

WYKORZYSTAJ POTENCJAŁ LIDERA
NA RYNKU JĘZYKÓW PROGRAMOWANIA!

 PRENTICE
HALL

JAVATM

Podstawy

WYDANIE IX



CAY S. HORSTMANN · GARY CORNELL

 Helion

Tytuł oryginału: Core Java Volume I--Fundamentals (9th Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-7761-0

Authorized translation from the English language edition, entitled CORE JAVA VOLUME I – FUNDAMENTALS, 9TH EDITION; ISBN 0137081898; by Cay S. Horstmann; and Gary Cornell; published by Pearson Education, Inc, publishing as Prentice Hall. Copyright © 2013 by Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065.

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 © 2013.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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 właścicielami.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przykłady/javpd9.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javpd9_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

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

Spis treści

Wstęp	13
Podziękowania	19
Rozdział 1. Wstęp do Javy	21
1.1. Java jako platforma programistyczna	21
1.2. Słowa klucze białej księgi Javy	22
1.2.1. Prosty	23
1.2.2. Obiektowy	24
1.2.3. Sieciowy	24
1.2.4. Niezawodny	24
1.2.5. Bezpieczny	25
1.2.6. Niezależny od architektury	26
1.2.7. Przenośny	26
1.2.8. Interpretowany	27
1.2.9. Wysokowydajny	27
1.2.10. Wielowątkowy	28
1.2.11. Dynamiczny	28
1.3. Apletty Javy i internet	28
1.4. Krótka historia Javy	30
1.5. Główne nieporozumienia dotyczące Javy	32
Rozdział 2. Środowisko programistyczne Javy	37
2.1. Instalacja oprogramowania Java Development Kit	38
2.1.1. Pobieranie pakietu JDK	38
2.1.2. Ustawianie ścieżki dostępu	39
2.1.3. Instalacja bibliotek i dokumentacji	41
2.1.4. Instalacja przykładowych programów	42
2.1.5. Drzewo katalogów Javy	42
2.2. Wybór środowiska programistycznego	43
2.3. Używanie narzędzi wiersza poleceń	44
2.3.1. Rozwiązywanie problemów	45
2.4. Praca w zintegrowanym środowisku programistycznym	47
2.4.1. Znajdowanie błędów kompilacji	49

2.5.	Uruchamianie aplikacji graficznej	50
2.6.	Tworzenie i uruchamianie apletów	53

Rozdział 3. Podstawowe elementy języka Java **57**

3.1.	Prosty program w Javie	58
3.2.	Komentarze	61
3.3.	Typy danych	62
3.3.1.	Typy całkowite	62
3.3.2.	Typy zmennoprzecinkowe	63
3.3.3.	Typ char	65
3.3.4.	Typ boolean	66
3.4.	Zmienne	66
3.4.1.	Inicjacja zmiennych	68
3.4.2.	Stałe	68
3.5.	Operatory	69
3.5.1.	Operatory inkrementacji i dekrementacji	71
3.5.2.	Operatory relacyjne i logiczne	71
3.5.3.	Operatory bitowe	72
3.5.4.	Funkcje i stałe matematyczne	73
3.5.5.	Konwersja typów numerycznych	74
3.5.6.	Rzutowanie	75
3.5.7.	Nawiąsy i priorytety operatorów	76
3.5.8.	Typ wyliczeniowy	77
3.6.	Łańcuchy	77
3.6.1.	Podłańcuchy	78
3.6.2.	Konkatenacja	78
3.6.3.	Łańcuchów nie można modyfikować	79
3.6.4.	Porównywanie łańcuchów	79
3.6.5.	Łańcuchy puste i łańcuchy null	81
3.6.6.	Współrzędne kodowe znaków i jednostki kodowe	81
3.6.7.	API String	83
3.6.8.	Dokumentacja API w internecie	85
3.6.9.	Składanie łańcuchów	86
3.7.	Wejście i wyjście	89
3.7.1.	Odbieranie danych wejściowych	89
3.7.2.	Formatowanie danych wyjściowych	91
3.7.3.	Zapis i odczyt plików	96
3.8.	Przepływ sterowania	98
3.8.1.	Zasięg blokowy	98
3.8.2.	Instrukcje warunkowe	99
3.8.3.	Pętle	101
3.8.4.	Pętle o określonej liczbie powtórzeń	106
3.8.5.	Wybór wielokierunkowy — instrukcja switch	109
3.8.6.	Instrukcje przerywające przepływ sterowania	111
3.9.	Wielkie liczby	114
3.10.	Tablice	116
3.10.1.	Pętla typu for each	117
3.10.2.	Inicjowanie tablic i tworzenie tablic anonimowych	118
3.10.3.	Kopiowanie tablicy	119
3.10.4.	Parametry wiersza poleceń	120
3.10.5.	Sortowanie tablicy	121

