

The Helion logo, featuring the word "Helion" in a white sans-serif font followed by a stylized white "V" or checkmark symbol, all contained within a red rectangular box.

Helion



JAVA

Podstawy

WYDANIE XII

ORACLE

Cay S. Horstmann

Tytuł oryginału: Core Java, Volume I: Fundamentals, 12th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-9479-7

Authorized translation from the English language edition, entitled Core Java, Volume I: Fundamentals, 12th Edition by Cay Horstmann, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2022 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 © 2022.

Figure 1.1: © Jmol

Figures 2.1-2.3, 2.9: © Microsoft 2021

Figures 2.5-2.8: © Eclipse Foundation

Figures 3.2-3.5, 4.9, 3.2-3.5, 4.9, 5.4, 7.2, 7.3, 10.1, 10.3, 10.8, 10.10, 10.11, 10.14-10.16, 11.4, 11.5, 11.8-11.34, 12.5, 12.6: © 2021 Oracle

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.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/javp12>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: <https://ftp.helion.pl/przyklady/javp12.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://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	26
1.2.3. Sieciowość	26
1.2.4. Niezawodność	26
1.2.5. Bezpieczeństwo	27
1.2.6. Niezależność od architektury	27
1.2.7. Przenośność	28
1.2.8. Interpretacja	29
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	34
Rozdział 2. Środowisko programistyczne Javy	38
2.1. Instalacja oprogramowania Java Development Kit	38
2.1.1. Pobieranie pakietu JDK	39
2.1.2. Instalacja pakietu JDK	39
2.1.3. Instalacja plików źródłowych i dokumentacji	40
2.2. Używanie narzędzi wiersza poleceń	42
2.3. Praca w zintegrowanym środowisku programistycznym	47
2.4. JShell	50

Rozdział 3. Podstawowe elementy języka Java	53
3.1. Prosty program w Javie	54
3.2. Komentarze	57
3.3. Typy danych	58
3.3.1. Typy całkowite	58
3.3.2. Typy zmiennoprzecinkowe	59
3.3.3. Typ char	61
3.3.4. Unicode i typ char	62
3.3.5. Typ boolean	63
3.4. Zmienne i stałe	64
3.4.1. Deklarowanie zmiennych	64
3.4.2. Inicjalizacja zmiennych	65
3.4.3. Stałe	66
3.4.4. Typ wyliczeniowy	67
3.5. Operatory	67
3.5.1. Operatory arytmetyczne	67
3.5.2. Funkcje i stałe matematyczne	68
3.5.3. Konwersja typów numerycznych	70
3.5.4. Rzutowanie	71
3.5.5. Przypisanie	72
3.5.6. Operatory inkrementacji i dekrementacji	72
3.5.7. Operatory relacyjne i logiczne	73
3.5.8. Operator warunkowy	74
3.5.9. Wyrażenia switch	74
3.5.10. Operatory bitowe	75
3.5.11. Nawiasy i priorytety operatorów	76
3.6. Łącuchy	77
3.6.1. Podłącuchy	77
3.6.2. Konkatenacja	78
3.6.3. Łącuchów nie można modyfikować	79
3.6.4. Porównywanie łańcuchów	80
3.6.5. Łącuchy puste i łańcuchy null	81
3.6.6. Współrzędne kodowe znaków i jednostki kodowe	82
3.6.7. API String	83
3.6.8. Dokumentacja API w internecie	86
3.6.9. Składanie łańcuchów	86
3.6.10. Bloki tekstowe	90
3.7. Wejście i wyjście	92
3.7.1. Odbieranie danych wejściowych	92
3.7.2. Formatowanie danych wyjściowych	94
3.7.3. Zapis i odczyt plików	97
3.8. Sterowanie wykonywaniem programu	99
3.8.1. Zasięg blokowy	99
3.8.2. Instrukcje warunkowe	100
3.8.3. Pętle	102
3.8.4. Pętle o określonej liczbie powtórzeń	106
3.8.5. Wybór wielokierunkowy — instrukcja switch	110
3.8.6. Instrukcje przerywające przepływ sterowania	114

3.9.	Wielkie liczby	117
3.10.	Tablice	120
3.10.1.	Deklarowanie tablic	120
3.10.2.	Dostęp do elementów tablicy	121
3.10.3.	Pętla typu for each	122
3.10.4.	Kopiowanie tablicy	123
3.10.5.	Parametry wiersza poleceń	124
3.10.6.	Sortowanie tablicy	125
3.10.7.	Tablice wielowymiarowe	128
3.10.8.	Tablice postrzępione	130
Rozdział 4.	Obiekty i klasy	134
4.1.	Wstęp do programowania obiektowego	135
4.1.1.	Klasy	136
4.1.2.	Obiekty	137
4.1.3.	Identyfikacja klas	137
4.1.4.	Relacje między klasami	138
4.2.	Używanie klas predefiniowanych	139
4.2.1.	Obiekty i zmienne obiektów	140
4.2.2.	Klasa LocalDate	143
4.2.3.	Metody udostępniające i zmieniające wartość elementu	145
4.3.	Definiowanie własnych klas	148
4.3.1.	Klasa Employee	148
4.3.2.	Używanie wielu plików źródłowych	151
4.3.3.	Analiza klasy Employee	152
4.3.4.	Pierwsze kroki w tworzeniu konstruktorów	152
4.3.5.	Deklarowanie zmiennych lokalnych za pomocą słowa kluczowego var	154
4.3.6.	Praca z referencjami null	154
4.3.7.	Parametry jawne i niejawne	156
4.3.8.	Korzyści z hermetyzacji	157
4.3.9.	Przywileje klasowe	159
4.3.10.	Metody prywatne	160
4.3.11.	Stałe jako pola klasy	160
4.4.	Pola i metody statyczne	161
4.4.1.	Pola statyczne	161
4.4.2.	Stałe statyczne	162
4.4.3.	Metody statyczne	163
4.4.4.	Metody fabryczne	164
4.4.5.	Metoda main	165
4.5.	Parametry metod	168
4.6.	Konstruowanie obiektów	174
4.6.1.	Przeciążanie	174
4.6.2.	Domyślna inicjalizacja pól	174
4.6.3.	Konstruktor bezargumentowy	175
4.6.4.	Jawna inicjalizacja pól	176
4.6.5.	Nazywanie parametrów	177
4.6.6.	Wywoływanie innego konstruktora	178
4.6.7.	Bloki inicjalizujące	178
4.6.8.	Niszczenie obiektów i metoda finalize	183

4.7.	Rekordy	183
4.7.1.	Koncepcja rekordu	184
4.7.2.	Konstruktory: kanoniczny, niestandardowy i kompaktowy	186
4.8.	Pakiety	188
4.8.1.	Nazwy pakietów	188
4.8.2.	Importowanie klas	188
4.8.3.	Importowanie statyczne	190
4.8.4.	Dodawanie klasy do pakietu	191
4.8.5.	Dostęp do pakietu	194
4.8.6.	Ścieżka klas	195
4.8.7.	Ustawianie ścieżki klas	197
4.9.	Pliki JAR	198
4.9.1.	Tworzenie plików JAR	198
4.9.2.	Manifest	198
4.9.3.	Wykonywalne pliki JAR	200
4.9.4.	Pliki JAR z wieloma wersjami klas	201
4.9.5.	Kilka uwag na temat opcji wiersza poleceń	202
4.10.	Komentarze dokumentacyjne	203
4.10.1.	Wstawianie komentarzy	203
4.10.2.	Komentarze do klas	204
4.10.3.	Komentarze do metod	205
4.10.4.	Komentarze do pól	205
4.10.5.	Komentarze ogólne	206
4.10.6.	Komentarze do pakietów	207
4.10.7.	Pobieranie komentarzy	207
4.11.	Porady dotyczące projektowania klas	208
Rozdział 5.	Dziedziczenie	211
5.1.	Klasy, nadklasy i podklasy	212
5.1.1.	Definiowanie podklas	212
5.1.2.	Przesłanianie metod	214
5.1.3.	Konstruktory podklas	215
5.1.4.	Hierarchia dziedziczenia	219
5.1.5.	Polimorfizm	220
5.1.6.	Zasady wywoływania metod	221
5.1.7.	Wyłączanie dziedziczenia — klasy i metody finalne	224
5.1.8.	Rzutowanie	225
5.1.9.	Operator instanceof i dopasowywanie wzorców	227
5.1.10.	Ograniczanie dostępu	229
5.2.	Kosmiczna klasa wszystkich klas — Object	230
5.2.1.	Zmienne typu Object	231
5.2.2.	Metoda equals	231
5.2.3.	Porównywanie a dziedziczenie	233
5.2.4.	Metoda hashCode	236
5.2.5.	Metoda toString	239
5.3.	Generyczne listy tablicowe	244
5.3.1.	Deklarowanie list tablicowych	245
5.3.2.	Dostęp do elementów listy tablicowej	247
5.3.3.	Zgodność pomiędzy typowanymi a surowymi listami tablicowymi	250

5.4.	Opakowania obiektów i automatyczne pakowanie	252
5.5.	Metody ze zmienną liczbą parametrów	255
5.6.	Klasy abstrakcyjne	257
5.7.	Klasy wyliczeniowe	262
5.8.	Klasy zapieczętowane	264
5.9.	Refleksja	269
5.9.1.	Klasa Class	270
5.9.2.	Podstawy deklarowania wyjątków	273
5.9.3.	Zasoby	274
5.9.4.	Zastosowanie refleksji w analizie funkcjonalności klasy	276
5.9.5.	Refleksja w analizie obiektów w czasie działania programu	283
5.9.6.	Zastosowanie refleksji w generycznym kodzie tablicowym	288
5.9.7.	Wywoływanie dowolnych metod i konstruktorów	291
5.10.	Porady projektowe dotyczące dziedziczenia	295
Rozdział 6.	Interfejsy, wyrażenia lambda i klasy wewnętrzne	298
6.1.	Interfejsy	299
6.1.1.	Koncepcja interfejsu	299
6.1.2.	Własności interfejsów	305
6.1.3.	Interfejsy a klasy abstrakcyjne	306
6.1.4.	Metody statyczne i prywatne	308
6.1.5.	Metody domyślne	308
6.1.6.	Wybieranie między metodami domyślnymi	310
6.1.7.	Interfejsy i wywołania zwrotne	311
6.1.8.	Interfejs Comparator	314
6.1.9.	Klonowanie obiektów	315
6.2.	Wyrażenia lambda	322
6.2.1.	Po co w ogóle są lambdy	322
6.2.2.	Składnia wyrażeń lambda	323
6.2.3.	Interfejsy funkcyjne	326
6.2.4.	Referencje do metod	328
6.2.5.	Referencje do konstruktorów	331
6.2.6.	Zakres dostępności zmiennych	332
6.2.7.	Przetwarzanie wyrażeń lambda	334
6.2.8.	Poszerzenie wiadomości o komparatorach	338
6.3.	Klasy wewnętrzne	339
6.3.1.	Dostęp do stanu obiektu w klasie wewnętrznej	340
6.3.2.	Specjalne reguły składniowe dotyczące klas wewnętrznych	343
6.3.3.	Czy klasy wewnętrzne są potrzebne i bezpieczne?	344
6.3.4.	Lokalne klasy wewnętrzne	346
6.3.5.	Dostęp do zmiennych finalnych z metod zewnętrznych	346
6.3.6.	Anonimowe klasy wewnętrzne	348
6.3.7.	Statyczne klasy wewnętrzne	351
6.4.	Moduły ładowania usług	355
6.5.	Klasy pośredniczące	357
6.5.1.	Kiedy używać klas pośredniczących	358
6.5.2.	Tworzenie obiektów pośredniczących	358
6.5.3.	Właściwości klas pośredniczących	362

Rozdział 7. Wyjątki, asercje i dzienniki	365
7.1. Obsługa błędów	366
7.1.1. Klasyfikacja wyjątków	367
7.1.2. Deklarowanie wyjątków kontrolowanych	369
7.1.3. Zgłaszanie wyjątków	371
7.1.4. Tworzenie klas wyjątków	373
7.2. Przechwytywanie wyjątków	374
7.2.1. Przechwytywanie wyjątku	374
7.2.2. Przechwytywanie wielu typów wyjątków	376
7.2.3. Powtórne generowanie wyjątków i budowanie łańcuchów wyjątków	377
7.2.4. Klauzula finally	378
7.2.5. Instrukcja try z zasobami	381
7.2.6. Analiza danych ze stosu wywołań	382
7.3. Wskazówki dotyczące stosowania wyjątków	387
7.4. Asercje	390
7.4.1. Koncepcja asercji	390
7.4.2. Włączanie i wyłączanie asercji	391
7.4.3. Zastosowanie asercji do sprawdzania parametrów	392
7.4.4. Zastosowanie asercji do dokumentowania założeń	394
7.5. Dzienniki	395
7.5.1. Podstawy zapisu do dziennika	396
7.5.2. Zaawansowane techniki zapisu do dziennika	396
7.5.3. Zmiana konfiguracji menedżera dzienników	398
7.5.4. Lokalizacja	400
7.5.5. Obiekty typu Handler	401
7.5.6. Filtry	405
7.5.7. Formatery	405
7.5.8. Przepis na dziennik	405
7.6. Wskazówki dotyczące debugowania	414
Rozdział 8. Programowanie generyczne	420
8.1. Dlaczego programowanie generyczne	421
8.1.1. Zalety parametrów typów	421
8.1.2. Dla kogo programowanie generyczne	422
8.2. Definicja prostej klasy generycznej	423
8.3. Metody generyczne	425
8.4. Ograniczenia zmiennych typowych	427
8.5. Kod generyczny a maszyna wirtualna	429
8.5.1. Wymazywanie typów	429
8.5.2. Translacja wyrażeń generycznych	430
8.5.3. Translacja metod generycznych	431
8.5.4. Używanie starego kodu	433
8.6. Ograniczenia i braki	434
8.6.1. Nie można podawać typów prostych jako parametrów typowych	434
8.6.2. Sprawdzanie typów w czasie działania programu jest możliwe tylko dla typów surowych	435
8.6.3. Nie można tworzyć tablic typów generycznych	435
8.6.4. Ostrzeżenia dotyczące zmiennej liczby argumentów	436
8.6.5. Nie wolno tworzyć egzemplarzy zmiennych typowych	437

8.6.6.	Nie można utworzyć egzemplarza generycznej tablicy	438
8.6.7.	Zmiennych typowych nie można używać w statycznych kontekstach klas generycznych	440
8.6.8.	Obiektów klasy generycznej nie można generować ani przechwytywać	440
8.6.9.	Można wyłączyć sprawdzanie wyjątków kontrolowanych	441
8.6.10.	Uważaj na konflikty, które mogą powstać po wymazaniu typów	442
8.7.	Zasady dziedziczenia dla typów generycznych	443
8.8.	Typy wieloznaczne	445
8.8.1.	Koncepcja typu wieloznacznego	445
8.8.2.	Ograniczenia nadtypów typów wieloznacznych	447
8.8.3.	Typy wieloznaczne bez ograniczeń	450
8.8.4.	Chwywanie typu wieloznacznego	450
8.9.	Refleksja a typy generyczne	453
8.9.1.	Generyczna klasa Class	453
8.9.2.	Zastosowanie parametrów Class<T> do dopasowywania typów	454
8.9.3.	Informacje o typach generycznych w maszynie wirtualnej	454
8.9.4.	Literały typowe	458
Rozdział 9.	Kolekcje	464
9.1.	Architektura kolekcji Javy	465
9.1.1.	Oddzielenie warstwy interfejsów od warstwy klas konkretnych	465
9.1.2.	Interfejs Collection	467
9.1.3.	Iteratory	468
9.1.4.	Generyczne metody użytkowe	470
9.2.	Interfejsy w systemie kolekcji Javy	473
9.3.	Konkretne klasy kolekcyjne	475
9.3.1.	Listy powiązane	476
9.3.2.	Listy tablicowe	486
9.3.3.	Zbiór HashSet	486
9.3.4.	Zbiór TreeSet	490
9.3.5.	Kolejki Queue i Deque	494
9.3.6.	Kolejki priorytetowe	496
9.4.	Słowniki	497
9.4.1.	Podstawowe operacje słownikowe	497
9.4.2.	Modyfikowanie wpisów w słowniku	501
9.4.3.	Widoki słowników	502
9.4.4.	Klasa WeakHashMap	504
9.4.5.	Klasy LinkedHashMap i LinkedHashMap	505
9.4.6.	Klasy EnumSet i EnumMap	506
9.4.7.	Klasa IdentityHashMap	507
9.5.	Kopie i widoki	509
9.5.1.	Małe kolekcje	509
9.5.2.	Niemodyfikowalne kopie i widoki	511
9.5.3.	Przedziały	513
9.5.4.	Widoki kontrolowane	513
9.5.5.	Widoki synchronizowane	514
9.5.6.	Uwagi dotyczące operacji opcjonalnych	514
9.6.	Algorytmy	519
9.6.1.	Dlaczego algorytmy generyczne	519
9.6.2.	Sortowanie i tasowanie	520

9.6.3.	Wyszukiwanie binarne	523
9.6.4.	Proste algorytmy	524
9.6.5.	Operacje zbiorowe	526
9.6.6.	Konwersja pomiędzy kolekcjami a tablicami	527
9.6.7.	Pisanie własnych algorytmów	528
9.7.	Stare kolekcje	529
9.7.1.	Klasa Hashtable	529
9.7.2.	Wyliczenia	530
9.7.3.	Słowniki własności	531
9.7.4.	Stosy	534
9.7.5.	Zbiory bitów	535
Rozdział 10.	Graficzne interfejsy użytkownika	539
10.1.	Historia zestawów narzędzi do tworzenia interfejsów użytkownika	539
10.2.	Wyświetlanie ramki	541
10.2.1.	Tworzenie ramki	541
10.2.2.	Właściwości ramki	543
10.3.	Wyświetlanie informacji w komponencie	547
10.3.1.	Figury 2D	551
10.3.2.	Kolory	557
10.3.3.	Czcionki	559
10.3.4.	Wyświetlanie obrazów	565
10.4.	Obsługa zdarzeń	566
10.4.1.	Podstawowe koncepcje obsługi zdarzeń	567
10.4.2.	Przykład — obsługa kliknięcia przycisku	569
10.4.3.	Zwiąże definiowanie procedur nasłuchowych	572
10.4.4.	Klasy adaptacyjne	573
10.4.5.	Akcje	575
10.4.6.	Zdarzenia generowane przez mysz	580
10.4.7.	Hierarchia zdarzeń w bibliotece AWT	585
10.5.	API Preferences	587
Rozdział 11.	Komponenty Swing interfejsu użytkownika	595
11.1.	Swing i wzorec model-widok-kontroler	596
11.2.	Wprowadzenie do zarządzania rozkładem	599
11.2.1.	Zarządcy układu	600
11.2.2.	Rozkład brzegowy	601
11.2.3.	Rozkład siatkowy	603
11.3.	Wprowadzanie tekstu	604
11.3.1.	Pola tekstowe	605
11.3.2.	Etykiety komponentów	606
11.3.3.	Pola haseł	608
11.3.4.	Obszary tekstowe	608
11.3.5.	Panele przewijane	609
11.4.	Komponenty umożliwiające wybór opcji	611
11.4.1.	Pola wyboru	612
11.4.2.	Przełączniki	614
11.4.3.	Obramowanie	617
11.4.4.	Listy rozwijane	620
11.4.5.	Suwaki	623

11.5.	Menu	629
11.5.1.	Tworzenie menu	629
11.5.2.	Ikony w elementach menu	631
11.5.3.	Pola wyboru i przełączniki jako elementy menu	632
11.5.4.	Menu podręczne	634
11.5.5.	Mnemoniki i akceleratory	635
11.5.6.	Aktywowanie i dezaktywowanie elementów menu	637
11.5.7.	Paski narzędzi	641
11.5.8.	Dymki	643
11.6.	Zaawansowane techniki zarządzania rozkładem	644
11.6.1.	Rozkład GridBagLayout	644
11.6.2.	Niestandardowi zarządcy rozkładu	653
11.7.	Okna dialogowe	657
11.7.1.	Okna dialogowe opcji	658
11.7.2.	Tworzenie okien dialogowych	663
11.7.3.	Wymiana danych	666
11.7.4.	Okna dialogowe wyboru plików	672
Rozdział 12.	Współbieżność	680
12.1.	Czym są wątki	681
12.2.	Stany wątków	686
12.2.1.	Wątki tworzone za pomocą operatora new	686
12.2.2.	Wątki RUNNABLE	686
12.2.3.	Wątki BLOCKED i WAITING	687
12.2.4.	Zamykanie wątków	688
12.3.	Własności wątków	689
12.3.1.	Przerywanie wątków	689
12.3.2.	Wątki demony	692
12.3.3.	Nazwy wątków	692
12.3.4.	Procedury obsługi nieprzechwyconych wyjątków	693
12.3.5.	Priorytety wątków	694
12.4.	Synchronizacja	695
12.4.1.	Przykład sytuacji powodującej wyścig	695
12.4.2.	Wyścigi	698
12.4.3.	Obiekty klasy Lock	699
12.4.4.	Warunki	702
12.4.5.	Słowo kluczowe synchronized	707
12.4.6.	Bloki synchronizowane	711
12.4.7.	Monitor	713
12.4.8.	Pola ulotne	714
12.4.9.	Zmienne finalne	716
12.4.10.	Zmienne atomowe	716
12.4.11.	Zakleszczenia	718
12.4.12.	Dlaczego metody stop i suspend są wycofywane	720
12.4.13.	Inicjalizacja na żądanie	722
12.4.14.	Zmienne lokalne wątków	722
12.5.	Kolekcje bezpieczne wątkowo	724
12.5.1.	Kolejki blokujące	725
12.5.2.	Szybkie słowniki, zbiory i kolejki	731
12.5.3.	Atomowe modyfikowanie elementów słowników	733

12.5.4.	Operacje masowe na współbieżnych słownikach skrótów	736
12.5.5.	Współbieżne widoki zbiorów	738
12.5.6.	Tablice kopiowane przy zapisie	739
12.5.7.	Równoległe algorytmy tablicowe	739
12.5.8.	Starsze kolekcje bezpieczne wątkowo	740
12.6.	Zadania i pule wątków	741
12.6.1.	Interfejsy Callable i Future	742
12.6.2.	Klasa Executors	743
12.6.3.	Kontrolowanie grup zadań	746
12.6.4.	Metoda rozgałęzienie-złączenie	751
12.7.	Obliczenia asynchroniczne	754
12.7.1.	Klasa CompletableFuture	754
12.7.2.	Tworzenie obiektów CompletableFuture	756
12.7.3.	Czasochłonne zadania w wywołaniach zwrotnych interfejsu użytkownika	762
12.8.	Procesy	769
12.8.1.	Budowanie procesu	769
12.8.2.	Uruchamianie procesu	771
12.8.3.	Uchwyty procesów	772
Dodatek		776
Skorowidz		780

5

Dziedziczenie

W tym rozdziale:

- 5.1. Klasy, nadklasy i podklasy
- 5.2. Kosmiczna klasa wszystkich klas — Object
- 5.3. Generyczne listy tablicowe
- 5.4. Opakowania obiektów i automatyczne pakowanie
- 5.5. Metody ze zmienną liczbą parametrów
- 5.6. Klasy abstrakcyjne
- 5.7. Klasy wyliczeniowe
- 5.8. Klasy zapieczętowane
- 5.9. Refleksja
- 5.10. Porady projektowe dotyczące dziedziczenia

Rozdział 4. wprowadził pojęcia klas i obiektów. Ten rozdział wprowadza kolejne zagadnienie o fundamentalnym znaczeniu dla programowania obiektowego — **dziedziczenie** (ang. *inheritance*). Z założenia technika ta umożliwia tworzenie nowych klas na bazie klas już istniejących. Klasa, która dziedziczy po innej klasie, przejmuje jej metody i pola oraz dodaje własne metody i pola, które służą przystosowaniu do nowych zadań. Technika ta ma kluczowe znaczenie dla programowania w Javie.

Dodatkowo rozdział ten opisuje **refleksję** (ang. *reflection*), czyli technikę inspekcji klas w trakcie działania programu. Mimo że refleksja daje ogromne możliwości, jest bez wątpienia techniką skomplikowaną. Ponieważ ma ona większe znaczenie dla twórców narzędzi niż programistów aplikacji, tę część rozdziału można przeczytać pobieżnie i wrócić do niej w razie potrzeby.

5.1. Klasy, nadklasy i podklasy

Wróćmy do omawianej w poprzednim rozdziale klasy `Employee`. Wyobraźmy sobie, że (niestety) pracujemy w firmie, w której kierownicy są traktowani inaczej niż pozostali pracownicy. Oczywiście istnieje między nimi też wiele podobieństw. Zarówno zwykli pracownicy, jak i kierownictwo dostają wypłatę. Jednak podczas gdy zwykli pracownicy, aby otrzymać pensję, muszą ukończyć powierzone im zadania, kierownicy, jeśli osiągną zamierzony cel, dostają *dodatek* do wypłaty. Jest to typowa sytuacja, w której należy wykorzystać dziedziczenie. Dlaczego? Oczywiście można utworzyć całkiem nową klasę o nazwie `Manager` o odpowiednich właściwościach. Można jednak wykorzystać część kodu, który został napisany wcześniej w klasie `Employee`. *Wszystkie* pola oryginalnej klasy zostałyby zachowane. Stosując bardziej abstrakcyjną terminologię, istnieje oczywisty związek „jest” pomiędzy klasami `Manager` i `Employee`. Każdy kierownik (ang. *manager*) *jest* pracownikiem (ang. *employee*). Relacja typu „jest” stanowi cechę charakterystyczną dziedziczenia.



W rozdziale tym omawiamy klasyczny przykład z pracownikami i dyrektorami, ale pamiętaj, że należy do niego podchodzić z przymrużeniem oka. W prawdziwym świecie pracownik może zostać dyrektorem, więc dyrektor powinien być rolą pracownika, a nie podklasą. W naszym przykładzie jednak zakładamy, że świat korporacji zamieszkują tylko dwa rodzaje ludzi — ci, którzy po wsze czasy będą szeregowymi pracownikami, i ci, którzy od zawsze byli dyrektorami.

5.1.1. Definiowanie podklas

Do wyrażania relacji dziedziczenia służy słowo kluczowe `extends`. W poniższym przykładowym kodzie klasa `Manager` dziedziczy po klasie `Employee`.

```
class Manager extends Employee
{
    Dodatkowe metody i pola.
}
```



Dziedziczenie w Javie i C++ jest podobne, jednak w Javie wyraża się je za pomocą słowa kluczowego `extends`, a w C++ za pomocą symbolu `:`. W Javie dziedziczenie może być wyłącznie publiczne. Nie ma w tym języku odpowiedników znanych z C++ dziedziczenia prywatnego i chronionego.

Słowo kluczowe `extends` oznacza, że tworzona jest nowa klasa na podstawie istniejącej klasy. Klasa istniejąca nazywana jest **nadklasą** (ang. *superclass*), **klasą bazową** (ang. *base class*) lub **klasą macierzystą** (ang. *parent class*). Nowo utworzona klasa nazywa się **podklasą** (ang. *subclass*), **klasą pochodną** (ang. *derived class*) lub **klasą potomną** (ang. *child class*). Większość programistów Javy używa terminów „nadklasa” i „podklasa”, ale niektórym bardziej odpowiada analogia klasy macierzystej i potomnej, która bardzo dobrze pasuje do koncepcji dziedziczenia.

Klasa `Employee` jest nadklasą, ale nie ze względu na to, że jest wyższą rangą albo ma większą funkcjonalność. *W rzeczywistości jest odwrotnie*: podklasa ma *większą* funkcjonalność niż nadklasa. Będzie można się o tym przekonać, kiedy przejdziemy do analizy klasy `Manager`, która zawiera więcej danych i ma większą funkcjonalność niż jej nadklasa `Employee`.



Przedrostki *nad-* i *pod-* pochodzą od sposobu opisu zbiorów w informatyce teoretycznej i matematyce. Zbiór wszystkich pracowników zawiera zbiór wszystkich kierowników, czyli zbiór pracowników jest **nadzbiorem** zbioru kierowników. Albo w drugą stronę — zbiór kierowników jest **podzbiorem** zbioru pracowników.

Klasa `Manager` zawiera dodatkowe pole do przechowywania dodatku do pensji i nową metodę do ustawiania jego wysokości:

```
public class Manager extends Employee
{
    private double bonus;

    . . .
    public void setBonus(double bonus)
    {
        this.bonus = bonus;
    }
}
```

Powyższego pola i powyższej metody nie wyróżnia nic szczególnego. Po utworzeniu obiektu klasy `Manager` można na jego rzecz wywołać metodę `setBonus`:

```
Manager boss = . . .;
boss.setBonus(5000);
```

Oczywiście na rzecz obiektu klasy `Employee` nie można wywołać metody `setBonus`. Nie ma jej wśród metod tej klasy.

Natomiast na rzecz obiektów klasy `Manager` *można* wywoływać metody `getName` i `getHi reDay`, mimo że nie zostały one zdefiniowane bezpośrednio w tej klasie. Są dostępne, ponieważ zostały odziedziczone po klasie `Employee`.

Każdy obiekt klasy `Manager` ma cztery pola: `name`, `salary`, `hireDay` i `bonus`. Pola `name`, `salary` i `hireDay` pochodzą z nadklasy.



W specyfikacji języka Java napisano: „Składowe klasy, które są zadeklarowane jako prywatne, nie są dziedziczone przez podklasy tej klasy”. To zdanie od lat wprowadza w konsternację wielu moich czytelników. W specyfikacji słowo „dziedziczenie” jest używane w bardzo wąskim znaczeniu. Prywatne pola zostały opisane jako niedziedziczone, ponieważ klasa `Manager` nie ma do nich bezpośredniego dostępu. W związku z tym każdy obiekt klasy `Manager` ma trzy pola z nadklasy, ale klasa `Manager` ich nie „dziedziczy”.

W definicji klasy rozszerzającej inną klasę konieczne jest podanie tylko *różnic* pomiędzy tymi klasami. Metody ogólnego przeznaczenia należy umieszczać w nadklasie, a metody bardziej

wyspecjalizowane — w podklasie. Wydzielanie wspólnego kodu i umieszczanie go w nadklasie jest często stosowaną techniką programowania obiektowego.



W rozdziale 4. zostały opisane **rekordy**, czyli klasy, których stan w całości określały parametry konstruktora. Rekordu nie można rozszerzyć i rekord nie może rozszerzać żadnej innej klasy.

5.1.2. Przesłanianie metod

Niektóre obecne w klasie nadrzędnej metody są niewłaściwe dla klasy `Manager`. Jest tak w przypadku metody `getSalary`, która powinna zwracać sumę podstawy wynagrodzenia i dodatku. Konieczne jest napisanie nowej metody **przesłaniającej** (ang. *override*) metodę z nadklasy:

```
public class Manager extends Employee
{
    . . .
    public double getSalary()
    {
        . . .
    }
    . . .
}
```

Jak powinna wyglądać implementacja tej metody? Na pierwszy rzut oka wydaje się to proste — wystarczy zwrócić sumę pól `salary` i `bonus`:

```
public double getSalary()
{
    return salary + bonus;    // nie działa
}
```

Tak się jednak nie da. Tylko metody klasy `Employee` mają bezpośredni dostęp do prywatnych pól klasy `Employee`. Oznacza to, że metoda `getSalary` klasy `Manager` nie ma bezpośredniego dostępu do pola `salary`, mimo że każdy obiekt tej klasy zawiera pole o nazwie `salary`. Jeśli metody klasy `Manager` chcą uzyskać do nich dostęp, muszą zrobić to co wszystkie inne metody — użyć interfejsu publicznego. W tym przypadku konieczne jest użycie metody `getSalary` klasy `Employee`.

Spróbujmy jeszcze raz. Zamiast bezpośrednio odwoływać się do pola `salary`, użyjemy metody `getSalary`:

```
public double getSalary()
{
    double baseSalary = getSalary();    // nadal nie działa
    return baseSalary + bonus;
}
```

Problem polega na tym, że metoda `getSalary` wywołuje *sama siebie*, ponieważ klasa `Manager` zawiera metodę o takiej nazwie (dokładniej mówiąc, jest to ta metoda, którą próbujemy zaimplementować). W ten sposób powstała nieskończona seria wywołań jednej metody, która prowadzi do załamania programu.

Musimy zaznaczyć, że odwołujemy się do metody `getSalary` klasy `Employee`, a nie klasy, w której się znajdujemy. Do tego celu służy słowo kluczowe `super`. Instrukcja:

```
super.getSalary()
```

wywoła metodę `getSalary` klasy `Employee`. Poniżej znajduje się poprawna wersja metody `getSalary` w klasie `Manager`:

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```



Niektórzy programiści uważają, że słowo kluczowe `super` jest odpowiednikiem referencji `this`. Nie jest to jednak precyzyjne porównanie, ponieważ słowo `super` nie jest referencją do obiektu. Nie można na przykład przypisać jego wartości do innej zmiennej obiektowej. `super` to słowo kluczowe, które nakazuje kompilatorowi wywołanie metody z nadklasy.

Wiemy już, że do podklasy można *dodawać* nowe pola oraz *dodawać* lub *przesłaniać* metody z nadklasy. Dziedziczenie nie umożliwia natomiast pozbycia się żadnych metod ani pól.



W Javie do wywoływania metod nadklasy służy słowo kluczowe `super`. W C++ w takiej sytuacji należy użyć nazwy nadklasy z operatorem `::`. Na przykład metoda `getSalary` klasy `Manager` wywoływałaby `Employee::getSalary` zamiast `super.getSalary`.

5.1.3. Konstruktory podklas

Na koniec dodamy konstruktor.

```
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

W tym miejscu słowo kluczowe `super` znaczy co innego. Instrukcja:

```
super(name, salary, year, month, day);
```

mówi: „wywołaj konstruktor nadklasy `Employee` z parametrami `n`, `s`, `year`, `month` i `day`”.

Ponieważ konstruktor `Manager` nie ma dostępu do pól prywatnych klasy `Employee`, musi je zainicjalizować poprzez inny konstruktor. Konstruktor jest wywoływany za pomocą specjalnej składni z użyciem słowa kluczowego `super`. Wywołanie z tym słowem kluczowym musi być pierwszą instrukcją konstruktora podklasy.

Jeśli obiekt podklasy jest tworzony bez jawnego wywołania konstruktora nadklasy, to nadklasa musi mieć konstruktor bezargumentowy, który jest wywoływany przed konstruktorem podklasy.



Przypomnijmy, że słowo `this` ma dwa zastosowania: określa referencję do parametru niejawnego i wywołuje inny konstruktor tej samej klasy. Podobnie dwa zastosowania ma słowo `super`: wywołuje metody nadklasy i konstruktory nadklasy. W przypadku wywoływania konstruktorów słowa te są blisko spokrewnione. Wywołanie konstruktora może następować tylko w pierwszej instrukcji innego konstruktora. Parametry konstrukcyjne są przekazywane albo do innego konstruktora tej samej klasy (`this`), albo do konstruktora nadklasy (`super`).



W języku C++ nie używa się w konstruktorze słowa kluczowego `super`, tylko składni listy inicjalizującej nadklasy. Konstruktor `Manager` w C++ wyglądałby następująco:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

Po ponownym zdefiniowaniu metody `getSalary` dla obiektów `Manager` dodatki do pensji kierowników będą dodawane *automatycznie*.

Oto praktyczny przykład obrazujący powyższe rozważania. Utworzymy nowy obiekt klasy `Manager` i ustawimy dodatek dla kierownika:

```
Manager boss = new Manager("Karol Parol", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Tworzymy tablicę trzech pracowników:

```
Employee[] staff = new Employee[3];
```

Wstawiamy do niej obiekty zwykłych pracowników i kierowników:

```
staff[0] = boss;
staff[1] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
staff[2] = new Employee("Artur Kwiatkowski", 40000, 1990, 3, 15);
```

Wyświetlamy pensję każdego z nich:

```
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
```

Powyższa pętla zwraca następujące dane:

```
Karol Parol 85000.0
Henryk Kwiatek 50000.0
Artur Kwiatkowski 40000.0
```

Obiekty `staff[1]` i `staff[2]` drukują podstawowe wynagrodzenie, ponieważ są to obiekty klasy `Employee`. Natomiast obiekt `Staff[0]` jest obiektem klasy `Manager`, a więc jego metoda `getSalary` dolicza dodatek do podstawowego wynagrodzenia.

Na uwagę zasługuje fakt, że wywołanie:

```
e.getSalary()
```

wybiera *odpowiednią* metodę `getSalary`. Warto zauważyć, że zmienna `e` jest *zadeklarowana* jako typ `Employee`, ale *rzeczywistym* typem, do którego odwołuje się ta zmienna, może być albo `Employee`, albo `Manager`.

