `next` z interfejsu `Iterator`.

Jeśli znajdziesz ten interfejs ze starymi klasami, to przy użyciu metody `Collections.list` możesz zbierać elementy w kolekcji `ArrayList`. Na przykład klasa `LogManager` ukazuje nazwy loggerów tylko w postaci obiektów typu `Enumeration`. Tak można pobrać je wszystkie:

```
ArrayList<String> loggerNames = Collections.list(LogManager.getLoggerNames());
```

Ewentualnie wyliczenie można zamienić w iterator:

```
LogManager.getLoggerNames().asIterator().forEachRemaining(n -> { . . . });
```

Czasami spotyka się stare metody wymagające wyliczenia jako parametru. Statyczna metoda `Collections.enumeration` tworzy obiekt wyliczeniowy zapełniony elementami pobranymi z kolekcji. Na przykład:

```
List<InputStream> streams = . . . ;
var in = new SequenceInputStream(Collections.enumeration(streams));
// Konstruktor SequenceInputStream wymaga na wejściu typu wyliczeniowego.
```



W języku C++ jako parametry często stosuje się iteratory. Na szczęście w Javie wielu programistów stosuje tę technikę. O wiele lepiej jest zamiast iteratora przekazać kolekcję. Kolekcja jest o wiele bardziej użyteczna. Odbiorca może zawsze w razie potrzeby wydobyć iterator z obiektu kolekcji, a poza tym ma do dyspozycji wszystkie metody tej kolekcji. W starszym kodzie można jednak spotkać wyliczenia, ponieważ były one jedynym sposobem uzyskania kolekcji generycznych przed wprowadzeniem Java Collections Framework w Javie 1.2.

`java.util.Enumeration<E>` **1.0**

■ `boolean hasMoreElements()`

Zwraca wartość `true`, jeśli są jeszcze jakieś elementy.

■ `E nextElement()`

Zwraca kolejny element do odwiedzenia. Nie należy wywoływać tej metody, jeśli metoda `hasMoreElements()` zwróci wartość `false`.

- default Iterator<E> asIterator() 9

Zwraca iterator przeglądający elementy wyliczenia.

java.util.Collections 1.2

- static <T> Enumeration<T> enumeration(Collection<T> c)

Zwraca wyliczenie zawierające elementy c.

- public static <T> ArrayList<T> list(Enumeration<T> e)

Zwraca listę tablicową zawierającą elementy znajdujące się w e.

9.7.3. Słowniki własności

Słownik własności (ang. *property map*) to struktura danych o bardzo szczególnych cechach. Ma trzy wyróżniające ją właściwości:

- Klucze i wartości są łańcuchami.
- Słownik ten można łatwo zapisać w pliku i załadować z pliku.
- Istnieje dodatkowa tablica z wartościami domyślnymi.

Klasa Javy implementująca słownik własności nosi nazwę `Properties`. Słowniki własności są powszechnie wykorzystywane do określania opcji konfiguracyjnych programów, na przykład:

```
var settings = new Properties();
settings.setProperty("width", "600.0");
settings.setProperty("filename", "/home/cay/books/corejava/code/v1ch09/raven.html");
```

Za pomocą metody `getProperty` można wyszukać wartość odpowiadającą kluczowi.

```
String filename = settings.getProperty("filename");
```



Z powodów historycznych klasa `Properties` implementuje `Map<Object, Object>`. Dzięki temu można używać metod `get` i `put` interfejsu `Map`. Tylko że metoda `get` zwraca typ `Object`, a metoda `put` pozwala na wstawienie dowolnego obiektu. Lepiej jest trzymać się metod `getProperty` i `setProperty`, które działają na łańcuchach, nie obiektach.

Za pomocą metody `store` można zapisać własności w pliku. Drugi argument to komentarz, który zostanie dodany do pliku.

```
var out = new FileWriter("program.properties", StandardCharsets.UTF_8);
settings.store(out, "Właściwości programu");
```

Przykładowy zbiór daje następujący wynik:

```
#Program Properties
#Sun Dec 31 12:54:19 PST 2023
top=227.0
left=1286.0
```

```
width=423.0
height=547.0
filename=/home/cay/books/corejava/code/v1ch09/raven.html
```

Aby załadować własności z pliku, należy użyć następującej instrukcji:

```
var in = new FileReader("program.properties");
settings.load(in);
```



Gdy metody `load` i `store` są używane ze strumieniami wejścia i wyjścia, to stosowane jest archaiczne kodowanie znaków ISO-8859-1 i znaki `> U+00FF` są zarezerwowane jako symbole zastępcze Unicode. Aby włączyć kodowanie UTF-8, należy używać operacji odczytu i zapisu, jak w powyższych przykładach. Przed Javą 18 należy ręcznie ustawić kodowanie na `StandardCharsets.UTF_8`.

Klasa `Properties` ma dwa mechanizmy dostarczania wartości domyślnych. Po pierwsze, przy każdym wyszukiwaniu wartości łańcucha można określić wartość domyślną, która ma zostać użyta, jeśli dany klucz nie zostanie znaleziony.

```
String filename = settings.getProperty("filename", "");
```

Jeśli w słowniku własności znajduje się własność `"filename"`, zostanie ona ustawiona na określony łańcuch. W przeciwnym przypadku zostanie ustawiona na pusty łańcuch.

Jeśli określanie wartości domyślnej w każdym wywołaniu metody `getProperty` jest zbyt żmudne, wszystkie wartości domyślne można umieścić w dodatkowym słowniku własności i przekazać go do konstruktora pierwszego słownika własności.

```
var defaultSettings = new Properties();
defaultSettings.setProperty("width", "600");
defaultSettings.setProperty("height", "400");
defaultSettings.setProperty("filename", "");
...
var settings = new Properties(defaultSettings);
```

Można nawet zdefiniować domyślne wartości dla domyślnych wartości, jeśli przekaże się inny argument słownika własności do konstruktora `defaultSettings`, ale zwykle nikt z tej możliwości nie korzysta.

W plikach z przykładowym kodem znajduje się program przedstawiający sposób użycia własności do przechowywania i ładowania stanu programu. Wykorzystuje on program `ImageViewer` z rozdziału 2. oraz zapamiętuje położenie ramki, rozmiar ramki i ostatnio załadowany plik. Uruchom program, załaduj plik, a następnie przesuń okno i zmień jego rozmiar. Potem zamknij program i otwórz go ponownie, aby przekonać się, że zapamiętał otwarty wcześniej plik oraz ustawienia okna. Można też ręcznie edytować plik `.corejava/ImageViewer.properties` znajdujący się w katalogu głównym.

Własności to proste tabele bez hierarchicznej struktury. Często imituje się w nich hierarchię poprzez wprowadzenie nazw kluczy, na przykład `window.main.color`, `window.main.title` itd. Tylko że klasa `Properties` nie zawiera żadnych metod, które mogłyby być pomocne w organizacji takich hierarchii. Dlatego do przechowywania skomplikowanych konfiguracji preferencji lepiej używać klasy `Preferences` — patrz rozdział 10 w tomie drugim.

java.util.Properties 1.0

- **Properties()**
Tworzy pustą mapę właściwości.
- **Properties(Properties defaults)**
Tworzy pustą mapę właściwości z określonymi wartościami domyślnymi.
- **String getProperty(String key)**
Pobiera właściwość. Zwraca łańcuch powiązany z kluczem (key) lub – jeśli go nie znajdzie w tabeli – łańcuch powiązany z kluczem w domyślnej tabeli albo null, jeśli klucza nie było także w domyślnej tabeli.
- **String getProperty(String key, String defaultValue)**
Pobiera właściwość o domyślnej wartości, jeśli klucz (key) nie zostanie znaleziony. Zwraca łańcuch powiązany z kluczem lub domyślny łańcuch, jeśli nie było go w tabeli.
- **Object setProperty(String key, String value)**
Ustawia właściwość. Zwraca poprzednio ustawioną wartość danego klucza.
- **Set<String> stringPropertyNames() 6**
Zwraca zbiór wszystkich kluczy, włącznie z kluczami z domyślnego słownika.
- **void load(Reader in) throws IOException 6**
Wczytuje słownik właściwości z czytnika.
- **void store(Writer out, String header) 6**
Zapisuje słownik właściwości w writerze. header znajduje się w pierwszym wierszu zapisanego pliku.

9.7.4. Właściwości systemowe

Metoda `System.getProperties` zwraca obiekt typu `Properties` zawierający informacje dotyczące systemu. Na przykład katalog główny ma klucz `"user.home"`.

Można go odczytać za pomocą metody `getProperty` zwracającej klucz jako łańcuch:

```
String userDir = System.getProperty("user.home");
```

Aby sprawdzić wersję Javy w maszynie wirtualnej, należy znaleźć właściwość `"java.version"`. W odpowiedzi zostanie zwrócony łańcuch w rodzaju `"21.0.1"` (albo `"1.8.0"` do Javy 8).



Jak widać, w Javie 9 zmieniono sposób numeracji wersji. Ta pozornie drobna zmiana narobiła kłopotów wielu narzędziom, które korzystały ze starego formatu. Dlatego, jeśli przetwarzasz łańcuch wersji, koniecznie przeczytaj dokument JEP 322 na stronie <http://openjdk.java.net/jeps/322>, aby dowiedzieć się, jak będą formatowane łańcuchy wersji w przyszłości — albo przynajmniej do następnej zmiany formatu numerowania.



Aby wydrukować wszystkie właściwości systemowe, wykonaj poniższe polecenie:

```
java -XshowSettings:properties
```

Właściwość systemowa `java.version` jest ustawiana przez maszynę wirtualną i nie należy jej zmieniać. Inne właściwości systemowe można zmieniać za pomocą opcji wiersza poleceń `-D`:

```
java -Duser.language=fr -Duser.country=CA MyProg
```

Aby uczynić właściwości dostępnymi w `jshe11`, użyj opcji `-R` w następujący sposób:

```
jshe11 -R-Duser.language=fr -R-Duser.country=CA
```

Aby uzyskać liczbową lub logiczną wartość właściwości systemowej, użyj metod statycznych `Boolean.getBoolean()`, `Integer.getInteger()` i `Long.getLong()`:

```
Integer version = Integer.getInteger("java.specification.version");
```

java.lang.System 1.0

■ `Properties getProperties()`

Pobiera wszystkie własności systemowe. Aplikacja musi mieć uprawnienia do pobierania wszystkich własności albo zostanie zgłoszony wyjątek bezpieczeństwa.