3.10.6. Tablice wielowymiarowe	124
3.10.7. Tablice postrzepione	127
Rozdział 4. Obiekty i klasy	131
4.1. Wstęp do programowania obiektowego	132
4.1.1. Klasы	132
4.1.2. Obiekty	133
4.1.3. Identyfikacja klas	134
4.1.4. Relacje między klasami	135
4.2. Używanie klas predefiniowanych	137
4.2.1. Obiekty i zmienne obiektów	137
4.2.2. Klasa GregorianCalendar	139
4.2.3. Metody udostępniające i zmieniające wartość elementu	141
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	151
4.3.4. Pierwsze kroki w tworzeniu konstruktorów	152
4.3.5. Parametry jawne i niejawne	153
4.3.6. Korzyści z hermetyzacji	155
4.3.7. Przywileje klasowe	157
4.3.8. Metody prywatne	157
4.3.9. Stałe jako pola klasy	158
4.4. Pola i metody statyczne	158
4.4.1. Pola statyczne	159
4.4.2. Stałe statyczne	159
4.4.3. Metody statyczne	160
4.4.4. Metody fabryczne	161
4.4.5. Metoda main	162
4.5. Parametry metod	164
4.6. Konstruowanie obiektów	171
4.6.1. Przeciążanie	171
4.6.2. Inicjacja pól wartościami domyślnymi	171
4.6.3. Konstruktor bezargumentowy	172
4.6.4. Jawna inicjacja pól	172
4.6.5. Nazywanie parametrów	174
4.6.6. Wywoływanie innego konstruktora	174
4.6.7. Bloki inicjujące	175
4.6.8. Niszczenie obiektów i metoda finalize	179
4.7. Pakiety	180
4.7.1. Importowanie klas	180
4.7.2. Importy statyczne	182
4.7.3. Dodawanie klasy do pakietu	182
4.7.4. Zasięg pakietów	185
4.8. Ścieżka klas	187
4.8.1. Ustawianie ścieżki klas	189
4.9. Komentarze dokumentacyjne	190
4.9.1. Wstawianie komentarzy	190
4.9.2. Komentarze do klas	191
4.9.3. Komentarze do metod	191
4.9.4. Komentarze do pól	192

4.9.5. Komentarze ogólne	192
4.9.6. Komentarze do pakietów i ogólne	194
4.9.7. Generowanie dokumentacji	194
4.10. Porady dotyczące projektowania klas	195
Rozdział 5. Dziedziczenie	199
5.1. Klasy, nadklasy i podklasy	200
5.1.1. Hierarchia dziedziczenia	206
5.1.2. Polimorfizm	207
5.1.3. Wiązanie dynamiczne	209
5.1.4. Wyłączanie dziedziczenia — klasy i metody finalne	211
5.1.5. Rzutowanie	212
5.1.6. Klasy abstrakcyjne	214
5.1.7. Ochrona dostępu	219
5.2. Klasa bazowa Object	220
5.2.1. Metoda equals	221
5.2.2. Porównywanie a dziedziczenie	222
5.2.3. Metoda hashCode	225
5.2.4. Metoda toString	228
5.3. Generyczne listy tablicowe	233
5.3.1. Dostęp do elementów listy tablicowej	236
5.3.2. Zgodność pomiędzy typowanymi a surowymi listami tablicowymi	239
5.4. Osłony obiektów i autoboxing	241
5.5. Metody ze zmienną liczbą parametrów	244
5.6. Klasy wyliczeniowe	245
5.7. Refleksja	247
5.7.1. Klasa Class	248
5.7.2. Podstawy przechwytywania wyjątków	250
5.7.3. Zastosowanie refleksji w analizie funkcjonalności klasy	252
5.7.4. Refleksja w analizie obiektów w czasie działania programu	257
5.7.5. Zastosowanie refleksji w generycznym kodzie tablicowym	261
5.7.6. Wywoływanie dowolnych metod	264
5.8. Porady projektowe dotyczące dziedziczenia	268
Rozdział 6. Interfejsy i klasy wewnętrzne	271
6.1. Interfejsy	272
6.1.1. Właściwości interfejsów	276
6.1.2. Interfejsy a klasy abstrakcyjne	279
6.2. Klonowanie obiektów	280
6.3. Interfejsy a sprzężenie zwrotne	286
6.4. Klasy wewnętrzne	289
6.4.1. Dostęp do stanu obiektu w klasie wewnętrznej	289
6.4.2. Specjalne reguły składniowe dotyczące klas wewnętrznych	293
6.4.3. Czy klasy wewnętrzne są potrzebne i bezpieczne?	294
6.4.4. Lokalne klasy wewnętrzne	296
6.4.5. Dostęp do zmiennych finalnych z metod zewnętrznych	297
6.4.6. Anonimowe klasy wewnętrzne	300
6.4.7. Statyczne klasy wewnętrzne	303
6.5. Klasy proxy	306
6.5.1. Właściwości klas proxy	311