Kiedy zmienna `e` odwołuje się do obiektu klasy `Employee`, wywołanie `e.getSalary()` wywołuje metodę `getSalary` klasy `Employee`. Jeśli jednak `e` odwołuje się do obiektu klasy `Manager`, metoda `getSalary` jest wywoływana z klasy `Manager`. Maszyna wirtualna rozpoznaje, do jakiego typu odwołuje się zmienna `e`, dzięki czemu może wywołać odpowiednią metodę.

Możliwość odwoływania się przez obiekty (jak zmienna `e`) do wielu różnych typów nosi nazwę **polimorfizmu** (ang. *polymorphism*). Automatyczny dobór odpowiednich metod w trakcie działania programu nazywa się **wiązaniem dynamicznym** (ang. *dynamic binding*). Oba te zagadnienia zostały szczegółowo opisane w tym rozdziale.



W języku C++ wiązaniu dynamicznemu podlegają metody, które mają w deklaracji słowo kluczowe `virtual`. W Javie wiązanie dynamiczne jest domyślne. Aby metoda **nie była** wirtualna, należy użyć słowa kluczowego `final` (opis słowa kluczowego `final` znajduje się dalej w tym rozdziale).

Listing 5.1 przedstawia program demonstrujący różnicę naliczania pensji dla obiektów klasy `Employee` (listing 5.2) i `Manager` (listing 5.3).

Listing 5.1. inheritance/ManagerTest.java

```
package inheritance;

/**
 * Ten program demonstruje dziedziczenie.
 * @version 1.21 2004-02-21
 * @author Cay Horstmann
 */
public class ManagerTest
{
    public static void main(String[] args)
    {
        // Tworzenie obiektu klasy Manager.
        var boss = new Manager("Karol Parol", 80000, 1987, 12, 15);
        boss.setBonus(5000);

        var staff = new Employee[3];

        // Wstawienie obiektów klas Manager i Employee do tablicy staff.

        staff[0] = boss;
        staff[1] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
        staff[2] = new Employee("Artur Kwiatkowski", 40000, 1990, 3, 15);

        // Wyświetlanie informacji o wszystkich obiektach klasy Employee.
```

```
        for (Employee e : staff)
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
    }
}
```

Listing 5.2. inheritance/Employee.java

```
package inheritance;

import java.time.*;

public class Employee
{
    private String name;
    private double salary;
    private LocalDate hireDay;

    public Employee(String name, double salary, int year, int month, int day)
    {
        this.name = name;
        this.salary = salary;
        hireDay = LocalDate.of(year, month, day);
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public LocalDate getHireDay()
    {
        return hireDay;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }
}
```

Listing 5.3. inheritance/Manager.java

```
package inheritance;

public class Manager extends Employee
{
    private double bonus;
    /**
     * @param n imię i nazwisko pracownika
     * @param s pensja
     * @param year rok przyjęcia do pracy
     */
}
```

```

    * @param month miesiąc przyjęcia do pracy
    * @param day dzień przyjęcia do pracy
    */
    public Manager(String n, double s, int year, int month, int day)
    {
        super(n, s, year, month, day);
        bonus = 0;
    }

    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

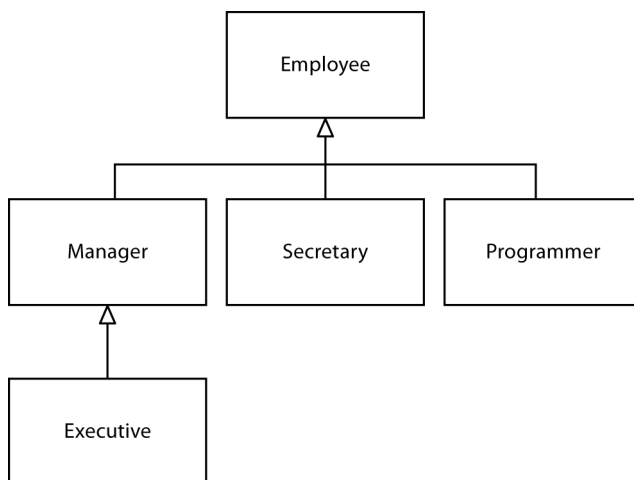
    public void setBonus(double b)
    {
        bonus = b;
    }
}

```

5.1.4. Hierarchia dziedziczenia

Dziedziczenie nie dotyczy tylko jednego poziomu klas. Na przykład rozszerzeniem klasy `Manager` może być klasa `Executive` (dyrektor). Strukturę klas i ich drogi dziedziczenia od wspólnej klasy bazowej nazywa się **hierarchią dziedziczenia** (ang. *inheritance hierarchy*) — zobacz rysunek 5.1. Ścieżka od danej klasy do jej przodków w hierarchii dziedziczenia ma nazwę **łańcucha dziedziczenia** (ang. *inheritance chain*).

Rysunek 5.1.
Hierarchia
dziedziczenia
klasy `Employee`



Łańcuchów dziedziczenia może być wiele. Na przykład klasę `Employee` mogą rozszerzać klasy `Programmer` i `Secretary`, które nie muszą mieć nic wspólnego z klasą `Manager` (albo ze sobą nawzajem). Struktura ta może mieć dowolną długość.



Java nie obsługuje dziedziczenia wielokrotnego. Sposoby naśladowania techniki dziedziczenia wielokrotnego są opisane w części o interfejsach w podrozdziale 6.1.

5.1.5. Polimorfizm

W podjęciu decyzji dotyczącej tego, czy w danym przypadku zastosować dziedziczenie, pomaga prosta zasada. Reguła „jest” mówi, że każdy obiekt podklasy jest obiektem nadklasy. Na przykład każdy kierownik jest pracownikiem. W związku z tym klasa `Manager` może być podklasą klasy `Employee`. Oczywiście twierdzenia tego nie można odwrócić — nie każdy pracownik jest kierownikiem.

Regułę relacji „jest” można także sformułować jako **zasadę zamienialności** (ang. *substitution principle*). Zasada ta głosi, że wszędzie tam, gdzie można użyć obiektu nadklasy, można użyć obiektu podklasy.

Można na przykład przypisać obiekt podklasy do zmiennej nadklasy.

```
Employee e;  
e = new Employee(. . .); // Powinien być obiekt klasy Employee.  
e = new Manager(. . .);  // OK, obiekt klasy Manager też może być.
```

W Javie zmienne obiektowe są **polimorficzne**. Zmienna typu `Employee` może odwoływać się do dowolnego obiektu klasy `Employee`, jak również do każdego obiektu podklasy klasy `Employee` (jak `Manager`, `Executive`, `Secretary` itd.).

Skorzystaliśmy z tej zasady w programie z listingu 5.1:

```
Manager boss = new Manager(. . .);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

W tym przypadku zmienne `staff[0]` i `boss` odwołują się do tego samego obiektu. Jednak dla kompilatora `staff[0]` jest obiektem wyłącznie klasy `Employee`.

Oznacza to, że można użyć poniższego wywołania:

```
boss.setBonus(5000); // OK
```

Ale poniższe spowoduje błąd:

```
staff[0].setBonus(5000); // błąd
```

Typ zadeklarowany zmiennej `staff[0]` to `Employee`, a metoda `setBonus` nie jest obecna w tej klasie.

Nie można przypisać referencji nadklasy do zmiennej podklasy. Na przykład poniższe przypisanie jest niedozwolone:

```
Manager m = staff[i]; // błąd
```

Powód jest jasny — nie wszyscy pracownicy są kierownikami. Gdyby powyższe przypisanie udało się i zmienna `m` byłaby referencją do obiektu klasy `Employee`, który nie jest kierownikiem, to możliwe byłoby też wywołanie `m.setBonus(...)`, a to z kolei spowodowałoby błąd czasu wykonania.



W Javie możliwa jest konwersja tablic referencji do obiektów podklasy na tablicę referencji do obiektów nadklasy bez rzutowania. Spójrzmy na poniższą tablicę obiektów klasy `Manager`:

```
Manager[] managers = new Manager[10];
```

Można ją przekonwertować na tablicę `Employee[]`:

```
Employee[] staff = managers; // OK
```

Można pomyśleć, czemu nie. Przecież każdy kierownik jest też pracownikiem. Jednak dzieje się tu coś zaskakującego. Pamiętajmy, że zmienne `managers` i `staff` są referencjami do tej samej tablicy. Spójrzmy teraz na poniższą instrukcję:

```
staff[0] = new Employee("Henryk Kwiatek", ...);
```

Kompilator nie zgłosi żadnego sprzeciwu przy kompilacji tego przypisania, ale zmienne `staff[0]` i `managers[0]` są tą samą referencją, a więc wygląda na to, że awansowaliśmy zwykłego pracownika na stanowisko kierownicze. Byłaby to bardzo zła sytuacja. Wywołanie `managers[0].setBonus(1000)` usiłowałoby uzyskać dostęp do nieistniejącego pola i spowodowałoby uszkodzenie okolicznej pamięci.

Aby uniknąć takiego zniszczenia, wszystkie tablice pamiętają, z jakim typem zostały utworzone, i pilnują, aby przechowywane w nich były tylko odpowiednie referencje. Na przykład tablica utworzona za pomocą instrukcji `new Manager[10]` pamięta, że jest tablicą kierowników. Próba zapisania w niej referencji do typu `Employee` spowoduje wyjątek `ArrayStoreException`.

5.1.6. Zasady wywoływania metod

Programista musi dokładnie wiedzieć, jak wywołanie metody jest stosowane do obiektu. Powiedzmy, że mamy w programie wywołanie `x.f(arg)` i że niejawni parametr `x` jest zadeklarowany jako obiekt klasy `C`. Oto co się dzieje:

1. Kompilator sprawdza zadeklarowany typ obiektu i nazwę metody. Pamiętajmy, że metod o nazwie `f` może być kilka, a różnica między nimi polega na tym, że mają różne listy parametrów. Na przykład mogą to być metody `f(int)` i `f(String)`. Kompilator tworzy listę wszystkich metod o nazwie `f` dostępnych w klasie `C` i wszystkich dostępnych metod o tej nazwie w nadklasie klasy `C` (prywatne metody nadklasy są niedostępne).

W ten sposób powstaje lista wszystkich potencjalnych metod do wywołania.

2. Następnie kompilator sprawdza typy parametrów podanych w wywołaniu metody. Jeśli wśród zebranych metod o nazwie `f` znajduje się taka, której parametry pasują dokładnie do podanych parametrów, to zostaje ona wywołana. Proces ten nazywa się **rozstrzygnięciem przeciążania**. Na przykład dla wywołania `x.f("Witaj")`

kompilator wybierze metodę `f(String)`, a nie `f(int)`. Sytuacja może się skomplikować ze względu na konwersję typów (`int` na `double`, `Manager` na `Employee` itd.). Jeśli kompilator nie może znaleźć metody pasującej do podanych parametrów lub znajdzie kilka pasujących metod, zgłasza błąd.

W ten sposób kompilator znajduje metodę, którą należy wywołać.

3. Jeśli metoda jest prywatna, statyczna lub finalna albo jest konstruktorem, kompilator wie, którą dokładnie metodę wywołać (modyfikator `final` jest opisany w kolejnym podrozdziale). Nazywa się to **wiązaniem statycznym** (ang. *static binding*). W przeciwnym przypadku to, która metoda zostanie wywołana, zależy od rzeczywistego typu parametru niejawnego; musi też być zastosowane wiązanie dynamiczne w trakcie działania programu. W naszym przykładzie kompilator wygenerowałby instrukcję wywołującą metodę `f(String)` za pomocą wiązania dynamicznego.



Przypomnijmy, że nazwa oraz lista typów parametrów metody są nazywane **sygnaturą** metody. Na przykład metody `f(int)` i `f(String)` mają takie same nazwy, ale różne sygnatury. Jeśli w podklasie zostanie zdefiniowana metoda o takiej samej sygnaturze jak metoda w klasie nadrzędnej, dana metoda nadklasy zostanie przesłonięta.

Typ zwrotny nie jest częścią sygnatury, ale musi się on w metodzie przesłaniającej zgadzać z tym w metodzie przesłoniętej. Podklasa może zmienić typ zwrotny na podtyp oryginalnego typu. Wyobraźmy sobie na przykład, że klasa `Employee` zawiera metodę

```
public Employee getKumpel() { ... }
```

Menedżer nie chciałby się kolegować ze zwykłym pracownikiem. Dlatego w podklasie metoda ta może zostać przesłonięta:

```
public Manager getKolega() { ... } // Można zmienić typ zwrotny
```

Mówi się, że obie metody `getKolega` mają **kowariantne** typy zwrotne (ang. *covariant return types*).

4. Kiedy program działa i wywołuje metodę za pomocą wiązania dynamicznego, maszyna wirtualna musi wywołać tę wersję niniejszej metody, która odpowiada *rzeczywistemu* typowi obiektu, do którego odwołuje się zmienna `x`. Niech tym typem będzie `D` — podklasa klasy `C`. Jeśli klasa `D` zawiera definicję metody `f(String)`, wywołana zostaje właśnie ta metoda. W przeciwnym przypadku metoda ta jest szukana w nadklasie klasy `D` itd.

Gdyby to wyszukiwanie było przeprowadzane przy każdym wywołaniu tej metody, tracono by dużo czasu. Dlatego maszyna wirtualna tworzy na początku **tabelę metod** zawierającą sygnatury i ciała wszystkich metod, które mogą być wywołane.

Maszyna wirtualna może utworzyć tabelę metod po załadowaniu klasy, przez połączenie metod znalezionych w pliku klasy z tabelą metod nadklasy.

Kiedy dana metoda jest wywoływana, maszyna wirtualna odszukuje ją w swojej tabeli. W naszym przykładzie maszyna wirtualna przeszukuje tabelę metod klasy `D` i odnajduje metodę `f(String)`. Może to być metoda `D.f(String)` albo `x.f(String)`,

gdzie X to jedna z nadklas klasy D . Scenariusz ten może się zmienić w jednej sytuacji. Jeśli wywołanie ma postać `super.f(param)`, to maszyna wirtualna przeszukuje tabelę metod nadklasy.

Przeanalizujmy ten proces na przykładzie wywołania `e.getSalary()` z listingu 5.1. Obiekt `e` jest typu `Employee`. Klasa `Employee` zawiera tylko jedną metodę o nazwie `getSalary`, która nie ma parametrów. Dzięki temu w tym przypadku nie ma problemu z rozstrzygnięciem przeciążania.

Ponieważ metoda `getSalary` nie jest prywatna, statyczna ani finalna, jest wiązana dynamicznie. Maszyna wirtualna tworzy tabele metod dla klas `Employee` i `Manager`. Tabela `Employee` wskazuje, że wszystkie jej metody są zdefiniowane w samej klasie `Employee`:

```
Employee:
  getName() -> Employee.getName()
  getSalary() -> Employee.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
```

To jednak nie wszystko. Jak się niebawem dowiemy, klasa `Employee` ma nadklasę `Object`, po której dziedziczy kilka metod. Na razie ignorujemy te dodatkowe metody.

Tabela metod klasy `Manager` wygląda nieco inaczej. Trzy metody są odziedziczone, jedna prze-definiowana, a jedna została dodana.

```
Manager:
  getName() -> Employee.getName()
  getSalary() -> Manager.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
  setBonus(double) -> Manager.setBonus(double)
```

Oto co się dzieje z wywołaniem `e.getSalary()` w trakcie działania programu:

1. Maszyna wirtualna tworzy tabelę metod dla rzeczywistego typu `e`. Może to być tabela klasy `Employee` lub jednej z jej podklas, np. `Manager`.
2. Następnie maszyna wirtualna szuka klasy zawierającej definicję metody o sygnaturze `getSalary()`. W ten sposób znajduje metodę, którą ma wywołać.
3. Ostatecznie maszyna wirtualna wywołuje odpowiednią metodę.

Wiązanie dynamiczne ma jedną bardzo ważną cechę: umożliwia *rozszerzanie* programów bez modyfikacji istniejącego kodu. Przypuśćmy, że została utworzona klasa `Executive` i istnieje możliwość, że zmienna `e` odwołuje się do obiektu tej klasy. Kod zawierający wywołanie `e.getSalary()` nie musi być ponownie kompilowany. Metoda `Executive.getSalary()` zostanie wywołana automatycznie, jeśli zmienna `e` odwołuje się do obiektu typu `Executive`.



Kiedy metoda jest przesłaniana, jej odpowiednik w podklasie musi mieć *przynajmniej taką samą widoczność* jak oryginał. Jeśli metoda w nadklasie jest publiczna, w podklasie również musi być publiczna. Pominięcie specyfikatora `public` w metodzie podklasy jest częstym błędem. W takim przypadku kompilator informuje, że usiłowano zastosować niższy przywilej dostępu.

5.1.7. Wyłączanie dziedziczenia — klasy i metody finalne

Zdarzają się sytuacje, kiedy chcemy, aby nie tworzono podklas jednej z klas. Klasy, których nie można rozszerzać, nazywają się klasami **finalnymi** (ang. *final*), a do ich oznaczania służy modyfikator `final`. Załóżmy na przykład, że nie chcemy, aby klasa `Executive` była rozszerzana. W tym celu należy w jej definicji użyć modyfikatora `final`:

```
public final class Executive extends Manager
{
    . . .
}
```

Finalna może też być metoda w klasie. W takim przypadku nie można jej przesłonić w żadnej z podklas (wszystkie metody w klasie finalnej są finalne). Na przykład:

```
public class Employee
{
    . . .
    public final String getName()
    {
        return name;
    }
    . . .
}
```



Przypomnijmy, że pola również mogą być finalne. Wartość takiego pola nie może być zmieniana po utworzeniu obiektu. Jeśli klasa jest finalna, tylko jej metody są automatycznie finalne, nie dotyczy to pól.

Jest tylko jeden powód, dla którego warto zdefiniować klasę lub metodę jako finalną: aby zapewnić, że żadna podklasa nie zmieni semantyki. Na przykład metody `getTime` i `setTime` klasy `Calendar` są finalne. Oznacza to, że projektanci tej klasy wzięli na siebie ciężar odpowiedzialności za konswersję pomiędzy klasą `Date` a stanem kalendarza. Żadna podklasa nie powinna mieć możliwości mieszania się w to. Klasa `String` też jest finalna. Oznacza to, że nie można utworzyć jej podklas. Innymi słowy, jeśli mamy referencję do obiektu `String`, wiemy, że jest to obiekt klasy `String` i nic innego.

Według niektórych programistów wszystkie metody powinny być finalne, chyba że istnieje dobry powód, dla którego potrzebny jest w danym przypadku polimorfizm. W C++ i C# metody nie są polimorficzne, dopóki jawnie się tego nie zażąda. Może to jest w pewnym sensie skrajne podejście, ale zgadzamy się, że przy projektowaniu hierarchii klas należy poważnie rozważyć możliwość zastosowania modyfikatora `final` dla klas i metod.

Na początku istnienia Javy niektórzy programiści używali słowa kluczowego `final` w nadziei, że unikną narzutu powodowanego przez wiązanie dynamiczne. Jeśli metoda nie jest przesłonięta i jest krótka, kompilator może zoptymalizować jej wywołanie poprzez proces polegający na wstawieniu jej kodu w **miejsce wywołania** (ang. *inlining*). Na przykład wywołanie metody `e.getName()` zostałoby zastąpione dostępem do pola `e.name`. Jest to godna uwagi poprawa — procesory nie przepadają za rozgałęzianiem, ponieważ stoi ono w sprzeczności z ich strategią

pobierania zawczasu kolejnej funkcji podczas przetwarzania innej. Jeśli jednak metoda `getName` może być przesłonięta w innej klasie, kompilator nie może zastosować wstawiania kodu, ponieważ nie wie, jak działa ta przesłonięta wersja.

Na szczęście kompilator JIT w maszynie wirtualnej spisuje się lepiej niż zwykły kompilator. Wie dokładnie, które klasy są rozszerzeniem danej klasy, i ma możliwość sprawdzenia, czy dana metoda jest przesłonięta w którejś z tych klas. Jeśli metoda jest krótka, często wywoływana i nie jest przesłonięta, kompilator JIT może zastosować inlining. Co się stanie, jeśli maszyna wirtualna załaduje inną podklasę, która przesłania naszą metodę? Wtedy proces inliningu musi zostać cofnięty. Jest to operacja powolna, ale zdarza się bardzo rzadko.



Wyliczenia i rekordy zawsze są finalne, tzn. nie można ich rozszerzać.

5.1.8. Rzutowanie

Przypomnijmy z rozdziału 3., że proces wymuszania konwersji pomiędzy dwoma typami nazywa się rzutowaniem (ang. *casting*). W Javie dostępna jest specjalna notacja oznaczająca rzutowanie. Na przykład:

```
double x = 3.405;
int nx = (int) x;
```

W powyższym kodzie wartość zmiennej `x` została przekonwertowana na typ całkowitoliczbowy, co spowodowało utratę części ułamkowej.

Podobnie jak od czasu do czasu konieczna jest konwersja typu `double` na `int`, tak samo bywa, że trzeba przekonwertować referencję do obiektu jednej klasy na referencję do obiektu innej klasy. Po raz kolejny posłużę się przykładem tablicy zawierającej mieszankę obiektów klas `Employee` i `Manager`:

```
var staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 15);
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Do rzutowania referencji do obiektów używa się podobnej notacji jak do rzutowania typów liczbowych. Nazwę klasy docelowej należy umieścić w nawiasach i wstawić przed referencją, która ma być rzutowana. Na przykład:

```
Manager boss = (Manager) staff[0];
```

Tego typu rzutowanie może być potrzebne tylko w jednego rodzaju sytuacji — aby w pełni wykorzystać obiekt, którego typ został chwilowo zgubiony. Na przykład w klasie `TestManager` tablica `staff` musiała być typu `Employee`, ponieważ *niektóre* z przechowywanych w niej obiektów reprezentowały zwykłych pracowników. Aby uzyskać dostęp do nowych składowych obiektów klasy `Manager`, konieczne by było ich przekonwertowanie z powrotem na typ `Manager` (w zaprezentowanym wcześniej przykładowym kodzie specjalnie uniknęliśmy rzutowania, inicjalizując

zmienną `boss` obiektem klasy `Manager` przed zapisaniem jej w tablicy — aby ustawić dodatek do wypłaty, potrzebny jest odpowiedni typ obiektu).

Wiadomo, że każdy obiekt w Javie ma typ. Typ ten określa rodzaj obiektu, do którego odwołuje się zmienna, i wskazuje, co obiekt ten może robić. Na przykład zmienna `staff[i]` odwołuje się do obiektu klasy `Employee` (a więc może także odwoływać się do obiektu klasy `Manager`).

Kompilator sprawdza, czy programista nie wymaga zbyt wiele, zapisując wartość w zmiennej. Jeśli przypisze referencję do obiektu podklasy do zmiennej obiektowej nadklasy, zmniejsza swoje wymagania, więc kompilator bez problemu się na to zgodzi. Jeśli jednak referencja do obiektu nadklasy zostanie przypisana zmiennej obiektu podklasy, zwiększa swoje wymagania. W takim przypadku konieczne jest zastosowanie rzutowania, które umożliwi sprawdzenie tych wymagań w trakcie działania programu.

Co się stanie, jeśli programista spróbuje wykonać rzutowanie w dół łańcucha dziedziczenia i „oszuka” w kwestii zawartości obiektu?

```
Manager boss = (Manager) staff[1]; // Błąd
```

W trakcie działania programu systemy wykonawcze Javy wykryją niedorzeczne wymagania i wygenerują wyjątek `ClassCastException`. Jeśli wyjątek nie zostanie przechwycony, program zostanie zamknięty. W związku z tym do dobrych praktyk programistycznych należy sprawdzenie, czy rzutowanie się powiedzie, przed jego zastosowaniem. Do tego celu służy operator `instanceof`. Na przykład:

```
if (staff[1] instanceof Manager)
{
    boss = (Manager) staff[1];
    . . .
}
```

I wreszcie, kompilator nie pozwoli na rzutowanie, które nie ma szans powodzenia. Na przykład rzutowanie:

```
Date c = (Date) staff[1];
```

spowoduje błąd kompilacji, ponieważ `Date` nie jest podklasą klasy `Employee`.

Podsumowując:

- Rzutowanie jest możliwe wyłącznie w obrębie hierarchii dziedziczenia.
- Przy rzutowaniu typu nadklasy na typ podklasy należy sprawdzić wykonalność operacji za pomocą operatora `instanceof`.



Test:

```
x instanceof C
```

nie wygeneruje wyjątku, jeśli zmienna `x` ma wartość `null`, tylko zwróci wartość `false`. Sens tego zachowania polega na tym, że skoro `null` oznacza, iż referencja nie wskazuje na żaden obiekt, z pewnością nie odwołuje się do obiektu klasy `C`.

Faktem jest, że konwersja typu obiektu za pomocą rzutowania nie jest z reguły dobrym pomysłem. W naszym przykładzie rzadko potrzebne jest rzutowanie obiektu klasy `Employee` na obiekt klasy `Manager`. Metoda `getSalary` działa prawidłowo na obiektach obu tych klas. Wiązanie dynamiczne, na którym opiera się polimorfizm, automatycznie lokalizuje odpowiednią metodę.

Jedyna sytuacja, w której potrzebne jest takie rzutowanie, ma miejsce wtedy, gdy chcemy użyć metody dostępnej tylko dla obiektów klasy `Manager`, czyli `setBonus`. Jeśli wywołanie metody `setBonus` na rzecz obiektów klasy `Employee` okaże się konieczne, należy rozważyć możliwość, że nadklasa została źle zaprojektowana. Niewykluczone, że dobrym rozwiązaniem okaże się dodanie do nadklasy metody `setBonus`. Pamiętajmy, że do przerwania działania programu wystarczy jeden nieprzechwycony wyjątek. Zasadniczo najlepiej jest wystrzegać się rzutowania i operatora `instanceof`.



Składnia rzutowania w Javie pochodzi ze „starego i złego” języka C, natomiast działanie tego mechanizmu jest podobne do bezpiecznego rzutowania `dynamic_cast` w C++. Na przykład:

```
Manager boss = (Manager) staff[1]; // Java
```

jest odpowiednikiem:

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

Jest tylko jedna różnica. Jeśli rzutowanie nie powiedzie się, nie powstaje obiekt `null`, ale wyjątek. W tym sensie przypomina to znane z C++ rzutowanie *referencji*. Jest to jedna z bolączek. W C++ można zadbać o test typu i konwersję w jednej operacji.

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
if (boss != NULL) . . .
```

W Javie konieczne jest użycie operatora `instanceof` i zastosowanie rzutowania:

```
if (staff[1] instanceof Manager)
{
    Manager boss = (Manager) staff[1];
    . . .
}
```

5.1.9. Operator `instanceof` i dopasowywanie wzorców

Kod

```
if (staff[i] instanceof Manager)
{
    Manager boss = (Manager) staff[i];
    boss.setBonus(5000);
}
```

nie grzeszy zwięzłością. Czy naprawdę musimy powtarzać nazwę podklasy `Manager` aż trzy razy?

Od Javy 16 można to uprościć. Zmienną podklasy można zadeklarować wprost w teście operatora `instanceof`:

```

if (staff[i] instanceof Manager boss)
{
    boss.setBonus(5000);
}

```

Jeśli `staff[i]` jest egzemplarzem klasy `Manager`, to zostaje on przypisany zmiennej `boss` i można jej używać jako obiektu klasy `Manager`. Pominęliśmy rzutowanie.

Jeśli `staff[i]` nie odnosi się do obiektu klasy `Manager`, zmienna `boss` nie otrzymuje przypisania i operator `instanceof` zwraca wartość `false`. Treść głównej instrukcji `if` zostaje pominięta.



W większości sytuacji, w których jest używany operator `instanceof`, konieczne jest zastosowanie metody podklasy. W takich przypadkach, zamiast stosować rzutowanie, należy używać tej wersji operatora „z dopasowywaniem wzorców”.

Bezsensowne użycie wzorca operatora `instanceof` jest błędem:

```

Manager boss = . . . ;
if (boss instanceof Employee e) . . . // BŁĄD: to oczywiste, że to jest obiekt klasy Employee

```



Równie bezsensowna konstrukcja

```
if (boss instanceof Employee) . . .
```

jest dozwolona ze względu na zgodność z Javą 1.0.

Jeśli wzorec operatora `instanceof` wprowadza zmienną, to można jej od razu użyć w tym samym wyrażeniu:

```

Employee e;
if (e instanceof Manager m && m.getBonus() > 10000) . . .

```

To działa, ponieważ prawa strona wyrażenia `&&` jest obliczana tylko wtedy, gdy lewa strona jest prawdziwa. Jeśli prawa strona zostanie obliczona, to znaczy, że zmienna `m` została powiązana z egzemplarzem klasy `Manager`.

Jednak poniższy kod spowoduje błąd kompilacji:

```
if (e instanceof Manager m || m.getBonus() > 10000) . . . // BŁĄD
```

Prawa strona wyrażenia `||` jest wykonywana, gdy lewa strona jest fałszywa, a więc zmienna `m` nie jest z niczym powiązana.

Poniżej znajduje się kolejny przykład, w którym użyto operatora warunkowego:

```
double bonus = e instanceof Manager m ? m.getBonus() : 0;
```

Zmienna `m` jest zdefiniowana w części wyrażenia za znakiem `?`, ale nie ma jej w części za znakiem `:`.



Wersja operatora `instanceof` z deklaracją zmiennej to tzw. wersja „z dopasowywaniem wzorca”, ponieważ jest podobna do wzorców typów w konstrukcji `switch`, która na próbę została dodana do Javy 17. Nie opisuję szczegółowo próbnych konstrukcji, ale poniżej przedstawiam przykład składni:

```
String description = switch (e)
{
    case Executive exec -> "Dyrektor o efektownym tytule " + exec.getTitle();
    case Manager m -> "Kierownik, któremu przysługuje premia " + m.getBonus();
    default -> "Zwykły pracownik zarabiający " + e.getSalary();
}
```

Podobnie jak w przypadku wzorca operatora `instanceof` każdy wzorec typu deklaruje zmienną.



Zmienna lokalna zdefiniowana przez wzorec operatora `instanceof` jak każda zmienna lokalna może zasłaniać pole. Na przykład:

```
class Value
{
    private double v;
    public boolean equals(Value other)
    {
        if (other instanceof LabeledValue v)
            // v jest taka sama jak other
        else
            // v oznacza pole
        {
            . . .
        }
    }
}
```

5.1.10. Ograniczanie dostępu

Wiemy już, że najlepiej jest, kiedy pola metody są prywatne, a metody publiczne. Wszystko, co jest oznaczone słowem kluczowym `private`, jest niewidoczne dla innych klas. Na początku tego rozdziału dowiedzieliśmy się też, że powyższa zasada dotyczy także podklas — podklasa nie ma dostępu do pól prywatnych swojej nadklasy.

W niektórych sytuacjach konieczne jest ograniczenie widoczności metody tylko do podklas lub, rzadziej, zezwolenie metodom podklas na dostęp do pól nadklasy. W takim przypadku należy użyć słowa kluczowego `protected`. Jeśli na przykład klasa `Employee` zawiera pole `hireDay` zadeklarowane jako `protected`, a nie `private`, metody klasy `Manager` mają do tego pola bezpośredni dostęp.

W Javie pole chronione jest dostępne dla każdej klasy znajdującej się w tym samym pakiecie. Teraz wyobraź sobie podklasę `Administrator` z innego pakietu. Jej metody mogą zaglądać tylko do pola `hireDay` klasy `Administrator`, ale nie klasy `Employee`. Ograniczenie to ma zapobiec nadużywaniu mechanizmu ochrony w celu tworzenia podklas tylko po to, aby uzyskać dostęp do chronionych pól.

W praktyce z pól chronionych należy korzystać ostrożnie. Załóżmy, że nasza klasa, która zawiera pola chronione, jest używana przez innych programistów. Ci programiści mogą tworzyć bez naszej wiedzy klasy dziedziczące po naszej klasie i w ten sposób uzyskać dostęp do chronionych pól naszej klasy. W takiej sytuacji zmiana implementacji owej nadklasy pociągałaby za sobą problemy u wspomnianych programistów. Takie działanie jest sprzeczne z ideą OOP, która opiera się na hermetyzacji danych.

Więcej sensu ma tworzenie chronionych metod. Metoda może być chroniona, jeśli jej użycie może sprawiać problemy. Oznacza to, że podklasy (które prawdopodobnie dobrze znają swoich przodków) z pewnością użyją danej metody prawidłowo, podczas gdy metody innych klas niekoniecznie.

Dobrym przykładem tego rodzaju metody jest metoda `clone` z klasy `Object` — więcej szczegółów znajdziesz w rozdziale 6.



Składowe chronione klasy w Javie są widoczne dla wszystkich jej podklas i innych klas w pakiecie. Jest to nieco inne pojęcie ochrony niż w C++ i powoduje ono, że składowe chronione w Javie są jeszcze mniej bezpieczne niż w C++.

Oto zestawienie wszystkich czterech modyfikatorów dostępu Javy służących do kontroli widoczności:

1. `private` — widoczność w obrębie klasy;
2. `public` — widoczność wszędzie;
3. `protected` — widoczność w pakiecie i wszystkich podklasach;
4. widoczność w obrębie pakietu — (niefortunne) zachowanie domyślne, które nie wymaga żadnego modyfikatora.

5.2. Kosmiczna klasa wszystkich klas — `Object`

Klasa `Object` jest podstawą wszystkich pozostałych klas w Javie. Każda klasa w tym języku rozszerza klasę `Object`. Nigdy jednak nie trzeba pisać czegoś takiego:

```
public class Employee extends Object
```

Jeśli żadna nadklasa nie jest jawnie podana, to automatycznie jest nią klasa `Object`. Ponieważ *każda* klasa w Javie stanowi rozszerzenie klasy `Object`, trzeba się zapoznać z usługami świadczonymi przez tę klasę. W tym rozdziale opisujemy tylko podstawowe zagadnienia, a po szczegółowe informacje odsyłamy do kolejnych rozdziałów lub dokumentacji internetowej (niektóre metody klasy `Object` mogą być używane tylko podczas pracy z wątkami — więcej o wątkach dowiesz się w rozdziale 12.).

5.2.1. Zmienne typu Object

Za pomocą zmiennej typu `Object` można się odwoływać do wszystkich typów obiektów:

```
Object obj = new Employee("Henryk Kwiatek", 35000);
```

Oczywiście zmienna typu `Object` jest jedynie generycznym kontenerem dla dowolnych wartości. Aby zrobić z niej użytek, trzeba posiadać wiedzę na temat oryginalnego typu i wykonać rzutowanie:

```
Employee e = (Employee) obj;
```

W Javie tylko *typy podstawowe* (liczby, znaki i wartości logiczne) nie są obiektami.

Wszystkie typy tablicowe — bez względu na to, czy przechowują obiekty, czy typy podstawowe — są typami klasowymi rozszerzającymi klasę `Object`.

```
Employee[] staff = new Employee[10];
obj = staff;           // OK
obj = new int[10];     // OK
```



W języku C++ nie ma uniwersalnej klasy bazowej, jednak każdy wskaźnik można przekonwertować na wskaźnik `void*`.

5.2.2. Metoda equals

Dostępna w klasie `Object` metoda `equals` porównuje dwa obiekty. Jej implementacja w klasie `Object` sprawdza, czy dwie referencje do obiektów są identyczne. Jest to bardzo rozsądne działanie domyślne — jeśli dwa obiekty są identyczne, powinny być sobie równe. W przypadku wielu klas nie potrzeba nic więcej. Na przykład porównywanie dwóch obiektów `PrintStream` pod kątem równości nie ma większego sensu, jednak często potrzebne jest porównywanie stanów, czyli sytuacji, w której dwa obiekty są równe, jeśli mają ten sam stan.

Na przykład dwóch pracowników uznamy za równych, jeśli mają identyczne imię i nazwisko, pensję i zostali zatrudnieni w tym samym czasie (w prawdziwej bazie danych pracowników lepiej byłoby porównać identyfikatory pracowników; ten przykład ma na celu zobrazowanie mechanizmów implementacyjnych metody `equals`).