■ `String getProperty(String key)`

Pobiera własność systemową o określonym kluczu. Poniższe własności zawsze są obecne:

```
java.version
java.version.date
java.vendor
java.vendor.url
java.home
java.class.path
java.library.path
java.class.version
java.specification.version
java.specification.vendor
java.specification.name
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.io.tmpdir
user.name
user.home
user.dir
```

```
native.encoding
stdout.encoding
stderr.encoding
```

java.lang.Boolean 1.0

- static boolean getBoolean(String name)

Zwraca true, jeśli właściwość systemowa o określonej nazwie ma wartość "true" (ignorując wielkość liter), i false w przeciwnym przypadku.

java.lang.Integer 1.0

- static Integer getInteger(String nm)
- static Integer getInteger(String nm, int val)
- static Integer getInteger(String nm, Integer val)

Zwraca wynik wywołania metody Integer.decode na właściwości systemowej o określonej nazwie. Jeśli nie ma właściwości systemowej o takiej nazwie lub jeśli jej wartości nie można rozkodować, metoda ta zwraca wartość domyślną albo – jeśli żadna nie jest podana – null.

java.lang.Long 1.0

- static Long getLong(String nm)
- static Long getLong(String nm, int val)
- static Long getLong(String nm, Long val)

Zwraca wynik wywołania metody Long.decode na właściwości systemowej o określonej nazwie. Jeśli nie ma właściwości systemowej o takiej nazwie lub jeśli jej wartości nie można rozkodować, metoda ta zwraca wartość domyślną albo – jeśli żadna nie jest podana – null.

9.7.5. Stosy

API Javy od wersji 1.0 zawiera klasę Stack udostępniającą powszechnie znane metody, takie jak push i pop. Klasa ta dziedziczy jednak po klasie Vector, która z teoretycznego punktu widzenia nie jest zadowalająca — pozwala na stosowanie takich niewłaściwych stosom metod jak insert i remove, wstawiających i usuwających wartości w dowolnym miejscu, nie tylko na wierzchu stosu.

java.util.Stack<E> 1.0

- E push(E item)

Wstawia element na stos i go zwraca.

- E pop()

Usuwa i zwraca element znajdujący się na wierzchu stosu. Nie należy używać tej metody na pustym stosie.

■ `E peek()`

Zwraca element z wierzchu stosu, nie usuwając go. Nie należy wywoływać tej metody na pustym stosie.

9.7.6. Zbiory bitów

Klasa `BitSet` umożliwia przechowywanie szeregów bitowych zapakowanych w „słowa” zależne od implementacji, które obecnie są wartościami typu `long`. To o wiele efektywniejsze rozwiązanie niż użycie tablicy lub wartości logicznej.

Klasa `BitSet` udostępnia wygodny interfejs do odczytu, ustawiania oraz resetowania poszczególnych bitów. Interfejs ten pozwala uniknąć tworzenia masek bitowych i innych operacji na bitach, które są konieczne przy przechowywaniu bitów w zmiennych typu `int` lub `long`.

Na przykład dla zbioru `BitSet` o nazwie `bucketOfBits` poniższa instrukcja zwróci wartość `true`, jeśli *i*-ty bit jest włączony, lub `false` w przeciwnym przypadku.

```
bucketOfBits.get(i)
```

Podobnie poniższa instrukcja włącza *i*-ty bit.

```
bucketOfBits.set(i)
```

Z kolei poniższa instrukcja wyłącza *i*-ty bit.

```
bucketOfBits.clear(i)
```

Za pomocą obiektu typu `BitSet` można reprezentować zbiór nieujemnych liczb całkowitych o ograniczonej wartości maksymalnej. Ustaw *i*-ty bit, aby pokazać, że liczba *i* znajduje się w zbiorze. Jest to o wiele wydajniejsze od użycia `Set<Integer>`, jeżeli górna granica nie jest zbyt duża i zbiór ma wiele elementów.



Szablon `bitset` w C++ ma taką samą funkcjonalność jak klasa `BitSet` w Javie.

`java.util.BitSet` 1.0

■ `BitSet(int initialCapacity)`

Tworzy zbiór bitów.

■ `int cardinality()` 1.4

Zwraca liczbę ustawionych bitów lub, w przypadku traktowania ich jako zbioru liczb całkowitych, liczb elementów

■ `int length()` 1.2

Zwraca „logiczną długość” (1 plus indeks najwyższego ustawionego). Ta operacja może być używana do iteracji przez elementy.

- `int size()`

Zwraca liczbę bitów obecnie dostępnych w wewnętrznej strukturze danych, nie liczbę elementów zbioru.

- `boolean get(int bit)`

Pobiera bit.

- `void set(int bit)`

Ustawia bit.

- `void clear(int bit)`

Usuwa bit.

- `void and(BitSet set)`

Tworzy sumę logiczną dwóch zbiorów bitów.

- `void or(BitSet set)`

Tworzy alternatywę logiczną dwóch zbiorów.

- `void xor(BitSet set)`

Wykonuje działanie logiczne XOR (lub wykluczające) na dwóch zbiorach bitów.

- `void andNot(BitSet set)`

Usuwa wszystkie bity ze zbioru bitów, które są ustawione w innym zbiorze bitów.

- `IntStream stream() 8`

Zwraca strumień ustawionych wartości indeksów bitów, lub, w przypadku traktowania ich jako zbioru liczb całkowitych, strumień elementów

Zastosowanie zbiorów bitów zademonstrujemy za pomocą algorytmu do znajdowania liczb pierwszych, czyli sita Eratostenesa (liczba pierwsza to taka, która dzieli się bez reszty tylko przez siebie samą i jeden, a sito Eratostenesa jest pierwszą odkrytą metodą obliczania tych liczb). Nie jest to najlepszy algorytm do znajdowania liczb pierwszych, ale z niewyjaśnionych powodów zyskał dużą popularność jako test wydajności kompilatorów (nie jest to też dobry test wydajności, ponieważ testuje przede wszystkim operacje bitowe).

Jest to jednak ukłon w stronę tradycji — przedstawiamy implementację tego algorytmu. Prezentowany program zlicza wszystkie liczby pierwsze w zakresie od 2 do 2 000 000 (jest ich 148 933, a więc pewnie lepiej ich wszystkich nie drukować).

Nie zagłębiając się zbyt w szczegóły implementacyjne tego programu, jego zadaniem jest przejście przez zbiór bitów zawierający dwa miliony elementów. Najpierw włączamy wszystkie bity. Następnie wyłączamy bity będące wielokrotnościami liczb, które wiadomo, że są pierwsze. Liczby odpowiadające bitom pozostałym po tym procesie są pierwsze. Listing 9.8 przedstawia implementację tego programu w języku Java, a listing 9.9 — w języku C++.



Mimo że sito nie jest dobrym testem wydajności, nie mogliśmy się oprzeć pokusie zmierzenia czasu obu implementacji algorytmu. Oto wyniki uzyskane na komputerze z procesorem i7-1165G7, 32 GB pamięci RAM i systemem Ubuntu 22.04:

- C++ (g++ 11.4.0): 70 milisekund,
- Java (Java 21): 20 milisekund.

Test ten przeprowadziłem w trzynastu wydaniach tej książki i w ostatnich dziewięciu Java bije C++ na głowę. Trzeba jednak ujawnić, że jeśli zoptymalizujemy kompilator C++, pobije on Javę, uzyskując czas 16 milisekund. Taki wynik w Javie można by było uzyskać tylko wtedy, gdyby program działał na tyle długo, aby włączył się kompilator JIT Hotspot.

Listing 9.8. sieve/Sieve.java

```
package sieve;

import java.util.*;

/**
 * Program wykonujący test porównawczy na bazie algorytmu sita Eratostenesa. Oblicza wszystkie
 * liczby pierwsze do 2 000 000.
 * @version 1.22 2021-06-17
 * @author Cay Horstmann
 */
public class Sieve
{
    public static void main(String[] s)
    {
        int n = 2000000;
        long start = System.nanoTime();
        var bitSet = new BitSet(n + 1);
        int i;
        for (i = 2; i <= n; i++)
            bitSet.set(i);
        i = 2;
        while (i * i <= n)
        {
            if (bitSet.get(i))
            {
                int k = i * i;
                while (k <= n)
                {
                    bitSet.clear(k);
                    k += i;
                }
            }
            i++;
        }
        long end = System.nanoTime();
        System.out.println(bitSet.cardinality() + " liczb pierwszych");
        System.out.println((end - start) / 1000 + " milisekund");
    }
}
```

Listing 9.9. Sieve.cpp

```
/**
 * @version 1.22 2021-06-17
 * @author Cay Horstmann
 */
#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    const int N = 2000000;
    clock_t cstart = clock();
    bitset<N + 1> b;
    int i;
    for (i = 2; i <= N; i++)
        b.set(i);
    i = 2;
    while (i * i <= N)
    {
        if (b.test(i))
        {
            int k = i * i;
            while (k <= N)
            {
                b.reset(k);
                k += i;
            }
        }
        i++;
    }

    clock_t cend = clock();
    double millis = 1000.0 * (cend - cstart) / CLOCKS_PER_SEC;

    cout << b.count() << " liczb pierwszych\n" << millis << " milisekund\n";

    return 0;
}
```

Na tym kończy się nasza podróż po architekturze kolekcji w języku Java. Jak widać udostępnia ona bogaty zestaw klas kolekcyjnych, mających na celu zaspokajanie potrzeb programisty.