Rozdział 7. Grafika	313
7.1. Wprowadzenie do pakietu Swing	314
7.2. Tworzenie ramki	318
7.3. Pozycjonowanie ramki	321
7.3.1. Właściwości ramek	322
7.3.2. Określanie rozmiaru ramki	323
7.4. Wyświetlanie informacji w komponencie	327
7.5. Figury 2D	332
7.6. Kolory	340
7.7. Czcionki	343
7.8. Wyświetlanie obrazów	351
Rozdział 8. Obsługa zdarzeń	355
8.1. Podstawy obsługi zdarzeń	355
8.1.1. Przykład — obsługa kliknięcia przycisku	357
8.1.2. Nabycianie biegłości w posługiwaniu się klasami wewnętrznymi	362
8.1.3. Tworzenie słuchaczy zawierających jedno wywołanie metody	364
8.1.4. Przykład — zmiana stylu	366
8.1.5. Klasы адаптacyjные	369
8.2. Akcje	373
8.3. Zdarzenia generowane przez mysz	380
8.4. Hierarchia zdarzeń w bibliotece AWT	387
8.4.1. Zdarzenia semantyczne i niskiego poziomu	388
Rozdział 9. Komponenty Swing interfejsu użytkownika	391
9.1. Swing a wzorzec projektowy Model-View-Controller	392
9.1.1. Wzorce projektowe	392
9.1.2. Wzorzec Model-View-Controller	393
9.1.3. Analiza MVC przycisków Swing	397
9.2. Wprowadzenie do zarządzania rozkładem	398
9.2.1. Rozkład brzegowy	400
9.2.2. Rozkład siatkowy	402
9.3. Wprowadzanie tekstu	406
9.3.1. Pola tekstowe	406
9.3.2. Etykiety komponentów	408
9.3.3. Pola haseł	410
9.3.4. Obszary tekstowe	410
9.3.5. Panele przewijane	411
9.4. Komponenty umożliwiające wybór opcji	413
9.4.1. Pola wyboru	413
9.4.2. Przełączniki	415
9.4.3. Obramowanie	419
9.4.4. Listy rozwijalne	423
9.4.5. Suwaki	426
9.5. Menu	432
9.5.1. Tworzenie menu	432
9.5.2. Ikony w elementach menu	435
9.5.3. Pola wyboru i przełączniki jako elementy menu	436
9.5.4. Menu podręczne	437
9.5.5. Mnemoniki i akceleratory	438