```
public class Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        // Szybkie sprawdzenie, czy obiekty są identyczne.
        if (this == otherObject) return true;

        // Musi zwrócić false, jeśli parametr jawny ma wartość null.
        if (otherObject == null) return false;

        // Jeśli klasy nie pasują, nie mogą być równe.
```

```

    if (getClass() != otherObject.getClass())
        return false;

    // Wiadomo, że otherObject nie jest obiektem null klasy Employee.
    Employee other = (Employee) otherObject;

    // Sprawdzenie, czy pola mają identyczne wartości.
    return name.equals(other.name)
        && salary == other.salary
        && hireDay.equals(other.hireDay);
}

```

Metoda `getClass` zwraca nazwę klasy obiektu — szczegółowy opis tej metody znajduje się w dalszej części tego rozdziału. W naszym teście dwa obiekty mogą być równe tylko wtedy, kiedy należą do tej samej klasy.



Aby zabezpieczyć się na wypadek, gdyby zmienne `name` lub `hireDay` były `null`, można użyć metody `Objects.equals`. Wywołanie `Objects.equals(a, b)` zwraca `true`, jeśli oba argumenty są `null`, `false` — jeśli jeden z argumentów jest `null`, a w pozostałych przypadkach wywołuje `a.equals(b)`. Przy użyciu tej metody ostatnią instrukcję w metodzie `Employee.equals` można zmienić następująco:

```

return Objects.equals(name, other.name)
    && salary == other.salary
    && Objects.equals(hireDay, other.hireDay);

```

Implementując metodę `equals` w podklasie, najpierw należy wywołać metodę `equals` należącą do nadklasy. Jeśli test zakończy się niepowodzeniem, obiekty nie mogą być równe. Jeśli pola nadklasy są równe, można porównywać składowe obiektów podklasy.

```

public class Manager extends Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        // Metoda super.equals sprawdziła, czy this i otherObject należą do tej samej klasy.
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}

```



Przypomnę z rozdziału 4., że rekord jest specjalnym rodzajem niezmiennej klasy, której stan jest w całości zdefiniowany przez pola ustawione w konstruktorze „kanonicznym”. Rekordy automatycznie definiują metodę `equals`, która porównuje pola. Dwa egzemplarze rekordu są równe, gdy odpowiednie wartości pól są równe.

5.2.3. Porównywanie a dziedziczenie

Jak powinna się zachować metoda `equals`, jeśli parametry jawny i niejawny nie należą do tej samej klasy? Jest to dość kontrowersyjna kwestia. W poprzednim przykładzie metoda `equals` zwracała wartość `false`, jeśli klasy nie były identyczne. Jednak wielu programistów zamiast tej metody używa operatora `instanceof`:

```
if (!(otherObject instanceof Employee)) return false;
```

Dzięki temu obiekt `otherObject` może należeć do podklasy klasy `Employee`. Metoda ta może jednak sprawiać problemy. Specyfikacja języka Java wymaga, aby metoda `equals` miała następujące własności:

- 1. Zwrotność:** `x.equals(x)` powinno zwracać `true`, jeśli `x` nie ma wartości `null`.
- 2. Symetria:** dla dowolnych referencji `x` i `y`, `x.equals(y)` powinno zwrócić wartość `true` wtedy i tylko wtedy, gdy `y.equals(x)` zwróci wartość `true`.
- 3. Przemienność:** dla dowolnych referencji `x`, `y` i `z`, jeśli `x.equals(y)` zwraca wartość `true` i `y.equals(z)` zwraca `true`, to `x.equals(z)` zwraca tę samą wartość.
- 4. Niezmiennosc:** jeśli obiekty, do których odwołują się zmienne `x` i `y`, nie zmieniły się, kolejne wywołania `x.equals(y)` zwracają tę samą wartość.
- 5.** Dla każdego `x` różnego od `null`, wywołanie `x.equals(null)` powinno zwrócić wartość `false`.

Powyższe reguły są oczywiście podyktowane zdrowym rozsądkiem. Programista biblioteki nie powinien być zmuszany do tracenia czasu na podejmowanie decyzji, czy wywołać `x.equals(y)`, czy `y.equals(x)`, aby zlokalizować jakiś element w strukturze danych.

Jednak w przypadku gdy parametry należą do różnych klas, reguła symetrii może pociągnąć za sobą pewne konsekwencje. Przeanalizujmy poniższe wywołanie:

```
e.equals(m)
```

gdzie `e` jest obiektem klasy `Employee`, a `m` — klasy `Manager`. Tak się składa, że każdy z nich ma takie samo imię i nazwisko, pensję i datę zatrudnienia. Jeśli wywołanie `Employee.equals` użyje operatora `instanceof`, zostanie zwrócona wartość `true`. Oznacza to jednak, że wywołanie odwrotne:

```
m.equals(e)
```

także musi zwrócić wartość `true` — reguła symetrii nie zezwala na zwrócenie wartości `false` ani spowodowanie wyjątku.

To krępuje klasę `Manager`. Jej metoda `equals` zmusza ją do porównywania się z klasą `Employee` bez brania pod uwagę informacji właściwych tylko kierownikom! Nagle operator `instanceof` przestał wydawać się tak atrakcyjny!

Niektórzy twierdzą, że test `getClass` jest błędny, ponieważ łamie regułę zamienialności. Często przytaczany jest przykład metody `equals` w klasie `AbstractSet`, która sprawdza, czy dwa zbiory

mają te same elementy. Klasa `AbstractSet` ma dwie konkretne podklasy: `TreeSet` i `HashSet`. Każda z nich lokalizuje elementy za pomocą innego algorytmu. Bez względu na implementację potrzebna jest możliwość porównywania zbiorów.

Jednak przykład zbioru dotyczy raczej wąskiej specjalizacji. Można by było zdefiniować metodę `AbstractSet.equals` jako finalną, ponieważ nikt nie powinien zmieniać semantyki równości zbiorów (obecnie metoda ta nie jest finalna, dzięki czemu możliwe jest zaimplementowanie w podklasie bardziej efektywnego algorytmu porównującego).

Z naszego punktu widzenia możliwe są dwa odrębne scenariusze:

- Jeśli podklasy mają własny mechanizm porównywania, reguła symetrii zmusza nas do użycia testu `getClass`.
- Jeśli mechanizm porównujący jest ustalony w nadklasie, można użyć operatora `instanceof`, pozwalając, aby obiekty różnych podklas były równe.

W przypadku klas `Employee` i `Manager` dwa obiekty uznajemy za równe, jeśli mają takie same pola. Jeśli dwa obiekty klasy `Manager` mają takie same imiona i nazwiska, pensje i daty zatrudnienia, ale różne dodatki do pensji, chcemy, aby były uznane za różne. Dlatego użyliśmy testu `getClass`.

Założmy jednak, że do porównywania użyliśmy identyfikatorów pracowników. Ten rodzaj porównania jest odpowiedni dla wszystkich podklas. W takim przypadku moglibyśmy zastosować operator `instanceof`, a metodę `Employee.equals` zadeklarować jako finalną.



Standardowa biblioteka Javy zawiera ponad 150 implementacji metody `equals`. Znajdują się wśród nich wersje z operatorem `instanceof`, wywołaniem `getClass`, przechwytywaniem wyjątku `ClassCastException` i nierobiące nic. W dokumentacji API klasy `java.sql.Timestamp` można znaleźć notatkę, w której implementatorzy sami ze wstydem przyznają, że zapędzili się w kozi róg. Klasa `java.sql.Timestamp` dziedziczy po klasie `java.util.Date`, której metoda `equals` wykorzystuje operator `instanceof`. Nie da się przesłonić metody `equals`, aby była dokładna i symetryczna.

Poniżej znajduje się opis procedury pisania idealnej metody `equals`:

1. Nadaj parametrowi `otherObject` nazwę `otherObject` — później konieczne będzie rzutowanie go na inną zmienną, która powinna mieć nazwę `other`.

2. Sprawdź, czy `this` i `otherObject` są identyczne:

```
if (this == otherObject) return true;
```

Ta instrukcja służy tylko do optymalizacji. Jest to dość często spotykany przypadek. Łatwiej sprawdzić tożsamość obiektów, niż porównywać ich pola.

3. Sprawdź, czy obiekt `otherObject` jest równy `null`, i zwróć wartość `false`, jeśli jest.

```
if (otherObject == null) return false;
```

4. Porównaj klasy obiektów `this` i `otherObject`. Jeśli semantyka metody `equals` może zmienić się w podklasach, użyj testu `getClass`:

```
if (getClass() != otherObject.getClass()) return false;
ClassName other = (ClassName) otherObject;
```

Jeśli *wszystkie* podklasy korzystają z tej samej semantyki, można użyć operatora `instanceof`:

```
if (!(otherObject instanceof ClassName)) return false;
```

Zwróć uwagę, że test operatora `instanceof` w przypadku powodzenia ustawia zmienną `other` na `otherObject`. Nie trzeba stosować rzutowania.

5. Rzutuj obiekt `otherObject` na zmienną typu swojej klasy:

```
ClassName other = (ClassName) otherObject
```

6. Porównaj pola zgodnie z własnymi wymaganiami. Dla pól typów podstawowych użyj operatora `==`, a metody `Objects.equals` dla obiektów. Zwróć wartość `true`, jeśli wszystkie pola się zgadzają, lub `false` w przeciwnym przypadku.

```
return field1 == other.field1
    && field2.equals(other.field2)
    && . . .;
```

Jeśli przeddefiniujesz metodę `equals` w podklasie, użyj odwołania `super.equals(other)`.



Do porównania elementów dwóch tablic można użyć statycznej metody `Arrays`.
 ➔ `equals`. W przypadku tablic wielowymiarowych należy używać metody `Arrays`.
 ➔ `deepEquals`.



Poniżej znajduje się często spotykany błąd w implementacji metody `equals`. Na czym on polega?

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return other != null
            && getClass() == other.getClass()
            && Objects.equals(name, other.name)
            && salary == other.salary
            && Objects.equals(hireDay, other.hireDay);
    }
    ...
}
```

Ta metoda deklaruje parametr jawny jako typ `Employee`. W wyniku tego nie przesłania ona metody `equals` z klasy `Object`, ale tworzy zupełnie nową metodę.

Można chronić się przed tego typu błędem, oznaczając metody mające przesłaniać metody nadklasy znacznikiem `@Override`:

```
@Override public boolean equals(Object other)
```

Jeśli zostanie popełniony błąd i zdefiniowana nowa metoda, kompilator zgłosi błąd. Przypuśćmy na przykład, że dodano poniższą deklarację do klasy `Employee`:

```
@Override public boolean equals(Employee other)
```

Zostanie zgłoszony błąd, ponieważ ta metoda nie przesłania żadnej metody w nadklasie `Object`.

java.util.Arrays 1.2

■ static boolean equals(`xxx[] a, xxx[] b`) 5

Zwraca wartość `true`, jeśli tablice są równe pod względem liczby elementów i elementy te są takie same na odpowiadających sobie pozycjach w obu tablicach. Tablice te mogą przechowywać elementy następujących typów: `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, `double`.

java.util.Objects 7

■ static boolean equals(`Object a, Object b`)

Zwraca wartość `true`, jeśli `a` i `b` są `null`, `false` — jeśli `a` lub `b` jest `null`, oraz `a.equals(b)` w pozostałych przypadkach.

5.2.4. Metoda `hashCode`

Wartość skrótu (ang. *hash code*) obiektu to obliczona dla niego specjalna liczba całkowita. Skróty powinny mieć różne wartości. To znaczy, że jeśli `x` i `y` to dwa różne obiekty, powinno istnieć wysokie prawdopodobieństwo, że `x.hashCode()` i `y.hashCode()` to dwie różne liczby. Tabela 5.1 przedstawia trzy przykładowe wartości skrótu zwrócone przez metodę `hashCode` klasy `String`.

Tabela 5.1. Wartości skrótu zwrócone przez metodę `hashCode`

Łańcuch	Wartość skrótu
Witaj	83588971
Henryk	-2137002381
Kwiatek	1350142454

Klasa `String` oblicza wartości skrótu za pomocą następującego algorytmu:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

Metoda `hashCode` znajduje się w klasie `Object`. Dlatego każdy obiekt ma domyślny skrót, który jest obliczany na podstawie adresu obiektu w pamięci. Przeanalizujmy poniższy kod:

```
var s = "Ok";
var sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
```

```
var t = new String("Ok");
var tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Wynik przedstawia tabela 5.2.

Tabela 5.2. Skróty łańcuchów i obiektów klasy `StringBuilder`

Obiekt	Wartość skrótu	Obiekt	Wartość
s	2556	t	2556
sb	20526976	tb	20527144

Należy zauważyć, że łańcuchy s i t mają takie same wartości skrótu, ponieważ skróty łańcuchów są pochodnymi ich *zawartości*. Obiekty sb i tb mają różne skróty, ponieważ dla klasy `String` ↪ `Builder` nie zdefiniowano metody `hashCode`. W związku z tym domyślna metoda `hashCode` klasy `Object` wygenerowała ich wartości skrótu na podstawie adresów w pamięci.

W przypadku przededefiniowania metody `equals` należy także przededefiniować metodę `hashCode` dla obiektów, które użytkownicy mogą wstawiać do tablicy skrótów (ang. *hash table*) — tablice skrótów zostały opisane w rozdziale 9.

Metoda `hashCode` powinna zwracać liczbę całkowitą (może być ujemna). Aby wartości skrótu różnych obiektów były różne, wystarczy użyć kombinacji skrótów pól tych obiektów.

Poniżej znajduje się przykładowa metoda `hashCode` klasy `Employee`:

```
public class Employee
{
    public int hashCode()
    {
        return 7 * name.hashCode()
            + 11 * Double.valueOf(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    . . .
}
```

Można tu jednak dokonać dwóch udoskonaleń. Po pierwsze, lepiej użyć zabezpieczonej przed `null` metody `Objects.hashCode`, która zwraca 0, jeśli jej argument jest `null`, albo wynik wywołania `hashCode` na tym argumencie w pozostałych przypadkach. Dodatkowo można użyć statycznej metody `Double.hashCode`, aby nie musieć tworzyć obiektu typu `Double`:

```
public int hashCode()
{
    return 7 * Objects.hashCode(name)
        + 11 * Double.hashCode(salary)
        + 13 * Objects.hashCode(hireDay);
}
```

Po drugie, gdy trzeba połączyć kilka wartości skrótu (ang. *hash value*), można wywołać metodę `Objects.hash`, przekazując jej wszystkie te wartości. Spowoduje to wywołanie metody `Objects.hashCode` dla każdego argumentu i połączenie wartości. Wówczas metoda `Employee.hashCode` może być o wiele prostsza:

```
public int hashCode()
{
    return Objects.hash(name, salary, hireDay);
}
```

Definicje metod `equals` i `hashCode` muszą się ze sobą zgadzać — jeśli `x.equals(y)` zwraca wartość `true`, to `x.hashCode()` musi mieć taką samą wartość jak `y.hashCode()`. Jeśli na przykład metoda `Employee.equals` porównuje identyfikatory pracowników, metoda `hashCode` musi obliczać skróty na podstawie identyfikatorów, a nie imion i nazwisk pracowników czy adresów w pamięci.



Jeśli pola są typu tablicowego, można użyć metody `Arrays.hashCode`, która oblicza skrót złożony ze skrótów elementów tablicy.



Typ `record` automatycznie dostarcza metodę `hashCode`, która generuje kod skrótu z kodów skrótu wartości pól.



Jeśli zmienne egzemplarza mają małe zakresy możliwych wartości, to należy osiągnąć możliwie jak najwięcej osobnych kodów skrótu. Weźmy na przykład daty z kalendarza. Operacja $7 * \text{rok} + 11 * \text{miesiąc} + 13 * \text{dzień}$ generuje wiele kolizji. Natomiast $31 * 12 * \text{rok} + 31 * \text{miesiąc} + \text{dzień}$ jest „idealną funkcją skrótu”. Przy założeniu rozsądnego zakresu lat żadne dwie daty nie mają takiego samego kodu skrótu. (Metoda `hashCode` z klasy `LocalDate`, która obsługuje zakres $\pm 999\,999\,999$ lat, jest nieco bardziej skomplikowana).

`java.lang.Object` **1.0**

■ `int hashCode()`

Zwraca wartość skrótu dla obiektu. Skrót ten może być każdą dodatnią lub ujemną liczbą całkowitą. Identyczne obiekty powinny mieć takie same skróty.

`java.lang.Objects` **7**

■ `int hash(Object... objects)`

Zwraca wartość skrótu będącą kombinacją wartości skrótu wszystkich podanych obiektów.

■ `static int hashCode(Object a)`

Zwraca 0, jeśli `a` jest `null`, lub `a.hashCode()` w pozostałych przypadkach.

`java.lang.(Integer|Long|Short|Byte|Double|Float|Character|Boolean)` **1.0**

■ `static int hashCode(XXX value)` **8**

Zwraca skrót podanej wartości. W tym przypadku `xxx` oznacza typ podstawowy odpowiadający danemu typowi opakowania.


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

■ `static int hashCode(XXX[] a)` 5

Oblicza skrót tablicy `a`, która może przechowywać elementy następujących typów: `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, `double`.

5.2.5. Metoda `toString`

Kolejną ważną metodą klasy `Object` jest metoda `toString`, która zwraca obiekt reprezentujący wartość obiektu. Z typowym przykładem jej zastosowania mamy do czynienia, gdy metoda `toString` klasy `Point` zwraca następujący łańcuch:

```
java.awt.Point[x=10,y=20]
```

Większość metod `toString` (ale nie wszystkie) ma następujący format: nazwa klasy plus wartości pól wymienione w nawiasach kwadratowych. Poniżej znajduje się implementacja metody `toString` w klasie `Employee`:

```
public String toString()
{
    return "Employee[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

Można to jednak zrobić nieco lepiej. Zamiast na sztywno wpisywać nazwę klasy w metodzie `toString`, nazwę tę można pobrać za pomocą wywołania `getClass().getName()`.

```
public String toString()
{
    return getClass().getName()
        + "[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

Dzięki temu metoda ta będzie działała także w podklasach.

Oczywiście twórca podklasy powinien zdefiniować własną metodę `toString`, uwzględniając pola tej klasy. Jeśli w nadklasie użyto wywołania `getClass().getName()`, w podklasie można użyć wywołania `super.toString()`. Poniżej znajduje się przykładowa metoda `toString` klasy `Manager`:

```
public class Manager extends Employee
{
    ...
    public String toString()
    {
        return super.toString()
            + "[bonus=" + bonus
            + "];"
    }
}
```

Wydruk zawartości obiektu Manager wyglądałby następująco:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Metoda `toString` jest bardzo często używana z jednego ważnego powodu: za każdym razem, gdy obiekt jest łączony z łańcuchem za pomocą operatora `+`, kompilator automatycznie wywołuje metodę `toString`, aby utworzyć łańcuchową reprezentację obiektu. Na przykład:

```
var p = new Point(10, 20);
String message = "Aktualne położenie to " + p;
// Automatyczne wywołanie p.toString().
```



Zamiast `x.toString()` można napisać `" " + x`. Ta instrukcja łączy pusty łańcuch z łańcuchową reprezentacją `x`, czyli robi dokładnie to samo co `x.toString()`. W przeciwieństwie do metody `toString`, ta instrukcja działa nawet wtedy, gdy `x` jest typu podstawowego.

Jeśli `x` jest dowolnego typu, w wywołaniu:

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

metoda `println` wywołuje `x.toString()` i drukuje powstały w ten sposób łańcuch.

Klasa `Object` zawiera metodę `toString`, która drukuje nazwę klasy i skrót obiektu. Na przykład wywołanie:

```
System.out.println(System.out)
```

zwróci wynik podobny do poniższego:

```
java.io.PrintStream@187aeca
```

Jest to spowodowane tym, że programista implementujący klasę `PrintStream` nie zadał sobie trudu, aby przesłonić metodę `toString`.



Niestety tablice dziedziczą metodę `toString` po klasie `Object`, przez co typ tablicy jest drukowany w przestarzałym formacie. Na przykład:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

Powyższy kod zwraca łańcuch typu `[I@e48e1b` (przedrostek `[I` oznacza tablicę liczb całkowitych). Można temu zaradzić poprzez wywołanie metody `Arrays.toString`. Poniższy kod:

```
String s = Arrays.toString(luckyNumbers);
```

zwróci łańcuch `[2, 3, 5, 7, 11, 13]`.

Aby prawidłowo wydrukować zawartość tablicy wielowymiarowej (tzn. tablicy tablic), należy użyć metody `Arrays.deepToString`.

Metoda `toString` jest doskonałym narzędziem do rejestracji danych. Wiele klas biblioteki standardowej zawiera metodę `toString`, która umożliwia uzyskanie informacji na temat stanu obiektu. Jest to szczególnie przydatne w przypadku komunikatów rejestracyjnych typu:

```
System.out.println("Aktualne położenie = " + position);
```

Jak piszemy w rozdziale 7., jeszcze lepszym rozwiązaniem jest użycie obiektu klasy `Logger` i zastosowanie następującego wywołania:

```
Logger.global.info("Aktualne położenie = " + position);
```



Zdecydowanie zalecam dodawanie metody `toString` do każdej nowej klasy. Każdy, kto będzie tych klas używał, z pewnością doceni ułatwienia dotyczące rejestracji danych.

W typach `record` metoda `toString` jest od razu dostępna. Zwraca ona nazwę klasy oraz nazwy i zamienione na łańcuchy wartości pól.

Program z listingu 5.4 testuje metody `equals`, `hashCode` oraz `toString` w klasach `Employee` (listing 5.5) i `Manager` (listing 5.6).

Listing 5.4. `equals/EqualsTest.java`

```
package equals;

/**
 * Jest to program demonstrujący użycie metody equals.
 * @version 1.12 2012-01-26
 * @author Cay Horstmann
 */
public class EqualsTest
{
    public static void main(String[] args)
    {
        var alice1 = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
        var alice2 = alice1;
        var alice3 = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
        var bob = new Employee("Bartosz Borkowski", 50000, 1989, 10, 1);

        System.out.println("alice1 == alice2: " + (alice1 == alice2));

        System.out.println("alice1 == alice3: " + (alice1 == alice3));

        System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));

        System.out.println("alice1.equals(bob): " + alice1.equals(bob));

        System.out.println("bob.toString(): " + bob);

        var carl = new Manager("Karol Krakowski", 80000, 1987, 12, 15);
        var boss = new Manager("Karol Krakowski", 80000, 1987, 12, 15);
        boss.setBonus(5000);
        System.out.println("boss.toString(): " + boss);
        System.out.println("carl.equals(boss): " + carl.equals(boss));
    }
}
```

```
        System.out.println("alice1.hashCode(): " + alice1.hashCode());
        System.out.println("alice3.hashCode(): " + alice3.hashCode());
        System.out.println("bob.hashCode(): " + bob.hashCode());
        System.out.println("carl.hashCode(): " + carl.hashCode());
    }
}
```

Listing 5.5. equals/Employee.java

```
package equals;

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

public class Employee
{
    private String name;
    private double salary;
    private LocalDate hireDay;

    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public LocalDate getHireDay()
    {
        return hireDay;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    public boolean equals(Object otherObject)
    {
        // Sprawdzenie, czy obiekty są identyczne
        if (this == otherObject) return true;

        // Musi zwrócić false, jeśli jawny parametr jest null
        if (otherObject == null) return false;

        // Jeśli klasy nie zgadzają się, nie mogą być jednakowe
        if (getClass() != otherObject.getClass()) return false;
```

```

// Teraz wiadomo, że otherObject jest typu Employee i nie jest null
var other = (Employee) otherObject;

// Sprawdzenie, czy pola mają identyczne wartości
return Objects.equals(name, other.name) && salary == other.salary &&
    ↳ Objects.equals(hireDay, other.hireDay);
}

public int hashCode()
{
    return Objects.hash(name, salary, hireDay);
}

public String toString()
{
    return getClass().getName() + "[name=" + name + ",salary=" + salary +
        ↳ ",hireDay=" + hireDay
        + "];"
}
}

```

Listing 5.6. equals/Manager.java

```

package equals;

public class Manager extends Employee
{
    private double bonus;

    public Manager(String n, double s, int year, int month, int day)
    {
        super(n, s, year, month, day);
        bonus = 0;
    }

    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
        this.bonus = bonus;
    }

    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        var other = (Manager) otherObject;
        // Metoda super.equals określiła, że obiekty należą do tej samej klasy
        return bonus == other.bonus;
    }

    public int hashCode()
    {
        return super.hashCode() + 17 * new Double(bonus).hashCode();
    }
}

```

```
    }

    public String toString()
    {
        return java.util.Objects.hash(super.hashCode(), bonus);
    }
}
```

java.lang.Object 1.0**■ Class getClass()**

Zwraca obiekt `Class` zawierający informacje dotyczące klasy obiektu. W dalszej części tego rozdziału dowiemy się, że w Javie istnieje zamknięta w klasie `Class` reprezentacja czasu wykonywania klas.

■ boolean equals(Object otherObject)

Porównuje dwa obiekty. Zwraca wartość `true`, jeśli oba obiekty wskazują ten sam obszar pamięci, lub `false` w przeciwnym przypadku. We własnych klasach powinno się tę metodę przesłaniać.

■ String toString()

Zwraca łańcuch reprezentujący wartość obiektu. We własnych klasach powinno się tę metodę przesłaniać.

java.lang.Class 1.0**■ String getName()**

Zwraca nazwę klasy.

■ Class getSuperClass()

Zwraca nadklasę danej klasy w postaci obiektu klasy `Class`.

5.3. Generyczne listy tablicowe

W niektórych językach programowania, zwłaszcza w C, rozmiary wszystkich tablic muszą być ustalone w czasie kompilacji. Nie podoba się to programistom, ponieważ zmusza ich to do niewygodnych kompromisów. Ilu pracowników będzie zatrudniał dany dział? Z pewnością nie więcej niż 100. Co zrobić, jeśli jeden dział będzie zatrudniać aż 150 pracowników? Czy dla każdego działu z tylko 10 pracownikami konieczne jest marnowanie 90 pozostałych miejsc?

W Javie sytuacja ta przedstawia się nieco lepiej. Rozmiar tablicy można ustawić w trakcie działania programu.

```
int actualSize = . . . ;
var staff = new Employee[actualSize];
```

Oczywiście powyższy fragment kodu nie rozwiązuje w pełni problemu dynamicznej zmiany rozmiaru tablic w trakcie działania programu. Kiedy rozmiar tablicy jest ustalony, nie można go łatwo zmienić. W Javie najprostszy sposób na poradzenie sobie z taką często spotykaną sytuacją jest użycie klasy o nazwie `ArrayList`. Obiekty tej klasy są podobne do tablic, z tą różnicą, że automatycznie dostosowują swoje rozmiary w wyniku dodawania i odejmowania elementów. Nie wymaga to żadnych modyfikacji kodu.

`ArrayList` jest **klasą generyczną z parametrem typu**. Aby określić typ obiektów przechowywanych przez listę tablicową, należy podać nazwę klasy w nawiasach ostrych, np. `ArrayList <Employee>`. Sposób definiowania klas generycznych został opisany w rozdziale 8., choć znajomość tych technik nie jest konieczna, aby móc używać typu `ArrayList`.

W kolejnych punktach dowiesz się, jak posługiwać się listami tablicowymi.

5.3.1. Deklarowanie list tablicowych

Poniższy kod deklaruje i tworzy listę tablicową przechowującą obiekty klasy `Employee`:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Od Javy 10 dobrym pomysłem jest używanie słowa `var`, aby uniknąć powielania nazwy klasy:

```
var staff = new ArrayList<Employee>();
```

Jeśli nie używasz słowa kluczowego `var`, parametr po prawej stronie możesz opuścić.

```
ArrayList<Employee> staff = new ArrayList<>();
```

Jest to tzw. składnia diamentowa (ang. *diamond syntax*), nazwana tak ze względu na to, że pusty nawias `<>` przypomina kształtem diament. Należy ją stosować w połączeniu z operatorem `new`. Kompilator sprawdza, co się dzieje z nową wartością. Jeżeli zostaje przypisana do zmiennej, przekazana do metody lub zwrócona przez metodę, to kompilator sprawdza generyczny typ tej zmiennej lub tego parametru albo tej metody. Następnie wstawia ten typ w nawias `<>`. W przedstawionym przykładzie znajduje się przypisanie `new ArrayList()` do zmiennej typu `ArrayList <Employee>`, więc typ generyczny to `Employee`.



Jeśli deklarujesz listę `ArrayList` przy użyciu słowa kluczowego `var`, nie stosuj składni diamentowej. Deklaracja:

```
var elements = new ArrayList<>();
```

da w wyniku `ArrayList<Object>`.



Przed wersją 5.0 w Javie nie było klas generycznych. Zamiast tego była sama klasa `ArrayList`, która mogła przechowywać elementy typu `Object`. Nadal można używać klasy `ArrayList` bez `<...>`. Jest to tak zwany typ surowy (ang. *raw*), w którym usunięto parametr typu.



W jeszcze starszych wersjach Javy tablice dynamiczne tworzone za pomocą klasy `Vector`. Klasa `ArrayList` jest jednak bardziej efektywna i nie ma powodu do używania starej klasy `Vector`.

Nowe elementy do listy są dodawane za pomocą metody `add`. Poniższy fragment kodu uzupełnia listę tablicową pracownikami:

```
staff.add(new Employee("Henryk Kwiatek", . . .));
staff.add(new Employee("Waldemar Kowalski", . . .));
```

Lista tablicowa zawiera wewnętrzną tablicę referencji do obiektów. Kiedy skończy się miejsce w tej tablicy, lista wykonuje swoje magiczne sztuczki. Jeśli wewnętrzna tablica jest pełna i zostanie wywołana metoda `add`, lista automatycznie utworzy większą tablicę i skopiuje do niej wszystkie obiekty.

Jeśli liczba elementów, które będą przechowywane w tablicy, jest znana, przynajmniej w przybliżeniu, przed uzupełnieniem listy należy wywołać metodę `ensureCapacity`.

```
staff.ensureCapacity(100);
```

Powyższe wywołanie zarezerwuje miejsce dla wewnętrznej tablicy mogącej przechowywać 100 elementów. Dzięki temu 100 pierwszych wywołań metody `add` nie spowoduje czasochłonnej realokacji.

Początkową pojemność listy można także przekazać do konstruktora klasy `ArrayList`:

```
ArrayList<Employee> staff = new ArrayList<>(100);
```



Alokacja listy tablicowej:

```
new ArrayList<>(100) // Pojemność 100
```

nie jest tym samym co alokacja nowej tablicy:

```
new Employee[100] // Rozmiar 100
```

Pomiędzy pojemnością listy tablicowej a rozmiarem tablicy jest duża różnica. Tablica o rozmiarze 100 zawiera 100 miejsc, w których może przechowywać dane. Lista tablicowa o pojemności 100 ma *możliwość* przechowywania 100 elementów (a nawet więcej kosztem dodatkowej realokacji). Jednak na początku, nawet po jej utworzeniu, lista tablicowa nie zawiera żadnych elementów.

Metoda `size` sprawdza liczbę elementów w liście tablicowej. Na przykład wywołanie:

```
staff.size()
```

zwróci bieżącą liczbę elementów w liście tablicowej `staff`. To wywołanie jest odpowiednikiem wywołania:

```
a.length
```

dla tablicy `a`.

Kiedy jest już pewne, że rozmiar listy tablicowej się nie zmieni, można wywołać metodę `trimToSize`. Metoda ta dostosowuje rozmiar bloku pamięci przechowującego listę dokładnie do aktualnego rozmiaru listy. Nadmiar pamięci zostanie wyczyszczony przez system zbierania nieużytków.

Jeśli po dopasowaniu rozmiaru listy zostaną dodane do niej nowe elementy, blok ponownie zostanie przeniesiony, co zabiera czas. Metody `trimToSize` należy używać wyłącznie wtedy, gdy jest pewne, że do listy tablicowej nie będą dodawane już nowe elementy.



Klasa `ArrayList` przypomina dostępny w C++ szablon `vector`. Jedno i drugie jest typem generycznym. Ale szablon `vector` przeciąża operator `[]`, dzięki czemu dostęp do elementów jest wygodniejszy. Ponieważ w Javie nie można przeciążać operatorów, trzeba używać do tego celu metod. Ponadto wektory w C++ są kopiowane przez wartość. Jeśli `a` i `b` są wektorami, przypisanie `a = b` tworzy nowy wektor `a` o takiej samej długości jak `b` i kopiuje wszystkie elementy z `b` do `a`. Takie samo przypisanie w Javie powoduje, że zarówno `a`, jak i `b` odwołują się do tej samej listy tablicowej.

`java.util.ArrayList<E>` 1.2

■ `ArrayList<E>()`

Tworzy pustą listę tablicową.

■ `ArrayList<E>(int initialCapacity)`

Tworzy pustą listę tablicową o podanej pojemności.

■ `boolean add(E obj)`

Dodaje obiekt na końcu listy. Zawsze zwraca wartość `true`.

■ `int size()`

Zwraca liczbę elementów aktualnie przechowywanych w liście (wartość ta nie może być większa niż pojemność listy).

■ `void ensureCapacity(int capacity)`

Zapewnia, że lista będzie miała wystarczającą pojemność do przechowywania danej liczby elementów, bez potrzeby realokacji wewnętrznej tablicy.

■ `void trimToSize()`

Redukuje pojemność listy do jej aktualnego rozmiaru.

5.3.2. Dostęp do elementów listy tablicowej

Niestety nie ma nic za darmo. Ceną za wygodę związaną z automatycznym powiększaniem się listy tablicowej jest bardziej skomplikowany dostęp do jej elementów. Powód jest taki, że klasa `ArrayList` nie należy do języka Java — została utworzona przez jakiegoś programistę i dodana do standardowej biblioteki.

Zamiast składni z operatorem `[]`, aby uzyskać dostęp do obiektów w celu ich odczytania lub modyfikacji, konieczne jest stosowanie metod `get` i `set`.

Aby na przykład ustawić wartość i -tego elementu, należy napisać:

```
staff.set(i, harry);
```

Powyższy zapis jest odpowiednikiem poniższego:

```
a[i] = harry;
```

dla tablicy `a` (tak samo jak w tablicach, numerowanie indeksów zaczyna się od zera).



Nie należy wywoływać `list.set(i, x)`, jeśli *rozmiar* tablicy nie jest większy od i . Na przykład poniższy kod jest błędny:

```
var list = new ArrayList<Employee>(100);    // Pojemność 100, rozmiar 0
list.set(0, x);                             // Nie ma jeszcze elementu 0
```

Do zapewniania tablicy używaj metody `add` zamiast `set`, którą należy stosować tylko w celu podmiany wcześniej dodanego elementu.

Aby pobrać wartość elementu listy tablicowej, należy napisać:

```
Employee e = staff.get(i);
```

Zapis ten jest odpowiednikiem poniższego:

```
Employee e = a[i];
```



Gdy nie było generycznych klas, metoda `get` surowej klasy `ArrayList` nie miała innego wyjścia, jak zwracać obiekty klasy `Object`. Z tego powodu wywołujący tę metodę musiał rzutować zwróconą wartość na odpowiedni typ:

```
Employee e = (Employee) staff.get(i);
```

Ponadto surowa klasa `ArrayList` nie jest w pełni bezpieczna. Jej metody `add` i `set` przyjmują obiekty każdego typu. Poniższe wywołanie:

```
staff.set(i, "Henryk Kwiatek");
```

w czasie kompilacji spowoduje tylko ostrzeżenie, a problemy zaczną się dopiero po uzyskaniu obiektu i próbie rzutowania go. Przy użyciu `ArrayList<Employee>` kompilator wykryje ten błąd.

Czasami możliwe jest wzięcie tego, co najlepsze, z tablic i list tablicowych — elastyczności i wygodnego dostępu do elementów. Trzeba zastosować następującą sztuczkę. Należy utworzyć listę tablicową i wstawić do niej wszystkie potrzebne elementy:

```
var list = new ArrayList<X>();
while ( . . . )
{
    x = . . . ;
    list.add(x);
}
```

Następnie wszystkie elementy należy skopiować do tablicy za pomocą metody `toArray`:

```
var a = new X[list.size()];
list.toArray(a);
```

Aby dodać element w środku listy, należy użyć metody `add` z parametrem określającym indeks:

```
int n = staff.size() / 2;
staff.add(n, e);
```

Elementy znajdujące się na pozycjach od `n` do góry są przesuwane, aby zrobić miejsce dla nowego elementu. Jeśli nowy rozmiar listy przekracza jej pojemność, następuje realokacja wewnętrznej tablicy.

Podobnie ze środka listy można usunąć dowolny element:

```
Employee e = staff.remove(n);
```

Elementy znajdujące się nad nim zostaną skopiowane w dół, a rozmiar tablicy zostanie zmniejszony o jeden.

Operacje wstawiania i usuwania elementów nie należą do najefektywniejszych. W przypadku małych list nie ma raczej czym się przejmować, ale jeśli elementów jest dużo i są one często wstawiane do środka kolekcji i z niej usuwane, należy rozważyć użycie listy dwukierunkowej (ang. *linked list*). Zagadnienia związane z listami dwukierunkowymi zostały poruszone w rozdziale 9.

Zawartość listy tablicowej można przemierzać za pomocą pętli typu `for each`:

```
for (Employee e : staff)
    działania związane z e
```

Ta pętla daje taki sam rezultat jak poniższa:

```
for (int i = 0; i < staff.size(); i++)
{
    Employee e = staff.get(i);
    działania związane z e
}
```

Listing 5.7 przedstawia zmodyfikowaną wersję programu *EmployeeTest* z rozdziału 4. Tablica `Employee[]` została zastąpiona listą tablicową `ArrayList<Employee>`. Należy zwrócić uwagę na następujące zmiany:

- Nie ma konieczności określenia rozmiaru tablicy.
- Za pomocą metody `add` można dodać dowolną liczbę elementów.
- Do sprawdzenia liczby elementów została użyta metoda `size()` zamiast metody `length`.
- Dostęp do elementu daje wywołanie `a.get(i)` zamiast `a[i]`.

Listing 5.7. `arrayList/ArrayListTest.java`

```
package arrayList;

import java.util.*;
```

```
/**
 * Ten program demonstruje użycie klasy ArrayList.
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class ArrayListTest
{
    public static void main(String[] args)
    {
        // Wstawienie do listy staff trzech obiektów klasy Employee.
        var staff = new ArrayList<Employee>();

        staff.add(new Employee("Karol Krakowski", 75000, 1987, 12, 15));
        staff.add(new Employee("Henryk Kwiatek", 50000, 1989, 10, 1));
        staff.add(new Employee("Waldemar Kowalski", 40000, 1990, 3, 15));

        // Zwiększenie pensji wszystkich pracowników o 5%.
        for (Employee e : staff)
            e.raiseSalary(5);

        // Drukowanie informacji o wszystkich obiektach Employee.
        for (Employee e : staff)
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary()
                + ",hireDay="
                + e.getHireDay());
    }
}
```

java.util.ArrayList<E> 1.2**■ E set(int index, T obj)**

Wstawia wartość obj do listy tablicowej w miejscu o określonym indeksie, zwraca poprzednią zawartość.

■ E get(int index)

Pobiera wartość zapisaną w określonym indeksie.

■ void add(int index, E obj)

Przesuwa elementy, aby dodać obj w określonym miejscu.

■ E remove(int index)

Usuwa element o określonym indeksie i przesuwają w dół wszystkie elementy, które znajdują się nad nim. Usunięty element jest zwracany.

5.3.3. Zgodność pomiędzy typowanymi a surowymi listami tablicowymi

Pisząc własny program, ze względów bezpieczeństwa należy zawsze używać parametrów typu. W tej części dowiesz się, jak korzystać ze starego kodu, w którym parametry te nie zostały użyte.

Mając poniższą starą klasę:

```
public class EmployeeDB
{
    public void update(ArrayList list) { ... }
    public ArrayList find(String query) { ... }
}
```

listę tablicową z typem można przekazać do metody update bez rzutowania.

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

Do metody update przekazywany jest obiekt staff.



Mimo że kompilator nie zgłasza żadnego błędu ani ostrzeżenia, to wywołanie nie jest w pełni bezpieczne. Metoda update może dodać do listy tablicowej elementy, które nie są typu Employee. Kiedy te elementy są pobierane, powstaje wyjątek. Mimo że brzmi to strasznie, działanie to jest dokładnie takie samo jak przed Java SE 5.0. Integralność maszyny wirtualnej nigdy nie jest zagrożona. W tej sytuacji nie zostaje naruszone bezpieczeństwo, ale nie ma też żadnych korzyści z testów przeprowadzanych w czasie kompilacji.

Jeśli natomiast obiekt surowej klasy ArrayList zostanie przypisany do typu z parametrem, zostanie wyświetlone ostrzeżenie.