Skorowidz

A

- abstrakcja, 265
- adnotacja, annotation, 448
 - @Deprecated, 675
 - @Documented, 675, 677
 - @FunctionalInterface, 675
 - @Generated, 675, 676
 - @Inherited, 675, 678
 - @LogEntry, 689
 - @Override, 243, 675, 676
 - @Persistent, 678
 - @Retention, 675
 - @SafeVarargs, 461, 675, 676
 - @Serial, 676
 - @SuppressWarnings, 675, 678
 - @Target, 675
- adnotacje, 33, 114, 666
 - definiowanie, 672
 - deklaracji, 668
 - do generacji kodu źródłowego, 684
 - elementy, 667
 - kontenerowe, 678
 - metaadnotacje, 677
 - mnożenie, 668
 - na poziomie kodu bajtowego, 689
 - na poziomie kodu źródłowego, 682
 - powielanie, 668
 - procesory, 682
 - przetwarzanie, 678, 681, 682
 - regularne, 675
 - typów, 668
 - typy elementów, 667, 673
 - w API Javy, 675
 - w działającym programie, 678
 - zastosowań typów, 669
- Adoptium, 39
- agregacja, 140
- akcesory, 147, 159, 160
- algorytm, 137
 - QuickSort, 127
 - znajdowania liczb pierwszych, 559
 - wyszukiwania binarnego, 544
- algorytmy
 - generyczne, 540
 - proste, 546
 - równoległe, 640
 - sortowania, 542
 - tworzenie, 550
- analiza
 - danych ze stosu wywołań, 402
 - funkcjonalności klasy, 294
 - obiektów w czasie działania programu, 301
- anonimowe klasy wewnętrzne, 367
- API, Application Programming Interface, 20
- API String, 85
- aplety, 30
- aplikacja ImageViewer, 44
- argument
 - jawny, explicit parameter, 158
 - niejawny, implicit parameter, 158
- argumenty, 57
- ASCII, 62
- asercje, 410
 - sprawdzanie parametrów, 412
 - stosowanie, 412
 - włączanie, 411
 - wyłączanie, 411
 - zastosowanie, 414
- asocjacja, 140
- asynchroniczne obliczenia, 642

atomowe
 modyfikowanie elementów słowników, 634
 zmiennne, 621
autoboxing, 33, 260
automatyczne
 opakowywanie, autoboxing, 260
 usuwanie nieużytków, 24
autowrapping, 260

B

backend, 415
bezpieczeństwo, 26, 36
 wątków, 619
biblioteka
 BCEL, 689
 JCommander, 666
 JSON, 274
 picocli, 666
 refleksyjna, 287
biblioteki Javy, 35
blok, 54, 102
 inicjujący, 180
 inicjujący obiektów, 183
 synchronizowany, 616
 try-catch, 393
 try-finally, 400
blokady, 601, 602, 611, 618
 ad hoc, 616
 jawne, 612
 licznik wywołań, 603
 wewnętrzne, 612, 613
 wielowejściowe, reentrant, 603, 604
 z zasadą uczciwości, 604
bloki tekstowe, 91
blokowanie po stronie klienta, 617
błędy, 385
 danych wejściowych, 385
 debugowanie, 429
 kod źródłowy, 385
 pomyłka o jeden, 541
 urządzenia, 385

C

callback, 330
ciało metody, 56

D

dane
 wejściowe, 93
 wyjściowe, 93
 formatowanie, 96
 ze stosu wywołań, 402
debuger, 429
debugowanie, 429
dedukcja typów, 468
definiowanie
 adnotacji, 672
 interfejsu, 317
 klasy, 150
 klasy generycznej, 438
 stałej klasowej, 68
 zmiennej w C, 67
deklarowanie
 list tablicowych, 253
 rekordu, 186
 tablic, 122
 wyjątków, 290
 wyjątków kontrolowanych, 388
 zmiennej tablicowej, 122
 zmiennych, 65
dekrementacja, 74
demony, 575
diagram przepływu sterowania
 instrukcja
 if, 103
 if-else, 104
 if-else if, 105
 switch, 113
 pętla
 do-while, 109
 for, 110
 while, 106
diagramy klas, 141
dokumentacja, 41
 adnotacji, 677
 API Javy, 87
dopasowywanie do wzorca, 234, 280
dostęp
 chroniony, 236
 do elementów listy tablicowej, 256
 do elementów tablicy, 124
 do klasy publicznej, 191
 do pakietu, 196
 do plików, 100

- do pól, 159
- do zmiennych, 351
- swobodny, random access, 492
- drzewa czerwono-czarne, 509
- duże liczby, big numbers, 58
- dynamiczność, 30
- dyrektywa import, 95
- dziedziczenie, 138, 140, 217, 313
 - dostęp chroniony, 236
 - hierarchia, 225
 - interfejs, 324
 - klasy abstrakcyjne, 265
 - klasy finalne, 229
 - klasy zapieczętowane, 274
 - metody finalne, 229
 - metody przesłaniające, 219
 - polimorfizm, 222, 225
 - porównywanie obiektów, 240
 - przesłanie metod, 221
 - rzutowanie, 232
 - wielokrotne, multiple inheritance, 325
- dzielenie modułu, 69
- dzienniki, 415, 416
 - API rejestrowania platformy, 416
 - backend, 418
 - filtry, 421
 - formatery, 421
 - handlery, 419
 - konfiguracja, 418
 - nazwy plików, 421
 - parametry konfiguracyjne obiektu Handler, 420

E

- Eclipse, 39
 - edycja kodu źródłowego, 49
 - komunikaty o błędach, 50
 - konfigurowanie projektu, 48
 - tworzenie projektu, 47
- egzekutor, 583, 588
- egzemplarz klasy, 137
- eksport kwalifikowany, 719
- epoka, 144
- etykieta, 118

F

- fabryki wątków, 579
- flagi wiersza poleceń, 716

- formatowanie danych wyjściowych, 96
- fragmenty kodu, code snippets, 212
- frontend, 415
- funkcje
 - czysto wirtualne, 267
 - matematyczne, 70
- funkcyjne interfejsy, 344

G

- Garbage Collector, 524
- generator liczb losowych, 625
- generowanie
 - dokumentacji javadoc, 212
 - liczb losowych, 182
- generyczne
 - listy tablicowe, 253
 - metody użytkowe, 487
 - parametry zmienne, 461
 - typy, 33
 - typy listowe, 451
 - wzorce rekordów, 449
- GraalVM, 39
- graf modułu, 703, 722

H

- hash code, 504
- hermetyzacja, 138, 158
- hierarchia
 - dziedziczenia, inheritance hierarchy, 225
 - interfejsów, 324
 - klas JSON, 277
 - wyjątków, 386, 408
- historia Javy, 31
- Hotspot, 39
- HTML, 35

I

- identyfikacja klas, 139
- identyfikatory, 65
- IEEE 754, 60
- implementacja
 - interfejsu, 316, 318
 - kolejki, 483
- import
 - klas, 191
 - statyczny, 193

- indeksy, 81
- informacje o typach czasu wykonywania, 287
- inicjalizacja
 - na żądanie, 623
 - pól
 - domyślna, 177
 - jawna, 178
 - z podwójną klamrą, 370
 - zmiennych, 66
 - zmiennych składowych, 183
- inkrementacja, 74
- instalacja
 - pakietu JDK, 39
 - plików źródłowych, 41
- instrukcja, *Patrz także* klauzula
 - break, 101, 117
 - break z etykietą, 117
 - continue, 118
 - goto, 116
 - if, 103
 - if-else, 104
 - if-else if, 105
 - switch, 113, 282, 283
 - switch rozszerzona, 282
 - try, 400
- instrukcje
 - sterujące, 101
 - warunkowe, 102
- interfejs, interface, 316, 270, 309
 - ActionListener, 330, 345
 - AnnotatedElement, 679, 682
 - Annotation, 672, 674
 - AutoCloseable, 401
 - BlockingDeque<E>, 632
 - BlockingQueue<E>, 631
 - Callable<V>, 580, 582
 - Cloneable, 336, 337
 - Collection, 327, 484, 491, 500
 - Comparable, 317, 322, 442
 - Comparator, 332, 357
 - Condition, 608, 612
 - Deque, 512
 - Enumerable, 481
 - Enumeration, 486, 552
 - ExecutorService, 585, 586
 - Formattable, 97
- funkcyjny
 - BiConsumer<T, U>, 355
 - BiFunction<T, U, R>, 345, 355
 - BinaryOperator<T>, 355
 - BiPredicate<T, U>, 355
 - BooleanSupplier, 356
 - Consumer<T>, 355
 - Function<T, R>, 355
 - ObjPCConsumer<T>, 356
 - PBinaryOperator, 356
 - PConsumer, 356
 - PFunction<T>, 356
 - PPredicate, 356
 - Predicate<T>, 346, 355
 - PSupplier, 356
 - PToQFunction, 356
 - PUnaryOperator, 356
 - Runnable, 355, 563
 - Supplier<T>, 346, 355
 - ToPBiFunction<T, U>, 356
 - ToPFunction<T>, 356
 - UnaryOperator<T>, 355
- Future<V>, 580–582, 656
- GenericArrayType, 472
- Instrumentation, 695
- InvocationHandler, 377, 382
- Iterable, 124, 485
- Iterator, 484
- List, 492, 504
- List<E>, 536, 539
- List<T>, 450
- ListIterator, 497
- Lock, 612
- Map, 491, 515
- Map<K, V>, 517, 536
- NavigableMap<K, V>, 493, 539
- NavigableSet<E>, 493, 510, 511, 539
- ParameterizedType, 472
- ProcessHandle, 661, 664
- ProcessHandle.Info, 665
- Queue<E>, 482, 512
- RandomAccess, 492
- RandomGenerator, 182, 184
- SequencedCollection<E>, 512, 539
- SequencedMap<E>, 540
- SequencedSet<E>, 540
- Set<E>, 493, 536
- SortedMap<K, V>, 493, 519, 539
- SortedSet<E>, 493, 539
- Thread.Builder, 579
- Thread.UncaughtExceptionHandler, 576, 577
- ThreadFactory, 579

- TransferQueue<E>, 628, 632
- Type, 472, 473
- TypeVariable, 472
- WildcardType, 472
- interfejsy
 - dziedziczenie, 324
 - funkcyjne, 344, 345
 - metoda abstrakcyjna, 344
 - wrażenia lambda, 344
 - hierarchia, 324
 - implementacja, 316, 318
 - Java Collections Framework, 491
 - klasy abstrakcyjne, 325
 - kolekcyjne, 541
 - konflikt nazw, 328
 - metody, 324
 - domyślne, 326, 328
 - prywatne, 326
 - statyczne, 326
 - programowania aplikacji, API, 20, 683
 - sprzężenie zwrotne, 330
 - użytkownika
 - wywołania zwrotne, 651
 - własności, 323
 - zapięczętowane, 325
 - zmienne, 324
 - znacznikowe, tagging interface, 337
- internet, 30
- interpolator STR, 193
- interpreter, 28
- ISO 8859-1, 62
- ISO 8859-2, 62
- iterator zbioru HashSet, 506
- iteratory, 484
 - słabo spójne, 633
 - do metod, 207
 - do pól, 208
 - ogólne, 209, 211
- znacznik
 - @author, 211
 - @deprecated, 211
 - @link, 210
 - @param, 207
 - @return, 208
 - @see, 210
 - @snippet, 212
 - @throws, 208
 - @version, 211
- znaczniki
 - blokowe, 207
 - dokumentacyjne, 206
 - śródliniowe, 207
- JDK, 433
- język
 - C#, 30
 - C++, 25
 - HTML, 35
 - J++, 30
 - Java, 15, 23
 - JavaScript, 37
 - programowania, 23
 - UML, 140
- języki
 - interpretowane, 36
 - programowania, 35
 - ściśle typizowane, 37
- JIT, just-in-time, 27
- JShell, 49
- JVM, 203

J

- JAR, Java archive, 198
- Java, 15, 23
 - Development Kit, JDK, 38
 - instalacja pakietu, 39
 - pobieranie pakietu, 38
 - Runtime Environment, JRE, 39
- javadoc, 213
 - generowanie dokumentacji, 212
 - kod HTML, 209
 - komentarze, 206
 - do klas, 207