9.5.6. Aktywowanie i dezaktywowanie elementów menu	440
9.5.7. Paski narzędzi	444
9.5.8. Dymki	446
9.6. Zaawansowane techniki zarządzania rozkładem	448
9.6.1. Rozkład GridLayout	449
9.6.2. Rozkład grupowy	459
9.6.3. Nieużywanie żadnego zarządcy rozkładu	468
9.6.4. Niestandardowi zarządcy rozkładu	469
9.6.5. Kolejka dostępu	472
9.7. Okna dialogowe	474
9.7.1. Okna dialogowe opcji	474
9.7.2. Tworzenie okien dialogowych	484
9.7.3. Wymiana danych	489
9.7.4. Okna dialogowe wyboru plików	495
9.7.5. Okna dialogowe wyboru kolorów	505
Rozdział 10. Przygotowywanie apletów i aplikacji do użytku	511
10.1. Pliki JAR	512
10.1.1. Manifest	512
10.1.2. Wykonywalne pliki JAR	514
10.1.3. Zasoby	515
10.1.4. Pieczętowanie pakietów	518
10.2. Java Web Start	519
10.2.1. Piaskownica	522
10.2.2. Podpisywanie kodu	523
10.2.3. API JNLP	525
10.3. Aplet	533
10.3.1. Prosty aplet	533
10.3.2. Znacznik applet i jego atrybuty	537
10.3.3. Znacznik object	540
10.3.4. Parametry przekazujące informacje do apletów	541
10.3.5. Dostęp do obrazów i plików audio	546
10.3.6. Środowisko działania apletu	547
10.4. Zapisywanie preferencji użytkownika	549
10.4.1. Mapy własności	550
10.4.2. API Preferences	555
Rozdział 11. Wyjątki, dzienniki, asercje i debugowanie	563
11.1. Obsługa błędów	564
11.1.1. Klasifikacja wyjątków	565
11.1.2. Deklarowanie wyjątków kontrolowanych	567
11.1.3. Zgłaszanie wyjątków	569
11.1.4. Tworzenie klas wyjątków	570
11.2. Przechwytywanie wyjątków	571
11.2.1. Przechwytywanie wielu typów wyjątków	574
11.2.2. Powtórne generowanie wyjątków i budowanie łańcuchów wyjątków	575
11.2.3. Klauzula finally	576
11.2.4. Instrukcja try z zasobami	580
11.2.5. Analiza danych ze śledzenia stosu	581
11.3. Wskazówki dotyczące stosowania wyjątków	584

11.4. Asercje	587
11.4.1. Włączanie i wyłączanie asercji	588
11.4.2. Zastosowanie asercji do sprawdzania parametrów	589
11.4.3. Zastosowanie asercji do dokumentowania założeń	590
11.5. Dzienniki	591
11.5.1. Podstawy zapisu do dziennika	592
11.5.2. Zaawansowane techniki zapisu do dziennika	592
11.5.3. Zmiana konfiguracji menedżera dzienników	594
11.5.4. Lokalizacja	596
11.5.5. Obiekty typu Handler	596
11.5.6. Filtry	600
11.5.7. Formatery	600
11.5.8. Przepis na dziennik	601
11.6. Wskazówki dotyczące debugowania	609
11.7. Wskazówki dotyczące debugowania aplikacji z GUI	614
11.7.1. Zaprzeganie robota AWT do pracy	617
11.8. Praca z debuggerem	621
Rozdział 12. Programowanie ogólne	627
12.1. Dlaczego programowanie ogólne	628
12.1.1. Dla kogo programowanie ogólne	629
12.2. Definicja prostej klasy ogólnej	630
12.3. Metody ogólne	632
12.4. Ograniczenia zmiennych typowych	633
12.5. Kod ogólny a maszyna wirtualna	635
12.5.1. Translacja wyrażeń generycznych	637
12.5.2. Translacja metod ogólnych	637
12.5.3. Używanie starego kodu	639
12.6. Ograniczenia i braki	641
12.6.1. Nie można podawać typów prostych jako parametrów typowych	641
12.6.2. Sprawdzanie typów w czasie działania programu jest możliwe tylko dla typów surowych	641
12.6.3. Nie można tworzyć tablic typów ogólnych	642
12.6.4. Ostrzeżenia dotyczące zmiennej liczby argumentów	642
12.6.5. Nie wolno tworzyć egzemplarzy zmiennych typowych	643
12.6.6. Zmiennych typowych nie można używać w statycznych kontekstach klas ogólnych	645
12.6.7. Obiektów klasy ogólnej nie można generować ani przechwytywać	646
12.6.8. Uważaj na konflikty, które mogą powstać po wymazaniu typów	648
12.7. Zasady dziedziczenia dla typów ogólnych	649
12.8. Typy wieloznaczne	650
12.8.1. Ograniczenia nadtypów typów wieloznacznych	652
12.8.2. Typy wieloznaczne bez ograniczeń	655
12.8.3. Chwytanie typu wieloznacznego	655
12.9. Refleksja a typy ogólne	658
12.9.1. Zastosowanie parametrów Class<T> do dopasowywania typów	659
12.9.2. Informacje o typach generycznych w maszynie wirtualnej	659
Rozdział 13. Kolekcje	665
13.1. Interfejsy kolekcyjne	665
13.1.1. Oddzielenie warstwy interfejsów od warstwy klas konkretnych	666
13.1.2. Interfejsy Collection i Iterator	668