```
ArrayList<Employee> result = employeeDB.find(query); // Powoduje ostrzeżenie
```



Aby ostrzeżenie się pojawiło, należy podać kompilatorowi opcję `-Xlint:unchecked`.

Zastosowanie rzutowania nie powoduje zniknięcia ostrzeżenia.

```
ArrayList<Employee> result = (ArrayList<Employee>)
employeeDB.find(query); // Powoduje kolejne ostrzeżenie
```

Zostanie wyświetlone inne ostrzeżenie informujące, że rzutowanie może wprowadzać w błąd.

Jest to spowodowane niezbyt udanym ograniczeniem typów generycznych w Javie. Ze względów zgodności kompilator konwertuje wszystkie listy tablicowe z typem na surowe obiekty klasy ArrayList po uprzednim sprawdzeniu, że reguły dotyczące typów nie zostały złamane. W działającym programie wszystkie listy tablicowe są takie same — w maszynie wirtualnej nie ma parametrów określających typ. W związku z tym rzutowania (ArrayList) i (ArrayList<Employee>) powodują przeprowadzenie identycznych sprawdzeń w trakcie działania programu.

Niewiele można z tym zrobić. Pracując ze starszym kodem, należy przyglądać się ostrzeżeniom zgłaszanym przez kompilator i pocieszać się tym, że nie mają one wielkiego znaczenia.

Gdy już osiągniesz zadowalający rezultat, możesz oznaczyć rzutowaną zmienną adnotacją @SuppressWarnings("unchecked"):

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =
(ArrayList<Employee>) employeeDB.find(query); // Powoduje zgłoszenie kolejnego ostrzeżenia
```

5.4. Opakowania obiektów i automatyczne pakowanie

Od czasu do czasu konieczna jest konwersja typu podstawowego, jak `int`, na obiekt. Każdy typ podstawowy ma swój odpowiednik w postaci klasy. Na przykład typowi `int` odpowiada klasa `Integer`. Tego typu klasy często są nazywane klasami **opakowującymi** (ang. *wrapper*). Nazwy klas opakowujących są oczywiste: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` i `Boolean` (sześć pierwszych dziedziczy po wspólnej nadklasie `Number`). Klasy te są niezmiennie, tzn. nie można zmienić opakowanej wartości po utworzeniu opakowania. Ponadto są finalne, co oznacza, że nie można tworzyć ich podklas.

Załóżmy, że potrzebujemy tablicy liczb całkowitych. Niestety parametr typu w nawiasach ostrych nie może określać typu podstawowego. Nie można utworzyć listy `ArrayList<int>`. W takiej sytuacji przydatna okazuje się klasa opakowująca. Listę obiektów klasy `Integer` można zadeklarować bez problemu.

```
var list = new ArrayList<Integer>();
```



Lista `ArrayList<Integer>` jest znacznie mniej wydajna niż tablica `int[]`, ponieważ każda wartość jest osobno zapakowana wewnątrz obiektu. Tego typu konstrukcji należy używać wyłącznie w przypadku małych kolekcji, kiedy wygoda programisty jest ważniejsza od wydajności.

Na szczęście istnieje technika umożliwiająca łatwe dodawanie i pobieranie elementów z tablicy. Wywołanie:

```
list.add(3);
```

jest automatycznie konwertowane na:

```
list.add(new Integer(3));
```

Tego typu konwersja nazywa się **automatycznym opakowywaniem** (ang. *autoboxing*).



Mogłoby się wydawać, że bardziej odpowiednim terminem byłoby **autowrapping**, ale człon **boxing** został przejęty z języka C#.

Jeśli natomiast obiekt klasy `Integer` zostanie przypisany do wartości `int`, zostanie automatycznie odpakowany. To znaczy kompilator przekonwertuje:

```
int n = list.get(i);
```

na:

```
int n = list.get(i).intValue();
```

Automatyczne opakowywanie i odpakowywanie działa nawet w przypadku operacji arytmetycznych. Można na przykład zastosować do referencji do obiektu opakowującego operator inkrementacji:

```
Integer n = 3;
n++;
```

Kompilator automatycznie wstawi instrukcje odpakowujące obiekt, zwiększające opakowaną w nim wartość i opakowującą ją z powrotem.

W większości przypadków wydaje się, że typy podstawowe i ich opakowania są jednym i tym samym. Jest między nimi tylko jedna znacząca różnica: tożsamość. Jak wiadomo, operator == zastosowany do obiektów opakowujących sprawdza tylko, czy obiekty te mają identyczne lokalizacje w pamięci. W związku z tym poniższe porównanie prawdopodobnie zakończyłoby się niepowodzeniem:

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

Jednak implementacja Javy *może*, jeśli tak zdecyduje, opakować często pojawiające się wartości w identyczne obiekty i wtedy takie porównanie zakończyłoby się powodzeniem. Taka dwuznaczność nie jest pożądana. Rozwiązaniem problemu jest porównywanie obiektów opakowujących za pomocą metody equals.



Specyfikacja automatycznego opakowywania wymaga, aby typy boolean, byte, char o wartościach mniejszych bądź równych 127 oraz short i int w przedziale od -128 do 127 były opakowywane w ustalone obiekty. Jeśli na przykład w powyższym fragmencie kodu a i b zostałyby zainicjalizowane wartością 100, porównywanie musiałoby się zakończyć powodzeniem.



Nigdy nie polegaj na tożsamości obiektów opakowujących. Nie porównuj ich za pomocą operatora == i nie używaj ich jako blokad (rozdział 14.).

Nie używaj konstruktorów klas opakowujących. Są wycofywane i przeznaczone do usunięcia. Na przykład pisz Integer.valueOf(1000), a nie new Integer(1000). Ewentualnie polegaj po prostu na automatycznym opakowywaniu: Integer a = 1000.

Z automatycznym pakowaniem wiążą się jeszcze pewne inne subtelne kwestie. Przede wszystkim referencje do obiektów klasy opakowującej mogą mieć wartość null, więc operacja automatycznego odpakowywania może spowodować zgłoszenie wyjątku NullPointerException.

```
Integer n = null;
System.out.println(2 * n); // powoduje wyjątek NullPointerException
```

A jeśli zmiesza się typy Integer i Double w wyrażeniu warunkowym, to wartość Integer zostanie opakowana, przekonwertowana na double i spakowana w Double:

```
Integer n = 1;
Double x = 2.0;
System.out.println(true ? n : x); // drukuje 1.0
```

Należy również zaznaczyć, że opakowywanie i odpakowywanie zawdzięczamy „uprzejmości” kompilatora, a nie maszyny wirtualnej. Kompilator wstawia odpowiednie wywołania, kiedy generuje kod bajtowy klasy. Rola maszyny wirtualnej sprowadza się tylko do wykonywania tego kodu.



W przyszłej wersji Javy programista będzie mógł definiować własne typy danych o takich samych cechach jak podstawowe typy danych, czyli nie przechowujące wartości w obiektach. Na przykład wartość podstawowego typu danych `Point` zawierającego pola `x` i `y` typu `double` jest w pamięci 16-bajtowym blokiem obejmującym dwie przylegające do siebie wartości typu `double`. Można go skopiować, ale nie można utworzyć referencji do niego.

Jeśli potrzebna jest referencja, należy użyć automatycznie wygenerowanej klasy towarzyszącej (zgodnie z obecnymi propozycjami miałyby nazwę `Point.ref`). Pakowanie i rozpakowywanie odbywa się automatycznie, tak samo jak w przypadku wszystkich typów podstawowych.

Kiedyś klasy opakowujące typów podstawowych zostaną zunifikowane z tymi klasami. Na przykład `Double` będzie aliasem `double.ref`.

Obiekty opakowujące typów liczbowych są także często używane do innego celu. Projektanci Javy odkryli, że obiekty opakowujące są dobrym miejscem na przechowywanie niektórych podstawowych metod, jak te, które służą do konwersji łańcuchów cyfr na liczby.

Aby przekonwertować łańcuch na liczbę, należy użyć następującej instrukcji:

```
int x = Integer.parseInt(s);
```

Kod ten nie ma nic wspólnego z obiektami klasy `Integer` — metoda `parseInt` jest statyczna. Jednak klasa `Integer` była dobrym miejscem na przechowywanie tej metody.

W opisie API znajdują się informacje o innych ważniejszych metodach klasy `Integer`. Pozostałe klasy odpowiadające typom liczbowym zawierają podobne metody.



Niektórzy programiści uważają, że klas opakowujących można używać do implementacji metod, które mogą modyfikować parametry liczbowe. Są jednak w błędzie. Pamiętamy z rozdziału 4., że w Javie nie można napisać metody zwiększającej parametr liczbowy, ponieważ parametry są zawsze przekazywane do metod przez wartość.

```
public static void triple(int x)           // nie zadziała
{
    x = 3 * x;                             // modyfikacja lokalnej zmiennej
}
```

Czy można to ominąć, stosując typ `Integer` zamiast `int`?


```
public static void triple(Integer x)    // nie zadziała
{
    ...
}
```

Problem polega na tym, że obiekty klasy `Integer` są **niezmienne** — informacje zawarte w obiekcie opakującym nie mogą być zmieniane. Nie można użyć tych klas opakujących do tworzenia metod modyfikujących parametry liczbowe.

`java.lang.Integer` 1.0

- `int intValue()`

Zwraca wartość obiektu `Integer` jako liczbę typu `int` (przesłania metodę `intValue` z klasy `Number`).

- `static String toString(int i)`

Zwraca nowy obiekt klasy `String` reprezentujący liczbę `i` w systemie dziesiętnym.

- `static String toString(int i, int radix)`

Zwraca reprezentację liczby `i` w systemie określonym przez parametr `radix`.

- `Static int parseInt(String s)`

- `Static int parseInt(String s, int radix)`

Zwraca liczbę całkowitą utworzoną z cyfr w łańcuchu `s`. Łańcuch ten musi reprezentować liczbę w systemie dziesiętnym (w przypadku pierwszej metody) lub w systemie określonym przez parametr `radix` (druga metoda).

- `static Integer valueOf(String s)`

- `static Integer valueOf(String s, int radix)`

Zwraca obiekt klasy `Integer` zainicjalizowany liczbą całkowitą reprezentowaną przez łańcuch `s`. Łańcuch ten musi reprezentować liczbę w systemie dziesiętnym (w przypadku pierwszej metody) lub w systemie określonym przez parametr `radix` (druga metoda).

`java.text.NumberFormat` 1.1

- `Number parse(String s)`

Zwraca wartość liczbową, jeśli łańcuch `s` reprezentuje liczbę.

5.5. Metody ze zmienną liczbą parametrów

Programista może tworzyć metody, które da się wywoływać z różną liczbą parametrów (można spotkać ich angielską nazwę **varargs**).

Znamy już metodę `printf`. Na przykład wywołania:

```
System.out.printf("%d", n);  
  
i  
  
System.out.printf("%d %s", n, "widgets");
```

dotyczą tej samej metody, mimo że pierwsze z nich ma dwa parametry, a drugie trzy.

Definicja metody `printf` wygląda następująco:

```
public class PrintStream  
{  
    public PrintStream printf(String fmt, Object... args)  
    {  
        return format(fmt, args);  
    }  
}
```

W powyższym kodzie wielokropek (...) jest częścią kodu Javy. Określa on, że metoda może przyjmować dowolną liczbę obiektów (parametr `fmt` jest obowiązkowy).

Metoda `printf` w rzeczywistości przyjmuje dwa parametry — łańcuch formatu i tablicę `Object[]`, która zawiera wszystkie pozostałe parametry (jeśli zostanie podana wartość typu podstawowego, jak `int`, mechanizm automatycznego opakowywania zamieni ją na obiekt). Musi ona przeskanować łańcuch `fmt` i dopasować *i*-ty specyfikator formatu do wartości `args[i]`.

Innymi słowy, z punktu widzenia programisty implementującego metodę `printf` typ parametru `Object...` jest dokładnie tym samym co `Object[]`.

Kompilator musi przekonwertować każde wywołanie metody `printf`, pakując parametry do tablicy i wykonując w razie potrzeby automatyczne opakowywanie:

```
System.out.printf("%d %s", new Object[] { Integer.valueOf(n), "widgets" } );
```

Można definiować własne metody ze zmienną liczbą parametrów. Parametry te mogą być każdego typu, także podstawowego. Poniżej znajduje się prosty przykład takiej funkcji — zwraca największą liczbę w zbiorze o zmiennych rozmiarach:

```
public static double max(double... values)  
{  
    double largest = Double.NEGATIVE_INFINITY;  
    for (double v : values) if (v > largest) largest = v;  
    return largest;  
}
```

Należy ją wywołać w następujący sposób:

```
double m = max(3.1, 40.4, -5);
```

Kompilator przekazuje tablicę `new double[] { 3.1, 40.4, -5 }` do funkcji `max`.



Ostatnim parametrem metody o zmiennej liczbie parametrów może być tablica. Na przykład:

```
System.out.printf("%d %s", new Object[] { Integer.valueOf(1), "widgets" } );
```

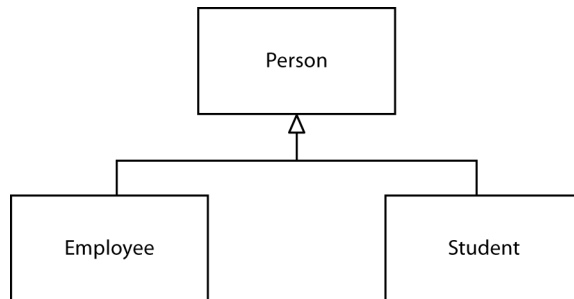
W związku z tym można przeddefiniować istniejącą funkcję, której ostatni parametr jest tablicą, na metodę ze zmienną liczbą parametrów, nie uszkadzając istniejącego kodu. Na przykład w ten sposób rozszerzono metodę `MessageFormat.format` w Java SE 5.0. Można nawet zadeklarować metodę `main` w następujący sposób:

```
public static void main(String... args)
```

5.6. Klasy abstrakcyjne

Im bliżej wierzchołka hierarchii dziedziczenia, tym klasy są bardziej ogólne i często bardziej abstrakcyjne. W pewnym momencie klasa nadrzędna staje się tak abstrakcyjna, że zaczyna być traktowana nie jako klasa do tworzenia obiektów o określonym przeznaczeniu, a jako podstawa do tworzenia innych klas. Przeanalizujemy na przykład rozszerzenie hierarchii klasy `Employee`. Pracownik (ang. *employee*), podobnie jak student, jest osobą. Poszerzymy naszą hierarchię klas o klasy `Person` (osoba) i `Student`. Rysunek 5.2 przedstawia relacje dziedziczenia zachodzące pomiędzy tymi klasami.

Rysunek 5.2.
Diagram
dziedziczenia
klasy `Person`
i jej podklas



Po co w ogóle zaprządać sobie głowę takim poziomem abstrakcji? Niektóre cechy ma każda osoba, np. nazwisko. Zarówno studenci, jak i pracownicy mają nazwiska, a więc wprowadzenie wspólnej nadklasy umożliwia wydzielenie metody `getName` i przeniesienie jej na wyższy poziom hierarchii dziedziczenia.

Dodamy teraz nową metodę o nazwie `getDescription`, która zwraca krótki opis osoby, np.:

```
pracownik zarabiający 50 000,00 zł
student specjalizacji informatyka
```

Implementacja tej metody dla klas `Employee` i `Student` jest łatwa. Jakie natomiast informacje można podać w klasie `Person`? Klasa ta ma jedynie informacje na temat nazwiska osoby. Oczywiście można zaimplementować metodę `Person.getDescription()`, która zwraca pusty łańcuch.

Istnieje jednak lepsze rozwiązanie. Dzięki użyciu słowa kluczowego `abstract` w ogóle nie ma potrzeby implementowania tej metody.

```
public abstract String getDescription();
// nie jest potrzebna żadna implementacja
```

Klasa zawierająca przynajmniej jedną metodę abstrakcyjną sama musi być abstrakcyjna.

```
public abstract class Person
{
    . . .
    public abstract String getDescription();
}
```

Poza metodami abstrakcyjnymi klasy abstrakcyjne mogą zawierać pola i metody konkretne. Na przykład klasa `Person` przechowuje nazwisko osoby i ma metodę konkretną, która zwraca te dane.

```
public abstract class Person
{
    private String name;

    public Person(String name)
    {
        this.name = name;
    }
    public abstract String getDescription();
    public String getName()
    {
        return name;
    }
}
```



Niektórzy programiści nie wiedzą, że klasy abstrakcyjne mogą zawierać metody konkretne. Należy zawsze przenosić wspólne pola i metody (bez względu na to, czy są abstrakcyjne, czy nie) do nadklasy (abstrakcyjnej lub nie).

Metody abstrakcyjne pełnią rolę symbolu zastępczego dla metod, które są implementowane w podklasach. Przy rozszerzaniu klasy abstrakcyjnej programista ma do wyboru jedną z dwóch opcji: może pozostawić niezdefiniowane niektóre lub wszystkie metody nadklasy — wtedy podklasa również musi być abstrakcyjna, albo zdefiniować wszystkie metody i wtedy podklasa nie jest abstrakcyjna.

Jako przykład zdefiniujemy klasę `Student`, która będzie rozszerzała abstrakcyjną klasę `Person` i implementowała metodę `getDescription`. Ponieważ żadna z metod klasy `Student` nie jest abstrakcyjna, klasa ta również nie musi być abstrakcyjna.

Klasę można zdefiniować jako abstrakcyjną, nawet jeśli nie zawiera żadnych metod abstrakcyjnych.

Nie można tworzyć obiektów klas abstrakcyjnych. To znaczy, że jeśli klasa ma w deklaracji słowo `abstract`, nie może mieć obiektów. Na przykład poniższe wyrażenie:

```
new Person("Wincenty Witos")
```

jest błędne. Można natomiast tworzyć obiekty podklas konkretnych.

Nadal można tworzyć zmienne obiektów klas abstrakcyjnych, ale takie zmienne muszą się odnosić do obiektu podklasy nieabstrakcyjnej. Na przykład:

```
Person p = new Student("Vince Vu", "Economics");
```

Tutaj zmienna p jest typu abstrakcyjnego Person, ale odnosi się do egzemplarza nieabstrakcyjnej podklasy Student.



W C++ metoda abstrakcyjna nazywa się **funkcją czysto wirtualną** i jest oznaczana końcowymi znakami =0:

```
class Person    // C++
{
public:
    virtual string getDescription() = 0;
    . . .
};
```

W C++ klasa jest abstrakcyjna, jeśli zawiera co najmniej jedną funkcję czysto wirtualną. Nie ma w tym języku specjalnego słowa kluczowego określającego klasę abstrakcyjną.

Zdefiniujemy konkretną podklasę Student, która rozszerza abstrakcyjną klasę Person:

```
public class Student extends Person
{
    private String major;

    public Student(String name, String major)
    {
        super(name);
        this.major = major;
    }
    public String getDescription()
    {
        return "student specjalizacji " + major;
    }
}
```

Klasa Student zawiera definicję metody getDescription. W związku z tym wszystkie metody tej klasy są konkretne, czyli nie jest ona abstrakcyjna.

Program przedstawiony na listingach 5.8, 5.9, 5.10 i 5.11 definiuje abstrakcyjną nadklasę o nazwie Person i dwie konkretne podklasy o nazwach Employee i Student. Do tablicy referencji typu Person wstawiane są obiekty klas Employee i Student:

```
var people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

Listing 5.8. abstractClasses/PersonTest.java

```
package abstractClasses;

/**
 * Ten program demonstruje klasy abstrakcyjne.
 * @version 1.01 2004-02-21
 * @author Cay Horstmann
 */
public class PersonTest
{
    public static void main(String[] args)
    {
        var people = new Person[2];

        // Wstawienie do tablicy people obiektów Student i Employee.
        people[0] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
        people[1] = new Student("Maria Mrozowska", "informatyka");

        // Drukowanie imion i nazwisk oraz opisów wszystkich obiektów klasy Person.
        for (Person p : people)
            System.out.println(p.getName() + ", " + p.getDescription());
    }
}
```

Listing 5.9. abstractClasses/Person.java

```
package abstractClasses;

public abstract class Person
{
    public abstract String getDescription();
    private String name;

    public Person(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Listing 5.10. abstractClasses/Employee.java

```
package abstractClasses;

import java.time.*;

public class Employee extends Person
{
    private double salary;
    private LocalDate hireDay;
```

```

public Employee(String n, double s, int year, int month, int day)
{
    super(name);
    this.salary = salary;
    hireDay = LocalDate.of(year, month, day);
}

public double getSalary()
{
    return salary;
}

public Date getHireDay()
{
    return hireDay;
}

public String getDescription()
{
    return String.format("pracownik zarabiający %.2f zł", salary);
}

public void raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
}
}

```

Listing 5.11. abstractClasses/Student.java

```

package abstractClasses;

public class Student extends Person
{
    private String major;

    /**
     * @param n imię i nazwisko studenta
     * @param m specjalizacja studenta
     */
    public Student(String name, String major)
    {
        // Przekazanie name do konstruktora nadklasy.
        super(name);
        this.major = major;
    }

    public String getDescription()
    {
        return "student specjalizacji " + major;
    }
}

```

Poniższy fragment kodu odpowiada za wydruk imion i nazwisk oraz opisów powyższych obiektów:

```

for (Person p : people)
    System.out.println(p.getName() + ", " + p.getDescription());

```

Wątpliwości może budzić poniższe wywołanie:

```
p.getDescription()
```

Czy nie jest to wywołanie niezdefiniowanej metody? Należy pamiętać, że zmienna `p` nie odwołuje się nigdy do obiektu `Person`, ponieważ nie można utworzyć obiektu klasy abstrakcyjnej `Person`. Zmienna `p` zawsze odwołuje się do obiektu konkretnej podklasy, jak `Employee` lub `Student`. Dla tych obiektów metoda `getDescription` jest zdefiniowana.

Czy można by było pominąć abstrakcyjną metodę nadklasy abstrakcyjnej `Person` i zdefiniować metody `getDescription` w podklasach `Employee` i `Student`? Gdybyśmy tak zrobili, niemożliwe byłoby wywołanie metody `getDescription` na rzecz zmiennej `p`, ponieważ kompilator zezwala na wywoływanie tylko tych metod, które są zdefiniowane w danej klasie.

Metody abstrakcyjne są bardzo ważnym elementem języka programowania Java. Najczęściej występują w *interfejsach*. Więcej informacji na temat interfejsów zawiera rozdział 6.

5.7. Klasy wyliczeniowe

W rozdziale 3. nauczyliśmy się definiować typy wyliczeniowe. Oto typowy przykład:

```
public enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE}
```

Typ zdefiniowany przez powyższą deklarację jest w rzeczywistości klasą. Ma ona dokładnie cztery egzemplarze — nie można tworzyć jej nowych obiektów.

W związku z tym do porównywania typów wyliczeniowych nie trzeba używać metody `equals`. Wystarczy operator `==`.

Do typu wyliczeniowego można dodać konstruktory, metody i pola. Oczywiście konstruktory są wywoływane tylko wówczas, gdy są konstruowane stałe wyliczeniowe. Na przykład:

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    Size(String abbreviation) { this.abbreviation = abbreviation; }
    // prywatna automatycznie
    public String getAbbreviation() { return abbreviation; }
}
```

Konstruktor wyliczenia zawsze jest prywatny. Modyfikator `private` można opuścić, jak pokazano w poprzednim przykładzie. Zadeklarowanie konstruktora wyliczenia jako publicznego lub chronionego stanowi błąd składni.

Wszystkie typy wyliczeniowe są podklasami abstrakcyjnej klasy `Enum`. Dziedziczą po niej kilka metod. Do najbardziej przydatnych należy metoda `toString`, która zwraca nazwę stałej wyliczeniowej. Na przykład wywołanie `Size.SMALL.toString()` zwraca łańcuch `"SMALL"`.

Przeciwnieństwem metody `toString` jest statyczna metoda `valueOf`. Na przykład poniższa instrukcja ustawia `s` na `Size.SMALL`.

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Każdy typ wyliczeniowy ma statyczną metodę `values`, która zwraca wszystkie wartości wyliczenia. Na przykład wywołanie:

```
Size[] values = Size.values();
```

zwraca tablicę zawierającą następujące elementy: `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE` oraz `Size.EXTRA_LARGE`.

Metoda `ordinal` zwraca położenie stałej wyliczeniowej w deklaracji `enum`, zaczynając liczenie od zera. Na przykład wywołanie `Size.MEDIUM.ordinal()` zwraca wartość 1.

Krótki program przedstawiony na listingu 5.12 demonstruje zastosowanie typów wyliczeniowych.



Klasa `Enum` ma parametr typu, który dla uproszczenia pominęliśmy. Na przykład typ wyliczeniowy `Size` w rzeczywistości rozszerza `Enum<Size>`. Parametr typu jest używany przez metodę `compareTo` (metodę `compareTo` opisujemy w rozdziale 6., a parametry typu w rozdziale 8.).

Listing 5.12. `enums/EnumTest.java`

```
package enums;

import java.util.*;

/**
 * Ten program demonstruje typy wyliczeniowe.
 * @version 1.0 2004-05-24
 * @author Cay Horstmann
 */
public class EnumTest
{
    public static void main(String[] args)
    {
        var in = new Scanner(System.in);
        System.out.print("Podaj rozmiar: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
        String input = in.next().toUpperCase();
        Size size = Enum.valueOf(Size.class, input);
        System.out.println("rozmiar=" + size);
        System.out.println("skrót=" + size.getAbbreviation());
        if (size == Size.EXTRA_LARGE)
            System.out.println("Dobra robota -- nie pominąłeś znaku podkreślenia _.");
    }
}

enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
```

```

    public String getAbbreviation() { return abbreviation; }

    private String abbreviation;
}

```

java.lang.Enum<E> 5

- `static Enum valueOf(Class enumClass, String name)`

Zwraca stałą wyliczeniową danej klasy o podanej nazwie.

- `String toString()`

Zwraca nazwę stałej wyliczeniowej.

- `int ordinal()`

Zwraca położenie w deklaracji enum (licząc od zera) stałej wyliczeniowej.

- `int compareTo(E other)`

Zwraca ujemną liczbę całkowitą, jeśli stała wyliczeniowa występuje przed `other`, zero — jeśli `this == other`, lub liczbę dodatnią w przeciwnym przypadku. Kolejność stałych jest określana przez deklarację enum.

5.8. Klasy zapieczętowane

Jeśli klasa nie jest zadeklarowana jako finalna, to każdy może utworzyć jej podklasę. Czy można mieć nad tym bardziej szczegółową kontrolę? Powiedzmy na przykład, że chcemy napisać własną bibliotekę JSON, ponieważ żadna z istniejących nie odpowiada w pełni naszym potrzebom.

W standardzie JSON znajduje się zapis informujący, że wartością JSON jest tablica, liczba, łańcuch, wartość logiczna, obiekt lub null. Oczywistym wyborem do reprezentowania tych wartości są klasy `JSONArray`, `JSONNumber` itd., które rozszerzają klasę `JSONValue`:

```

public abstract class JSONValue
{
    // Metody dotyczące wszystkich wartości JSON
}
public final class JSONArray extends JSONValue
{
    . . .
}
public final class JSONNumber extends JSONValue
{
    . . .
}

```

Przez dodanie słowa `final` do deklaracji klas `JSONArray`, `JSONNumber` itd. uniemożliwiamy komukolwiek utworzenie ich podklas. Nie możemy natomiast zabronić utworzenia kolejnej podklasy klasy `JSONValue`.

Do czego mogłaby być nam potrzebna taka możliwość? Spójrz na poniższy kod:

```
JSONValue v = . . . ;
if (v instanceof JSONArray a) . . .
else if (v instanceof JSONNumber n) . . .
else if (v instanceof JSONString s) . . .
else if (v instanceof JSONBoolean b) . . .
else if (v instanceof JSONObject o) . . .
else . . . // Musi być JSONNull
```

W tym przypadku przepływ sterowania implikuje, że znamy wszystkie bezpośrednie podklasy klasy `JSONValue`. To nie jest otwarta hierarchia. Standard JSON się nie zmienia, a jeśli tak, to my, jako implementatorzy biblioteki, dodamy siódmą podklasę. Nie chcemy, aby ktokolwiek inny mieszał w tej hierarchii klas.

Mechanizmem pozwalającym kontrolować dziedziczenie w Javie jest **pieczętowanie** klas. Technika ta została wprowadzona na próbę w wersji 15 i ostatecznie zaimplementowana w wersji 17 języka.

Poniżej znajduje się przykład zapieczętowania klasy `JSONValue`:

```
public abstract sealed class JSONValue
    permits JSONArray, JSONNumber, JSONString, JSONBoolean, JSONObject, JSONNull
{
    . . .
}
```

Próba zdefiniowania klasy spoza dozwolonej listy jest błędem:

```
public class JSONComment extends JSONValue { . . . } // Błąd
```

W tym przypadku to bardzo dobrze, bo standard JSON nie przewiduje komentarzy. Zatem klasy zapieczętowane umożliwiają precyzyjne modelowanie ograniczeń dziedziny.

Dopuszczalne podklasy klasy zapieczętowanej muszą być dostępne. Nie mogą być prywatnymi konstrukcjami zagnieżdżonymi w innej klasie ani jednostkami widocznymi tylko w obrębie innego pakietu.

Reguły dotyczące dozwolonych podklas publicznych są jeszcze surowsze. Te klasy muszą się znajdować w tym samym pakiecie co klasa zapieczętowana. Jeśli jednak w użyciu są moduły (rozdział 9. tomu drugiego), to wystarczy, aby taka klasa znajdowała się w tym samym module.



Klasę zapieczętowaną można zadeklarować bez klauzuli `permits`. W takim przypadku wszystkie jej bezpośrednie podklasy muszą być zadeklarowane w tym samym pliku. Programiście nie mający do niego dostępu nie mogą tworzyć podklas.

W pliku może się znajdować tylko jedna klasa publiczna, a więc zasada ta wydaje się dotyczyć wyłącznie sytuacji, w której podklasy nie są przeznaczone do publicznego użytku.

Z drugiej strony w następnym rozdziale poznasz techniki używania klas wewnętrznych jako publicznych podklas.

Ważnym argumentem przemawiającym za klasami zapieczętowanymi jest możliwość przeprowadzania kontroli przez kompilator. Spójrz na poniższą metodę z klasy `JSONValue`, która zawiera wyrażenie `switch` z dopasowywaniem wzorca (próbny dodatek do Javy 17):

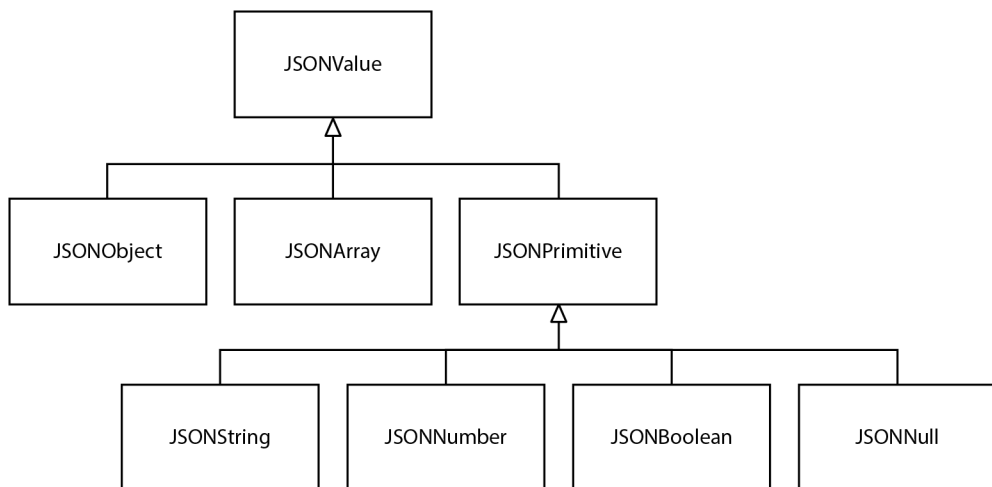
```
public String type()
{
    return switch (this)
    {
        case JSONArray j -> "array";
        case JSONNumber j -> "number";
        case JSONString j -> "string";
        case JSONBoolean j -> "boolean";
        case JSONObject j -> "object";
        case JSONNull j -> "null";
        // Klauzula domyślna jest niepotrzebna
    };
}
```

Skoro wszystkie bezpośrednie podklasy klasy `JSONValue` zostały ujęte w klauzulach `case`, kompilator będzie „wiedział”, że klauzula `default` jest niepotrzebna.



Powyższa metoda `type` nie wygląda na konstrukcję w technice obiektowej. Zgodnie z założeniami tej metodyki należałoby raczej utworzyć osobną metodę `type` w każdej z sześciu klas, z wykorzystaniem polimorfizmu, a nie konstrukcji `switch` — takie podejście dobrze się sprawdza w przypadku hierarchii otwartych. Kiedy natomiast zestaw klas jest określony na sztywno, często wygodniej jest zdefiniować wszystkie alternatywne opcje w jednej metodzie.

Na pierwszy rzut oka można odnieść wrażenie, że podklasa klasy zapieczętowanej musi być finalna. Jednak na potrzeby testów kompletności musimy znać tylko wszystkie bezpośrednie podklasy. Nie jest problemem, jeśli mają one jakieś dalsze podklasy. Naszą hierarchię klas JSON możemy na przykład przeorganizować w sposób pokazany na rysunku 5.3.



Rysunek 5.3. Kompletna hierarchia klas reprezentujących wartości JSON

W tej hierarchii klasa `JSONValue` może mieć trzy podklasy:

```
public abstract sealed class JSONValue permits JSONObject, JSONArray, JSONPrimitive
{
    . . .
}
```

Klasa `JSONPrimitive` też jest zapieczętowana:

```
public abstract sealed class JSONPrimitive extends JSONValue
    permits JSONString, JSONNumber, JSONBoolean, JSONNull
{
    . . .
}
```

Podklasa zapieczętowanej klasy musi określać, czy jest zapieczętowana, finalna czy otwarta na dalsze tworzenie podklas. W drugim z wymienionych przypadków konieczna jest deklaracja, że klasa nie jest zapieczętowana (ang. *non-sealed*).



Słowo kluczowe `non-sealed` to pierwsze słowo kluczowe z łącznikiem w języku Java. Może zwiastować początek przyszłego trendu. Dodawanie słów kluczowych do języka programowania zawsze jest ryzykowne, ponieważ może spowodować trudności z kompilacją istniejącego kodu. Dlatego też `sealed` jest słowem „kontekstowym”, tzn. nadal można go używać do nazywania zmiennych i metod:

```
int sealed = 1; // Można użyć kontekstowego słowa kluczowego jako identyfikatora
```

W przypadku gdy zostanie użyty łącznik, nie trzeba się o to martwić. Jedyna dwuznaczność może dotyczyć operacji odejmowania:

```
int non = 0;
non = non-sealed; // Odejmowanie, nie słowo kluczowe
```

Po co w ogóle ktoś miałby tworzyć niezapieczętowaną podklasę? Wyobraź sobie klasę węzła XML, która ma sześć bezpośrednich podklas:

```
public abstract sealed class Node permits Element, Text, Comment,
    CDATASection, EntityReference, ProcessingInstruction
{
    . . .
}
```

Zdejmujemy ograniczenie dotyczące podklas z klasy `Element`:

```
public non-sealed class Element extends Node
{
    . . .
}
public class HTMLDivElement extends Element
{
    . . .
}
```

W tym podrozdziale opisałem klasy zapieczętowane, a w następnym rozdziale poznasz *interfejsy*, które są uogólnieniem klas abstrakcyjnych. W Javie interfejsy także mogą mieć podtypy.

Zapieczętowane interfejsy mają dokładnie takie same cechy jak zapieczętowane klasy w zakresie kontroli nad tworzeniem bezpośrednich podtypów.

Przykładowy program przedstawiony na listingu 5.13 ilustruje implementację hierarchii klas reprezentujących typy JSON. W implementacji klasy `JSONObject` została wykorzystana klasa `HashMap`, której opis znajduje się w rozdziale 9. W przykładzie użyto interfejsów zamiast klas abstrakcyjnych, dzięki czemu klasy `JSONNumber` i `JSONString` mogą być rekordami, a klasy `JSONBoolean` i `JSONNull` mogą być wyliczeniami. Rekordy i wyliczenia mogą implementować interfejsy, ale nie mogą rozszerzać klas.

Listing 5.13. `sealed/SealedTest.java`

```
package sealed;

import java.util.*;

sealed interface JSONValue permits JSONArray, JSONObject, JSONPrimitive
{
    public default String type()
    {
        if (this instanceof JSONArray) return "array";
        else if (this instanceof JSONObject) return "object";
        else if (this instanceof JSONNumber) return "number";
        else if (this instanceof JSONString) return "string";
        else if (this instanceof JSONBoolean) return "boolean";
        else return "null";
    }
}

final class JSONArray extends ArrayList<JSONValue> implements JSONValue {}

final class JSONObject extends HashMap<String, JSONValue> implements JSONValue
{
    public String toString()
    {
        StringBuilder result = new StringBuilder();
        result.append("{");
        for (Map.Entry<String, JSONValue> entry : entrySet())
        {
            if (result.length() > 1) result.append(",");
            result.append(" ");
            result.append(entry.getKey());
            result.append(": ");
            result.append(entry.getValue());
        }
        result.append("}");
        return result.toString();
    }
}

sealed interface JSONPrimitive extends JSONValue
    permits JSONNumber, JSONString, JSONBoolean, JSONNull
{
}

final record JSONNumber(double value) implements JSONPrimitive
```

```

{
    public String toString() { return "" + value; }
}

final record JSONString(String value) implements JSONPrimitive
{
    public String toString() { return "\"" + value.translateEscapes() + "\""; }
}

enum JSONBoolean implements JSONPrimitive
{
    FALSE, TRUE;
    public String toString() { return super.toString().toLowerCase(); }
}

enum JSONNull implements JSONPrimitive
{
    INSTANCE;
    public String toString() { return "null"; }
}

public class SealedTest
{
    public static void main(String[] args)
    {
        JSONObject obj = new JSONObject();
        obj.put("name", new JSONString("Harry"));
        obj.put("salary", new JSONNumber(90000));
        obj.put("married", JSONBoolean.FALSE);
        JSONArray arr = new JSONArray();
        arr.add(new JSONNumber(13));
        arr.add(JSONNull.INSTANCE);

        obj.put("luckyNumbers", arr);
        System.out.println(obj);
        System.out.println(obj.type());
    }
}

```

5.9. Refleksja

Biblioteka refleksyjna dostarcza bogaty i zaawansowany zestaw narzędzi do pisania programów, które dynamicznie zarządzają kodem Javy. Z mechanizmów refleksji w Javie korzystają kreatory interfejsów, mapery obiektowo-relacyjne i inne narzędzia programistyczne, które dynamicznie sprawdzają właściwości klas.

Program, który może analizować funkcjonalność klas, nazywa się **programem refleksyjnym**. Mechanizm refleksji ma bardzo duże możliwości. W kolejnych podrozdziałach opisujemy jego następujące zastosowania:

- Analiza właściwości klasy w trakcie działania programu.
- Inspekcja obiektów w czasie działania programu, na przykład do napisania jednej metody `toString`, która działa we *wszystkich* klasach.
- Implementacja generycznego kodu manipulującego tablicami.
- Wykorzystanie obiektów `Method`, które działają tak jak wskaźniki do funkcji w innych językach, np. C++.

Refleksja to wszechstronny i skomplikowany mechanizm. Jest interesująca przede wszystkim dla twórców narzędzi, mniej dla programistów aplikacji. Osoby, które są zainteresowane pisaniem aplikacji, a nie narzędzi dla innych programistów Javy, mogą pominąć resztę rozdziału i wrócić do niego kiedy indziej.

5.9.1. Klasa `Class`

Kiedy uruchomiony jest program, system wykonawczy Javy cały czas przechowuje informacje o typach wszystkich obiektów. Do informacji tych zaliczają się nazwy klas, do których należą obiekty. Informacje o typach czasu wykonywania programu są wykorzystywane przez maszynę wirtualną do wyboru odpowiednich metod do wykonania.

Do informacji tych można jednak uzyskać dostęp także dzięki specjalnej klasie. Klasa przechowująca te informacje ma nazwę `Class`. Metoda `getClass()` klasy `Object` zwraca egzemplarz klasy `Class`.

```
Employee e;  
.  
.  
.  
Class cl = e.getClass();
```

Podobnie jak obiekt klasy `Employee` opisuje cechy określonego pracownika, obiekt klasy `Class` opisuje cechy określonej klasy. Chyba najczęściej używaną metodą klasy `Class` jest metoda `getName`, która zwraca nazwę klasy. Na przykład poniższa instrukcja:

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

drukuje:

```
Employee Henryk Kwiatek
```

jeśli `e` jest zwykłym pracownikiem, lub

```
Manager Henryk Kwiatek
```

jeśli `e` jest kierownikiem.

Jeśli klasa należy do jakiegoś pakietu, nazwa tego pakietu stanowi część nazwy tej klasy:

```
var generator = new Random();  
Class cl = generator.getClass();  
String name = cl.getName(); // name ustawiono na "java.util.Random"
```


Obiekt klasy `Class` odpowiadający nazwie wybranej klasy można utworzyć za pomocą statycznej metody `forName`.

```
String className = "java.util.Date";
Class c1 = Class.forName(className);
```

Metody tej należy użyć, jeśli nazwa klasy jest przechowywana w łańcuchu, który zmienia się w czasie działania programu. Powyższy kod działa, jeśli `className` jest nazwą klasy lub interfejsu. W przeciwnym przypadku metoda `forName` powoduje **wyjątek kontrolowany** (ang. *checked exception*). Informacje na temat **obsługi wyjątków** podczas używania tej metody znajdują się w punkcie 5.9.2, „Podstawy deklarowania wyjątków”.

Trzecia metoda tworzenia obiektu typu `Class` jest wygodnym skrótem. Jeśli `T` jest dowolnym typem Javy (lub słowem kluczowym `void`), `T.class` jest odpowiadającym mu obiektem klasy `Class`. Na przykład:

```
Class c11 = Random.class;    // Należy zaimportować java.util.*.
Class c12 = int.class;
Class c13 = Double[].class;
```

Należy zauważyć, że obiekt klasy `Class` w rzeczywistości oznacza *typ*, który może, ale nie musi być klasą. Na przykład `int` nie jest klasą, ale `int.class` jest z pewnością obiektem typu `Class`.



Klasa `Class` jest sparametryzowana. Na przykład `Employee.class` jest typu `Class` `<Employee>`. Nie będziemy drążyć tego tematu, ponieważ jeszcze bardziej skomplikowalibyśmy i tak już wystarczająco abstrakcyjną koncepcję. Dla praktycznych celów można zignorować parametr typu i pracować na surowym typie `Class`. Więcej informacji na ten temat znajduje się w rozdziale 8.



Z powodów historycznych metoda `getName` zwraca nieco dziwne nazwy typów tablicowych:

- `Double[].class.getName()` zwraca `[Ljava.lang.Double;`,
- `int[].class.getName()` zwraca `[I`.

Maszyna wirtualna obsługuje unikatowy obiekt `Class` dla każdego typu. W związku z tym do porównywania obiektów `class` można używać operatora `==`. Na przykład:

```
if (e.getClass() == Employee.class) . . .
```

Wynik tego testu będzie pozytywny, gdy `e` będzie egzemplarzem klasy `Employee`. W odróżnieniu od warunku `e instanceof Employee`, w tym przypadku wynik będzie negatywny, jeśli `e` będzie egzemplarzem podklasy, np. `Manager`.

Jeśli masz obiekt typu `Class`, przy jego użyciu możesz tworzyć egzemplarze tej klasy. Wywołaj metodę `getConstructor`, aby otrzymać obiekt typu `Constructor`, a następnie za pomocą metody `newInstance` utwórz nowy egzemplarz. Na przykład:

```
var className = "java.util.Random"; // lub nazwa jakiegokolwiek innej klasy
                                   // z konstruktorem bezargumentowym
Class c1 = Class.forName(className);
Object obj = c1.getConstructor().newInstance();
```

Jeżeli klasa nie ma konstruktora bezargumentowego, metoda `getConstructor` zgłasza wyjątek. W punkcie 5.9.7, „Wywoływanie dowolnych metod i konstruktorów”, dowiesz się, jak wywoływać inne konstruktory.



Istnieje wycofywana z użycia metoda `Class.newInstance`, która także tworzy egzemplarz przy użyciu konstruktora bez argumentów. Jeśli jednak konstruktor zgłosi wyjątek kontrolowany, zostaje on ponownie zgłoszony bez sprawdzenia. To stanowi złamanie zasady sprawdzania wyjątków w czasie kompilacji. Natomiast `Constructor.newInstance` opakowuje każdy wyjątek konstruktora w obiekt `InvocationTargetException`.



Metoda `newInstance` jest odpowiednikiem **konstruktora wirtualnego** w C++. Jednak konstruktory wirtualne w tym języku nie są właściwością języka, a tylko idiomem, który musi być obsługiwany przez specjalną bibliotekę. Klasa `Class` jest podobna do klasy `TypeInfo` w C++, a metoda `getClass` jest odpowiednikiem operatora `typeid`. Klasa `Class` Javy jest jednak nieco bardziej wszechstronna niż `TypeInfo`. Ta druga potrafi tylko zwrócić łańcuch z nazwą typu. Nie tworzy nowych obiektów tego typu.

java.lang.Class 1.0

- `static Class.forName(String className)`

Zwraca obiekt klasy `Class` reprezentujący klasę o nazwie `className`.

- `Constructor getConstructor(Class... parameterTypes)` 1.1

Zwraca obiekt opisujący konstruktor za pomocą podanych typów parametrów. Zobacz punkt 5.9.7 „Wywoływanie dowolnych metod i konstruktorów”, aby się dowiedzieć, jak dostarczać te typy parametrów.

java.lang.reflect.Constructor 1.1

- `Object newInstance(Object[] args)`

Tworzy nowy egzemplarz klasy zawierającej deklarację konstruktora, przekazując do niego parametry. Więcej informacji na temat przekazywania parametrów znajduje się w punkcie 5.9.7.

java.lang.Throwable 1.0

- `void printStackTrace()`

Drukuje obiekt `Throwable` i dane ze stosu do standardowego strumienia błędów.

5.9.2. Podstawy deklarowania wyjątków

Techniki przechwytywania wyjątków zostały opisane w rozdziale 7., ale zanim do niego dojdziemy, napotkamy po drodze kilka metod, które grożą, że mogą spowodować wyjątek.

Kiedy w czasie działania programu występuje błąd, program może spowodować wyjątek. Mechanizm wyjątków zapewnia większą elastyczność niż kończenie programu, ponieważ można napisać procedurę, która *przechwyci taki* wyjątek i go odpowiednio obsłuży.

Jeśli nie ma procedury obsługi wyjątku, program zostaje zakończony, a w konsoli zostaje wydrukowany komunikat informujący o jego typie. Komunikat taki może się pojawić w wyniku przypadkowego użycia referencji `null` lub przekroczenia rozmiaru tablicy.

Są dwa rodzaje wyjątków: **niekontrolowane** (ang. *unchecked*) i **kontrolowane** (ang. *checked*). W przypadku tych drugich kompilator sprawdza, czy programista wie o wyjątku i czy jest przygotowany na konsekwencje jego zgłoszenia. Jednak wiele wyjątków, na przykład błędy zakresu i dostęp do referencji `null`, jest niekontrolowanych, tzn. kompilator nie oczekuje, że programista dostarczy dla nich procedurę obsługi. W ich przypadku energię należy poświęcić na uniknięcie ich powstania, a nie na tworzenie procedur do ich obsłużenia.

Nie wszystkich błędów można jednak uniknąć. Jeśli mimo najlepszych chęci programisty wyjątek może się pojawić, większość API Javy zgłasza wyjątek kontrolowany. Przykładem jest metoda `Class.forName`. Nie da się zagwarantować, że będzie istniała klasa o określonej nazwie. W rozdziale 7. znajduje się opis kilku technik obsługi wyjątków. Poniżej pokazana jest tylko najprostsza strategia.

Jeśli metoda zawiera instrukcję mogącą zgłosić kontrolowany wyjątek, do nazwy tej metody należy dodać klauzulę `throws`.

```
public static void doSomethingWithClass(String name)
    throws ReflectiveOperationException
{
    Class cl = Class.forName(name); // może zgłosić wyjątek
    jakies operacje na cl
}
```

Każda metoda wywołująca tę metodę również musi mieć deklarację `throws`. Dotyczy to także metody `main`. Jeśli rzeczywiście wystąpi wyjątek, metoda `main` zakończy działanie, wyświetlając dane ze stosu. (W rozdziale 7. dowiesz się, jak przechwytywać wyjątki, zamiast pozwalać im doprowadzić do zamknięcia programu).

Klauzula `throws` jest wymagana tylko dla wyjątków kontrolowanych. To, które metody zgłaszają takie wyjątki, można łatwo sprawdzić – kompilator zawsze informuje o próbie wywołania metody grożącej wyjątkiem kontrolowanym, gdy programista nie dostarczy odpowiedniej procedury obsługi tego wyjątku.

5.9.3. Zasoby

Klasy często mają powiązane pliki danych, np.:

- pliki graficzne i dźwiękowe,
- pliki tekstowe z komunikatami i etykietami przycisków.

W Javie takie pliki powiązane nazywają się **zasobami**.

Spójrz na przykład na okno dialogowe z napisem widoczne na rysunku 5.4.

Rysunek 5.4.
Wyświetlony
obraz i tekst



Oczywiście tytuł książki i informacja o prawach autorskich w następnym wydaniu książki zostaną zmienione. Aby ułatwić sobie zmianę tych informacji, umieścimy ten tekst w pliku, zamiast kodować go w łańcuchu.

Tylko gdzie umieścić taki plik jak *about.txt*? Najwygodniej byłoby go wrzucić razem z pozostałymi plikami programu do archiwum JAR.

Klasa `Class` zawiera praktyczną usługę do znajdowania plików zasobów. Poniżej znajduje się lista czynności, jakie należy wykonać, aby z niej skorzystać:

1. Zdobądź obiekt klasy `Class` zawierający interesujący Cię zasób — na przykład `ResourceTest.class`.
2. Niektóre metody, takie jak `getImage` z klasy `ImageIcon`, przyjmują adresy URL określające położenie zasobów. Wówczas należy zastosować wywołanie `URL url = cl.getResource("about.gif");`.
3. W przeciwnym przypadku należy użyć metody `getResourceAsStream`, aby uzyskać strumień wejściowy do wczytywania danych z pliku.

Maszyna wirtualna Javy potrafi znaleźć klasę, a więc potem może także poszukać powiązanych z nią zasobów w tej samej lokalizacji. Załóżmy, że klasa `ResourceTest` znajduje się w pakiecie `resources`. Wówczas plik `ResourceTest.class` znajduje się w katalogu `resources`, w którym należy też umieścić plik ikony.

Ewentualnie, zamiast umieszczać plik zasobu w tym samym katalogu, w którym znajduje się klasa, można podać względną lub bezwzględną ścieżkę, na przykład:

```
data/about.txt
/corejava/title.txt
```

Jedynym zadaniem funkcji ładowania zasobów jest automatyzacja ładowania plików. Nie ma żadnych standardowych metod do interpretacji zawartości plików zasobów. Każdy program musi mieć zdefiniowane własne techniki.

Zasoby są często wykorzystywane także do internacjonalizacji programów. Teksty, które powinny występować w wielu językach, na przykład powiadomienia i nazwy elementów interfejsu użytkownika, przechowuje się w plikach zasobów — po jednym dla każdego języka. **API internacjonalizacji**, którego opis znajduje się w rozdziale 7. tomu II, obsługuje standardową metodę organizacji i dostępu do plików lokalizacyjnych.

Na listingu 5.14 znajduje się program demonstrujący zastosowanie techniki ładowania zasobów. (Nie przejmuj się kodem wczytującym tekst i wyświetlającym okna dialogowe — wszystko wyjaśni się później). Skompiluj program, utwórz plik JAR i uruchom go:

```
javac resource/ResourceTest.java
jar cvfe ResourceTest.jar resources.ResourceTest \
resources/*.class resources/*.gif resources/data/*.txt corejava/*.txt
java -jar ResourceTest.jar
```

Przenieś plik JAR do innego katalogu i uruchom go ponownie, aby sprawdzić, czy program wczytuje zasoby z archiwum, a nie z bieżącego katalogu.

Listing 5.14. resources/ResourceTest.java

```
package resources;

import java.io.*;
import java.net.*;
import java.nio.charset.*;
import javax.swing.*;

/**
 * @version 1.5 2018-03-15
 * @author Cay Horstmann
 */
public class ResourceTest
{
    public static void main(String[] args) throws IOException
    {
        Class c1 = ResourceTest.class;
        URL aboutURL = c1.getResource("about.gif");
        var icon = new ImageIcon(aboutURL);

        InputStream stream = c1.getResourceAsStream("data/about.txt");
        var about = new String(stream.readAllBytes(), "UTF-8");

        InputStream stream2 = c1.getResourceAsStream("/corejava/title.txt");
        var title = new String(stream2.readAllBytes(),
            ↪StandardCharsets.UTF_8).trim();

        JOptionPane.showMessageDialog(
            null, about, title, JOptionPane.INFORMATION_MESSAGE, icon);
    }
}
```

java.lang.Class 1.0

- URL getResource(String name) **1.1**
- InputStream getResourceAsStream(String name) **1.1**

Znajduje zasób w tym samym miejscu, w którym znajduje się klasa, i zwraca URL lub strumień wejściowy, przy użyciu którego można załadować ten zasób. Jeśli nie znajdzie zasobu, zwraca `null`, a więc nie zgłasza wyjątku w przypadku błędu wejścia-wyjścia.

5.9.4. Zastosowanie refleksji w analizie funkcjonalności klasy

Poniżej znajduje się zwięzły opis najważniejszych funkcji mechanizmu refleksji, które umożliwiają analizę struktury klasy.

Trzy klasy — `Field`, `Method` i `Constructor` — dostępne w pakiecie `java.lang.reflect` opisują odpowiednio pola, metody i konstruktory klasy. Każda z nich ma metodę o nazwie `getName`, która zwraca nazwę odpowiedniego elementu. Klasa `Field` ma metodę `getType`, która zwraca obiekt typu `Class`, zawierający informacje o typie pola. Klasy `Method` i `Constructor` mają metody informujące o typach parametrów, a klasa `Method` informuje dodatkowo o typie zwrotnym. Każda z trzech wymienionych klas ma metodę `getModifiers`, która zwraca liczbę całkowitą z włączonymi i wyłączonymi różnymi bitami, określającą użyte modyfikatory, jak `public` i `static`. Do analizy liczby zwróconej przez tę metodę można użyć metod statycznych klasy `Modifier` dostępnej w pakiecie `java.lang.reflect`. Aby sprawdzić, czy metoda lub konstruktor miał modyfikator `public`, `private` lub `final`, należy użyć metod `isPublic`, `isPrivate` lub `isFinal` dostępnych w klasie `Modifier`. Jedyne, co jest potrzebne, to odpowiednia metoda w klasie `Modifier` działająca na liczbie zwróconej przez metodę `getModifiers`. Modyfikatory można także drukować za pomocą metody `Modifier.toString`.

Metody `getFields`, `getMethods` i `getConstructors` klasy `Class` zwracają tablice *publicznych* pól, metod i konstruktorów klasy. Do tego wliczają się publiczne składowe nadklasy. Metody `getDeclaredFields`, `getDeclaredMethods` i `getDeclaredConstructors` klasy `Class` zwracają tablice zawierające wszystkie pola, metody i konstruktory zadeklarowane w klasie. Wliczają się do nich składowe prywatne, pakietów i chronione, jak również składowe z dostępem pakietowym, ale nie nadklasy.

Listing 5.15 prezentuje sposób wydrukowania wszystkich informacji o klasie. Ten program monituje o podanie nazwy klasy, po czym drukuje sygnatury wszystkich metod i konstruktorów oraz nazwy wszystkich pól klasy. Jeśli na przykład programowi zostanie podana klasa `java.lang.Double`, wydrukuje on następujące dane:

```
public final class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(double);
    public java.lang.Double(java.lang.String);

    public boolean equals(java.lang.Object);
```

```

    public static java.lang.String toString(double);
    public java.lang.String toString();
    public static int hashCode(double);
    public int hashCode();
    public static double min(double, double);
    public static double max(double, double);
    public static native long doubleToRawLongBits(double);
    public static long doubleToLongBits(double);
    public static native double longBitsToDouble(long);
    public int compareTo(java.lang.Double);
    public volatile int compareTo(java.lang.Object);
    public static int compare(double, double);
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static java.lang.Double valueOf(java.lang.String);
    public static java.lang.Double valueOf(double);
    public static java.lang.String toHexString(double);
    public volatile java.lang.Object resolveConstantDesc(
        java.lang.invoke.MethodHandles$Lookup);
    public java.lang.Double resolveConstantDesc(java.lang.invoke.MethodHandles$Lookup);
    public java.util.Optional describeConstable();
    public boolean isNaN();
    public static boolean isNaN(double);
    public static double sum(double, double);
    public boolean isInfinite();
    public static boolean isInfinite(double);
    public static boolean isFinite(double);
    public static double parseDouble(java.lang.String);

    public static final double POSITIVE_INFINITY;
    public static final double NEGATIVE_INFINITY;
    public static final double NaN;
    public static final double MAX_VALUE;
    public static final double MIN_NORMAL;
    public static final double MIN_VALUE;
    public static final int MAX_EXPONENT;
    public static final int MIN_EXPONENT;
    public static final int SIZE;
    public static final int BYTES;
    public static final java.lang.Class TYPE;
    private final double value;
    private static final long serialVersionUID;
}

```

Należy zauważyć, że program ten potrafi przeanalizować każdą klasę, którą interpreter Javy potrafi załadować, a nie tylko te klasy, które były dostępne w czasie kompilacji. Ten program będziemy wykorzystywać w kolejnym rozdziale, w którym będziemy zaglądać do wnętrza klas wewnętrznych generowanych automatycznie przez kompilator.

java.lang.Class 1.0

■ Field[] getFields() 1.1

Listing 5.15. reflection/ReflectionTest.java

```
package reflection;

import java.util.*;
import java.lang.reflect.*;

/**
 * Ten program wykorzystuje technikę refleksji do wydrukowania pełnych informacji o klasie.
 * @version 1.12 2021-06-15
 * @author Cay Horstmann
 */
public class ReflectionTest
{
    public static void main(String[] args)
    {
        // Wczytanie nazwy klasy z argumentów wiersza poleceń lub danych od użytkownika.
        String name;
        if (args.length > 0) name = args[0];
        else
        {
            var in = new Scanner(System.in);
            System.out.println("Podaj nazwę klasy (np. java.util.Date): ");
            name = in.next();
        }

        // drukowanie modyfikatorów i nazwy klasy oraz nazwy nadklasy (if != Object)
        Class cl = Class.forName(name);
        String modifiers = Modifier.toString(cl.getModifiers());
        if (modifiers.length() > 0) System.out.print(modifiers + " ");
        if (cl.isSealed())
            System.out.print("sealed ");
        if (cl.isEnum())
            System.out.print("enum " + name);
        else if (cl.isRecord())
            System.out.print("record " + name);
        else if (cl.isInterface())
            System.out.print("interface " + name);
        else
            System.out.print("class " + name);
        Class supercl = cl.getSuperclass();
        if (supercl != null && supercl != Object.class) System.out.print(" extends "
            + supercl.getName());

        printInterfaces(cl);
        printPermittedSubclasses(cl);

        System.out.print("\n{\n");
        printConstructors(cl);

        System.out.println();
        printMethods(cl);
        System.out.println();
        printFields(cl);
        System.out.println("}");
    }
}
```



```

/**
 * Drukowanie wszystkich konstruktorów klasy.
 * @param cl klasa
 */
public static void printConstructors(Class cl)
{
    Constructor[] constructors = cl.getDeclaredConstructors();

    for (Constructor c : constructors)
    {
        String name = c.getName();
        System.out.print("  ");
        String modifiers = Modifier.toString(c.getModifiers());
        if (modifiers.length() > 0) System.out.print(modifiers + " ");
        System.out.print(name + "(");

        // Drukowanie typów parametrów.
        Class[] paramTypes = c.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++)
        {
            if (j > 0) System.out.print(", ");
            System.out.print(paramTypes[j].getName());
        }
        System.out.println(")");
    }
}

/**
 * Drukuje wszystkie metody klasy.
 * @param cl klasa
 */
public static void printMethods(Class cl)
{
    Method[] methods = cl.getDeclaredMethods();

    for (Method m : methods)
    {
        Class retType = m.getReturnType();
        String name = m.getName();

        System.out.print("  ");
        // Drukowanie modyfikatorów, typu zwrotnego i nazwy metody.
        String modifiers = Modifier.toString(m.getModifiers());
        if (modifiers.length() > 0) System.out.print(modifiers + " ");
        System.out.print(retType.getName() + " " + name + "(");

        // Drukowanie typów parametrów.
        Class[] paramTypes = m.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++)
        {
            if (j > 0) System.out.print(", ");
            System.out.print(paramTypes[j].getName());
        }
        System.out.println(")");
    }
}

```

```
/**
 * Drukowanie wszystkich pól klasy.
 * @param cl klasa
 */
public static void printFields(Class cl)
{
    Field[] fields = cl.getDeclaredFields();

    for (Field f : fields)
    {
        Class type = f.getType();
        String name = f.getName();
        System.out.print(" ");
        String modifiers = Modifier.toString(f.getModifiers());
        if (modifiers.length() > 0) System.out.print(modifiers + " ");
        System.out.println(type.getName() + " " + name + ";");
    }
}

/**
 * Drukuje wszystkie dozwolone podtypy zapieczętowanej klasy
 * @param cl a class
 */
public static void printPermittedSubclasses(Class cl)
{
    if (cl.isSealed())
    {
        Class<?>[] permittedSubclasses = cl.getPermittedSubclasses();
        for (int i = 0; i < permittedSubclasses.length; i++)
        {
            if (i == 0)
                System.out.print(" permits ");
            else
                System.out.print(", ");
            System.out.print(permittedSubclasses[i].getName());
        }
    }
}

/**
 * Drukuje wszystkie bezpośrednio zaimplementowane interfejsy klasy
 * @param cl a class
 */
public static void printInterfaces(Class cl)
{
    Class<?>[] interfaces = cl.getInterfaces();
    for (int i = 0; i < interfaces.length; i++)
    {
        if (i == 0)
            System.out.print(cl.isInterface() ? " extends " : " implements ");
        else
            System.out.print(", ");
        System.out.print(interfaces[i].getName());
    }
}
}
```

■ `Field[] getDeclaredFields()` **1.1**

Metoda `getFields` zwraca tablicę zawierającą obiekty `Field` reprezentujące pola publiczne klasy lub nadklasy. Metoda `getDeclaredFields` zwraca tablicę obiektów `Field` reprezentujących wszystkie pola klasy. Obie metody zwracają tablicę o zerowej długości, jeśli nie ma takich pól lub obiekt `Class` reprezentuje typ podstawowy bądź tablicowy.

■ `Method[] getMethods` **1.1**

■ `Method[] getDeclaredMethods()` **1.1**

Zwraca tablicę obiektów `Method`. Metoda `getMethods` zwraca metody publiczne, wliczając metody odziedziczone. Metoda `getDeclaredMethods` zwraca wszystkie metody klasy lub interfejsu, ale nie uwzględnia metod odziedziczonych.

■ `Constructor[] getConstructors()` **1.1**

■ `Constructor[] getDeclaredConstructors()` **1.1**

Zwraca tablicę obiektów `Constructor` reprezentujących wszystkie konstruktory publiczne (`getConstructors`) lub wszystkie konstruktory w ogóle (`getDeclaredConstructors`) klasy reprezentowanej przez obiekt `Class`.

■ `isInterface()`

Zwraca `true`, jeśli ten obiekt `Class` opisuje interfejs (interfejsy są opisane w rozdziale 6.).

■ `isEnum()` **1.5**

Zwraca `true`, jeśli ten obiekt `Class` opisuje wyliczenie.

■ `isRecord()` **16**

Zwraca `true`, jeśli ten obiekt `Class` opisuje rekord.

■ `RecordComponent[] getRecordComponents()` **16**

Zwraca tablicę obiektów `RecordComponent`, które opisują pola rekordu, lub `null`, jeśli ta klasa nie jest rekordem.

■ `String getPackageName()` **9**

Pobiera nazwę pakietu zawierającego ten typ lub pakietu typu elementu, jeśli jest to typ tablicowy, lub `"java.lang"`, jeśli jest to typ podstawowy.

```
java.lang.reflect.Field 1.1
java.lang.reflect.Method 1.1
java.lang.reflect.Constructor 1.1
```

■ `Class getDeclaringClass()`

Zwraca obiekt klasy `Class` reprezentujący klasę, która definiuje dany konstruktor, metodę lub pole.

- `Class[] getExceptionTypes()` (tylko klasy `Constructor` i `Method`)

Zwraca tablicę obiektów `Class`, które reprezentują typy wyjątków powodowanych przez metodę.

- `int getModifiers()`

Zwraca liczbę całkowitą opisującą modyfikatory konstruktora, metody lub pola. Do analizy zwróconej wartości służą metody klasy `Modifier`.

- `String getName()`

Zwraca w postaci łańcucha nazwę konstruktora, metody lub pola.

- `Class[] getParameterTypes()` (tylko klasy `Constructor` i `Method`)

Zwraca tablicę obiektów klasy `Class` reprezentujących typy parametrów.

- `Class getReturnType()` (tylko w klasie `Method`)

Zwraca obiekt klasy `Class` reprezentujący typ zwrrotny.

`java.lang.reflect.RecordComponent` **16**

- `String getName()`

- `Class<?> getType()`

Pobiera nazwę i typ tego komponentu rekordu.

- `Method getAccessor()`

Zwraca obiekt `Method` dający dostęp do tego komponentu rekordu.

`java.lang.reflect.Modifier` **1.1**

- `static String toString(int modifiers)`

Zwraca w postaci łańcucha modyfikatory odpowiadające bitom ustawionym przez metodę `modifiers`.

- `static boolean isAbstract(int modifiers)`

- `static boolean isFinal(int modifiers)`

- `static boolean isInterface(int modifiers)`

- `static boolean isNative(int modifiers)`

- `static boolean isPrivate(int modifiers)`

- `static boolean isProtected(int modifiers)`

- `static boolean isPublic(int modifiers)`

- `static boolean isStatic(int modifiers)`

- `static boolean isStrict(int modifiers)`

- `static boolean isSynchronized(int modifiers)`

- `static boolean isVolatile(int modifiers)`

Sprawdza bit w wartości `modifiers` odpowiadający modyfikatorowi znajdującemu się w nazwie metody.

5.9.5. Refleksja w analizie obiektów w czasie działania programu

W poprzednim podrozdziale nauczyliśmy się sprawdzać *nazwy* i *typy* pól egzemplarza:

- Tworzymy odpowiedni obiekt klasy `Class`.
- Wywołujemy na rzecz obiektu `Class` metodę `getDeclaredFields`.

Teraz pójdziemy o krok dalej i dobierzemy się do **zawartości** pól danych. Oczywiście zawartość określonego pola obiektu o znanych w trakcie pisania programu typie i nazwie można podejrzec bez trudu. Jednak refleksja umożliwia uzyskanie informacji o polach obiektów, które w czasie kompilacji nie były jeszcze znane.

Kluczowe znaczenie ma w tym przypadku metoda `get` z klasy `Field`. Jeśli `f` jest obiektem typu `Field` (na przykład utworzonym za pomocą metody `getDeclaredFields`), a `obj` jest obiektem klasy, której polem jest `f`, wywołanie `f.get(obj)` zwraca obiekt, którego wartością jest aktualna wartość pola obiektu `obj`. Przeanalizujmy to nieco skomplikowane zagadnienie na przykładzie.

```
var harry = new Employee("Henryk Kwiatek", 35000, 10, 1, 1989);
Class cl = harry.getClass();
// Obiekt Class reprezentujący pracownika.
Field f = cl.getDeclaredField("name");
// Pole name klasy Employee.
Object v = f.get(harry);
// Wartość pola name obiektu harry
// tj. obiekt klasy String "Henryk Kwiatek".
```

Oczywiście wartości, które można pobrać, można też ustawiać. Wywołanie `f.set(obj, value)` ustawia wartość pola reprezentowanego przez `f` obiektu `obj` na nową wartość.

Ten kod sprawia jednak jeden problem. Ponieważ pole `name` jest prywatne, metody `get` i `set` spowodują wyjątek `IllegalAccessException`. Za pomocą tych metod można uzyskać dostęp tylko do wartości dostępnych pól. Zabezpieczenia w Javie zezwalają na sprawdzenie, jakie pola zawiera obiekt, ale nie pozwalają na sprawdzenie ich wartości bez odpowiednich uprawnień dostępu.

Przy standardowych ustawieniach mechanizm refleksji honoruje mechanizmy ochronne Javy. Można jednak ominąć ustawienia ochrony dostępu. W tym celu należy wywołać metodę `setAccessible` na rzecz obiektu klasy `Field`, `Method` lub `Constructor`. Na przykład:

```
f.setAccessible(true); // Teraz można wywołać f.get(harry);.
```

Metoda `setAccessible` należy do klasy `AccessibleObject`, która jest wspólną nadklasą klas `Field`, `Method` i `Constructor`. Została ona utworzona z myślą o debuggerach, schowkach i podobnych mechanizmach. Nieco dalej używamy tej metody dla generycznej metody `toString`.

Wywołanie metody `setAccessible` powoduje zgłoszenie wyjątku, jeśli dostęp nie zostanie udzielony. Dostępu może odmówić system modułów (rozdział 9. w tomie II) lub menedżer zabezpieczeń (rozdział 10. w tomie II). Menedżery zabezpieczeń są używane rzadko i w Javie 17 otrzymały status wycofywanych. Natomiast od Javy 9 każdy program zawiera moduły, ponieważ API został na nie podzielony.

Na przykład program znajdujący się na końcu tego punktu zagląda do wnętrza obiektów `ArrayList` i `Integer`. Kiedy go uruchomisz w wersji Javy od 9 do 16, pojawi się następujące zło-wieszcze ostrzeżenie:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by objectAnalyzer.ObjectAnalyzer
↳(file:/home/cay/books/cj11/code/v1ch05/bin/) to field
↳java.util.ArrayList.serialVersionUID
WARNING: Please consider reporting this to the maintainers of
↳objectAnalyzer.ObjectAnalyzer
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
↳access operations
WARNING: All illegal access operations will be denied in a future release
```

Jeśli uruchomisz ten program w Javie 17, zostanie zgłoszony wyjątek `InaccessibleObjectException`.