K

- kalendarz, 146, 149
- catalog bazowy drzewa pakietu, 198
- klasa, 56, 137, 698, *Patrz także* kolekcja
 - AbstractCollection, 327, 488, 500
 - AccessibleObject, 301, 305
 - ActionListener, 367
 - Array, 306, 309
 - ArrayBlockingQueue, 631
 - ArrayDeque<E>, 514
 - ArrayList, 253, 256, 259, 494
 - Arrays, 127, 129
 - AssertionError, 411

klasa

BigDecimal, 119, 121
BigInteger, 119, 121
Boolean, 557
Class, 287, 290, 292, 298
Class<T>, 470, 472, 479
ClassLoader, 414
Collection<E>, 489
CompletableFuture<T>, 644, 646
Console, 96
ConsoleHandler, 421, 427
Constructor, 290, 294, 299, 310
DataFlavor, 699
Date, 146, 160
Dictionary, 448
EntryLogger, 695
Enum, 274
EOFException, 390
Error, 387
Exception, 387, 404
Executable, 310
ExecutorCompletionService, 592
Executors, 583, 585
Field, 294, 299
FileHandler, 421, 427
Filter, 428
Formatter, 429
FutureTask<V>, 583
GenericArrayType, 480
GregorianCalendar, 146
Handler, 427
Integer, 262, 557
InvocationHandler, 382
Iterator<E>, 490
java.util.Timer, 331
javax.swing.Timer, 331
JOptionPane, 332, 702
JSONArray, 274
JSONPrimitive<T>, 277, 469
JSONValue, 276
LinkedBlockingQueue<E>, 631
LocalDate, 144–147, 149
Lock, 601–603, 608
LogRecord, 427
Long, 557
LongAccumulator, 622
LongAdder, 622
Math, 51, 70, 71
Method, 294, 299, 310, 479

Modifier, 300
NumberFormat, 264
Object, 138, 238, 382, 615
Pair<T>, 459, 467, 472
ParameterizedType, 480
Path, 101
PrintStream, 248
PrintWriter, 100, 101
PriorityBlockingQueue<E>, 631
Process, 657, 663
ProcessBuilder, 657, 662
Properties, 553
Provider<S>, 376
Proxy, 377, 382, 383
Random, 182, 185, 625
RandomGenerator, 182
RecordComponent, 300
ReentrantLock, 601, 604
RuntimeException, 387, 404
Scanner, 93, 95, 100
ServiceLoader<S>, 376, 720
StackTraceElement, 406
StackWalker, 402, 405
String, 79, 85, 88, 333
StringBuilder, 89, 90
SwingWorker, 655, 657
System, 556
Thread, 565, 575, 577
ThreadGroup, 577
ThreadLocal<T>, 594, 625
Throwable, 290, 386, 392, 402, 404
TimePrinter, 361, 366
Toolkit, 332
TypeLiteral<T>, 476
TypeVariable, 479
Vector, 254
WildcardType, 480

klasy

abstrakcyjne, 265, 325
 interfejs, 325
agregacja, 140
bazowe, 218
definiowanie, 150
diagramy, 141
do blokowania, 601
dodawanie do pakietu, 193
dziedziczenie, 138, 140, 217
egzemplarz, 137
finalne, 229

- generyczne, 253, 437
 - definicja, 438
 - dziedziczenie, 450
 - tworzenie egzemplarza, 439
- identyfikacja, 139
- konstruktory, 142, 154
- macierzyste, 218
- metody, 138
 - prywatne, 161
 - statyczne, 164
- nazwy, 54
- niezmienne, immutable classes, 162
- opakowujące, 260
 - stałe, 262
- plik źródłowy, 199
- pochodne, 218
- poła statyczne, 163
- pośredniczące, proxy classes, 377
 - obiekty obsługujące wywołanie, 377
 - obiekty pośredniczące, 377
 - przesłanie metod, 382
 - właściwości, 381
- potomne, 218
- powiązania, 140
- predefiniowane, 141
- projektowanie, 213
- publiczne, 191
- relacje między klasami, 140
- rozszerzanie, 138
- stałe pola, 67, 162
- statyczne, static classes, 371
- wewnętrzne
 - anonimowe, 367
 - dostęp do stanu obiektu, 359
 - dostęp do zmiennych finalnych, 366
 - lokalne, 365
 - metody, 359
 - referencja do klasy zewnętrznej, 362
 - referencja do obiektu zewnętrznego, 360
 - reguły składniowe, 362
 - składnia, 359
 - składowe statyczne, 363
 - statyczne, 371
 - zastosowanie, 363
- wewnętrzne, inner classes, 316, 358, 371
- wyliczeniowe, 270
- z systemu kolekcji, 495
- zagnieżdżone, nested classes, 359, 371
- zależność, 140
- zapięczętowane, 274
- zmienne, mutable classes, 162
- klasyfikacja wyjątków, 386
- kłauzula
 - default, 76
 - finally, 398, 400
 - import, 191
 - null, 76, 281
 - permits, 276
 - throws, 100, 291
- klonowanie obiektów, 334
- kod
 - bajtowy, 689
 - modyfikacja podczas ładowania, 695
 - wyjścia, exit code, 56
 - źródłowy, 54
- kodowanie
 - ASCII, 62
 - ISO-8859-1, 62, 554
 - ISO-8859-2, 62
 - Unicode, 61, 62
 - UTF-16, 63, 81
 - UTF-32, 63
 - UTF-8, 79, 554
- kolejka, queue, 482, 512, 632
 - blokująca, blocking queue, 626
 - ArrayBlockingQueue, 628, 631
 - BlockingDeque, 632
 - BlockingQueue, 631
 - LinkedBlockingDeque, 628
 - LinkedBlockingQueue, 628, 631
 - metody, 627
 - PriorityBlockingQueue, 628, 631
 - dwukierunkowa, deque, 512
 - priorytetowa, 514
 - harmonogram zadań, 514
 - TransferQueue, 632
- kolekcja, 124
 - ArrayDeque, 482, 484, 494
 - ArrayList, 494, 504, 641
 - Arrays, 538
 - BitSet, 481, 558
 - ConcurrentHashMap, 632–634, 640
 - ConcurrentLinkedQueue, 632, 633
 - ConcurrentSkipListMap, 632, 634
 - ConcurrentSkipListSet, 632, 634
 - CopyOnWriteArrayList, 640
 - CopyOnWriteArraySet, 640
 - Deque, 512, 513

kolekcja

- Enumeration, 552
- EnumMap, 494, 525, 528
- EnumSet, 494, 525, 527
- HashMap, 494, 515, 518, 641
- HashSet, 485, 494, 504, 506, 508
- Hashtable, 481, 552, 641
- IdentityHashMap, 494, 526, 528
- LinkedHashMap, 494, 524, 527
- LinkedHashSet, 494, 524, 527
- LinkedList, 482, 484, 494, 500
- List, 502, 536, 539
- Map, 491, 517, 520, 536
- NavigableMap, 493, 539
- NavigableSet, 493, 511, 539
- PriorityQueue, 494, 514
- Properties, 553
- Queue, 482, 512
- SequencedCollection, 513, 539
- SequencedMap, 540
- SequencedSet, 540
- Set, 493, 536
- SortedMap, 493, 519, 539
- SortedSet, 493, 511, 539
- Stack, 481, 557
- TreeMap, 494, 515, 518
- TreeSet, 494, 508
- Vector, 481, 504, 641
- WeakHashMap, 494, 523, 527

kolekcje, 481

- architektura, 481
- bezpieczne wątkowo, 626, 641
- dostęp swobodny, 492
- implementacja algorytmów, 540
- interfejsy, 491
- konwersja tablic na kolekcje, 549
- małe, 529
- niemodyfikowalne
 - kopie, 531
 - obiekty, 530
 - widoki, 531
- obiekty opakowujące, 528
- operacje opcjonalne, 535
- operacje zbiorcze, 548
- przedziały, 532
- słabo spójne iteratory, 633
- sortowanie, 542
- stare, 551

- tablice kopiowane przy zapisie, 640

- tasowanie, 542

- uporządkowane, 491, 497

- warstwa interfejsów, 482

- warstwa klas konkretnych, 482

- widoki

- kontrolowane, 534

- odwrócone, 534

- synchronizowane, 535

- wyszukiwanie binarne, 544

- kolizja nazw klas, 190

komentarze

- bloku programu, /* */, 57

- do klas, 207

- do końca wiersza, //, 57

- do metod, 207

- do pakietów, 209

- do pól, 208

- dokumentacyjne, /** */, 57, 206

- ogólne, 211

- komórki, 505

- komparatory, 333, 334

- tworzenie, 357

kompilator

- javac, 199

- JIT, 27, 29, 39

- opcja Xlint, 114

- z symbolem wieloznacznym, 153

- komponenty rekordu, 186

- konkatenacja, 79

- konstrukcje switch, 115

- konstruktor, 142, 154, 155, 166

- bezargumentowy, 177

- domyślny, 177, 183

- kanoniczny, 188

- klasy wyliczeniowej, 273

- kompaktowy, 188

- niestandardowy, 188

- podklas, 221

- przeciążanie, 178

- referencje, 350

- wątków, 579

- wirtualny, 290

konwersja

- łańcucha na liczbę, 262

- tablic na kolekcje, 549

- typów numerycznych, 71

kopiowanie
 głębokie, 335
 obiektów, 335
 płytkie, 336
 tablicy, 125
 kowariantne typy zwracane, 228
 kubelki, 633
 kwalifikator .this, 363

L

lambda, 340
 leniwa ewaluacja, 346
 licencja GPL, 36
 liczby
 całkowite, 58
 format zapisu, 59
 losowe, 182
 zaokrąglanie, 72
 zmiennoprzecinkowe, 60
 lista, 491, 494
 cykliczna, 482, 494
 dwukierunkowa, doubly linked list, 495
 powiązana, 482, 492, 494, 497
 dodawanie elementów, 497, 498
 ListIterator, 497
 usuwanie elementów, 496
 powiązana, linked list, 495
 tablicowa, 504
 deklarowanie, 253
 dostęp do elementów, 256
 generyczna, 253, 256
 surowa, 259
 typowana, 259
 uporządkowana, 494
 właściwości, 543
 literały
 typowe, 475
 typu char, 61
 lokalizacja, 99
 lokalne klasy wewnętrzne, 365
 LTS, Long Term Support, 34, 39

Ł

ładowanie
 usług, 374
 wtyczek, 374
 łańcuch dziedziczenia, inheritance chain, 225

łańcuchy, 63
 dzielenie, 80
 identyczność, 83
 klasa String, 79
 klasa StringBuilder, 89
 konkatenacja, 79
 niezmienialność, 82
 null, 84
 porównywanie, 83, 333
 puste, 84
 składanie, 82, 89
 współdzielenie, 82
 wyjątków, 397
 wzorce podziału, 80