13.2. Konkretne klasy kolekcyjne	674
13.2.1. Listy powiązane	674
13.2.2. Listy tablicowe	684
13.2.3. Zbiór HashSet	684
13.2.4. Zbiór TreeSet	688
13.2.5. Porównywanie obiektów	689
13.2.6. Kolejki Queue i Deque	694
13.2.7. Kolejki priorytetowe	696
13.2.8. Mapy	697
13.2.9. Specjalne klasy Set i Map	702
13.3. Architektura kolekcji	706
13.3.1. Widoki i obiekty opakowujące	709
13.3.2. Operacje zbiorcze	717
13.3.3. Konwersja pomiędzy kolekcjami a tablicami	718
13.4. Algorytmy	718
13.4.1. Sortowanie i tasowanie	720
13.4.2. Wyszukiwanie binarne	722
13.4.3. Proste algorytmy	723
13.4.4. Pisanie własnych algorytmów	725
13.5. Stare kolekcje	726
13.5.1. Klasa Hashtable	726
13.5.2. Wyliczenia	727
13.5.3. Mapy własności	728
13.5.4. Stosy	729
13.5.5. Zbiory bitów	729
Rozdział 14. Wielowątkowość	735
14.1. Czym są wątki	736
14.1.1. Wykonywanie zadań w osobnych wątkach	741
14.2. Przerywanie wątków	746
14.3. Stany wątków	749
14.3.1. Wątki NEW	749
14.3.2. Wątki RUNNABLE	750
14.3.3. Wątki BLOCKED i WAITING	750
14.3.4. Zamknięcie wątków	752
14.4. Właściwości wątków	752
14.4.1. Priorytety wątków	752
14.4.2. Wątki demony	753
14.4.3. Procedury obsługi nieprzechwyconych wyjątków	754
14.5. Synchronizacja	756
14.5.1. Przykład sytuacji powodującej wyścig	756
14.5.2. Wyścigi	760
14.5.3. Obiekty klasy Lock	762
14.5.4. Warunki	765
14.5.5. Słowo kluczowe synchronized	769
14.5.6. Bloki synchronizowane	774
14.5.7. Monitor	775
14.5.8. Pola ułotne	776
14.5.9. Zmienne finalne	777
14.5.10. Zmienne atomowe	777
14.5.11. Zakleszczenia	778

14.5.12. Zmienne lokalne wątków	781
14.5.13. Testowanie blokad i odmierzanie czasu	782
14.5.14. Blokady odczytu-zapisu	783
14.5.15. Dlaczego metody stop i suspend są wycofywane	784
14.6. Kolejki blokujące	786
14.7. Kolekcje bezpieczne wątkowo	794
14.7.1. Szybkie mapy, zbiory i kolejki	794
14.7.2. Tablice kopowane przy zapisie	796
14.7.3. Starsze kolekcje bezpieczne wątkowo	796
14.8. Interfejsy Callable i Future	797
14.9. Klasa Executors	802
14.9.1. Pule wątków	803
14.9.2. Planowanie wykonywania	807
14.9.3. Kontrolowanie grup zadań	808
14.9.4. Szkielet rozgałęzienie-złączenie	809
14.10. Synchronizatory	812
14.10.1. Semafora	812
14.10.2. Klasa CountDownLatch	813
14.10.3. Bariery	814
14.10.4. Klasa Exchanger	814
14.10.5. Kolejki synchroniczne	815
14.11. Wątki a biblioteka Swing	815
14.11.1. Uruchamianie czasochłonnych zadań	816
14.11.2. Klasa SwingWorker	820
14.11.3. Zasada jednego wątku	827
Dodatek A. Słowa kluczowe Javy	829
Skorowidz	831

Wstęp

Do Czytelnika

Jezyk programowania Java pojawił się na scenie pod koniec 1995 roku i od razu zyskał sobie reputację gwiazdy. Ta nowa technologia miała się stać uniwersalnym łącznikiem pomiędzy użytkownikami a informacją, bez względu na to, czy informacje te pochodzą z serwera sieciowego, bazy danych, serwisu informacyjnego, czy jakiegokolwiek innego źródła. I rzeczywiście, Java ma niepowtarzalną okazję spełnienia tych wymagań. Ten zaprojektowany z niezwykłą starannością język zyskał akceptację wszystkich największych firm z wyjątkiem Microsoftu. Wbudowane w język zabezpieczenia działają uspokajająco zarówno na programistów, jak i użytkowników programów napisanych w Javie. Dzięki wbudowanym funkcjom zaawansowane zadania programistyczne, takie jak programowanie sieciowe, łączność pomiędzy bazami danych i wielowątkowość, są znacznie prostsze.