Aby program nadal działał, „otwórz” pakiety `java.util` i `java.lang` w module `java.base` do „modułu bez nazwy”. Szczegółowe informacje znajdują się w rozdziale 9. tomu II. Oto cała składnia:

```
java --add-opens java.base/java.util=ALL-UNNAMED \
--add-opens java.base/java.lang=ALL-UNNAMED \
objectAnalyzer.ObjectAnalyzerTest
```



Przyszłe wersje bibliotek mogą używać *uchwyty* do zmiennych zamiast refleksji w celu odczytu i zapisu pól. Obiekt `VarHandle` jest podobny do `Field`. Przy jego użyciu możesz odczytać lub zapisać określone pole dowolnego egzemplarza wybranej klasy. Aby jednak uzyskać `VarHandle`, biblioteka musi mieć obiekt `Lookup`:

```
public Object getFieldValue(Object obj, String fieldName, Lookup lookup)
throws NoSuchFieldException, IllegalAccessException
{
    Class<?> cl = obj.getClass();
    Field field = cl.getDeclaredField(fieldName);
    VarHandle handle = MethodHandles.privateLookupIn(cl, lookup)
        .unreflectVarHandle(field);
    return handle.get(obj);
}
```

To zadziała, pod warunkiem że obiekt `Lookup` zostanie wygenerowany w module, który ma uprawnienia dostępu do pola. Metoda w module po prostu wywołuje `MethodHandles.lookup()`, która zwraca obiekt i posiada prawa dostępu wywołującego. W ten sposób jeden moduł może udzielić prawa dostępu do prywatnych składowych innego modułu. Praktycznym problemem jest to, jak te uprawnienia można nadać jak najmniejszym wysiłkiem.

Dopóki można jeszcze korzystać z tej techniki, przyjrzymy się generycznej metodzie `toString`, która działa dla każdej klasy (listing 5.16). Generyczna metoda `toString` pobiera wszystkie pola egzemplarza za pomocą metody `getDeclaredFields`. Następnie metoda `setAccessible` umożliwia dostęp do wszystkich tych pól. Pobierane są nazwa i wartość każdego pola. Program przedstawiony na listingu 5.16 rekurencyjnie wywołuje metodę `toString`, zamieniając każdą wartość na łańcuch.

Listing 5.16. `objectAnalyzerTest/ObjectAnalyzerTest.java`

```
package objectAnalyzer;

import java.util.*;

/**
 * Ten program analizuje obiekty za pomocą refleksji.
 * @version 1.13 2018-03-16
 * @author Cay Horstmann
 */
public class ObjectAnalyzerTest
{
    public static void main(String[] args)
        throws ReflectiveOperationException
    {
        var squares = new ArrayList<Integer>();
        for (int i = 1; i <= 5; i++)
            squares.add(i * i);
        System.out.println(new ObjectAnalyzer().toString(squares));
    }
}
```

Generyczna metoda `toString` musi rozwiązywać kilka problemów. Cykliczne odwołania mogą spowodować nieskończoną rekurencję. Dlatego klasa `ObjectAnalyzer` (listing 5.17) zapamiętuje obiekty, które były już odwiedzane. Aby zajrzeć do tablic, potrzebne jest zastosowanie innej metody. Więcej szczegółów na ten temat znajduje się w kolejnym podrozdziale.

Listing 5.17. `objectAnalyzer/ObjectAnalyzer.java`

```
package objectAnalyzer;

import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Array;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.ArrayList;

class ObjectAnalyzer
{
    private ArrayList<Object> visited = new ArrayList<>();

    /**
     * Konwertuje obiekt na łańcuch zawierający listę wszystkich pól.
     * @param obj obiekt
     * @return łańcuch zawierający nazwę klasy obiektu oraz nazwy i wartości wszystkich pól.
     */
}
```

```

public String toString(Object obj)
    throws ReflectiveOperationException
{
    if (obj == null) return "null";
    if (visited.contains(obj)) return "...";
    visited.add(obj);
    Class cl = obj.getClass();
    if (cl == String.class) return (String) obj;
    if (cl.isArray())
    {
        String r = cl.getComponentType() + "[]{";
        for (int i = 0; i < Array.getLength(obj); i++)
        {
            if (i > 0) r += ",";
            Object val = Array.get(obj, i);
            if (cl.getComponentType().isPrimitive()) r += val;
            else r += toString(val);
        }
        return r + "}";
    }

    String r = cl.getName();
    // Inspekcja pól tej klasy i wszystkich nadklas.
    do
    {
        r += "[";
        Field[] fields = cl.getDeclaredFields();
        AccessibleObject.setAccessible(fields, true);
        // Pobranie nazw i wartości wszystkich pól.
        for (Field f : fields)
        {
            if (!Modifier.isStatic(f.getModifiers()))
            {
                if (!r.endsWith("(")) r += ",";
                r += f.getName() + "=";
                Class t = f.getType();
                Object val = f.get(obj);
                if (t.isPrimitive()) r += val;
                else r += toString(val);
            }
        }
        r += "]";
        cl = cl.getSuperclass();
    }
    while (cl != null);

    return r;
}
}

```

Za pomocą metody `toString` można zajrzeć do środka każdego obiektu. Na przykład wywołanie:

```

var squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));

```


zwraca:

```
java.util.ArrayList[elementData=class java.lang.Object[] {java.lang.Integer[value=1] [],  
java.lang.Integer[value=4] [],java.lang.Integer[value=9] [],  
java.lang.Integer[value=16] [],  
java.lang.Integer[value=25] [],null,null,null,null,null},size=5][modCount=5] [] []
```

Przy użyciu generycznej metody `toString` można zaimplementować metody `toString` w poszczególnych klasach:

```
public String toString()  
{  
    return new ObjectAnalyzer().toString(this);  
}
```

Jest to bezproblemowa metoda na utworzenie uniwersalnej metody `toString`. Zanim jednak zaczniesz się cieszyć, że to koniec z samodzielnym implementowaniem metody `toString`, przypomnij sobie, że dni niekontrolowanego dostępu do mechanizmów wewnętrznych są już policzone.

java.lang.reflect.AccessibleObject 1.2

- `void setAccessible(boolean flag)`
Ustawia lub czyści znacznik dostępności obiektu lub zgłasza wyjątek `IllegalAccessException` w przypadku odmowy dostępu.
- `boolean trySetAccessible()` 9
Ustawia znacznik dostępności dla tego dostępnego obiektu lub zwraca fałsz w przypadku odmowy dostępu.
- `boolean canAccess()`
Sprawdza, czy wywołujący ma dostęp do obj przez ten obiekt pola, metody lub konstruktora. Dla statycznego pola lub metody albo dla konstruktora należy przekazać `null`.
- `static void setAccessible(AccessibleObject[] array, boolean flag)`
Ustawia znacznik dostępności obiektów tablicowych.

java.lang.Class 1.1

- `Field getField(String name)`
- `Field[] getFields()`
Zwraca pole publiczne o danej nazwie lub tablicę wszystkich pól.
- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`
Zwraca pole o danej nazwie zadeklarowane w klasie lub tablicę wszystkich pól.

```
java.lang.reflect.Field 1.1
```

■ `Object get(Object obj)`

Zwraca wartość pola reprezentowanego przez obiekt `Field` w obiekcie `obj`.

■ `void set(Object obj, Object newValue)`

Ustawia pole reprezentowane przez obiekt `Field` w obiekcie `obj` na nową wartość.

5.9.6. Zastosowanie refleksji w generycznym kodzie tablicowym

Klasa `Array` dostępna w pakiecie `java.lang.reflect` umożliwia dynamiczne tworzenie tablic. Możliwość ta jest wykorzystana na przykład w implementacji metody `copyOf` w klasie `Array`. Przypomnijmy sobie, jak użyć tej metody do powiększenia tablicy, która została wypełniona.

```
var a = new Employee[100];
...
// Tablica jest pełna.
a = Arrays.copyOf(a, 2 * a.length);
```

Jak napisać taką metodę? Pomocny jest fakt, że tablicę `Employee[]` można przekonwertować na tablicę `Object[]`. Brzmi obiecująco. Oto pierwsza próba.

```
public static Object[] badCopyOf(Object[] a, int newLength) // nieprzydatna
{
    var newArray = new Object[newLength];
    System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
    return newArray;
}
```

Niestety jest problem z *użyciem* powstałej tablicy. Ten kod zwraca tablicę *obiektów* (`Object[]`). Odpowiada za to poniższy wiersz:

```
new Object[newLength]
```

Tablice obiektów *nie można* rzutować na tablicę pracowników (`Employee[]`) — w czasie działania programu zostałby spowodowany wyjątek `ClassCastException`. Problem polega na tym, że jak nam wiadomo, tablice w Javie pamiętają typ swoich elementów, to znaczy typ elementu, który był użyty w wyrażeniu `new` podczas ich tworzenia. Można tymczasowo przekonwertować tablicę `Employee[]` na `Object[]`, a później wrócić do poprzedniego stanu. Niemożliwe jednak jest rzutowanie tablicy, która od chwili powstania jest typu `Object[]`, na typ `Employee[]`. Do napisania tego rodzaju generycznego kodu tablicy konieczna jest możliwość utworzenia nowej tablicy *tego samego* typu co oryginalna tablica. Do tego celu potrzebne są metody dostępne w klasie `Array` z pakietu `java.lang.reflect`. Kluczowe znaczenie ma metoda `newInstance` klasy `Array`, która tworzy nową tablicę. Metoda ta przyjmuje jako argumenty typ elementów i żądany rozmiar tablicy.

```
Object newArray = Array.newInstance(componentType, newLength);
```

Aby tego dokonać, trzeba znać typ elementów i rozmiar nowej tablicy.

Pierwszą wartość można uzyskać za pomocą wywołania `Array.getLength(a)`. Statyczna metoda `getLength` klasy `Array` zwraca rozmiar tablicy. Aby sprawdzić typ elementów nowej tablicy:

1. Utwórz obiekt klasowy `a`.
2. Sprawdź, czy to na pewno jest tablica.
3. Znajdź odpowiedni typ dla tablicy za pomocą metody `getComponentType` (która jest zdefiniowana tylko dla obiektów klasowych reprezentujących tablice) klasy `Class`.
4. Z drugiej strony dla każdego obiektu `Class` reprezentującego klasę `C` metoda `arrayType` zwraca obiekt `Class` reprezentujący `C[]`.

Dlaczego metoda `getLength` należy do klasy `Array`, a `getComponentType` do klasy `Class`? Nie wiadomo — czasami wydaje się, że rozkład w klasach metod refleksyjnych jest nieco przypadkowy.

Oto potrzebny kod:

```
public static Object goodCopyOf(Object a, int newLength)
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
    return newArray;
}
```

Należy zauważyć, że metoda `copyOf` może powiększać tablice każdego typu, nie tylko przechowujące obiekty.

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodCopyOf(a, 10);
```

Aby to było możliwe, parametr metody `goodCopyOf` jest typu `Object`, a *nie tablicą obiektów* (`Object[]`). Tablicę typu `int[]` można przekonwertować na `Object`, ale nie na tablicę obiektów!

Listing 5.18 demonstruje obie metody powiększania tablicy. Zauważ, że rzutowanie typu zwróconego metody `badCopyOf` spowoduje wyjątek.

Listing 5.18. `arrays/CopyOfTest.java`

```
package arrays;

import java.lang.reflect.*;
import java.util.*;

/**
 * Ten program demonstruje zastosowanie refleksji do manipulacji tablicami.
 * @version 1.2 2012-05-04
 * @author Cay Horstmann
 */
public class CopyOfTest
{
    public static void main(String[] args)
```

```

{
    int[] a = { 1, 2, 3 };
    a = (int[]) goodCopyOf(a, 10);
    System.out.println(Arrays.toString(a));

    String[] b = { "Tomek", "Daniel", "Henryk" };
    b = (String[]) goodCopyOf(b, 10);
    System.out.println(Arrays.toString(b));

    System.out.println("Poniższe wywołanie spowoduje wyjątek.");
    b = (String[]) badCopyOf(b, 10);
}

/**
 * Ta metoda próbuje powiększyć tablicę, tworząc nową tablicę i kopiując wszystkie elementy.
 * @param a tablica, która ma być powiększona.
 * @param newLength nowa długość tablicy
 * @return większa tablica zawierająca wszystkie elementy tablicy a. Zwrócona tablica jest
 * typu Object[], a nie takiego samego jak a.
 */
public static Object[] badCopyOf(Object[] a, int newLength) // nieprzydatna
{
    var newArray = new Object[newLength];
    System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
    return newArray;
}

/**
 * Ta metoda powiększa tablicę, tworząc nową tablicę tego samego typu
 * i kopiując wszystkie elementy.
 * @param a tablica, która ma być powiększona. Może to być tablica obiektów lub
 * elementów typu podstawowego.
 * @return większa tablica zawierająca wszystkie elementy tablicy a.
 */
public static Object goodCopyOf(Object a, int newLength)
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
    return newArray;
}
}

```

java.lang.Class 1.1

■ boolean isArray()

Zwraca true, jeśli ten obiekt reprezentuje typ tablicowy.

■ Class<?> getComponentType()

■ Class<?> componentType() 12

Zwraca obiekt Class opisujący typ komponentu, jeśli ten obiekt reprezentuje typ tablicowy, lub null w przeciwnym razie.

■ `Class<?> arrayType()` 12

Zwraca obiekt `Class` opisujący typ tablicy, którego typ komponentu jest reprezentowany przez ten obiekt.

`java.lang.reflect.Array` 1.1

■ `static Object get(Object array, int index)`

■ `static xxx getXxx(Object array, int index)`

`xxx` to jeden z typów podstawowych: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`. Te metody zwracają wartość przechowywaną w określonym indeksie tablicy.

■ `static void set(Object array, int index, Object newValue)`

■ `static setXxx(Object array, int index, xxx newValue)`

`xxx` to jeden z typów podstawowych: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`. Te metody zapisują nową wartość w danej tablicy w określonym indeksie.

■ `static int getLength(Object array)`

Zwraca długość tablicy.

■ `static Object newInstance(Class componentType, int length)`

■ `static Object newInstance(Class componentType, int[] lengths)`

Zwraca nową tablicę danego typu o określonych rozmiarach.

5.9.7. Wywoływanie dowolnych metod i konstruktorów

W językach C i C++ można wywołać dowolną funkcję, posługując się ustawionym na nią wskaźnikiem. Na pierwszy rzut oka wydaje się, że w Javie nie ma wskaźników do metod — umożliwiając one podanie lokalizacji metody innej metodzie, dzięki czemu ta druga może później wywołać tę pierwszą. Projektanci języka Java stwierdzili nawet, że wskaźniki do metod nie są bezpieczne, mogą bowiem stanowić źródło błędów, a lepszym od nich rozwiązaniem są **interfejsy** i wyrażenia lambda (opisane w kolejnym rozdziale). Jednak dzięki refleksji także w Javie można wywoływać dowolne metody.

Przypomnijmy sobie, że za pomocą metody `get` klasy `Field` można sprawdzić dowolne pole obiektu. Podobnie klasa `Method` zawiera metodę `invoke`, która umożliwia wywołanie metody zapakowanej w bieżący obiekt klasy `Method`. Sygnatura metody `invoke` wygląda następująco:

```
Object invoke(Object obj, Object... args)
```

Pierwszy jest parametr niejawny, a pozostałe obiekty to parametry jawne.

W przypadku metody statycznej pierwszy parametr jest ignorowany — można go ustawić na `null`.

Jeśli na przykład `m1` reprezentuje metodę `getName` klasy `Employee`, poniższy kod pokazuje, jak można ją wywołać:

```
String n = (String) m1.invoke(harry);
```

Jeśli typ zwrotny jest podstawowy, metoda `invoke` zwróci typ opakowania. Załóżmy na przykład, że `m2` reprezentuje metodę `getSalary` klasy `Employee`. Zwrócony obiekt będzie typu `Double` i trzeba wykonać odpowiednie rzutowanie. Można do tego celu zastosować automatyczne odpakowywanie.

```
double s = (Double) m2.invoke(harry);
```

Jak uzyskać obiekt klasy `Method`? Można oczywiście wywołać metodę `getDeclaredMethods` i przeszukać zwróconą tablicę w celu znalezienia żądanej metody. Można również wywołać metodę `getMethod` klasy `Class`. Jest ona podobna do metody `getField`, która przyjmuje nazwę pola w postaci łańcucha i zwraca obiekt typu `Field`. Jednak metod o takiej samej nazwie może być kilka, więc należy uważać, aby się nie pomylić. Z tego powodu konieczne jest podanie dodatkowo typów parametrów żądanej metody. Sygnatura metody `getMethod` jest następująca:

```
Method getMethod(String name, Class... parameterTypes)
```

Poniższy kod przedstawia sposób uzyskania wskaźników do metod `getName` i `raiseSalary` klasy `Employee`:

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

W podobny sposób można wywoływać dowolne konstruktory. Należy przekazać typy parametrów wybranego konstruktora do metody `Class.getConstructor` i dostarczyć ich wartości do metody `Constructor.newInstance`:

```
Class c1 = Random.class; // lub dowolna inna klasa z konstruktorem
                        // przyjmującym parametr typu long
Constructor cons = c1.getConstructor(long.class);
Object obj = cons.newInstance(42L);
```



Klasy `Method` i `Constructor` rozszerzają klasę `Executable`. Od Javy 17 klasa `Executable` jest zabezpieczona oraz zezwala na tworzenie tylko podklas `Method` i `Constructor`.

Skoro znamy już zasady używania obiektów klasy `Method`, wykorzystajmy je w praktyce. Listing 5.19 przedstawia program drukujący tabelę wartości funkcji matematycznych, jak `Math.sqrt` lub `Math.sin`. Wydruk wygląda następująco:

```
public static native double java.lang.Math.sqrt(double)
1.0000 | 1.0000
2.0000 | 1.4142
3.0000 | 1.7321
4.0000 | 2.0000
5.0000 | 2.2361
6.0000 | 2.4495
7.0000 | 2.6458
8.0000 | 2.8284
9.0000 | 3.0000
10.0000 | 3.1623
```

Oczywiście kod drukujący tabelę jest niezależny od samej funkcji, dla której zastosowano wcięcia.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
    double y = (Double) f.invoke(null, x);
    System.out.printf("%10.4f | %10.4f%n", x, y);
}
```

W powyższym kodzie `f` jest obiektem typu `Method`. Pierwszy parametr metody `invoke` ma wartość `null`, ponieważ wywoływana jest metoda statyczna.

Wyrównanie funkcji `Math.sqrt` zostało uzyskane poprzez ustawienie `f` na:

```
Math.class.getMethod("sqrt", double.class)
```

czyli metodę klasy `Math` o nazwie `sqrt`, z parametrem typu `double`.

Listing 5.19 przedstawia pełny kod generycznego tabulatora i kilka przykładowych wywołań.

Listing 5.19. `methods/MethodTableTest.java`

```
package methods;

import java.lang.reflect.*;

/**
 * Ten program demonstruje sposób wywoływania metod poprzez refleksję.
 * @version 1.2 2012-05-04
 * @author Cay Horstmann
 */
public class MethodTableTest
{
    public static void main(String[] args) throws ReflectiveOperationException
    {
        // Pobranie wskaźników do metod square i sqrt.
        Method square = MethodTableTest.class.getMethod("square", double.class);
        Method sqrt = Math.class.getMethod("sqrt", double.class);

        // Drukowanie tabel wartości x i y.
        printTable(1, 10, 10, square);
        printTable(1, 10, 10, sqrt);
    }

    /**
     * Zwraca kwadrat liczby.
     * @param x liczba
     * @return x podniesione do kwadratu
     */
    public static double square(double x)
    {
        return x * x;
    }

    /**
     * Drukuje tablicę wartości x i y dla danej metody.

```

```

    * @param od dolnej granicy wartości x
    * @param do górnej granicy wartości x
    * @param n liczba wierszy w tabeli
    * @param f metoda z parametrem i typem zwrrotnym typu double
    */
    public static void printTable(double from, double to, int n, Method f)
        throws ReflectiveOperationException
    {
        // Drukowanie metody jako nagłówek tabeli
        System.out.println(f);

        double dx = (to - from) / (n - 1);

        for (double x = from; x <= to; x += dx)
        {
            double y = (Double) f.invoke(null, x);
            System.out.printf("%10.4f | %10.4f%n", x, y);
        }
    }
}

```

Powyższy program pokazuje, że z obiektami klasy `Method` można zrobić wszystko to co ze wskaźnikami do funkcji w języku C (i delegacjami w C#). Podobnie jak w C, taki styl programowania jest niewygodny i zawsze podatny na błędy. Co się stanie, jeśli metoda zostanie wywołana przy użyciu nieprawidłowych parametrów? Metoda `invoke` spowoduje wyjątek.

Dodatkowo parametry i typy zwrotne metody `invoke` muszą być typu `Object`. Oznacza to konieczność częstego rzutowania w obie strony. W ten sposób kompilator zostaje pozbawiony możliwości sprawdzenia kodu. Przez to błędy ujawniają się tylko podczas testów, kiedy są trudniejsze do naprawienia. Ponadto kod tworzący wskaźnik do metody przy użyciu refleksji jest znacznie wolniejszy niż kod, który wywołuje metody bezpośrednio.

Z wymienionych powodów zalecamy używać obiektów klasy `Method` wyłącznie wtedy, kiedy jest to absolutnie niezbędne. Prawie zawsze lepiej jest używać interfejsów lub (od Java 8) wyrażeń `lambda` (które są tematem kolejnego rozdziału). Zgadza się zwłaszcza ze stanowiskiem projektantów języka Java i nie polecamy używania obiektów typu `Method` dla funkcji zwrrotnych. Dzięki użyciu interfejsów dla metod zwrrotnych (zobacz następny rozdział) kod jest znacznie szybszy i łatwiejszy w utrzymaniu.

java.lang.reflect.Method 1.1

■ `public Object invoke(Object implicitParameter, Object[] explicitParameters)`

Wywołuje metodę reprezentowaną przez obiekt, przekazując podane parametry, oraz zwraca wartość, którą zwraca ta metoda. W przypadku metod statycznych parametr niejawnny powinien mieć wartość `null`. Wartości typów podstawowych należy przekazywać w obiektach opakowujących. Wartości zwrotne typów podstawowych muszą być odpakowywane.

5.10. Porady projektowe dotyczące dziedziczenia

Na zakończenie tego rozdziału przedstawiamy kilka porad dotyczących dziedziczenia.

1. Wspólne metody i pola umieszczaj w nadklasie.

Z tego powodu pole `name` umieszczamy w klasie `Person` zamiast w klasach `Employee` i `Student`.

2. Nie używaj pól chronionych.

Niektórzy programiści uważają, że zdefiniowanie większości pól jako chronionych jest dobrym pomysłem, ponieważ dzięki temu podklasy mają w razie potrzeby do nich dostęp. Jednak mechanizm stojący za słowem kluczowym `protected` nie daje dobrej ochrony, i to z dwóch powodów. Po pierwsze, liczba podklas jest nieskończona — każdy może napisać podklasę naszej klasy, a następnie napisać kod uzyskujący bezpośredni dostęp do chronionych pól egzemplarzy, co stanowi złamanie zasad hermetyzacji. Po drugie, w Javie wszystkie klasy znajdujące się w jednym pakiecie mają dostęp do pól chronionych pozostałych klas, bez względu na to, czy są podklasami.

Użyteczne mogą natomiast być metody chronione, które nie są gotowe do użytku i powinny zostać ponownie zdefiniowane w podklasach.

3. Za pomocą dziedziczenia imituj relację „jest”.

Dziedziczenie umożliwia zaoszczędzenie wielu wierszy kodu, ale niestety jest często nadużywane. Wyobraźmy sobie na przykład, że potrzebujemy klasy o nazwie `Contractor` (pracownik kontraktowy). Pracownik kontraktowy ma imię i nazwisko oraz datę zatrudnienia, ale nie ma stałej pensji. Zamiast tego otrzymuje wynagrodzenie zależne od liczby przepracowanych godzin i nie pracuje na tyle długo, aby dostać podwyżkę. Istnieje więc pokusa, aby utworzyć podklasę `Contractor` klasy `Employee` i dodać pole `hourlyWage` (stawka godzinowa).

```
class Contractor extends Employee
{
    private double hourlyWage;
    . . .
}
```

Nie jest to jednak dobre rozwiązanie, ponieważ każdy obiekt pracownika kontraktowego będzie miał zarówno pole pensji, jak i stawki godzinowej. Stanie się to źródłem niekończących się problemów podczas implementacji metod generujących rachunki lub formularze podatkowe. Będzie konieczne napisanie większej ilości kodu, niż gdyby zrezygnowano na początku z dziedziczenia.

Relacja pracownik – pracownik kontraktowy nie jest typu „jest”. Pracownik kontraktowy nie jest specjalnym typem pracownika.

4. Nie używaj dziedziczenia, jeśli któraś z metod nie działa odpowiednio.

Wyobraźmy sobie, że chcemy napisać klasę o nazwie `Holiday`. Jak wiadomo, każde święto jest jakimś dniem, a dni można reprezentować jako obiekty klasy `GregorianCalendar`. W związku z tym możemy wykorzystać dziedziczenie.

```
class Holiday extends GregorianCalendar { . . . }
```

Niestety zbiór dni wolnych nie jest zamknięty dla wszystkich odziedziczonych metod. Jedną z publicznych metod klasy `GregorianCalendar` jest `add`. Za jej pomocą dni świąteczne można zamienić w dni nieświęteczne:

```
Holiday christmas;  
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Z tego powodu dziedziczenie nie jest właściwą techniką w tym przypadku.

Problem ten nie wystąpiłby w przypadku rozszerzenia klasy niezmiennej. Powiedzmy, że mamy niezmienną klasę reprezentującą datę, podobną do `LocalDate`, ale nie finalną. Jeśli utworzymy podklasę `Holiday`, to nie będzie takiej metody, która mogłaby zamienić święto na dzień nieświęteczny.

5. Przesłaniając metodę, nie zmieniaj jej spodziewanego działania.

Zasada zamienialności ma zastosowanie nie tylko do składni, ale co ważniejsze — do zachowania. Przesłaniając metodę, nie należy bez powodu zmieniać jej zachowania. Kompilator w takiej sytuacji nie pomoże, ponieważ nie może sprawdzić, czy nowa definicja metody jest właściwa. Na przykład problem z metodą `add` w klasie `Holiday` można rozwiązać, definiując tę metodę ponownie w taki sposób, aby nie robiła lub powodowała wyjątek czy też przechodziła do kolejnego święta.

Niestety ta poprawka łamie zasadę zamienialności. Poniższe instrukcje:

```
int d1 = x.get(Calendar.DAY_OF_MONTH);  
x.add(Calendar.DAY_OF_MONTH, 1);  
int d2 = x.get(Calendar.DAY_OF_MONTH);  
System.out.println(d2 - d1);
```

powinny *działać przewidywalnie*, bez względu na to, czy `x` jest typu `GregorianCalendar`, czy `Holiday`.

Oczywiście na ten temat można toczyć zażarte spory dotyczące tego, co należy uważać za przewidywalne działanie. Na przykład niektórzy programiści stwierdzą, że zasada zamienialności wymaga, aby metoda `Manager.equals` ignorowała pole `bonus`, ponieważ robi to także metoda `Employee.equals`. Takie dyskusje są bezcelowe, jeśli prowadzi się je bez odpowiedniego kontekstu. Przede wszystkim należy pamiętać, aby przesłaniając metody w podklasach, nie zacierać przeznaczenia oryginalnego projektu.

6. Wykorzystuj polimorfizm zamiast informacji o typach.

Kiedy napotkasz kod typu:

```

if (x jest typu 1)
    działanie1(x);
else if (x jest typu 2)
    działanie2(x);

```

zawsze pomyśl o polimorfizmie.

Czy *działanie₁* i *działanie₂* reprezentują wspólną koncepcję? Jeśli tak, zamień tę koncepcję na metodę nadklasy lub interfejsu wspólnego dla obu typów. Dzięki temu wystarczy wywołanie typu:

```
x.działanie();
```

Polimorficzny mechanizm dynamicznego przydzielania zadań wywoła odpowiednią metodę.

Kod wykorzystujący metody polimorficzne lub interfejsy jest zdecydowanie łatwiejszy do utrzymania i rozszerzania niż kod zawierający wiele sprawdzeń typów.

7. Nie nadużywaj refleksji.

Możliwość wykrywania pól i metod w czasie działania programu pozwala na pisanie programów o zadziwiającym stopniu uogólnienia. Ta funkcjonalność jest przydatna w programowaniu systemów, ale nie sprawdza się w aplikacjach. Refleksja jest wrażliwym mechanizmem — kompilator nie może pomóc w znajdowaniu błędów. Wszystkie błędy są odkrywane w czasie działania programu i wtedy powodują wyjątki.

Znamy już zasady działania fundamentów programowania obiektowego: klas, dziedziczenia i polimorfizmu. W kolejnym rozdziale opisujemy dwa zaawansowane zagadnienia, bardzo ważne dla tych, którzy chcą efektywnie wykorzystywać możliwości Javy. Są to interfejsy i wyrażenia lambda.

Skorowidz

A

- abstract, 258
- abstrakcja, 257
- accelerators, 636
- access modifier, 54
- accessor method, 145
- accessory component, 676
- action map, 579
- Action.MNEMONIC_KEY, 636
- adapter class, 573
- adnotacje, 434–437
 - @author, 204, 206
 - @link, 207
 - @Override, 235
 - @param, 204, 205
 - @return, 205
 - @SuppressWarnings, 434
 - @throws, 205
 - @version, 206
- agregacja, 138, 139
- akceleratorzy, 635, 636
- akcesorium, 676
- akcesory, 145, 158
- akcje, 575
- aktywność komponentu, 577
- aktywowanie elementów menu, 637
- algorytmy, 135, 519, 524
 - generyczne, 519
 - quick sort, 125, 521
 - równoległe, 739
 - sortowania, 520, 521
 - tasowania, 520
 - tworzenie, 528
 - wyszukiwania binarnego, 523
 - znajdowania liczb pierwszych, 536
- analiza
 - danych ze stosu wywołań, 382
 - funkcjonalności klasy, 276
 - obiektów w czasie działania programu, 283
- annotation, 434
- anonimowe klasy wewnętrzne, 348
- anonymous inner class, 348
- API Preferences, 587
 - dostęp do tablicy par klucz-wartość, 589
 - dostęp do węzła drzewa, 589
 - repozytorium, 589
 - zapis danych w repozytorium, 590
- aplety, 30
- argumenty, 56
- ascender, 561
- ascent, 561
- asercje, 390
 - dokumentacja założeń, 394
 - sprawdzanie parametrów, 392
 - stosowanie, 392
 - warunek wstępny, 393
 - włączanie, 391
 - wyłączanie, 392
- asocjacja, 139
- assert, 391
- asynchroniczne obliczenia, 754
- autoboxing, 33, 252, 253
- automatyczne
 - opakowywanie, 252
 - usuwanie nieużytków, 24

autowrapping, 252
 AWT, Abstract Window Toolkit, 539, 540
 dziedziczenie klas zdarzeniowych, 586
 hierarchia zdarzeń, 585
 zdarzenia niskiego poziomu, 586
 zdarzenia semantyczne, 586

B

base class, 212
 baseline, 561
 bazowy katalog drzewa pakietu, 195
 bezpieczeństwo, 27, 36
 biblioteka
 AWT, 586
 JSON, 264
 refleksyjna, 269
 Swing, 540
 big numbers, 58
 bitowe, *Patrz także* operatory
 alternatywa, |, 75
 koniunkcja, &, 75
 negacja, ~, 75
 przesunięcie, >>, 75
 blocking queue, 725
 blok, 55, 99
 inicjujący, 178
 synchronizowany, 711
 try-catch, 374
 try-finally, 380
 blokady, 701
 jawne, 708
 wewnętrzne, 708
 wielowejsciowe, 701
 bloki tekstowe, 90
 blokowanie po stronie klienta, 712
 błędy, 366
 danych wejściowych, 366
 debugowanie, 414
 kompilacji, 440
 nieprawidłowego typu, 446
 pomyłka o jeden, 519
 przekroczenie zakresu liczby całkowitej, 58
 urządzeń, 366
 w kodzie, 366
 BMP, Basic Multilingual Plane, 63
 boolean, 58, 63
 border layout manager, 601

bridge method, 432
 bucket, 487
 budowanie łańcuchów wyjątków, 377
 byte, 58

C

call
 by name, 168
 by reference, 168
 by value, 168
 callback, 311
 CamelCase, 54
 casting, 71, 225
 catch, 374
 char, 61, 62, 82
 checkbox, 611
 checked exception, 271, 368
 child class, 212
 chwyatanie typu wieloznacznego, 450
 ciało metody, 56
 class, 54, 148
 field, 162
 loader, 358, 391
 CLASSPATH, 197
 client-side locking, 712
 code
 planes, 63
 point, 62
 units, 63
 combo box, 620
 const, 67
 content pane, 547
 coupling, 139
 covariant return types, 222
 czcionki, 559
 opcje, 645
 Font, 560
 interlinia, 561
 linia
 bazowa, 561
 dolna pisma, 561
 górna pisma, 561
 logiczne nazwy, 560
 nazwa, 559
 rodzina, 559
 styl, 560

czcionki
 wydłużenie
 dolne, 561
 górne, 561
 wysokość, 561
czytanie danych, 92

D

dane
 XML, 34
 ze stosu wywołań, 382
deadlock, 704, 718
debugger, 414
debugowanie, 414
deep copying, 317
default, 110
 package, 191
definicja
 klasy, 148
 klasy uogólnionej, 422
 stałej klasowej, 66
 zmiennej, 66
deklaracja
 list tablicowych, 245
 tablic, 120
 wyjątków kontrolowanych, 369
 zmiennej tablicowej, 120
 zmiennych, 64
dekrementacja, --, 72
delegacja zdarzeń, 569
demony, 692
deque, 495
derived class, 212
descender, 561
descent, 561
dezaktywacja elementu menu, 587, 637
diagram
 dziedziczenia, 257
 klasy, 139
diamond syntax, 245
dokumentacja, 40
 API, 86
 generowanie, 207
 założeń, 394
dostęp
 chroniony, 229
 do elementów listy tablicowej, 247
 do pakietu, 194
 do plików, 98
 do pól, 157
 do zmiennych, 332
 swobodny, 474
double, 59
doubly linked list, 476
drukowanie, 33
drzewo czerwono-czarne, 490
duże liczby, 58
dymki, 643
dynamic binding, 217
dziedziczenie, 137–139, 211, 295, 421
 dostęp chroniony, 229
 hierarchia, 219
 interfejs, 305
 klasy
 abstrakcyjne, 257
 finalne, 224
 metody
 finalne, 224
 przesłaniające, 214, 215
 nadklasy, 212
 Object, 230
 podklasy, 212
 polimorfizm, 217, 220
 porównywanie obiektów, 233
 rzutowanie, 225
 super, 215
 typy uogólnione, 443
 wielokrotne, 307
dzielenie modulo, 67
dzienniki, 395, 405
 filtry, 405
 formatery, 405
 konfiguracja menedżera, 398
 lokalizacja, 400
 poziomy ważności komunikatów, 396
 rotacja plików, 402
 śledzenie przepływu wykonywania, 397
 zapis zaawansowany, 396

E

Eclipse, 47
 edytowanie kodu, 49
 komunikaty o błędach, 50
 konfiguracja projektu, 49
 okno New Project, 48
egzemplarz klasy, 136

elementy menu, 629
 eliminowanie wywołań funkcji, 29
 enum, 67, 262
 epoka, 143
 etykiety, 606
 event dispatch thread, 542, 721
 event object, 567
 exit code, 56
 explicit parameter, 156
 extends, 212, 305
 external padding, 647

F

factory method, 164
 false, 58, 63
 field accessor, 157
 figury
 2D, 551
 geometryczne, 551
 file chooser, 675
 filtr plików, 673
 filtry, 405
 float, 59, 70
 flow layout manager, 600
 formater, 405
 formatowanie danych wyjściowych, 94
 funkcje
 czysto wirtualne, 259
 matematyczne, 68

G

Garbage Collector, 24, 505
 GBC, *Patrz* klasa GridBagConstraints, 646–649, 653
 generator liczb losowych, 723
 generic
 class, 422, 423
 programming, 421
 types, 33
 generowanie dokumentacji javadoc, 207
 generyczne listy tablicowe, 244
 dostęp do elementów, 247
 glass pane, 547
 głęboka kopia, 317
 graficzny interfejs użytkownika, GUI, 539
 komponenty Swing, 595
 wywołania zwrotne, 762
 group layout manager, 644
 grupy przycisków radiowych, 614

H

hash
 code, 236, 487
 table, 237, 625
 heap, 124, 496
 hermetyzacja, 136, 157
 hierarchia
 dziedziczenia, 219
 klas ramek, 544
 klasy Component, 601
 interfejsów, 305
 klas JSON, 266
 wyjątków, 368
 zdarzeń w bibliotece AWT, 585
 historia Javy, 31
 HTML, 34

I

identyfikacja klas, 137
 IEEE 754, 60
 IFC, Internet Foundation Classes, 540
 ikony w elementach menu, 631
 immutable class, 161
 implementacja interfejsu, 298, 300
 implements, 300
 implicit parameter, 156
 import, 93, 189
 klas, 188
 statyczny, 190
 informacje o typach
 czasu wykonywania, 270
 generycznych w maszynie wirtualnej, 454
 inheritance, 211
 chain, 219
 hierarchy, 219
 inicjalizacja
 na żądanie, 722
 pól, 176
 pól wartościami domyślnymi, 174
 z podwójną klamrą, 351
 zmiennych, 65
 inkrementacja, ++, 72
 inline method, 157
 inlining, 29, 224
 inner class, 298, 339
 input
 dialog, 658
 maps, 578

- instalacja Java Development Kit, 38, 39
- instanceof, 226, 227, 318
- instrukcja, *Patrz także* słowo kluczowe
 - break z etykietą, 115
 - break, 99, 112, 115
 - case, 110
 - continue, 116
 - else, 101
 - goto, 114
 - if, 100
 - if-else if, 103
 - if-else, 101, 102
 - switch, 74, 110, 111
 - try, 374
 - try-finally, 380
 - yield, 113
- instrukcje
 - sterujące, 99, 110, 114
 - warunkowe, 100
- int, 58, 59
- interfejs, 262, 291, 298
 - Action, 575, 580, 630, 636
 - ActionListener, 312, 348, 568, 575, 588, 612
 - AdjustmentListener, 588
 - Array, 288, 289
 - BlockingDeque, 730
 - BlockingQueue, 730
 - ButtonModel, 598, 599, 615, 617
 - Callable, 742, 743
 - ChangeListener, 623
 - Cloneable, 318
 - Collection<E>, 467, 471–473, 482
 - Comparable, 299, 427, 496
 - Comparator, 314, 338
 - Condition, 707–710
 - Delayed, 730
 - Deque, 494
 - Enumerable, 465
 - Enumeration, 469, 530
 - ExecutorService, 744, 746, 750
 - Filter, 414
 - FocusListener, 588
 - Formattable, 96
 - Future, 742, 743, 767
 - GenericArrayType, 455, 463
 - Handler, 401, 402, 411
 - ProcessHandle.Info, 775
 - InvocationHandler, 358, 363
 - ItemListener, 588
 - ItemSelectable, 615
 - Iterable, 122, 469
 - Iterator<E>, 468–473
 - JSONValue, 264, 266
 - KeyListener, 588
 - LayoutManager, 653–657
 - List<E>, 484, 485, 515, 522, 526
 - ListIterator<E>, 479, 484, 485
 - Lock, 699, 702, 707, 709
 - Map, 473
 - MouseListener, 637
 - MouseEvent, 585, 587, 635
 - MouseListener, 581, 583, 588
 - MouseMotionListener, 581–583, 588
 - MouseWheelListener, 588
 - NavigableMap, 475, 518
 - NavigableSet, 475, 491, 494, 518
 - Object, 136, 230, 231, 238, 490, 711
 - ParameterizedType, 455, 463
 - ProcessHandle, 772–775
 - PropertyChangeListener, 677
 - Queue, 465, 494
 - RandomAccess, 475
 - Runnable, 441, 681, 686, 742, 745
 - ScheduledExecutorService, 745, 746
 - Set, 475, 515
 - Shape, 552
 - SortedMap, 475, 501–504, 518
 - SortedSet, 475, 494, 518
 - StackFrame, 385
 - Thread.UncaughtExceptionHandler, 693, 694
 - Type, 455
 - TypeVariable, 455, 462
 - WildcardType, 423, 455, 462
 - WindowEvent, 567, 573, 587
 - WindowFocusListener, 588
 - WindowListener, 573, 574, 588
 - WindowStateListener, 575, 588
- interfejsy, 262, 291, 298
 - architektury kolekcji, 474
 - definicja, 299
 - dziedziczenie, 305
 - ewolucja, 309
 - funkcyjne, 326, 335, 336
 - hierarchia, 305
 - implementacja, 298, 300
 - klasy abstrakcyjne, 306
 - metody, 306
 - nasłuchu, 567

- sprężenie zwrotne, 311
- własności, 305
- zmienne, 305
- znacznikowe, 318
- interlinia, 561
- internal padding, 647
- interpreter, 29
- interrupted state, 689
- ISO 8859-1, 62
- iteratory, 468
 - słabo spójne, 731

J

- JAR, 195, 198
 - opcje, 199
- Java, 23, 33, 43, 55
 - Archive, 198
 - Community Process, 36
 - wersje, 35
- java.lang.reflect, 276, 455
- java.util.concurrent, 687, 699, 726
- java.util.EventObject, 567
- java.util.logging.config.file, 399
- java.util.logging.LogManager, 400
- Java2D, 551
 - Ellipse2D, 552–554
 - figury geometryczne, 551
 - Line2D, 551
 - Rectangle2D, 551–554
 - rysowanie figur, 555
 - Shape, 552
 - współrzędne, 553
- javac, 43, 151
- javadoc, 203, 207
 - generowanie dokumentacji, 207
 - komentarze
 - do klas, 204
 - do metod, 205
 - do pakietów, 207
 - do pól, 205
 - ogólne, 206
 - wstawianie, 203
 - streszczenie, 204
 - znaczniki dokumentacyjne, 204
- JavaFX Script, 541
- javap, 699
- JavaScript, 37
- javax.swing, 312, 542

- jawna inicjalizacja pól, 176
- JDK, Java Development Kit, 27, 38, 39
- jednostki kodowe, 63, 82
- język
 - C++, 25
 - HTML, 34
 - interpretowany, 36
 - J++, 30
 - Java, 23
 - JavaScript, 37
 - UML, 139
- JRE, Java Runtime Environment, 39
- JShell, 50
- JSON, 264
- JUnit, 415
- just-in-time compilation, 28
- JVM, 200

K

- kalendarz, 143, 146
- katalog bazowy drzewa pakietu, 195
- keyboard focus, 577
- klasa, 54, 136, *Patrz także* kolekcja, wyjątek
 - AbstractAction, 576, 579, 630
 - AbstractButton, 617, 631, 632, 637
 - AbstractCollection, 471, 482
 - AccessibleObject, 287
 - ActionEvent, 567, 587
 - ActionMap, 579
 - AdjustmentEvent, 587
 - ArrayBlockingQueue, 729
 - ArrayDeque, 476, 494, 496
 - ArrayList<E>, 246, 247, 327, 421, 469, 486, 740
 - Arrays, 125, 127, 518, 739
 - AssertionError, 391
 - AWTEvent, 585
 - BigDecimal, 117, 119, 301
 - BigInteger, 117, 119
 - BitSet, 465, 535
 - BorderFactory, 617, 618
 - BorderLayout, 602, 603, 654
 - BoxLayout, 644
 - ButtonGroup, 614–617, 633
 - ChangeEvent, 623
 - Class, 270, 277, 359
 - Class<T>, 453–455, 462
 - ClassLoader, 394
 - Cleaner, 183

klasa

- Collections, 473, 516, 520–523, 741
- Color, 558
- ColorAction, 570, 577
- CompletableFuture, 754, 756
- Component, 545, 559, 600, 606
- ConcurrentHashMap, 731, 732, 741
- ConcurrentLinkedQueue, 732
- ConcurrentSkipListMap, 731, 732
- ConcurrentSkipListSet, 731, 733
- Console, 93, 94
- ConsoleHandler, 401, 405, 412
- Constructor, 272, 276, 281, 283
- Constructor<T>, 454
- Container, 572, 600, 601
- CopyOnWriteArrayList, 739
- CopyOnWriteArraySet, 739
- Cursor, 581
- Date, 140–144, 158
- DefaultButtonModel, 598, 599
- DelayQueue, 726, 730
- Dictionary, 433
- Dimension, 545
- Double, 59
- Ellipse2D, 552–554
- Ellipse2D.Double, 557
- EnumMap, 476, 506, 508
- EnumSet, 476, 506, 508
- Enum<T>, 264
- Error, 367
- EventObject, 567, 585
- ExecutorCompletionService, 747, 751
- Executors, 743, 745
- Field, 276, 281, 283, 288, 291
- File, 99
- FileFilter, 674, 678
- FileHandler, 402, 405
- FileInputStream, 369
- FileNameExtensionFilter, 679
- FileView, 675, 679
- FlowLayout, 601
- FocusEvent, 587
- Font, 560, 564
- FontMetrics, 565
- FontRenderContext, 561
- Formatter, 405, 414
- Frame, 541, 546
- FutureTask, 743
- Graphics, 547, 549, 565, 566
- Graphics2D, 551, 552, 558, 561, 562, 565
- GraphicsEnvironment, 559
- GridBagConstraints, 645–648, 652
- GridBagLayout, 644–650, 654
 - anchor, 647, 653
 - bottom, 647
 - dopełnienie, 647
 - dopełnienie wewnętrzne, 647
 - dopełnienie zewnętrzne, 647
 - fill, 647
 - GBC, 648
 - gridheight, 646, 647
 - gridwidth, 646, 647
 - gridx, 646, 647
 - gridy, 646, 647
 - ipadx, 647
 - ipady, 647
 - klasa pomocnicza, 648
 - left, 647
 - ograniczenia, 646
 - right, 647
 - top, 647
 - weight, 646
- GridLayout, 604
- HashMap, 476, 497, 500, 740
- HashSet, 469, 476, 486, 488–491
- Hashtable, 465, 529, 740
- IdentityHashMap, 476, 507, 509
- ImageIcon, 547, 565
- InputMap, 579
- Integer, 254, 255
- ItemEvent, 587
- JButton, 568–572, 577, 598, 672
- JCheckBox, 612, 614
- JCheckBoxMenuItem, 633
- JComboBox, 529, 620, 623
- JComponent, 547, 562, 578, 599, 606, 635, 672
- JDialog, 663, 666
- JFileChooser, 672–674, 677
- JFrame, 541, 544
- JLabel, 607
- JMenu, 630
- JMenuBar, 629
- JMenuItem, 631, 632, 635–638
- JOptionPane, 314, 658, 660, 663, 668
- JPanel, 603
- JPasswordField, 608
- JPopupMenu, 634
- JRadioButton, 614, 617

- JRadioButtonMenuItem, 633
- JRootPane, 672
- JScrollPane, 609, 611
- JSlider, 433, 623, 628
- JTextArea, 604, 608, 610
- JTextComponent, 604, 605
- TextField, 604, 605, 606
- JToolBar, 643
- KeyEvent, 587
- KeyStroke, 577, 580, 636
- Line2D, 551, 557
- LineBorder, 618, 620
- LineMetrics, 562, 565
- LocalDate, 143
- Logger, 410
- Math, 68–70, 162
- Method, 276, 281, 283, 292, 294, 462
- MouseEvent, 587
- NumberFormat, 164, 255
- PaintEvent, 586
- PasswordChooser, 668
- Point, 553
- Point2D, 553, 557
- Preferences, 589, 593
- PrintStream, 240
- PrintWriter, 97, 99
- Process, 769, 774
- ProcessBuilder, 769, 773
- Properties, 532, 533
- Proxy, 358, 362, 363
- Random, 182
- RecordComponent, 282
- Rectangle, 553
- Rectangle2D, 551–557
- RectangularShape, 553, 556
- ReentrantLock, 699–702
- Scanner, 92, 93, 99
- SocketHandler, 402
- SoftBevelBorder, 618, 619
- SpringLayout, 644
- Stack, 465, 534
- StackTraceElement, 386
- StreamHandler, 402, 404
- String, 77–80, 83, 236
- StringBuilder, 89
- SwingUtilities, 671
- SwingWorker, 766, 768
- Thread, 684, 692, 694
- ThreadDeath, 688
- ThreadGroup, 693, 694
- ThreadLocal, 722
- ThreadLocalRandom, 724
- ThreadPoolExecutor, 744, 746
- Throwable, 272, 367, 373, 384, 388
- Timer, 312, 314
- Toolkit, 314, 545, 546
- TreeMap, 475, 476, 498, 500
- TreeSet, 475, 476, 490–496
- Vector, 246, 465, 740
- WeakReference<T>, 505
- Window, 546
- WindowEvent, 573
- klasy, 54, 56, 136
 - abstrakcyjne, 257
 - interfejs, 305
 - adaptacyjne, 573
 - agregacja, 138
 - anonimowe, 348
 - bazowe, 212
 - definiowanie, 148
 - diagramy, 139
 - dodawanie do pakietu, 191
 - dziedziczenie, 138, 139, 212
 - egzemplarz, 136
 - finalne, 224
 - generyczne, 244, 422, 423
 - identyfikacja, 137
 - implementacja interfejsu, 298
 - komentarze, 204
 - konstruktory, 140, 152
 - ładowanie, 358
 - macierzyste, 212
 - metody, 136
 - prywatne, 160
 - statyczne, 163
 - nadklasy, 212
 - nazwy, 54
 - opakowujące, 252
 - parametryzowane, 422
 - plik źródłowy, 197
 - pochodne, 212
 - podklasy, 212
 - pola statyczne, 161
 - pośredniczące, 357
 - właściwości, 362
 - predefiniowane, 139
 - projektowanie, 208

- klasy
 - proxy
 - invocation handler, 358
 - ładowanie klas, 358
 - publiczne, 188
 - relacje między klasami, 138
 - rozszerzanie, 136
 - stałe, 66
 - stałe pola, 160
 - tworzenie egzemplarza, 424
 - uogólnione, 244, 422
 - definicja, 423
 - wewnętrzne, 298, 339
 - anonimowe klasy, 348
 - dostęp do stanu obiektu, 340
 - dostęp do zmiennych finalnych z metod zewnętrznzych, 346
 - lokalne, 346
 - metody, 339
 - referencja do klasy zewnętrznej, 343
 - referencja do obiektu zewnętrznego, 341
 - reguły składniowe, 343
 - składnia, 340
 - statyczne, 351
 - this, 343
 - zastosowanie, 344
 - wyliczeniowe, 262
 - zagnieżdżone, 339
 - zależność, 138
 - zapieczętowane, 264
- klasyfikacja wyjątków, 367
- klawisze, 577
- kliknięcie przycisku, 569
- klonowanie obiektów, 159, 315
- kod
 - bajtowy, 29
 - mieszący, 236, 487
 - uogólniony, 429
 - wyjścia, 56
- kodowanie
 - Unicode, 62
 - UTF-16, 63
- kolejka, 465, 494, 731
 - implementacje, 466
 - blokująca, 725
 - dwukierunkowa, 494
 - priorytetowa, 496
- kolekcja, 122, 464–538
 - ArrayDeque, 476, 494, 496
 - ArrayList, 245–252, 327, 421, 469, 740
 - BitSet, 465, 535
 - Collections, 516, 520–523, 741
 - ConcurrentHashMap, 731, 732, 741
 - ConcurrentLinkedQueue<E>, 732
 - ConcurrentSkipListMap, 731, 732
 - ConcurrentSkipListSet, 731, 733
 - CopyOnWriteArrayList, 739
 - CopyOnWriteArraySet, 739
 - Deque, 495
 - Enumeration, 469, 530
 - EnumMap, 476, 506, 508
 - EnumSet, 476, 506, 508
 - HashMap, 476, 497, 500, 740
 - HashSet, 469, 476, 486, 488–491
 - Hashtable, 465, 529, 740
 - IdentityHashMap, 476, 507, 509
 - LinkedBlockingDeque, 726, 729
 - LinkedBlockingQueue, 726
 - LinkedHashMap, 476, 505–508
 - LinkedHashSet, 476, 505, 507
 - LinkedList, 467, 476, 479, 482
 - LinkedList<E>, 485
 - List, 486, 515, 523
 - List<E>, 484, 526
 - Map, 473, 499
 - NavigableMap, 475, 518
 - NavigableSet, 475, 491, 494, 518
 - PriorityBlockingQueue, 726, 730
 - PriorityQueue, 476, 497
 - Properties, 531, 533
 - Queue, 465, 494
 - SortedMap, 475, 501–503, 518
 - SortedSet, 475, 494, 518
 - Stack, 465, 534
 - TreeMap, 475, 476, 498, 500
 - TreeSet, 475, 476, 490–496
 - Vector, 246, 465, 740
 - WeakHashMap, 476, 504, 507
- kolekcje, 122, 464–538
 - algorytmy, 519
 - bezpieczne wątkowo, 724
 - dostęp swobodny, 475
 - interfejsy, 465, 473
 - klasy, 475, 477
 - kolejność odwiedzania elementów, 469
 - konwersja pomiędzy kolekcjami a tablicami, 527
 - kopie niemodyfikowalne, 511
 - listy

- cykliczne, 467
- powiązane, 476
- tablicowe, 486
- mapy własności, 531
- obiekty opakowujące, 509
- ograniczone, 467
- operacje
 - opcjonalne, 514, 515
 - zbiorcze, 526
- słabo spójne iteratory, 731
- słowniki, 497
- sortowanie, 520
- uporządkowane, 474, 479
- stare, 529
- starsze kolekcje bezpieczne wątkowo, 740
- stos, 534
- tablice kopiowane przy zapisie, 739
- tasowanie, 520
- warstwa
 - interfejsów, 465
 - klas konkretnych, 465
- widoki, 509
 - kontrolowane, 513
 - niemodyfikowalne, 512
 - przedziałowe, 513
 - synchronizowane, 514
- wyliczenia, 530
- wyszukiwanie binarne, 523
- zbiory bitów, 535
- kolizja, 487
 - nazw, 188
- kolory, 557
 - Color, 558, 570, 577
 - definiowanie, 558
 - tło, 558
- komentarze
 - `/* */`, 57
 - `/** */`, 57
 - `//`, 57
 - do klas, 204
 - do metod, 205
 - do pakietów, 207
 - do pól, 205
 - dokumentacyjne, 57, 203
 - ogólne, 57, 206
- komórki, 487
- komparatory, 315, 338, 739
- kompilacja w czasie rzeczywistym, 28
- kompilator, 43
- komponenty, 548, 600
 - dymki, 643
 - etykiety, 606
 - listy rozwijalne, 620
 - menu, 629
 - menu podręczne, 634
 - obszary tekstowe, 608
 - panele przewijane, 609
 - paski narzędzi, 641
 - pola
 - hasel, 608
 - tekstowe, 605
 - wyboru, 612
 - przełączniki, 614
 - rekordy, 184
 - suwaki, 623
 - Swing, 595
 - wybór opcji, 611
 - zarządzanie rozkładem, 599
- komputer wieloprocesorowy, 714
- komunikat
 - ERROR_MESSAGE, 658
 - INFORMATION_MESSAGE, 658
 - QUESTION_MESSAGE, 658
 - WARNING_MESSAGE, 658
 - PLAIN_MESSAGE, 658
- komunikaty o błędach, 50, 658
- konfiguracja menedżera dzienników, 398
- konkatenacja, 78
- konstruktory, 140, 152
 - bezargumentowe, 175
 - domyślne, 175
 - kanoniczne, 186
 - kompaktowe, 186
 - niestandardowe, 186
 - podklas, 215
 - przeciążanie, 176
 - referencje, 331
 - wirtualne, 272
- kontener, 600
- kontrola grup zadań, 746
- kontroler, 596
- konwersja
 - automatyczne opakowywanie, 252
 - łańcucha na liczbę, 254
 - pośród kolekcjami a tablicami, 527
 - typów numerycznych, 70
- kończenie działania programu, 56

kopiowanie
 głębokie, 317
 obiektów, 316
 płytkie, 317
 tablicy, 123
kowariantne typy zwracane, 222, 433
kubelki, 487
kursory, 582

L

lambda, 322, 323
layered pane, 547
leniwa ewaluacja, 327
licencja GPL, 36
liczby
 całkowite, 58
 zmiennoprzecinkowe, 59
linia
 bazowa, 561
 dolna pisma, 561
 górna pisma, 561
linked list, 476
listener
 interface, 567
 object, 313
listy
 cykliczne, 467
 dwukierunkowe, 476, 478
 modyfikowalne, 521
 powiązane, 467, 476
 dodawanie elementów, 479, 480
 ListIterator, 479
 metody get i set, 483
 ogniwa, 476
 usuwanie elementów, 477, 478
 rozwijalne, 620
 tablicowe, 250, 486
 deklarowanie, 245
 dostęp do elementów, 248
 generyczne, 245
 zmiana rozmiaru, 521
literały typowe, 458
load factor, 488
logging proxy, 415
logic_error, 369
logiczne nazwy czcionek, 560
lokalizacja, 400

lokalne klasy wewnętrzne, 346
long, 58
long int, 28
LTS, Long Term Support, 39

Ł

ładowanie
 klas, 358
 usług, 355
łańcuchy, 77
 długość, 82
 dziedziczenia, 219
 identyczność, 80
 jednostki kodowe, 82
 konkatenacja, 78
 niezmienialność, 79
 null, 81
 podłańcuchy, 77
 porównywanie, 80
 składanie, 86
 String, 79, 83
 StringBuilder, 89
 współdzielenie, 79
 współrzędne kodowe znaków, 82
 wyjątków, 377

M

manifest, 198
mapa
 wejścia, 578
 własności, 531
marker interface, 318
maszyna wirtualna, 274, 429
 informacja o typach generycznych, 454
ME, Micro Edition, 39
mechanizm ładowania klas, 358
menedżer dzienników, 399
menu, 629
 akceleratorzy, 635, 636
 akcje, 630
 elementy, 629
 aktywowanie, 637
 dezaktywowanie, 637
 ikony, 631
 mnemoniki, 635
 pasek, 629
 podmenu, 629

- podręczne, 634
 - obsługa, 634
 - tworzenie, 634
- pola wyboru, 632
- przełączniki, 632
- separatory, 630
- skróty klawiszowe, 636
- tworzenie, 629
- metadane, 33
- metoda, 55, 136
 - accept(), 678
 - acceptEither(), 759
 - accumulateAndGet(), 717
 - actionPerformed(), 312, 347, 568, 575, 630
 - add(), 118, 472, 551, 617, 725
 - addActionListener(), 567, 612, 630, 664
 - addAll(), 472, 484, 525, 527
 - addChoosableFileFilter(), 675, 678
 - addFirst(), 486, 495
 - addHandler(), 411
 - addItem(), 621, 623
 - addLast(), 486, 495
 - addLayoutComponent(), 654, 657
 - addPropertyChangeListener(), 575
 - addSeparator(), 629, 631, 642, 643
 - addSuppressed(), 384
 - adjustmentValueChanged(), 588
 - allOf(), 508, 759
 - allProcesses(), 772, 775
 - and(), 536
 - andNot(), 536
 - anyOf(), 759
 - append(), 89, 611
 - appendCodePoint(), 89
 - applyToEither(), 759
 - arguments(), 775
 - Array.newInstance(), 439
 - ArrayAlg.getMiddle(), 426
 - arrayType(), 291
 - asIterator(), 530
 - asList(), 518
 - await(), 704–708
 - beep(), 314
 - binarySearch(), 127, 360, 362, 523, 524
 - BorderFactory.createCompoundBorder(), 618
 - BorderFactory.createEtchedBorder(), 618
 - BorderFactory.createTitledBorder(), 618
 - call(), 743
 - cancel(), 743, 745
 - cast(), 453, 454
 - ceiling(), 494
 - charAt(), 82, 84
 - checkedCollection(), 517
 - checkedList(), 517
 - checkedMap(), 517
 - checkedSet(), 517
 - checkedSortedMap(), 517
 - checkedSortedSet(), 517
 - Class.newInstance(), 437
 - clear(), 472, 535, 536
 - clearAssertionStatus(), 394
 - clone(), 316–320, 362
 - close(), 412
 - codePointAt(), 82, 84
 - codePointCount(), 82, 85
 - codePoints(), 83
 - command(), 776
 - commandLine(), 776
 - compare(), 304, 322
 - compareTo(), 84, 119, 263, 299–303, 424
 - completeOnTimeout(), 757
 - componentType(), 290
 - compute(), 502
 - computeIfAbsent(), 502
 - computeIfPresent(), 502
 - config(), 410
 - contains(), 472
 - containsAll(), 472
 - containsKey(), 500
 - containsValue(), 500
 - copy(), 525
 - copyArea(), 566
 - copyOf(), 123, 127, 515, 516
 - createBevelBorder(), 619
 - createCompoundBorder(), 619
 - createEmptyBorder(), 618
 - createEtchedBorder(), 619
 - createLineBorder(), 618
 - createLoweredBevelBorder(), 619
 - createMatteBorder(), 618
 - createRaisedBevelBorder(), 619
 - createTitledBorder(), 619
 - current(), 724, 775
 - currentThread(), 689, 692
 - delete(), 90
 - deriveFont(), 560, 564
 - descendants(), 775
 - descendingIterator(), 494

metoda

- destroy(), 774
- destroyForcibly(), 774
- disjoint(), 525
- divide(), 119
- doInBackground(), 766, 768
- Double.isNaN(), 60
- draw(), 552, 558
- drawImage(), 565, 566
- drawString(), 557–560, 565
- element(), 495, 725
- Employee.clone(), 433
- emptyEnumeration(), 517
- emptyIterator(), 517
- emptyList(), 517
- emptyListIterator(), 518
- emptyMap(), 517
- emptyNavigableMap(), 517
- emptyNavigableSet(), 517
- emptySet(), 517
- emptySortedMap(), 517
- emptySortedSet(), 517
- endsWith(), 84
- ensureCapacity(), 247
- entering(), 410
- entry(), 516
- entrySet(), 502, 503
- enumeration(), 530
- equals(), 80, 84, 127, 231–237, 362, 443, 475
- equalsIgnoreCase(), 80, 84
- exceptionally(), 757
- execute(), 768
- exiting(), 410
- exitValue(), 774
- exportNode(), 594
- exportSubtree(), 594
- fill(), 127, 525, 557–559
- finalize(), 183
- fine(), 410
- finer(), 410
- finest(), 410
- first(), 494
- firstKey(), 501
- flipDone(), 715
- floor(), 494
- flush(), 412
- focusGained(), 588
- focusLost(), 588
- forEach(), 385
- forEachRemaining(), 473
- format(), 405, 414
- formatMessage(), 405, 414
- forName(), 272
- frequency(), 525
- get(), 248, 288, 474, 499, 535, 589, 724, 743
- getActionCommand(), 617
- getActionCommands(), 585
- getActionMap(), 580
- getActualTypeArguments(), 463
- getAncestorOfClass(), 671
- getAscent(), 565
- getAvailableFontFamilyNames(), 559
- getBackground(), 559
- getBoolean(), 590, 593
- getBounds(), 462
- getByteArray(), 590, 593
- getCause(), 384
- getCenter(), 553
- getCenterX(), 553, 556
- getCenterY(), 553, 556
- getClass(), 232, 244, 270, 271, 362
- getClassName(), 385, 386
- getClickCount(), 585
- getColumns(), 606
- getComponentPopupMenu(), 635
- getComponentType(), 290
- getConstructor(), 453, 454
- getConstructors(), 276, 281
- getDay(), 144
- getDayOfMonth(), 144, 146, 148
- getDeclaredFields(), 287
- getDeclaredConstructor(), 453, 454
- getDeclaredConstructors(), 276, 281
- getDeclaredField(), 287
- getDeclaredFields(), 276, 281, 283, 285
- getDeclaredMethods(), 276, 281
- getDeclaringClass(), 281, 386
- getDefaultToolkit(), 314, 545, 546
- getDefaultUncaughtExceptionHandler(), 694
- getDelay(), 726, 730
- getDescent(), 565
- getDescription(), 678
- getDouble(), 589, 593
- getEnumConstants(), 453, 454
- getErrorMessage(), 774
- getExceptionTypes(), 282
- getFamily(), 564
- getField(), 287

getFields(), 276, 277, 287
 getFileName(), 385, 386
 getFilter(), 411, 412
 getFintName(), 564
 getFirst(), 486, 495
 getFloat(), 589, 593
 getFont(), 606
 getFontMetrics(), 562, 565
 getFontRenderContext(), 561, 562, 565
 getForeground(), 559
 getFormatter(), 412
 getGenericComponentType(), 463
 getGenericInterfaces(), 462
 getGenericParameterTypes(), 462
 getGenericReturnType(), 462
 getGenericSuperclass(), 462
 getHandlers(), 411
 getHead(), 405, 414
 getHeight(), 553, 557, 562, 565
 getIcon(), 608, 675, 679
 getIconImage(), 546
 getImage(), 547
 getImageURLs(), 757
 getInheritsPopupMenu(), 635
 getInputMap(), 578, 580
 getInputStream(), 774
 getInt(), 589, 593
 getKey(), 504
 getKeyStroke(), 577, 579, 580
 getLargestPoolSize(), 746
 getLast(), 486, 495
 getLeading(), 565
 getLength(), 289, 291
 getLevel(), 411–413
 getLineMetrics(), 562, 564
 getLineNumber(), 385, 386
 getLogger(), 410
 getLoggerName(), 412
 getLogManager(), 413
 getLong(), 589, 593
 getLongThreadID(), 413
 getLowerBounds(), 462
 getMaxX(), 556
 getMaxY(), 557
 getMessage(), 373, 413
 getMethod(), 292, 293
 getMethodName(), 386
 getMethods(), 276, 281
 getMillis(), 413
 getMinX(), 556
 getMinY(), 556
 getModifiers(), 276, 282
 getMonth(), 144
 getMonthValue(), 144, 146, 148
 getName(), 244, 271, 282, 462, 564, 679
 getOrDefault(), 499, 501
 getOutputStream(), 774
 getOwnerType(), 463
 getPackageName(), 281
 getPaint(), 558
 getParameters(), 413
 getParameterTypes(), 282
 getParent(), 411
 getPassword(), 608
 getPath(), 674
 getPoint(), 585
 getPredefinedCursor(), 581
 getPreferredSize(), 551
 getProperties(), 533
 getProperty(), 533
 getProxyClass(), 363
 getRawType(), 463
 getRecordComponents(), 281
 getResourceBundle(), 413
 getResourceBundleName(), 413
 getReturnType(), 282
 getRootPane(), 668, 672
 getScreenSize(), 545, 546
 getSelectedFile(), 674, 678
 getSelectedFiles(), 674, 678
 getSelectedItem(), 621, 623
 getSelectedObjects(), 615
 getSelection(), 615, 617
 getSequenceNumber(), 413
 getSource(), 585
 getSourceClassName(), 413
 getSourceMethodName(), 413
 getStackTrace(), 382, 384
 getState(), 689, 768
 getStringBounds(), 561, 564
 getSuperClass(), 244, 454
 getSuppressed(), 382, 385
 getTail(), 405, 414
 getText(), 604, 606, 607
 getThrown(), 413
 getTime(), 224
 getTitle(), 545, 546
 getTotalBalance(), 701

metoda

- getTypeDescription(), 675, 679
- getTypeParameters(), 462
- getUncaughtExceptionHandler(), 694
- getUpperBounds(), 463
- getUseParentHandlers(), 411
- getValue(), 504, 575, 576, 580
- getWidth(), 552, 553, 557, 562
- getX(), 557, 585
- getXxx(), 291
- getY(), 557, 562, 585
- getYear(), 144, 146, 148
- handle(), 757
- hashCode(), 236–238, 262, 487, 490
- hasMoreElements(), 469, 530
- hasNext(), 94, 468, 469, 473, 480
- hasNextDouble(), 94
- hasNextInt(), 94
- hasPrevious(), 479, 485
- headMap(), 513, 518
- headSet(), 513, 518
- higher(), 494
- identityHashCode(), 509
- importPreferences(), 594
- increment(), 717
- incrementAndGet(), 716
- indexOf(), 85, 174, 485
- indexOfSubList(), 525
- info(), 410, 775
- initCause(), 384
- initialValue(), 723
- insert(), 89, 631
- insertItemAt(), 621, 623
- insertSeparator(), 631
- Integer.parseInt(), 254
- interrupt(), 689, 692
- interrupted(), 691, 692
- intValue(), 255
- invoke(), 292, 294, 363
- invokeAll(), 751
- invokeAny(), 746
- invokeDefault(), 363
- isAbstract(), 282
- isAccessible(), 287
- isAlive(), 774
- isArray(), 290
- isBlank(), 84
- isCancelled(), 743, 745
- isDefaultButton(), 672
- isDone(), 715, 742–745
- isEditable(), 604, 623
- isEmpty(), 84, 472
- isEnabled(), 575, 580
- isEnum(), 281
- isFinal(), 276, 282