M

manifest, 202
 maszyna wirtualna Javy, 27, 432, 443
 informacje o typach generycznych, 472
 mechanizm
 ładowania klas, 378
 ładowania wtyczek, 374
 metaadnotacje, 672, 677
 metoda
 acceptEither, 647
 accessFlags, 300
 accumulateAndGet, 622
 acquire, 584
 actionPerformed, 330, 366, 367
 add, 120, 258, 489, 497, 502, 627
 addAll, 489, 502, 547, 549
 addFirst, 503, 513
 addLast, 503, 513
 addSuppressed, 404
 allOf, 527
 allProcesses, 664
 and, 559
 andNot, 559
 annotationType, 674
 append, 90, 91
 appendCodePoint, 90, 91
 applyToEither, 647
 Array.getLength, 306
 Arrays.copyOf, 126
 Arrays.parallelSort, 640
 Arrays.setAll, 347
 Arrays.sort, 127, 333, 345, 543
 arrayType, 308

metoda

- asList, 538
- await, 605–608, 612
- awaitTermination, 585, 587
- beep, 332
- binarySearch, 379, 381, 544, 545
- call, 580, 582
- canAccess, 305
- cancel, 581, 582, 656
- cardinality, 558
- cast, 471
- ceiling, 511
- charAt, 81, 85
- checkedCollection, 537
- checkedList, 538
- checkedMap, 538
- checkedNavigableMap, 538
- checkedNavigableSet, 538
- checkedQueue, 538
- checkedSet, 538
- checkedSortedMap, 538
- checkedSortedSet, 538
- children, 664
- Class.getConstructor, 310
- Class.newInstance, 290
- clear, 490, 558, 559
- clearAssertionStatus, 414
- clone, 334, 336, 337
- close, 427, 585, 587
- Collection.forEach, 485
- Collections.unmodifiableList, 531
- command, 665
- commandLine, 665
- Comparator.reverseOrder, 542
- compare, 322
- compareTo, 85, 121, 273, 317, 322, 439
- comparing, 357
- completeOnTimeout, 646
- compute, 520
- computeIfAbsent, 521, 635
- computeIfPresent, 520, 635
- ConsoleHandler, 427
- Constructor.newInstance, 310
- contains, 489
- containsAll, 488, 489
- containsKey, 518
- containsValue, 518
- copy, 547
- copyOf, 129, 523, 531, 537
- copyOfRange, 129
- current, 626, 664
- currentThread, 575
- daemon, 580
- delete, 91
- descendants, 664
- description, 231
- destroy, 660, 664
- destroyForcibly, 660, 664
- directory, 662
- disjoint, 548
- divide, 121
- doInBackground, 655
- element, 513, 627
- emptyEnumeration, 538
- emptyIterator, 538
- emptyList, 538
- emptyListIterator, 538
- emptyMap, 538
- emptyNavigableMap, 538
- emptyNavigableSet, 538
- emptySet, 538
- emptySortedMap, 538
- emptySortedSet, 538
- endsWith, 86
- ensureCapacity, 255
- entry, 536
- entrySet, 521, 522
- enumeration, 553
- environment, 663
- equals, 83–85, 130, 238, 243, 674
- equalsIgnoreCase, 83, 85
- exceptionally, 646
- exceptionNow, 581, 582
- execute, 657
- exitValue, 664
- factory, 579
- fill, 129, 547
- finalize, 185
- findFirst, 376
- first, 511
- firstKey, 519
- flipDone, 620
- floor, 512
- flush, 427
- forEach, 518, 638
- forEachRemaining, 484
- format, 421, 429
- formatMessage, 422, 429

formatted, 98
formatTo, 97
forName, 290
frequency, 548
generator.nextInt, 182
get, 256, 258, 305, 491, 517, 558, 581, 582
getAccessor, 300
getActualTypeArguments, 480
getAnnotation, 679–682
getAnnotations, 679–682
getAnnotationsByType, 679, 682
getBoolean, 557
getBounds, 480
getCause, 404
getClass, 239, 287, 289
getClassName, 405, 406
getComponentType, 308
getConstructor, 289, 471
getConstructors, 294, 299
getDay, 146
getDayOfMonth, 145, 149
getDayOfWeek, 149
getDeclaredAnnotation, 679
getDeclaredAnnotations, 679, 682
getDeclaredAnnotationsByType, 679
getDeclaredConstructor, 471
getDeclaredConstructors, 294, 299
getDeclaredField, 305
getDeclaredFields, 294, 298, 301, 302
getDeclaredMethods, 294, 299, 310
getDeclaringClass, 299, 405
getDefaultToolkit, 332
getDefaultUncaughtExceptionHandler, 577
getEnumConstants, 471
getErrorStream, 664
getExceptionTypes, 299
getField, 305, 310
getFields, 294, 298, 305
getFileName, 405, 406
getFilter, 427
getFirst, 504, 513
getFormatter, 427
getGenericComponentType, 480
getGenericInterfaces, 479
getGenericParameterTypes, 479
getGenericReturnType, 479
getGenericSuperclass, 479
getHead, 422, 429
getInputStream, 663
getInstance, 405
getInstant, 428
getInteger, 557
getKey, 523
getLast, 504, 513
getLength, 309
getLevel, 427
getLineNumber, 405, 406
getLoggerName, 428
getLong, 557
getLongThreadID, 428
getLowerBounds, 480
getMessage, 392, 428
getMethod, 310
getMethodName, 405, 406
getMethods, 294, 299
getMillis, 428
getModifiers, 294, 299
getMonth, 146
getMonthValue, 145, 149
getName, 289, 300, 426, 479
getOrDefault, 517
getOutputStream, 663
getOwnerType, 480
getPackageName, 299
getParameters, 428
getParameterTypes, 300
getProperties, 556
getProperty, 553–555
getProxyClass, 382, 383
getRawType, 480
getRecordComponents, 299
getResource, 293
getResourceBundle, 428
getResourceBundleName, 428
getReturnType, 300
getSequenceNumber, 428
getSourceClassName, 428
getSourceMethodName, 428
getStackTrace, 403, 404
getState, 657
getSuperclass, 252, 471
getSuppressed, 404
getTail, 422, 429
getThrown, 428
getTime, 230
getTotalBalance, 603
getTypeParameters, 479
getUncaughtExceptionHandler, 577

metoda

- getUpperBounds, 480
- getValue, 523
- getXxx, 309
- getYear, 145, 146, 149
- GregorianCalendar.add, 147
- group, 580
- handle, 646
- hashCode, 186, 244, 246, 493, 504, 674
- HashSet, 508
- hasMoreElements, 486, 552
- hasNext, 96, 327, 484, 486, 498
- hasNextDouble, 96
- hasNextInt, 96
- hasPrevious, 497, 503
- headMap, 533
- headSet, 532
- higher, 511
- identityHashCode, 252
- identityToString, 252
- in.close, 401
- incrementAndGet, 621
- indexOf, 86, 176, 503
- indexOfSubList, 547
- info, 665
- inheritIO, 663
- initCause, 404
- insert, 91
- instanceof, 337
- interrupt, 572, 575
- interrupted, 574, 575
- intValue, 263
- invoke, 309, 312, 382
- invokeAll, 587, 592, 595
- invokeAny, 587, 591, 592
- isAbstract, 300
- isAlive, 664
- isAnnotationPresent, 679, 682
- isBlank, 85
- isCancelled, 581, 582
- isDone, 581, 583, 620
- isEmpty, 85, 489
- isFinal, 294, 300
- isInterface, 300
- isInterrupted, 572–575
- isLoggable, 421, 426, 428
- isNative, 300
- isNativeMethod, 405, 406
- isPrivate, 294, 300
- isProtected, 300
- isProxyClass, 382, 383
- isPublic, 294, 300
- isStatic, 300
- isStrict, 300
- isSynchronized, 300
- isVirtual, 572
- isVolatile, 300
- iterator, 375, 484, 488, 489
- Iterator.forEachRemaining, 485
- join, 86
- keySet, 523, 528
- lastIndexOf, 86, 503
- lastIndexOfSubList, 547
- lastKey, 519
- length, 81, 85, 90, 124, 558
- LinkedList.add, 497
- list, 553
- List.of, 549
- listIterator, 498, 502
- load, 376, 554, 555
- lock, 601, 603
- log, 426
- lower, 511
- main, 54, 166
- Math.random, 129
- Math.round, 72
- max, 547
- merge, 520
- min, 547
- minusDays, 149
- mod, 121
- Modifier.toString, 294
- Module.getResourceAsStream, 712
- multiply, 120, 121
- name, 271, 274, 579
- nCopies, 538
- newCachedThreadPool, 583–586
- newCondition, 608
- newFixedThreadPool, 583–586
- newHashMap, 518
- newInstance, 290, 306, 309
- newKeySet, 533, 639
- newProxyInstance, 377, 382, 383
- newSetFromMap, 533, 537
- newSingleThreadExecutor, 583–586
- newThread, 579
- newThreadPoolExecutor, 586
- newVirtualThreadPoolExecutor, 583, 586

- next, 95, 327, 484, 486, 490
- nextDouble, 94, 95
- nextElement, 486, 552
- nextIndex, 500, 503
- nextInt, 94, 95
- nextLine, 93, 95
- noneOf, 527
- notify, 612, 615, 618
- notifyAll, 612, 615, 618
- now, 149
- Object.clone, 337, 447
- of, 149, 528, 536, 664
- ofEntries, 536
- offer, 512, 627, 631
- offerFirst, 513, 632
- offerLast, 513, 632
- ofPlatform, 579
- ofVirtual, 579
- onExit, 664
- or, 559
- ordinal, 272, 274
- orTimeout, 646
- paintComponent, 390
- parallelMultiply, 120
- parallelSetAll, 641
- parse, 264
- parseInt, 262, 263
- peek, 513, 558, 627
- peekFirst, 513
- peekLast, 513
- pid, 665
- plusDays, 149
- poll, 512, 592, 627, 632
- pollFirst, 512, 513, 632
- pollLast, 512, 513, 632
- pop, 557
- pow, 121
- premain, 695
- previous, 497, 500, 503
- previousIndex, 500, 503
- print, 57
- printf, 97, 98, 264
- println, 57, 381
- printStackTrace, 290, 402, 430, 431
- priority, 580
- process, 656, 657
- publish, 427, 657
- push, 557
- put, 491, 518, 627, 631
- putAll, 518
- putFirst, 632
- putIfAbsent, 521
- putLast, 632
- range, 527
- readLine, 96
- readPassword, 96
- redirectError, 659, 663
- redirectErrorStream, 659, 663
- redirectInput, 659, 663
- redirectOutput, 659, 663
- remove, 258, 484, 487, 490, 499, 502, 627
- removeAll, 488, 490
- removeEldestEntry, 525, 527
- removeFirst, 504, 513
- removeIf, 346, 490, 546, 548
- removeLast, 504, 513
- repeat, 87, 91
- replace, 86
- replaceAll, 521, 546–548
- requireNonNull, 169
- requireNonNullElse, 169
- requireNonNullElseGet, 169, 346
- res.close, 401
- resultNow, 581, 582
- retainAll, 488, 490, 548
- reverse, 91, 547
- reversed, 534, 539, 544
- rotate, 548
- run, 573
- runAfterBoth, 647
- runAfterEither, 647
- search, 638
- searchEntries, 638
- searchKeys, 638
- searchValues, 638
- set, 256, 258, 305, 309, 499, 502, 558
- setAccessible, 301, 305
- setClassAssertionStatus, 414
- setDaemon, 575
- setDefaultAssertionStatus, 414
- setDefaultUncaughtExceptionHandler, 576, 577
- setDone, 620
- setFilter, 427
- setFormatter, 427
- setLabelTable, 448
- setLevel, 427
- setName, 576
- setPackageAssertionStatus, 414