Do tej pory pojawiło się już osiem wersji pakietu Java Development Kit. Przez ostatnich osiemnaście lat interfejs programowania aplikacji (ang. *Application Programming Interface — API*) rozrosł się z około 200 do ponad 3000 klas. API to obejmuje obecnie tak różne aspekty tworzenia aplikacji, jak konstruowanie interfejsu użytkownika, zarządzanie bazami danych, międzynarodizacja, bezpieczeństwo i przetwarzanie XML.

Książka, którą trzymasz w ręce, jest pierwszym tomem dziewiątego wydania książki *Java. Podstawy*. Każda edycja tej książki następuje najszybciej, jak to tylko możliwe, po wydaniu kolejnej wersji pakietu Java Development Kit. Za każdym razem uaktualnialiśmy tekst książki z uwzględnieniem najnowszych narzędzi dostępnych w Javie. To wydanie opisuje Java Standard Edition (SE) 7.

Tak jak w przypadku poprzednich wydań tej książki, to również przeznaczone jest dla *poważnych programistów, którzy chcą wykorzystać technologię Java w rzeczywistych projektach*. Zakładamy, że odbiorca naszego tekstu jest programistą posiadającym duże doświadczenie w programowaniu w innym języku niż Java. Ponadto przόżno tu szukać dziecięcych przykładów (jak tostery, zwierzęta z zoo czy „rozbiegany tekst”). Nic z tych rzeczy tutaj nie znajdziesz.

Naszym celem było przedstawienie wiedzy w taki sposób, aby Czytelnik mógł bez problemu w pełni zrozumieć zasady rządzące językiem Java i jego biblioteką, a nie tylko myślał, że wszystko rozumie.

Książka ta zawiera mnóstwo przykładów kodu, obrazujących zasady działania niemal każdej opisywanej przez nas funkcji i biblioteki. Przedstawiane przez nas przykładowe programy są proste, ponieważ chcieliśmy się w nich skoncentrować na najważniejszych zagadnieniach. Niemniej znakomita większość z nich zawiera prawdziwy, nieskrócony kod. Powinny dobrze służyć jako punkt wyjścia do pisania własnych programów.

Wychodzimy z założenia, że osoby czytające tę książkę chcą (albo wręcz pragną) poznać wszystkie zaawansowane cechy Javy. Oto kilka przykładowych zagadnień, które opisujemy szczegółowo:

- programowanie obiektowe,
- mechanizm refleksji (ang. *reflections*) i obiekty proxy,
- interfejsy i klasy wewnętrzne,
- delegacyjny model obsługi zdarzeń,
- projektowanie graficznego interfejsu użytkownika za pomocą pakietu narzędzi Swing UI,
- obsługa wyjątków,
- programowanie generyczne,
- kolekcje,
- współbieżność.

Ze względu na niebywały wręcz rozwój biblioteki klas Javy opisanie w jednym tomie wszystkich własności tego języka, których potrzebuje poważny programista, graniczyłoby z cudem. Z tego powodu postanowiliśmy podzielić naszą książkę na dwa tomy. Pierwszy, który trzymasz w ręku, opisuje podstawy języka Java oraz najważniejsze zagadnienia związane z programowaniem interfejsu użytkownika. Tom drugi (który niebawem się ukaże) zawiera informacje dotyczące bardziej zaawansowanych tematów oraz opisuje złożone zagadnienia związane z programowaniem interfejsu użytkownika. Poruszane w nim tematy to:

- pliki i strumienie,
- obiekty rozproszone,
- bazy danych,
- zaawansowane komponenty GUI,
- metody rodzime,
- przetwarzanie dokumentów XML,
- programowanie sieciowe,
- zaawansowana obróbka grafiki,

- obsługa wielu języków,
- JavaBeans,
- adnotacje.