- isInterface(), 281, 282
- isInterrupted(), 689–692
- isLoggable(), 405, 414
- isNaN(), 60
- isNative(), 282
- isNativeMethod(), 386
- isPopupTrigger(), 635
- isPrivate(), 276, 282
- isProtected(), 282
- isProxyClass(), 363
- isPublic(), 276, 282
- isRecord(), 281
- isResizable(), 546
- isSelected(), 614, 615, 633
- isStatic(), 282
- isStrict(), 282
- isSynchronized(), 282
- isTraversable(), 675, 679
- isVisible(), 545
- isVolatile(), 283
- itemStateChanged(), 588
- iterator(), 468, 471, 472
- JMenu.remove(), 637
- join(), 688
- keyPressed(), 588
- keyReleased(), 588
- keys(), 593
- keySet(), 503, 504, 738
- keyTyped(), 588
- last(), 494
- lastIndexOf(), 85, 485
- lastIndexOfSubList(), 525
- lastKey(), 501
- layoutContainer(), 654, 657
- length(), 82, 85, 89, 122, 535
- listIterator(), 484
- lock(), 702
- log(), 410
- Logger.global.info(), 396
- Logger.global.setLevel(), 396
- logp(), 397, 410
- lower(), 494
- main(), 54, 55, 165

Math.round(), 71
Math.sqrt(), 293, 390
max(), 524, 525
menuCanceled(), 638
menuDeselected(), 638
menuSelected(), 638
merge(), 501
min(), 524
minimumLayoutSize(), 654, 657
minusDays(), 148
mod(), 119
Modifier.toString(), 276
mouseClicked(), 581, 588
mouseDragged(), 582, 588
mouseEntered(), 582, 588
mouseExited(), 582, 588
mouseMoved(), 582, 588
mousePressed(), 581, 588
mouseReleased(), 581, 588
mouseWheelMoved(), 588
multiply(), 118, 119
nCopies(), 517
newCachedThreadPool(), 743–745
newCondition(), 707
newFixedThreadPool(), 744, 745
newInstance(), 272, 291, 453, 454
newKeySet(), 738
newProxyInstance(), 358, 363
newScheduledThreadPool(), 744, 745
newSingleThreadExecutor(), 744, 745
newSingleThreadScheduledExecutor(), 744, 745
next(), 92, 468, 469, 470, 473
nextDouble(), 92, 94
nextElement(), 469, 530
nextIndex(), 482, 485
nextInt(), 92, 94
nextLine(), 92, 93
node(), 593
noneOf(), 508
notify(), 708, 711, 714
notifyAll(), 708–711, 714
now(), 148
Object.clone(), 318, 433
of(), 148, 515, 516
ofEntries(), 510, 516
off-by-one error, 519
offer(), 495, 725, 730
offerFirst(), 495, 731
offerLast(), 495, 731
offsetByCodePoints(), 82, 84
onExit(), 775
or(), 536
ordinal(), 264
orTimeout(), 757
pack(), 551
paintComponent(), 371, 547–551, 562, 659, 721
parallelSetAll(), 739
parse(), 255
parseInt(), 254, 255
peek(), 495, 534, 725
peekFirst(), 495
peekLast(), 496
pid(), 775
plusDays(), 145, 148
poll(), 495, 725, 730, 751
pollFirst(), 494, 495, 731
pollLast(), 494, 495, 731
pop(), 534
pow(), 68
Preferences.systemNodeForPackage(), 589
Preferences.systemRoot(), 589
Preferences.userNodeForPackage(), 589
Preferences.userRoot(), 589
preferredLayoutSize(), 654, 657
previous(), 479, 482, 485
previousIndex(), 482, 485
print(), 57, 94
printf(), 95, 256
println(), 68, 362
printStackTrace(), 272, 416
process(), 767, 768
process.onExit(), 772
publish(), 411, 768
push(), 534
put(), 473, 500, 590, 725, 730
putAll(), 500
putBoolean(), 594
putByteArray(), 594
putDouble(), 594
putFirst(), 730
putFloat(), 593
putIfAbsent(), 502
putInt(), 590, 593
putLast(), 730
putLong(), 593
putValue(), 575, 576, 580, 632, 643
random(), 126
range(), 508

metoda

- readConfiguration(), 413
- readLine(), 94
- readPage(), 757
- readPassword(), 94
- redirectError(), 770, 773
- redirectErrorStream(), 773
- redirectInput(), 773
- redirectOutput(), 773
- remove(), 468–474, 484, 495, 631, 725
- removeAll(), 472
- removeAllItems(), 623
- removeEldestEntry(), 508
- removeFirst(), 486, 495
- removeHandler(), 411
- removeIf(), 327, 472, 526
- removeItem(), 621, 623
- removeItemAt(), 621, 623
- removeLast(), 486, 495
- removeLayoutComponent(), 654, 657
- removePropertyChangeListener(), 575
- repaint(), 549, 551
- repeat(), 86
- replace(), 85
- replaceAll(), 502, 525, 526
- resetChoosableFileFilters(), 678
- resetChoosableFilters(), 675
- resume(), 689
- retainAll(), 473, 526
- revalidate(), 605, 606
- reverse(), 525
- reverseOrder(), 521
- reverseOrder(), 521
- rotate(), 525
- round(), 71
- run(), 685
- runAfterBoth(), 759
- runAfterEither(), 759
- schedule(), 746
- scheduleAtFixedRate(), 746
- scheduleWithFixedDelay(), 746
- set(), 248, 288, 291, 480, 485, 535, 724
- setAccelerator(), 636, 637
- setAcceptAllFileFilterUsed(), 675, 678
- setAccessible(), 283, 287, 288
- setAccessory(), 678
- setAction(), 631
- setActionCommand(), 617
- setAnchor(), 649
- setBackground(), 558, 559
- setBorder(), 618, 620
- setBounds(), 544–546
- setCharAt(), 89
- setClassAssertionStatus(), 394
- setColumns(), 605, 606, 609, 611
- setComponentPopupMenu(), 634, 635
- setCurrentDirectory(), 673, 677
- setCursor(), 585
- setDaemon(), 692
- setDefaultAssertionStatus(), 394
- setDefaultButton(), 668, 672
- setDefaultCloseOperation(), 543
- setDefaultUncaughtExceptionHandler(), 693, 694
- setDisplayedMnemonicIndex(), 636, 637
- setDone(), 715
- setEchoChar(), 608
- setEditable(), 604, 623
- setEnabled(), 575, 580, 638
- setFileFilter(), 678
- setFileSelectionMode(), 673, 677
- setFileView(), 676, 678
- setFill(), 649
- setFilter(), 405, 411, 412
- setFont(), 606
- setForeground(), 559
- setFormatter(), 412
- setFrameFromCenter(), 554
- setHorizontalTextPosition(), 632
- setIcon(), 607, 608
- setIconImage(), 544, 546
- setInheritsPopupMenu(), 635
- setInverted(), 628
- setJMenuBar(), 629, 631
- setLabelTable(), 625, 628
- setLayout(), 600, 601, 604
- setLevel(), 411, 412
- setLineWrap(), 609, 611
- setLocation(), 544, 546
- setLocationByPlatform(), 546
- setMajorTickSpacing(), 624, 628
- setMinorTickSpacing(), 624, 628
- setMnemonic(), 636, 637
- setMultiSelectionEnabled(), 673, 677
- setName(), 692
- setPackageAssertionStatus(), 394
- setPaint(), 557–559
- setPaintLabels(), 625, 628
- setPaintTicks(), 624, 628

setPaintTrack(), 628, 629
 setParent(), 411
 setPriority(), 694, 695
 setProperty(), 532
 setResizable(), 544, 546
 setRows(), 609, 611
 setSelected(), 612, 614, 633
 setSelectedFile(), 673, 677
 setSelectedFiles(), 677
 setSize(), 546
 setSnapToTicks(), 624, 628
 setTabSize(), 611
 setText(), 604–607, 667
 setTime(), 224
 setTitle(), 544–546
 setToolTipText(), 643, 644
 setUncaughtExceptionHandler(), 693, 694
 setUseParentHandlers(), 411
 setValue(), 504
 setVisible(), 543, 545, 664, 666, 667
 setWrapStyleWord(), 611
 setXxx(), 291
 severe(), 410
 show(), 634
 showConfirmDialog(), 659, 661
 DEFAULT_OPTION, 659
 OK_CANCEL_OPTION, 659
 YES_NO_CANCEL_OPTION, 659
 YES_NO_OPTION, 659
 showDialog(), 678
 showInputDialog(), 659, 662
 showInternalConfirmDialog(), 661
 showInternalInputDialog(), 662
 showInternalMessageDialog(), 661
 showInternalOptionDialog(), 662
 showMessageDialog(), 314, 660
 showOpenDialog(), 672, 673, 678
 showOptionDialog(), 659, 662
 showSaveDialog(), 673, 678
 shuffle(), 522
 shutdown(), 746
 signal(), 705–708, 720
 signalAll(), 704–708, 719
 singleton(), 517
 singletonList(), 517
 singletonMap(), 517
 size(), 247, 471, 472, 482
 sleep(), 685
 sort(), 125, 302, 520, 522
 sqrt(), 68, 119
 start(), 314, 682, 685
 startInstant(), 775
 stop(), 314, 688, 689, 720
 store(), 533
 stream(), 357, 536
 String.format(), 96
 stringPropertyNames(), 533
 stripLeading(), 85
 subList(), 513, 518
 subMap(), 513, 518
 submit(), 746, 751
 subSet(), 513, 518
 substring(), 77, 81, 85
 subtract(), 119
 super.clone(), 318
 supplyAsync(), 755, 756
 supportsNormalTermination(), 774
 suspend(), 689, 720
 swap(), 450, 525
 synchronizedCollection(), 516, 741
 synchronizedList(), 516, 741
 synchronizedMap(), 514, 516, 741
 synchronizedSet(), 516, 741
 synchronizedSortedMap(), 515, 741
 synchronizedSortedSet(), 516
 System.exit(), 56, 543
 System.out.print(), 94
 System.out.println(), 92
 systemNodeForPackage(), 593
 systemRoot(), 589, 593
 tailMap(), 513, 518, 519
 tailSet(), 513, 518
 take(), 725, 730, 751
 takeFirst(), 730
 takeLast(), 731
 text(), 94
 thenAccept(), 757
 thenAcceptBoth(), 759
 thenApply(), 757, 758
 thenCombine(), 759
 thenCompose(), 758
 thenRun(), 757
 Thread.dumpStack(), 416
 ThreadLocalRandom.current(), 723
 throwing(), 410
 toArray(), 439, 473
 toHandle(), 774
 toLowerCase(), 85

metoda

- toString(), 90, 96, 127, 239, 244, 255, 362, 386
- totalCpuDuration(), 776
- toUpperCase(), 85
- transfer(), 701
- trim(), 85, 606
- trimToSize(), 247
- trySetAccessible(), 287
- unlock(), 700, 702
- unmodifiableList(), 516
- unmodifiableMap(), 516
- unmodifiableSet(), 516
- unmodifiableSortedMap(), 516
- unmodifiableSortedSet(), 516
- updateConfiguration(), 414
- user(), 775
- userNodeForPackage(), 593
- userRoot(), 589, 593
- validate(), 606
- valueOf(), 119, 255, 264
- values(), 504
- wait(), 710, 711
- walk(), 385
- warning(), 410
- whenComplete(), 757–759
- windowActivated(), 573, 574, 588
- windowClosed(), 573, 574, 588
- windowClosing(), 573, 574, 588
- windowDeactivated(), 573, 588
- windowDeiconified(), 573, 574, 588
- windowGainedFocus(), 588
- windowIconified(), 573, 574, 588
- windowLostFocus(), 588
- windowOpened(), 573, 574, 588
- windowStateChanged(), 575, 588
- withInitial(), 724
- xor(), 536

metody, 55, 136

- abstrakcyjne, 258, 326
- akcesory, 145, 157
- ciało, 56
- domyślne, 308, 310
- dostęp do pól, 157
- fabryczne, 144, 164
- finalne, 224
- generyczne, 425, 455, 470
- klasy

- Executors, 744
 - Math, 70
 - String, 88

- kolejek blokujących, 726
 - komentarze, 205
 - parametry, 56, 168
 - parametryzowane, 426
 - pomostowe, 432
 - prywatne, 160, 308
 - przeciążanie, 174
 - przesłanianie, 214, 296
 - referencje, 328
 - rodzime, 163
 - statyczne, 68, 163
 - sygnatura, 174, 222
 - synchronizowane, 708, 709
 - udostępniające, 145
 - uogólnione, 425, 455, 470
 - typ w wywołaniu, 426
 - wnioskowanie o typie, 426
 - wstawiane, 157
 - wyjątki, 369
 - wywoływanie, 221, 292
 - zmieniające wartość elementu, 145
 - zmienna liczba parametrów, 255
- mnemoniki, 635
- modal dialog box, 657
- modalność, 663
- model, 596, 597
- modeless dialog box, 657
- moduł ładujący klasy, 391
- modyfikacja parametru obiektowego, 170
- modyfikatory dostępu, 54, 230
 - final, 716
 - volatile, 716
- monitor, 713
- multiple inheritance, 307
- multithreaded, 680
- mutable class, 161
- mutatory, 145, 158
- MVC, Model-View-Controller, 596
- mysz, 580, 587
 - kursory, 582

N

- nadklasy, 212
- nadtypy
 - ograniczenia, 447, 448
 - typów wieloznacznych, 447
- NaN, Not a Number, 60, 68
- narzędzia wiersza poleceń, 42

narzędzie, *Patrz* program
nawiasy, 76

klamrowe, 55, 458
kwadratowe, 120
ostre, 424

nazwy

klas, 210
metod, 210
pakietów, 188
parametrów, 177
rodziny czcionek, 559

niemodyfikowalne

kopie kolekcji, 511
widoki kolekcji, 512

nierówność, *Patrz* operatory relacyjne, 73

nieskończoność, 60

niezależność od architektury, 27

niezawodność, 26

niezmiennność, 233

niszczanie obiektów, 183

notacja wielbłądzia, 54

null, 81, 154, 174, 329, 502

0

obiektowość, 26

obiekty, 58, 137

autoboxing, 252

Handler, 401

hermetyzacja, 136

klasy uogólnionej, 440

klonowanie, 159, 315

konstruktory, 140

kopiowanie, 316

metody, 136

metody prywatne, 160

nasłuchujące, 313

niemodyfikowalne, 511

niszczenie, 183

obsługujące wywołanie, 358

opakowujące, 509

opakowywanie automatyczne, 253

polimorfizm, 217

porównywanie, 231

pośredniczące, 358

składowe, 136, 157

stan, 136, 137

tożsamość, 137

tworzenie, 140, 174

warunków, 702

właściwości, 136

zachowanie, 137

zdarzeń, 567

obliczenia, 68

asynchroniczne, 754

obramowanie, 617

obrazy, 565

wyświetlanie, 565

obsługa

błędów, 366

wyjątków, *Patrz* wyjątek, wyjątki

zdarzeń, 312, 588

akcje, 568, 575

hierarchia zdarzeń w AWT, 585

interfejs nasłuchu, 567

klasy, 567, 573, 585, 587, 623

kliknięcia przycisku, 569

mysz, 580

obiekt zdarzeń, 567

obszary

surogatów, 63

tekstowe, 608

odczyt

danych, 92

z pliku, 97

odmierzanie czasu, 312

odpakowywanie, 253

odrzucone metody, 144

ogniwa, 476

ograniczenia zmiennych typowych, 427

okna dialogowe, 657

akcesorium, 676

dane wejściowe, 658

klasy, 658, 672

klawisz wyzwolenia, 668

komunikaty, 658

modalne, 657, 663

niemodalne, 657

opcje, 658, 660

panel główny, 668

potwierdzenia, 660

przycisk domyślny, 668

przyciski, 659

ramka nadrzędna, 663

tworzenie, 663

wybór plików, 672

wymiana danych, 666

wyświetlanie, 667

- okno
 - New Project, 48
 - zdarzenia, 573
- OOP, Object Oriented Programming, 135
- opakowywanie obiektów, 252, 253
- operacje
 - masowe, 736
 - opcjonalne, 515
 - słownikowe, 497
 - zbiorcze, 526
- operator
 - ::, 328
 - >, 112
 - [], 124
 - dekrementacji, --, 72
 - inkrementacji, ++, 72
 - instanceof, 226, 305
 - new, 92, 120, 140
 - przypisania, =, +=, 72
 - trójargumentowy, ?:, 74
- operatory
 - arytmetyczne, +, -, *, /, %, 67
 - bitowe, &, ^, |, ~, 75
 - logiczne, &&, ||, !, 73
 - priorytety, 76
 - przesunięcia bitowe, >>, <<, >>>, <<<, 75
 - przyrostkowe, 73
 - relacyjne, !=, ==, >, <, <=, >=, 73
- opóźnione wykonywanie procedur, 334
- optional operations, 515
- overloading resolution, 174

P

- package, 191, 194
- packages, 188
- pakiet, 188
 - java.lang.reflect, 276, 455
 - java.util.concurrent, 687, 699, 726
 - java.util.concurrent.atomic, 716
 - java.util.EventObject, 567
 - java.util.logging.config.file, 399
 - java.util.logging.LogManager, 400
 - javax.swing, 312
 - JDK, 39
- pakiety
 - dodawanie klasy, 191
 - import klas, 188
 - komentarze, 207
 - lokalizacyjne, 400
 - modyfikatory dostępu, 194
 - nazwy, 188
 - nienazwane, 191
- panel przewijany, 609
- para klucz-wartość, 509, 589
- parametr, 56, 168, 255
 - super, 330
 - this, 330
- parametry
 - jawne, 156
 - nazwy, 177
 - niejawne, 156
 - przekazywane przez referencję, 168
 - przekazywane przez wartość, 168
 - typowe, 421
 - wiersza poleceń, 124
- parent class, 212
- pasek menu, 629
- paski narzędzi, 641
- pętle
 - for, 106, 107
 - for each, 33, 122, 249
 - do while, 105, 107
 - liczba iteracji, 106
 - przerywanie działania, 115, 116
 - REPL, 50
 - while, 102, 105
- pieczętowanie klas, 265
- pierwszy program, 54
- platforma programistyczna, 23
- pliki
 - .class, 54
 - .java, 54
 - dziennika, 403
 - JAR, 195, 198
 - manifest, 198
 - różne wersje klas, 201
 - sekcja główna, 198
 - wykonywalne, 200
 - zmiana zawartości pliku manifestu, 199
 - kompresja ZIP, 198
 - lokalizacyjne, 275
 - odczyt, 97
 - zapis, 97
 - źródłowe, 40
 - stosowanie, 151
- plytka kopia, 317

- pobieranie
 - danych, 92
 - pakietu JDK, 39
- podklasy, 212
- podkradanie pracy, 753
- podłańcuchy, 77
- podmenu, 629
- pola
 - hasel, 608
 - klasowe, 162
 - kombi, 620
 - statyczne, 161
 - tekstowe, 605
 - ulotne, 714
 - wyboru, 612
- polimorfizm, 217, 220, 296
- pop-up menu, 634
- pop-up trigger, 634
- porównywanie
 - łańcuchów, 80
 - obiektów, 231
- powiązania między klasami, 139
- powtórne generowanie wyjątków, 377
- pozycjonowanie ramki, 543
- preferencje użytkownika
 - API Preferences, 587
- priorytety
 - operatorów, 76, 77
 - wątków, 694
- private, 152, 160, 194, 229, 230
- procedura
 - nasłuchu zdarzeń, 572
 - obsługi błędów, 367
 - obsługi nieprzechwyconych wyjątków, 693
- proces, 680, 769
 - budowanie, 769
 - uchwyt, 772
 - uruchamianie, 771
- program
 - javadoc, 203
 - javap, 202, 344
 - jconsole, 418
 - junit, 415
 - jshell, 50
- programowanie
 - generyczne, *Patrz programowanie uogólnione*
 - proceduralne, 136
 - uogólnione, 421
 - dziedziczenie, 443
 - egzemplarze zmiennych typowych, 437
 - klasy uogólnione, 422, 423
 - konflikty, 442
 - maszyna wirtualna, 429
 - metody uogólnione, 425
 - ograniczenia, 434
 - parametry typowe, 421
 - refleksja, 453
 - sprawdzanie typów w czasie działania programu, 435
 - statyczny kontekst klas uogólnionych, 440
 - tablice typów uogólnionych, 435
 - translacja metod uogólnionych, 431
 - translacja wyrażeń generycznych, 430
 - typy proste jako parametry typowe, 434
 - typy surowe, 429
 - typy wieloznaczne, 445
 - wyjątki, 440
 - zastosowanie, 422
 - zachowawcze, 390
 - zorientowane obiektowo, 26, 135
- programy
 - refleksyjne, 269
 - wielowątkowe, 680
- projektowanie klas, 208
- property, 545
- property map, 531
- prostokąt, 553
- protected, 204, 229, 230
- próg równoległania, 736
- prywatne pola, 158
- przechwytywanie wyjątków, 273, 374, 376
- przeciążanie
 - konstruktorów, 176
 - metod, 174
- przekazywanie
 - przez nazwę, 168
 - przez referencję, 170
 - przez wartość, 171
 - wyjątków, 389
- przekierowywanie strumieni, 774
- przełączniki, 614
 - powiadamianie o zdarzeniach, 615
- przemienność, 233
- przenośność, 28
- przepełnienie stosu, 27

przepływ sterowania, 99
przerywanie
 działania pętli, 115
 przepływu sterowania, 114
 wątków, 689
przesłanianie
 metod, 214
 składowych obiektów, 177
przestrzeń numeracyjna, 63
przetwarzanie danych wejściowych, 93
przycisk radiowy, 611, 614
przyciski, 569, 614
przypisanie, =, 72
przywileje klasowe, 159
public, 54, 55, 152, 194, 230
publiczne
 metody akcesora, 158
 metody mutatora, 158
 pola danych, 158
puchnięcie kodu szablonów, 430
pule wątków, 742
 kontrola grup zadań, 747
 tworzenie, 743

Q

queue, 465, 494, 731
quick sort, 125, 521

R

race condition, 695
ragged arrays, 131
ramki, 541
 pozycjonowanie, 543
 struktura wewnętrzna, 548
 warstwy, 547
 wyświetlanie, 543
random access, 474
referencja, 142, 171
 listener, 347
 null, 142, 154
 this, 343
referencje
 do konstruktorów, 331
 do metod, 328, 330
 do obiektu, 142
 słabe, 505

refleksja, 211, 269, 297
 analiza
 funkcjonalności klasy, 276
 obiektów w czasie działania programu, 283
 generyczny kod tablicowy, 288
 informacje o typach generycznych, 454
 klasy, 270, 276
 nazwy pól, 283
 parametry Class<T>, 454
 typy
 pól, 283
 uogólnione, 453
 wskaźniki do metod, 291
rejestrujący obiekt pośredni, 415
rekordy, 183
 dziennika, 402
relacje
 między klasami, 138
 między modelem, widokiem i kontrolerem, 598
 między typami generycznymi, 445
REPL, read-evaluate-print loop, 50
repozytorium Preferences, 589
resource bundle, 400
rodzina czcionek, 559
root pane, 547
rozgałęzienie-złączenie, fork-join, 751
rozkład
 brzegowy, 601
 GridBagLayout, 644–650, 655
 siatkowy, 603
 sprężynowy, 644
rozmiar ekranu, 545
rozstrzyganie przeciążania, 174, 221
rozszerzanie klas, 136
równość, *Patrz* operatory relacyjne, 73
runtime_error, 369
rysowanie, 540, 547
 figury 2D, 551
rzutowanie, 71, 225, 227

S

scroll pane, 609
SE, Standard Edition, 39
short, 58
siatka, 646
sieciowość, 26
składanie łańcuchów, 79, 86, 561

- składnia
 - diamantowa, 245
 - Javy, 25
 - wyrażeń lambda, 323
- składowe, 136, 157
- skróty, 487
- slider, 611
- słowniki, 497, 731
 - akcji, 579
 - modyfikowanie atomowe, 733
 - modyfikowanie wpisów, 501
 - próg zrównoleglania, 737
 - sprawdzanie, 578, 579
 - tworzenie widoku, 502
 - wejściowe, 578
 - własności, 531
- słowo kluczowe, 776–779
 - abstract, 258
 - assert, 391
 - break, 112, 115
 - case, 110
 - catch, 374
 - class, 54
 - const, 67
 - continue, 116
 - default, 110
 - else, 101
 - enum, 67
 - extends, 212, 305
 - final, 66, 160, 217, 224
 - finally, 378, 380
 - if, 100
 - implements, 300
 - import, 93, 189
 - instanceof, 226
 - interface, 299
 - new, 92, 120, 140
 - non-sealed, 267
 - package, 191, 194
 - private, 152, 160, 229, 230
 - protected, 229, 230
 - public, 54, 152, 230
 - static, 162, 163
 - super, 215
 - switch, 74, 110
 - synchronized, 699, 707
 - this, 178, 178, 216, 330, 343
 - throw, 371
 - throws, 98, 369, 371
 - try, 374
 - try-finally, 380
 - var, 154
 - void, 56
 - volatile, 715
 - while, 102, 104
 - yield, 113
- sortowanie, 520
 - quick sort, 521
 - tablicy, 125
- specyfikacja wyjątku, 370
- specyfikator formatu, 95
- sprawdzanie
 - parametrów, 392
 - zakresu, [], 124
- sprzężenie zwrotne, 311
- stack trace, 382
- stałe, 66
 - interfejsu Action, 576
 - klasowe, 66, 160
 - łańcuchowe, 81
 - matematyczne, 68
 - statyczne, 162
- stan
 - obiektu, 136, 137
 - wątku, 686
- static, 162–164, 179, 353
 - binding, 222
 - final, 66
 - void, 55
- statyczne klasy wewnętrzne, 351
- statyczny inicjalizator, 722
- sterta, 124, 496
- STL, Standard Template Library, 465
- stos, 534
 - wywołań, 382
- stosowanie wyjątków, 387
- struktura danych, 464
 - drzewo czerwono-czarne, 490
 - kolejka, 465, 494
 - kolejka priorytetowa, 496
 - lista powiązana, 476
 - lista tablicowa, 486
 - słownik, 497
 - sterta, 124, 496
 - stos, 534
 - tablica mieszająca, 486
 - zbiór, 488

- strumień
 - System.err, 401
 - System.in, 92
 - System.out, 56, 98
 - subclass, 212
 - substitution principle, 220
 - super, 215
 - superclass, 212
 - supplementary characters, 63
 - surrogates area, 63
 - suwaki, 623
 - podziałka, 624
 - Swing
 - akcelerator, 636
 - czcionki, 559
 - dodawanie komponentów, 602
 - dymki, 643
 - etykiety, 606
 - figura 2D, 551
 - ikony, 544
 - klasy, 541, 598, 600, 602-609, 614, 635, 645, 672
 - kolory, 557
 - komponenty, 548, 595, 600
 - konfiguracja komponentów, 542
 - kontener, 600
 - listy rozwijalne, 620
 - mapa wejścia, 578
 - menu, 629
 - menu podręczne, 634
 - niestandardowi zarządcy rozkładu, 653
 - obramowanie, 617
 - obsługa zdarzeń, 548
 - obszary tekstowe, 608
 - okna dialogowe, 657
 - panel przewijany, 609
 - paski narzędzi, 641
 - pola
 - hasła, 608
 - tekstowe, 605
 - wyboru, 612
 - przełączniki, 614
 - ramki, 541
 - pozycjonowanie, 543, 544
 - warstwy, 547
 - wyświetlanie, 543
 - relacje pomiędzy modelem, widokiem i kontrolerem, 598
 - rozkład
 - brzegowy, 601
 - siatkowy, 603
 - rozmiar ekranu, 545
 - rysowanie, 547
 - suwaki, 623
 - tekst paska tytułu, 544
 - treść, 596
 - wątek dystrybucji zdarzeń, 542
 - wprowadzanie tekstu, 604
 - wybór opcji, 611
 - wygląd, 596
 - wymuszanie ponownego rysowania ekranu, 549
 - wysyłanie zdarzeń, 542
 - wyświetlanie informacji w komponencie, 547
 - zachowanie, 596
 - zamykanie ramki aplikacji, 543
 - zarządca rozkładu grupowego, 644
 - zarządzanie rozkładem, 599, 644
 - switch, 74, 110
 - break, 112
 - case, 110
 - default, 110
 - rodzaje konstrukcji, 113
 - sygnał SIGTERM, 772
 - sygnatura metody, 174, 222
 - symbole zastępcze znaków specjalnych, 61
 - symetria, 233
 - synchronizacja wątków, 695
 - synchronizacyjne obiekty opakowujące, 740
 - synchronized, 699, 707
- §**
- ścieżka do klasy, 195
 - ustawianie, 197
 - ściśła kontrola typów, 301
 - śledzenie przepływu wykonywania, 397
 - środowisko programistyczne, 38
- T**
- tabela metod, 222
 - tablica powiązana skrótów, 486, 505
 - tablice, 120
 - deklaracja, 120
 - for each, 122, 123
 - generyczne, 438
 - kopiowane przy zapisie, 739
 - kopiowanie, 123
 - liczba elementów, 122
 - numerowanie elementów, 121

mieszające, 237, 486
 kolizja, 487
 komórki, 487
 kubelki, 487
 reorganizacja, 488
 współczynnik zapęnlienia, 488
 postrzępione, 130
 przetwarzanie, 122
 sortowanie, 125
 tworzenie, 120, 438
 wielowymiarowe, 128
 tagging interface, 318
 tasowanie, 520
 tekst, 604
 pasku tytułu, 544
 template code bloat, 430
 thread of control, 680
 toolbar, 641
 tooltips, 643
 tożsamość obiektu, 137
 translacja
 metod uogólnionych, 431
 wyrażeń generycznych, 430
 trigger key, 668
 true, 58, 63
 tworzenie
 akceleratorów, 636
 algorytmów, 528
 egzemplarzy klas, 136
 egzemplarzy typu uogólnionego, 424
 GUI, 539
 klasy wyjątków, 373
 konstruktorów, 152
 menu, 629
 metod uogólnionych, 425
 obiektów, 140, 154, 174, 358, 756
 okna dialogowego, 663
 plików JAR, 198
 puli wątków, 743
 ramek, 541
 tablic, 120, 131, 438
 widoków słowników, 502
 zegara, 312
 type parameters, 421
 typy danych, 58
 generyczne, 244, 423, 435
 interfejsowe, 467
 łańcuchy, 77
 numeryczne, 59

konwersja, 70
 rzutowanie, 71
 sparametryzowane, 33, 444
 surowe, 246, 429, 435, 444
 uogólnione, 244, 423, 435
 refleksja, 453
 wieloznaczne, 423, 445
 bez ograniczeń, 450
 chwytanie, 450
 nadtypy, 447
 ograniczenia nadtypów, 447, 448
 wyliczeniowe, 67
 zmiennoprzecinkowe, 59

U

uchwyty procesów, 772
 ukrywanie danych, 136
 UML, Unified Modeling Language, 139
 unchecked exception, 368
 Unicode, 61, 62, 77
 uruchamianie
 aplikacji, 54
 procesu, 771
 usługi, 355
 moduły, 355
 usuwanie
 błędów, 414
 nieużytków, 24, 183
 UTC, Coordinated Universal Time, 143
 UTF-16, 63

V

varargs, 255, 398
 view, 509, 596
 void, 56
 volatile, 715

W

wait set, 704
 wartości
 domyślne, 174
 logiczne, 63
 wartość null, 81, 154, 174, 329, 502
 warunki, 702
 wątek
 dystrybucji zdarzeń, 542, 721
 sterowania, 680

- wątki, 680, 681
 - blok synchronizowany, 711
 - blokowanie po stronie klienta, 712
 - demony, 692
 - dostęp jednoczesny, 698
 - interfejs, 699, 707, 742
 - kolejka blokująca, 725
 - klasa, 743
 - metoda rozgałęzienie-złączenie, 751
 - monitor, 713
 - niesynchronizowane, 701
 - oczekujące, 704
 - podkradanie pracy, 753
 - poła ulotne, 714
 - priorytety, 694
 - MAX_PRIORITY, 694, 695
 - MIN_PRIORITY, 694, 695
 - NORM_PRIORITY, 694, 695
 - przerywanie, 682, 689
 - pule, 741
 - stany, 686
 - BLOCKED, 687, 690, 701
 - NEW, 686
 - RUNNABLE, 686
 - TERMINATED, 686
 - WAITING, 687
 - starsze kolekcje bezpieczne wątkowo, 740
 - status przerwania, 689
 - synchronizowane, 695, 701, 707
 - tablice kopiowane przy zapisie, 739
 - volatile, 715
 - warunki, 702
 - wyścig, 695, 698
 - wywłaszczanie, 686
 - wyzerowanie statusu przerwania, 691
 - zakleszczenia, 704, 718
 - zamykanie, 688
 - zatrzymanie wykonywania, 685
 - zmienne lokalne, 722
- weak reference, 505
- weakly consistent iterators, 731
- wejscie i wyjscie, 92
- wiązanie
 - dynamiczne, 217
 - statyczne, 222
- widoczność, 223
- widoki, 509, 596
 - kontrolowane, 513
 - przedziałowe, 513
 - słowników, 502
 - synchronizowane, 514
 - współbieżne zbiorów, 738
- wielkie liczby, 117
- wielkość liter, 54
- wielowątkowość, 29
- wiersz poleceń, 42
 - opcje, 202
 - parametry, 124
- własności
 - interfejsów, 305, 599
 - klas pośredniczących, 362
 - wątków, 689
- włączanie asercji, 391
- wprowadzanie tekstu, 604
- wskaźnik
 - do obiektu, 141
 - do metody, 292
- współbieżność, 680
- współrzędne, 553
 - kodowe znaków, 63, 82
- wstawianie komentarzy javadoc, 203
- wydajność kodu bajtowego, 29
- wyjątek, 271, 366, 387
 - ArrayIndexOutOfBoundsException, 368–370
 - ArrayStoreException, 221, 435, 437
 - BadCastException, 453
 - CancellationException, 767
 - ClassCastException, 226, 234, 288, 444, 514
 - CloneNotSupportedException, 319
 - ConcurrentModificationException, 481, 731
 - EmptyStackException, 389
 - EOFException, 371
 - Exception, 367, 385
 - FileNotFoundException, 369, 371
 - IllegalAccessException, 283
 - IllegalArgumentException, 390
 - IllegalStateException, 470
 - InterruptedException, 442, 682, 690, 742
 - IOException, 371, 375
 - NoSuchElementException, 473, 485, 519, 750
 - NullPointerException, 154, 368, 389, 393
 - NumberFormatException, 381, 388
 - RuntimeException, 367–369, 385, 388
 - ServletException, 377
 - TimeoutException, 742
 - UnsupportedOperationException, 503, 514

wyjątki, 271, 366, 387
 analiza danych ze stosu wywołań, 382
 blok
 try-catch, 374
 try-finally, 380
 budowanie łańcuchów wyjątków, 377
 diagram hierarchii, 367
 finally, 378
 hierarchia, 388
 klasy wyjątków, *Patrz* wyjątek
 klasyfikacja, 367
 kontrolowane, 271, 273, 368–370
 metody, 369
 niekontrolowane, 273, 368
 powtórne generowanie, 377
 przechwytywanie, 273, 374, 376
 przekazywanie, 389
 specyfikacja, 370
 stosowanie, 387
 tworzenie klas wyjątków, 373
 wyciszenie, 388
 wykonawcze, 367
 wyłączenie, 441
 zgłaszanie, 371
 wyliczenia, 67, 262, 530
 wyłączanie
 asercji, 391
 dziedziczenia, 224
 wymazywanie typów, 431
 wymiana danych, 666
 wyrażenia
 skręcanie, 72
 lambda, 322–334
 WYSIWYG, 597
 wysyłanie zdarzeń, 542
 wyszukiwanie
 binarne, 523
 błędów, 414
 wyścig, 695, 698
 wyświetlanie
 informacji w komponencie, 547
 obrazów, 565
 ramki, 543
 wywłaszczanie wątków, 686
 wywołania zwrotne, 311
 wywołanie
 przez nazwę, 168
 przez referencję, 168
 przez wartość, 168

wywoływanie
 konstruktorów, 178, 291
 metod, 221, 292
 wzorzec MVC, 596

X

XML, 34

Z

zachowanie obiektu, 137
 zadania, 741, 746
 czasochłonne, 762
 zakleszczenie, 704, 718
 zależność, 138
 zamiana parametrów obiektowych, 171
 zamykanie
 aplikacji, 543
 wątków, 688
 zapis
 do dziennika, 396
 do pliku, 97, 403
 zarządca rozkładu, 599, 644
 brzegowego, 601
 ciągłego, 600
 grupowego, 644
 siatkowego, 603
 zarządzanie rozkładem
 GridBagLayout, 644
 niestandardowe, 653
 zasada zamienialności, 220
 zasięg zmiennych, 99
 zasoby, 274
 zbiory, 488, 497, 731
 część wspólna, 526
 uporządkowane, 490
 widoki współbieżne, 738
 zdarzenia, 312, 566
 interfejs nasłuchu, 567
 klasy, 567, 573, 586, 587
 myszy, 580
 niskiego poziomu, 586
 obiekty nasłuchujące, 313
 obsługa, 312, 566, 588
 okna, 573
 semantyczne, 586
 zegar, 312
 zgłaszanie wyjątków, 371

zintegrowane środowisko programistyczne, IDE, 47
zmiana stanu okna, 587
zmienna liczba parametrów, 255
zmienne, 64, 332
 atomowe, 716
 definicja, 66
 deklaracja, 64, 66, 154
 finalne, 333, 346, 716
 inicjalizacja, 65
 interfejsowe, 305
 lokalne, 154, 722
 obiektywne, 140, 142
 polimorficzne, 220
 typu Object, 231
 typu T, 424, 427
 ograniczenia, 428
 używanie, 437
warunkowe, 702
zasięg, 99
znacznik
 @Override, 235
 append, 403
 dokumentacyjny, 204
znak
 echa, 608
 średnika, 64
znaki
 dodatkowe, 63
 konwersji, 95
 specjalne, 61
 Unicode, 64
zwracanie referencji, 159
zwrotność, 233

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:

biegle opanuj język mistrzów programowania!

W 1995 roku świat ujrzał przyszlą gwiazdę programowania: Javę. Dziś to język dojrzały i elastyczny, dzięki czemu może służyć do pisania dużych systemów, małych programów, aplikacji mobilnych i internetowych. Java została zaprojektowana z ogromną starannością. W język wbudowano wysublimowane zabezpieczenia, a także pewne zaawansowane funkcje, które docenia każdy programista tworzący systemy o skomplikowanej architekturze.

Ta książka jest kolejnym, zaktualizowanym i uzupełnionym wydaniem kultowego podręcznika dla profesjonalnych programistów Javy. To pierwszy tom, w którym opisano podstawy języka i najważniejsze zagadnienia związane z programowaniem interfejsu użytkownika, a także kolekcje, wyrażenia lambda, techniki programowania współbieżnego i funkcyjnego. W tym wydaniu poszczególne zagadnienia zoptymalizowano pod kątem Javy 17, opisano też takie nowości jak bloki tekstu, rozszerzenia konstrukcji `switch`, rekordy, dopasowywanie wzorców operatora `instanceof`, klasy zapieczętowane i wiele więcej. Podręcznik zawiera mnóstwo przykładów kodu obrazujących zasady działania niemal każdej opisywanej funkcji czy biblioteki. Aby nauka najważniejszych zagadnień była jeszcze łatwiejsza, przykładowe programy są proste i realistyczne.

W książce między innymi:

- składnia i najlepsze praktyki pisania kodu w języku Java
- interfejsy, klasy wewnętrzne i wyrażenia lambda
- obsługa wyjątków i skuteczne techniki debugowania
- korzystanie z typów generycznych i standardowych kolekcji Javy
- nowoczesne graficzne interfejsy użytkownika przy użyciu komponentów Swing
- stosowanie modelu współbieżności Javy

Cay S. Horstmann jest autorem najpopularniejszych w Polsce podręczników do nauki Javy, zdobywcą tytułu Java Champion i częstym gościem konferencji programistycznych. Urodził się w północnych Niemczech, obecnie mieszka w USA, gdzie pracuje jako profesor informatyki na Uniwersytecie Stanowym w San José.

**Helion**

 helion.pl

**HELION SA**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 250 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-9479-7



9 788328 394797

Cena: 129,00 zł