metoda

- setPriority, 578
- setProperty, 555
- setTime, 230
- setUncaughtExceptionHandler, 576, 577
- setValue, 523
- showMessageDialog, 332
- shuffle, 543, 544
- shutdown, 585, 587
- shutdownNow, 585, 587
- signal, 607, 609, 612
- signalAll, 605–609, 612
- singleton, 538
- singletonList, 538
- size, 255, 488, 489, 559
- sleep, 567
- sort, 129, 320, 341, 542, 544
- split, 80, 87
- sqrt, 121, 311
- stackSize, 580
- staff.toArray, 549
- start, 565, 580, 663
- startInstant, 665
- startPipeline, 663
- startsWith, 86
- startVirtualThread, 572
- state, 581–583
- static allOf, 647
- static anyOf, 647
- stop, 332
- store, 553, 555
- stream, 376, 559
- String.format, 98
- stringPropertyNames, 555
- strip, 86
- stripLeading, 86
- stripTrailing, 86
- subList, 532
- subMap, 533
- submit, 585, 586, 592
- subSet, 532
- substring, 86
- subtract, 121
- supportsNormalTermination, 664
- swap, 456, 547
- synchronizedCollection, 537
- synchronizedList, 537
- synchronizedMap, 535, 537
- synchronizedNavigableMap, 537
- synchronizedNavigableSet, 537
- synchronizedSet, 537
- synchronizedSortedMap, 537
- synchronizedSortedSet, 537
- System.getProperties, 555
- System.out.println, 93
- tailMap, 533
- tailSet, 532
- take, 592, 627, 631
- takeFirst, 632
- takeLast, 632
- thenAccept, 646
- thenAcceptBoth, 647
- thenApply, 646
- thenCombine, 647
- thenComparing, 357
- thenCompose, 646
- thenRun, 646
- Thread.currentThread, 572
- Thread.ofPlatform, 579
- Thread.ofVirtual, 579
- threadId, 576
- ThreadLocalRandom.current, 625
- toArray, 490
- toHandle, 664
- toLowerCase, 86
- toString, 90, 97, 129, 186, 247, 405, 406, 674
- totalCpuDuration, 665
- toUpperCase, 86
- transfer, 603, 632
- TreeSet, 511
- trimToSize, 255
- tryTransfer, 632
- type, 276
- uncaughtException, 577
- uncaughtExceptionHandler, 580
- unlock, 602, 603
- unmodifiableCollection, 532, 537
- unmodifiableList, 537
- unmodifiableNavigableMap, 537
- unmodifiableNavigableSet, 537
- unmodifiableSequencedCollection, 537
- unmodifiableSequencedMap, 537
- unmodifiableSequencedSet, 537
- unmodifiableSet, 537
- unmodifiableSortedSet, 537
- unstarted, 580
- user, 665
- valueOf, 121, 264, 274

- values, 272, 523
- wait, 612, 616, 618
- waitFor, 664
- walk, 405
- whenComplete, 646
- xor, 559
- metody, 138
 - abstrakcyjne, 266, 317, 344
 - akcesory, 147
 - asynchroniczne, 580
 - ciało, 56
 - domyślne, 326, 328
 - dostęp do prywatnych składowych, 161
 - dostępu do pól, field accessor, 159
 - fabryczne, factory methods, 145, 166, 583, 585
 - finalne, 229
 - generyczne, 440, 472, 487
 - definiowanie, 440
 - wnioskowanie o typie, 440
 - interfejsu, 324
 - AnnotatedElement, 679, 682
 - Collection<E>, 487
 - Deque<E>, 513
 - Future<V>, 581
 - ListIterator<E>, 503
 - Map, 517
 - Map.Entry<K, V>, 523
 - Map<K, V>, 520, 522
 - NavigableSet, 511
 - ProcessHandle, 664
 - ProcessHandle.Info, 665
 - Queue, 512
 - SequencedCollection<E>, 513
- klas pośredniczących, 382
- klasy
 - Array, 309
 - ArrayList, 258
 - Arrays, 129
 - BigDecimal, 121
 - BigInteger, 121
 - BitSet, 558
 - Class, 298, 305
 - Class<T>, 471
 - Collection<E>, 489
 - CompletableFuture, 646
 - Executors, 583, 585, 586
 - Field, 305
 - HashSet, 508
 - LinkedList, 500
 - LinkedList<E>, 503
 - List<E>, 502
 - LocalDate, 149
 - LogRecord, 427
 - Math, 51, 70, 71
 - Modifier, 300
 - Object, 382, 615
 - Objects, 409
 - Process, 663
 - ProcessBuilder, 662
 - Properties, 555
 - Scanner, 95
 - Stack, 557
 - StackTraceElement, 406
 - String, 85, 88
 - StringBuilder, 90
 - Thread, 576–578
- kolejek blokujących, 627
- mutatory, 147
- odradzane, 146
- opakowujące, 531, 532
- parametry, 169, 173
- pomostowe, 446
- prywatne, 161, 326
- przeciążanie, 176
- przesłanianie, 219
- referencje, 347, 349
- refleksyjne, 679
- rekordu, 186
- rodzime, 164
- statyczne, 70, 164, 165, 326
- sygnatura, 176, 228
- synchronizowane, 612, 613
- udostępniające wartość elementu, 146
- wywołanie na rzecz obiektu, 142
- zasady wywoływania, 227
- zmieniające wartość elementu, 146
- zmienna liczba argumentów, 264
- zmienna parametryczna, 85
- zwrotne, 643
- Micro Edition, ME, 39
- miejsce wywołania, inlining, 232
- modularne pliki JAR, 708
- moduły, 34, 698
 - agregujące, 718
 - automatyczne, automatic modules, 713
 - dodawanie w Eclipse, 708
 - eksport kwalifikowany, 719
 - eksportowanie pakietów, 704

moduły
 graf, 703, 723
 implementacja usługi, 721
 mechanizm refleksji, 710
 modularna wersja programu, 701
 modularne pliki JAR, 708
 nadawanie nazw, 700
 narzędzia, 722
 nienazwane, unnamed modules, 715
 otwieranie, 719
 warstwa startowa, boot layer, 709
 wymagania, 717
 żądanie użycia, 702
modyfikacja
 klas podczas ładowania, 696
 kodu bajtowego, 695
 parametru obiektowego, 172
modyfikator, 53, 237
 default, 326
 final, 621
 private, 196
 public, 196
 transitive, 718
 volatile, 620
monitor, 618
mutator, 147

N

nadklasa, superclass, 217, 218
nadtypy
 ograniczenia, 453
 typów wieloznacznych, 453
NaN, Not a Number, 60, 69
narzędzia wiersza poleceń, 42, 433
narzędzie
 jar, 201
 javadoc, 206
 jdeps, 722
 jlink, 724
 jmod, 724
 JUnit, 430
 VisualVM, 432
nawiasy
 klamrowe, 54, 55
 ostre, 253
nazwy
 klas, 43, 54, 215
 metod, 215

 pakietów, 190
 parametrów, 179
 plików, 43
nieskończoność, 60
niezależność od architektury, 27
niezawodność, 26
niszczenie obiektów, 185
notacja wielbłądzia, CamelCase, 54

O

obiekt, 58
 Callable, 580, 582, 588
 CompletableFuture, 645
 Condition, 613
 Console, 95
 ExecutorCompletionService, 588
 Future, 581, 582
 Lock, 601, 613
 nasłuchujący, listener object, 331
 Scanner, 100
 System.out, 57
obiektość, 25
obiekty, 138
 autoboxing, 260
 bezpiecznie opublikowane, 624
 hermetyzacja, 138
 klonowanie, 334
 kopiowanie, 335
 metody, 138
 niszczenie, 185
 obsługujące wywołanie, 377
 opakowujące, 528
 polimorfizm, 222
 porównywanie, 238
 pośredniczące, 377
 przekazanie do metody, 142
 składowe, 138
 stan, 138
 synchronizacyjne opakowujące, 641
 tożsamość, 138
 tworzenie, 89, 142, 175
 ustawienie wartości składowej, 159
 warunków, 604
 wielokrotnego użytku, 142
 właściwości, 138
 zachowanie, 138
obietnica, 644
obliczenia asynchroniczne, 642

obsługa

- błędów, 385
- dzienników aplikacji, 415
- wyjątków, 288, 384, 406
- wyjątków nieprzechwyconych, 576
- zdarzeń, 330

obszar surogatów, surrogates area, 63

odczyt

- danych, 93
- z pliku, 99

opakowywanie, 261

OOP, Object Oriented Programming, 136

opakowania obiektów, 260

OpenJ9, 39

OpenJDK, 39

operacje

- masowe, 637
- niepodzielne, 600, 621
- równoległe, 640
- zbiorcze, 548

operator

- /, 69
- ::, 348
- [], 124, 126
- alternatywy, ||, 74
- bitowa alternatywa, |, 77
- bitowa koniunkcja, &, 77
- bitowa negacja, ~, 77
- bitowe lub wykluczające, ^, 77
- bitowe przesunięcie z zerami, >>>, 77
- bitowe przesunięcie, >>, <<, 77
- dekrementacji, 74
- dzielenia modulo, %, 69
- inkrementacji, 74
- instanceof, 233, 234, 280, 324
- koniunkcji, &&, 75
- new, 122, 142, 143, 253
- nierówności, !=, 74
- przypisania złożony, 73
- równości, ==, 74, 289
- trójargumentowy, ?:, 75

operatory

- arytmetyczne, 69, 71
- bitowe, 77
- logiczne, 74
- priorytety, 78
- przeciążanie, 120
- relacyjne, 74

opóźnione wykonywanie procedur, 354

P

pakiet, 699

- Java Development Kit, 15
- java.lang, 191
- java.lang.reflect, 294, 472
- java.util, 95
- java.util.concurrent, 601, 613, 628
- java.util.concurrent.atomic, 621
- java.util.function, 345
- java.util.logging, 418
- javax.swing, 330

pakiety, packages, 190

- dodawanie klasy, 193
- katalog bazowy, 198
- nienazwane, 194

para klucz – wartość, 515, 521

parametry, 169

- jawne, 157
- nazwy, 179
- niejawne, 157
- odbiorcze, 671
- sprawdzanie, 412
- typów, 436
- wiersza poleceń, 126

pętla

- do-while, 107, 109
- for, 109, 111
- REPL, 49
- typu for each, 124, 131
- while, 105, 111

pętle

- instrukcja break, 117
- instrukcja continue, 118
- liczba iteracji, 109
- przerywanie działania, 117

pieczętowanie klas, 275

pierwszy program, 42, 53

platforma programistyczna, 23

pliki, 99

- .java, 54
- JAR, 198, 201
- manifest, 202
- modularne, 708
- sekcja główna, 202
- wykonywalne, 202
- z wieloma wersjami klas, 203
- zmienianie manifestu, 202