Podczas pisania książki nie da się uniknąć drobnych błędów i wpadek. Bardzo chcielibyśmy być o nich informowani. Jednak każdą taką informację wolelibyśmy otrzymać tylko jeden raz. W związku z tym na stronie <http://horstmann.com/corejava> zamieściliśmy listę najczęściej zadawanych pytań, obejść i poprawek do błędów. Formularz służący do wysyłania informacji o błędach i propozycji poprawek został celowo umieszczony na końcu strony z erratą, aby zachęcić potencjalnego nadawcę do wcześniejszego zapoznania się z istniejącymi już informacjami. Nie należy zrażać się, jeśli nie odpowiemy na każde pytanie lub nie zrobimy tego natychmiast. Naprawdę czytamy wszystkie przychodzące do nas listy i doceniamy wysiłki wszystkich naszych Czytelników wkładane w to, aby przyszłe wydania naszej książki były jeszcze bardziej zrozumiałe i zawierały jeszcze więcej pozytycznych informacji.

0 książce

Rozdział 1. stanowi przegląd właściwości języka Java, które wyróżniają go na tle innych języków programowania. Wyjaśniamy, co projektanci chcieli zrobić, a co się im udało. Następnie krótko opisujemy historię powstania Javy oraz sposob, w jaki ewoluowała.

W **rozdziale 2.** opisujemy proces pobierania i instalacji pakietu JDK (ang. *Java Development Kit*) oraz dołączonych do tej książki przykładów kodu. Następnie opisujemy krok po kroku komplikację i uruchamianie trzech typowych programów w Javie: aplikacji konsolowej, aplikacji graficznej i appletu. Naszymi narzędziami są czyste środowisko JDK, edytor tekstowy obsługujący Javę i zintegrowane środowisko programowania (ang. *Integrated Development Environment — IDE*) dla Javy.

Od **rozdziału 3.** zaczynamy opis języka programowania Java. Na początku zajmujemy się podstawami: zmiennymi, pętlami i prostymi funkcjami. Dla programistów języków C i C++ będzie to bułka z masłem, ponieważ Java i C w tych sprawach w zasadzie niczym się nie różnią. Programiści innych języków, takich jak Visual Basic, powinni bardzo starannie zapoznać się z treścią tego rozdziału.

Obecnie najpopularniejszą metodologią stosowaną przez programistów jest programowanie obiektowe, a Java to język w pełni obiektowy. W **rozdziale 4.** wprowadzamy pojęcie hermetyzacji (ang. *encapsulation*), która stanowi jeden z dwóch filarów programowania obiektowego, oraz piszemy o mechanizmach Javy służących do jej implementacji, czyli o klasach i metodach. Poza opisem zasad rządzących językiem Java dostarczamy także informacji na temat solidnego projektowania programów zorientowanych obiektowo. Na końcu poświęcamy nieco miejsca doskonałemu narzędziu o nazwie **javadoc**, służącemu do konwersji komentarzy zawartych w kodzie na wzajemnie połączone hiperlinkami strony internetowe. Osoby znające język C++ mogą przejrzeć ten rozdział pobieżnie. Programiści niemający doświadczenia w programowaniu obiektowym muszą się liczyć z tym, że opanowanie wiedzy przedstawionej w tym rozdziale zajmie im trochę czasu.

Klasy i hermetyzacja to dopiero połowa koncepcji programowania zorientowanego obiektowo. **Rozdział 5.** wprowadza drugą, czyli **dziedziczenie**. Mechanizm ten umożliwia modyfikację istniejących już klas do własnych specyficznych potrzeb. Jest to podstawowa technika programowania zarówno w Javie, jak i C++, a oba te języki mają pod tym względem wiele ze sobą wspólnego. Dlatego też programiści C++ mogą również w tym rozdziale skupić się tylko na różnicach pomiędzy tymi dwoma językami.

W **rozdziale 6.** nauczymy się posługiwać **interfejsami** w Javie, które wykraczają poza prosty model dziedziczenia opisywany w poprzednim rozdziale. Opanowanie technik związanych z interfejsami da nam pełny dostęp do możliwości, jakie stwarza w pełni obiektowe programowanie w Javie. Ponadto opisujemy bardzo przydatne w Javie **klasy wewnętrzne** (ang. *inner classes*). Pomagają one w pisaniu bardziej zwięzkiego i przejrzystego kodu.

Od **rozdziału 7.** zaczynamy poważne programowanie. Jako że każdy programista Javy powinien znać się na programowaniu GUI, w tym rozdziale opisujemy podstawy tego zagadnienia. Nauczysz się tworzyć okna, rysować w nich, rysować figury geometryczne, formatować tekst przy zastosowaniu różnych krojów pisma oraz wyświetlać obrazy.