- pliki
 - odczyt, 99
 - tekstowe, 101
 - zapis, 99
 - źródłowe, 41, 153, 199
- pobieranie danych, 93
- podklasa, subclass, 217, 218
- podłańcuchy, 81
- pola
 - finalne, 620
 - inicjalizacja domyślna, 177
 - inicjalizacja jawna, 178
 - klasowe, class field, 163
 - statyczne, 163, 181, 182
 - ulotne, 619
- polecenie
 - jar, 201
 - java NazwaKlasy, 54
 - javac, 42, 44
 - printf, 97
 - znaczniki, 98
 - znaki konwersji, 97
- polimorfizm, 222, 225, 315
- porównywanie
 - łańcuchów, 83, 333
 - obiektów, 238, 240
- powiązania między klasami, 140
- priorytety
 - operatorów, 78
 - wątków, 578
- procedura obsługi błędów, 386
- proces, 657
 - budowanie, 658
 - uchwyt, 661
 - uruchamianie, 660
- procesory adnotacji, 682, 685, 688
- procesy, 562
- program
 - jconsole, 433
 - JSHELL, 49
- programowanie
 - generyczne, generic programming, 435
 - dedukcja typów, 468
 - dziedziczenie, 449
 - egzemplarz generycznej tablicy, 463
 - egzemplarze zmiennych typowych, 462
 - generyczne parametry zmienne, 461
 - klasy generyczne, 437
 - konflikty, 467
 - maszyna wirtualna, 443
 - metody generyczne, 440
 - obiekty klasy generycznej, 465
 - ograniczenia, 458
 - ograniczenia zmiennych typowych, 441
 - parametry typów, 436
 - refleksja, 470
 - sprawdzanie typów w czasie działania programu, 459
 - sprawdzanie wyjątków kontrolowanych, 466
 - statyczny kontekst klas uogólnionych, 465
 - tablice typów generycznych, 459
 - translacja metod generycznych, 445
 - translacja wyrażeń generycznych, 445
 - typy proste jako parametry typowe, 459
 - typy surowe, 444
 - typy wieloznaczne, 437, 451
 - wyjątki, 465
 - zastosowanie, 437
 - zmienna liczba argumentów, 460
 - obiektove, 25, 136
 - proceduralne, 137
- programy
 - refleksyjne, 287
 - wielowątkowe, multithreaded, 562
- projektowanie klas, 213
- prywatne pola, 159
- przechwytywanie
 - wielu typów wyjątków, 395
 - wyjątków, 290, 392
- przeciążanie
 - konstruktorów, 178, 183
 - metod, 176
- przedziały, 532
- przekazywanie
 - przez wartość, 172
 - wyjątków, 409
- przenośność, 28
- przepływ sterowania, 101
- przesłanie
 - metod, 219, 229, 243
 - składowych, 179
- przestrzenie numeracyjne, code planes, 63
- przypisanie, 73
- przywileje klasowe, 161
- publiczne
 - metody akcesora, 159
 - metody mutatora, 159
- pula wątków, 583, 584
- kontrola grup zadań, 587

R

referencje
 do klasy zewnętrznej, 362
 do konstruktorów, 350
 do metod, 347, 349
 do obiektów, 172, 143
 do obiektu zewnętrznego, 360
 null, 156

refleksja, reflection, 217, 287, 315, 710
 analiza funkcjonalności klasy, 294
 analiza obiektów w czasie działania programu, 301
 generyczny kod tablicowy, 306
 informacje o typach generycznych, 472
 klasy, 287
 Constructor, 294
 Field, 294
 Method, 294
 nazwy pól, 301
 parametry Class<T>, 471
 typy pól, 301

rejestrujący obiekt pośredni, 430

rekordy, 186
 deklarowanie, 186
 komponenty, 186
 metody, 186

relacja typu „jest”, 217, 225, 313
 relacje między klasami, 140
 REPL, read-evaluate-print loop, 49
 rozstrzyganie przeciążania, 176, 227
 rozszerzanie klas, 138
 rzutowanie, casting, 72, 232, 234, 470

S

sieciowość, 26
 sito Eratostenesa, 559
 składanie łańcuchów, 82, 89
 składnia
 diamentowa, diamond syntax, 253
 Javy, 25
 wyrażeń lambda, 342
 składowe, 138, 159
 skrót, hash code, 244, 504
 słabe referencje, weak reference, 524

słowa kluczowe, 725–728
 słownik, 494
 skrótów, 633
 własności, property map, 553

słowniki, 632
 modyfikowanie atomowe, 634
 modyfikowanie wpisów, 519
 operacje masowe, 637
 podstawowe operacje, 515
 widoki, 521

słowo kluczowe
 abstract, 266
 assert, 410
 break, 114, 117
 case, 76
 catch, 393
 class, 53, 150
 const, 68
 continue, 118
 default, 76, 113
 else, 103
 enum, 68, 270
 exports, 701, 704, 706, 712
 extends, 218, 324
 final, 67, 162, 222, 229, 621
 finally, 398
 goto, 116
 implements, 318
 import, 95, 191
 instanceof, 233
 interface, 317
 module, 701
 new, 142
 non-sealed, 277
 opens, 711, 712, 720
 package, 193, 197
 permits, 276
 private, 154, 161, 236, 237
 protected, 206, 237
 public, 53, 54, 154, 237
 requires, 701, 706, 713
 sealed, 275
 static, 55, 67, 164, 165, 371
 strictfp, 69
 super, 220
 switch, 75, 112
 synchronized, 601, 611, 613
 this, 158, 165, 179, 180, 183, 220, 353, 363

słowo kluczowe
 throw, 390
 throws, 388
 transitive, 718
 try, 393
 var, 155, 253, 343, 369
 virtual, 222
 void, 55, 56, 288
 volatile, 619, 620
 when, 281
 yield, 115
sortowanie, 542
 tablicy, 127, 333
 za pomocą komparatora, 334
specyfikacja wyjątku, 389
specyfikator formatu, 97, 98
sprawdzanie zakresu, 126
stała
 Double.NaN, 61
 Double.NEGATIVE_INFINITY, 61
 Double.POSITIVE_INFINITY, 61
 Math.PI, 164
 MAX_VALUE, 262
 MIN_VALUE, 263
stałe, 67, 284
 klasowe, 67
 matematyczne, 70
 pola klasy, 162
 statyczne, 164
stan obiektu, 138
Standard Edition, SE, 39
stany wątków, 567
statyczne
 funkcje składowe, 56
 klasy wewnętrzne, 371
statyczny blok inicjalizujący, 183
sterta, heap, 126, 514
stos, stack, 126, 557
 wywołań, 402
stosowanie wyjątków, 406
strażnik, 281
struktura danych, *Patrz* kolekcja
strumień procesów, 659
strumień
 System.err, 431
 System.in, 93
sygnatura metody, 176, 228
symbole
 wieloznaczne, 153
 zastępcze znaków specjalnych, 61, 62

synchronizacja
 bloki synchronizowane, 616
 koszt, 607
 metody klasy Object, 615
 monitory, 618
 obiekty klasy lock, 601
 pola finalne, 620
 pola ulotne, 619
 słowo kluczowe synchronized, 611
 warunki, 604
 wątków, 597
 zakleszczenia, 609
 zmiennie atomowe, 621
system modułów platformy Javy, 698

Ś

ścieżka klas, 198
 ustawianie, 200
ściśła kontrola typów, 320
środowisko programistyczne, 38

T

tabela metod, 228
tablica
 skrótów, hash table, 244, 504
 skrótów powiązana, 524
tablice, 122
 anonimowe, 123
 deklaracja, 122
 dostęp do elementów, 124
 indeksy, 124
 kopiowanie, 125
 kopiowane przy zapisie, 640
 liczba elementów, 124
 list powiązanych, 505
 mieszające, 504
 kolizje, 505
 kubelki, 505
 reorganizacja, 506
 współczynnik wypełnienia, 506
operacje równoległe, 640
pętla typu for each, 125
postrzępione, 133
skrótów, 505
sortowanie, 127, 333
tworzenie, 122, 152
ustawianie rozmiaru, 253

- usuwanie elementu, 495
- wielowymiarowe, 130, 134
- tasowanie, 542
- Temurin, 39
- tożsamość obiektu, 138
- translacja
 - metod generycznych, 445
 - wyrażeń generycznych, 445
- tworzenie
 - algorytmów, 550
 - egzemplarza klasy, 137
 - klas wyjątków, 391
 - komparatorów, 357
 - konstruktorów, 154
 - obiektów, 89, 142, 143, 175, 645
 - obiektów pośredniczących, 377
 - plików JAR, 201
 - tablic, 122, 152
 - tablic postrzępionych, 133
 - wątków wirtualnych, 583
 - widoków słowników, 521
- typ danych, 58
 - byte, 58
 - char, 61, 62
 - double, 60
 - float, 60
 - int, 58
 - long, 58, 59
 - short, 58
 - string, 83
- typy
 - całkowite, 58
 - danych
 - boolean, 64
 - rzutowanie, 72
 - funkcji, 345
 - generyczne, 33
 - interfejsowe, 483
 - nieoznaczalne, 370
 - numeryczne, 59
 - podstawowe, 238
 - sparametryzowane, 33
 - surowe, 254, 444, 459
 - tablicowe, 238
 - wieloznaczne, 451
 - bez ograniczeń, 455
 - chwytanie, 456
 - ograniczenia nadtypów, 453

- wyliczeniowe, enumerated type, 68
- zmiennoprzecinkowe, 60
- zwracane kowariantne, 447

U

- uchwyty procesów, 661
- ukrywanie wyjątków, 408
- UML, Unified Modeling Language, 140, 141
- Unicode, 61, 62
- uruchamianie
 - aplikacji, 54
 - procesu, 660
- usługi, 720
 - moduły ładowania, 374
- usuwanie
 - błędów, 429
 - elementu z listy powiązanej, 496
 - elementu z tablicy, 495
 - nieużytków, Garbage Collector, 524
- UTC, Coordinated Universal Time, 144
- UTF-16, 63, 81
- UTF-32, 63
- UTF-8, 79

V

- varargs, 264

W

- wartości
 - logiczne, 64
 - zakresowe, scoped values, 625
- wartość
 - NaN, 69
 - null, 85, 156, 281, 282, 635
- warunek wstępny, precondition, 413
- warunki, 604
- wątek, thread, 562
 - stan
 - BLOCKED, 569
 - NEW, 568
 - RUNNABLE, 568
 - TERMINATED, 570
 - TIMED_WAITING, 570
 - WAITING, 569
 - sterowania, thread of control, 562

wątki

- bezpieczeństwo, 619
- blok synchronizowany, 616
- blokada, 601, 603, 608
 - wewnętrzna, 613
 - wielowejściowa, 603
- blokowanie po stronie klienta, 617
- demony, 575
- fabryki, 579
- identyfikatory, 576
- jednoczesny dostęp, 599
- kolejki blokujące, 626
- konstruktory, 579
- metody fabryczne, 583
- monitor, 618
- nazwy, 576
- nieprzechwycone, 576
- niesynchronizowane, 603
- nośnikowe, 572
- oczekujące, 569, 605
- planowanie kooperacyjne, 568
- planowanie wywłaszczające, 568
- platformy, 568, 571
- podkradanie pracy, work stealing, 597
- priorytety, 578
- procedury obsługi, 576
- przerywanie, 572
- pula, 583, 584
- stany, 567
- status przerwania, 572
- synchronizacja, 597, 603, 611
- tablice kopiowane przy zapisie, 640
- technika rozgałęzienie-złączenie, 594, 597
- tworzenie, 568
- uruchamianie, 563
- warunki, 604
- wirtualne, 568, 571, 583, 584, 613
- własności, 571
- współbieżność, 564
- współdzielenie zmiennych lokalnych, 625
- wykonywalne, 568
- wyścig, 597, 599
- wyzerowanie statusu przerwania, 574
- zablokowane, 569, 573
- zakleszczenia, 606, 609
- zamykanie, 569
- zatrzymanie wykonywania, 567
- zmienne lokalne, 593

wczytywanie usług, 720

wejście, 93

wersje języka Java, 34

wiązanie

- dynamiczne, dynamic binding, 222
- statyczne, static binding, 227

widok, view, 528

widoki

- kontrolowane, 534
- niemodyfikowalne, 531
- odwrócone, 534
- operacja zbiorcza, 549
- przedziały, 532
- słowników, 521
- synchronizowane, 535

wielkie liczby, 119

wielkość liter, 53

wielowątkowość, 29

wielozadaniowość, 562

wiersz poleceń, 42, 126

flagi, 716

opcje, 205

własności

interfejsów, 323

wątków, 571

właściwości systemowe, 555

współbieżność, 562

strukturalna, 591

współrzędna kodowa znaku, code point, 63

wybór wielokierunkowy, 112

wydajność, 29

wyjątek

- ArrayIndexOutOfBoundsException, 124, 387
- ArrayStoreException, 450, 460, 461
- CancellationException, 582, 656
- ClassCastException, 233, 24, 3061, 450, 460, 471, 534
- ConcurrentModificationException, 499, 633
- EmptyStackException, 409
- ExecutionException, 582, 587
- FileNotFoundException, 388, 390
- IllegalStateException, 487, 490, 503
- InterruptedException, 467, 564, 573, 581
- InvocationTargetException., 290
- IOException, 100, 390, 394
- MatchException, 281
- NoSuchElementException, 513
- NullPointerException, 76, 156, 231, 252, 281, 350, 409, 413, 519
- NumberFormatException, 408

ServletException, 397
 TimeoutException, 581
 UnsupportedOperationException, 530, 531, 535, 537
 wyjątki, 385
 analiza danych ze stosu wywołań, 402
 blok try-catch, 393
 blok try-finally, 400
 diagram hierarchii, 386
 hierarchia, 408
 IllegalAccessException, 301
 instrukcja try, 400
 klasa
 Error, 387
 Exception, 387
 RuntimeException, 387
 Throwable, 386
 klasyfikacja, 386
 klauzula finally, 398
 kontrolowane, checked exceptions, 288, 291, 387, 388, 389
 łańcuchy wyjątków, 397
 niekontrolowane, 291, 387
 powtórne generowanie, 397
 przechwytywanie, 290, 392, 395
 przekazywanie, 409
 specyfikacja, 389
 stosowanie, 406
 tworzenie klas wyjątków, 391
 wyciszanie, 408
 zgłaszanie, 390
 wyjście, 93
 wykonywalne pliki JAR, 202
 wyliczenia, 68, 270, 289, 552
 wyłączenie dziedziczenia, 229
 wyrażenia lambda, 309, 340
 interfejsy funkcyjne, 344
 przechwytywanie wartości, 352
 przetwarzanie, 354
 składnia, 342
 zmienna faktycznie finalna, 353
 wyrażenie, 73
 selektora, selector expression, 75
 switch, 112, 276, *Patrz także* konstrukcje switch
 dopasowywanie do wzorca, 280
 klauzula null, 281
 kompletność, 282
 kontrola dominacji, 283
 przekazywanie sterowania w dół, 284

 wzorce i stałe, 284
 ze strażnikiem, 281
 wyszukiwanie
 binarne, 544
 błędów, 429
 wyścig, race condition, 597, 599
 wywołania zwrotne, callback, 330
 wywołanie
 grup zadań, 587
 konstruktorów, 180, 183, 310
 metod, 309
 przez nazwę, 170
 przez referencję, 169
 przez wartość, 169
 wzorce, 280, 284
 numeryczne, 284
 rekordów, 236, 281, 284
 generyczne, 468, 449
 strzeżone, 282, 283
 typu, 236, 285

X

XML, 35

Z

zachowanie obiektu, 138
 zadania
 anulowanie, 581
 czasochłonne, 651
 koordynacja, 580
 współbieżne, 587
 wywoływanie jednoczesne, 587
 zadanie
 stan
 CANCELLED, 581
 FAILED, 581
 RUNNING, 581
 SUCCESS, 581
 zakleszczenie, deadlock, 606, 610
 zamiana parametrów obiektowych, 173
 zamknięcie, closure, 352
 zapis do pliku, 99
 zasada zamienialności, substitution principle, 225
 zasięg
 blokowy, 102
 zmiennych, 102, 284

- zasoby, 291
 - zatruta pigułka, 630
 - zbiory, 632
 - bitów, 558
 - zbiór, set, 493, 494, 497, 504, 506, 515
 - uporządkowany, 494, 508
 - dodawanie elementów, 509
 - widoki współbieżne, 639
 - zdarzenia, 330
 - obiekty nasłuchujące, 331
 - obsługa, 330
 - zegar, 330
 - zgłaszanie wyjątków, 390
 - zintegrowane środowisko programistyczne, IDE, 46
 - zmienna środowiskowa CLASSPATH, 200
 - zmienne, 65, 351
 - atomowe, 621
 - bez nazwy, 280
 - deklaracja, 65
 - faktycznie finalne, effectively final, 353, 366
 - inicjalizacja, 66
 - interfejsowe, 324
 - lokalne, 155, 177
 - lokalne wątków, 593, 625
 - obiektove, 142–144
 - polimorficzne, 226
 - typowe, 441
 - ograniczenia, 441
 - statyczny kontekst klas generycznych, 465
 - wymazywanie, 444
 - typu Object, 238
 - ulotne, 620
 - warunkowe, 604
 - zasięg, 102
 - znaczniki
 - blokowe, 207
 - dokumentacyjne, 206
 - polecenia printf, 98
 - śródliniowe, 207
 - znak
 - ", 92
 - \$, 65
 - *, 693
 - @, 206, 667
 - ^, 707
 - |, 498
 - nowego wiersza, \n, 57, 176, 203
 - podkreślenia, _, 236, 281, 343
 - strzałki, ->, 115, 342
 - wielokropka, ..., 264
 - znaki
 - białe, 93
 - dodatkowe, supplementary characters, 63
 - nowego wiersza, \n, 92
 - podkreślenia, __, 280
 - potrójne cudzysłowy, 92
 - specjalne, 61, 62
- Z**
- żądania sieciowe, 584

PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

JAVA:

doskonałość kodu zaczyna się od solidnych podstaw!

W 1995 roku rozpoczęła się nowa epoka programowania: świat poznał Javę. Dziś jest to potężny, dojrzały i wszechstronny język służący do pisania dużych systemów, małych programów, aplikacji mobilnych i internetowych. Java nieustannie się rozwija, a każda nowa wersja przynosi profesjonalnym programistom przemyślane funkcjonalności.

Ta książka jest kolejnym, zaktualizowanym i uzupełnionym wydaniem klasycznego podręcznika dla programistów, którzy szukają dokładnego opisu języka Java i jego platformy. Zawiera szczegółowe omówienie wszystkich jego składników, w tym najnowszych ulepszeń dodanych w wersji 21. W poszczególnych rozdziałach znajdują się przykłady kodu, które ilustrują najnowsze składniki obszernej biblioteki Javy — przystępne i praktyczne, stanowią świetny punkt wyjścia do pisania własnego kodu. W pierwszym tomie podręcznika znalazły się podstawowe zagadnienia związane z programowaniem w Javie, od programowania obiektowego, przez techniki refleksji i obiektów pośrednich, po wyrażenia lambda, adnotacje i system modułów platformy Java.

W książce między innymi:

- techniki, idiomy i najlepsze praktyki pisania kodu w języku Java
- interfejsy, klasy wewnętrzne i wyrażenia lambda
- zapieczętowane hierarchie klas i przetwarzanie danych strukturalnych
- obsługa wyjątków i skuteczne techniki debugowania
- korzystanie z typów generycznych i standardowych kolekcji Javy
- stosowanie modelu współbieżności Javy

Cay S. Horstmann jest profesorem informatyki na Uniwersytecie Stanowym w San José i autorem najpopularniejszych w Polsce podręczników do nauki Javy. Został wyróżniony tytułem Java Champion. Regularnie występuje na konferencjach programistycznych. Urodził się w północnej części Niemiec, obecnie mieszka w USA.

Helion 

 helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-289-3023-0



9 788328 930230

Cena: 149,00 zł