W **rozdziale 8.** szczegółowo opisujemy model zdarzeń AWT (ang. *Abstract Window Toolkit*). Nauczmy się pisać programy reagujące na zdarzenia, takie jak kliknięcie przyciskiem myszy albo naciśnięcie klawisza na klawiaturze. Dodatkowo opanujesz techniki pracy nad takimi elementami GUI jak przyciski i panele.

Rozdział 9. zawiera bardzo szczegółowy opis pakietu Swing. Pakiet ten umożliwia tworzenie niezależnych od platformy graficznych interfejsów użytkownika. Nauczysz się posługiwać różnego rodzaju przyciskami, komponentami tekstowymi, obramowaniami, suwakami, polami list, menu i oknami dialogowymi. Niektóre zaawansowane komponenty zostały opisane dopiero w drugim tomie.

Rozdział 10. zawiera informacje na temat wdrażania programów jako aplikacji lub appletów. Nauczysz się pakować programy do plików JAR oraz udostępniać aplikacje poprzez internet za pomocą mechanizmów Java Web Start i appletów. Na zakończenie opisujemy, w jaki sposób Java przechowuje i wyszukuje informacje na temat konfiguracji już po ich wdrożeniu.

Rozdział 11. poświęciliśmy obsłudze wyjątków — doskonałemu mechanizmowi pozwalającemu radzić sobie z tym, że z dobrymi programami mogą działać się zle rzeczy. Wyjątki są skutecznym sposobem na oddzielenie kodu normalnego przetwarzania od kodu obsługującego błędy. Oczywiście nawet zabezpieczenie w postaci obsługi wszystkich sytuacji wyjątkowych nie zawsze uchroni nas przed niespodziewanym zachowaniem programu. W drugiej części tego rozdziału zawarliśmy mnóstwo wskazówek dotyczących usuwania błędów z programu. Na końcu opisujemy krok po kroku całą sesję debugowania.

W **rozdziale 12.** przedstawiamy zarys **programowania ogólnego** — najważniejszej nowości w Java SE 5.0. Dzięki tej technice można tworzyć łatwiejsze do odczytu i bezpieczniejsze programy. Przedstawiamy sposoby stosowania ścisłej kontroli typów oraz pozbywania się szpetnych i niebezpiecznych konwersji. Ponadto nauczysz się radzić sobie w sytuacjach, kiedy trzeba zachować zgodność ze starszymi wersjami Javy.

Tematem **rozdziału 13.** są **kolekcje**. Chcąc zebrać wiele obiektów, aby móc ich później użyć, najlepiej posłużyć się kolekcją, zamiast po prostu wrzucać wszystkie obiekty do tablicy. W rozdziale tym nauczysz się korzystać ze standardowych kolekcji, które są wbudowane w język i gotowe do użytku.

Kończący książkę **rozdział 14.** zawiera opis wielowątkowości, która umożliwia programowanie w taki sposób, aby różne zadania były wykonywane jednocześnie (wątek to przepływy sterowania w programie). Nauczysz się ustawać wątki i panować nad ich synchronizacją. Jako że wielowątkowość w Java 5.0 uległa poważnym zmianom, opisujemy wszystkie nowe mechanizmy z nią związane.

Dodatek zawiera listę słów zarezerwowanych w języku Java.

Konwencje typograficzne

Podobnie jak w wielu książkach komputerowych, przykłady kodu programów pisane są czcionką o stałej szerokości znaków.



Taką ikoną opatrzone są uwagi.



Tą ikoną opatrzone są wskazówki.



Takiej ikony używamy, aby ostrzec przed jakimś niebezpieczeństwem.



W książce pojawia się wiele uwag wyjaśniających różnice pomiędzy Java a językiem C++. Jeśli nie znasz się na programowaniu w C++ lub na myśl o przykrych wspomnieniach z nim związanych dostajesz gęsiej skórkę, możesz je pominąć.

Java ma bardzo dużą bibliotekę programistyczną, czyli API (ang. *Application Programming Interface*). Kiedy po raz pierwszy używamy jakiegoś wywołania API, na końcu sekcji umieszcamy jego krótki opis. Opisy te są nieco nieformalne, ale staraliśmy się, aby zawierały więcej potrzebnych informacji niż te, które można znaleźć w oficjalnej dokumentacji API w internecie. Liczba znajdująca się za nazwą klasy, interfejsu lub metody odpowiada wersji JDK, w której opisywana własność została wprowadzona.

Interfejs programowania aplikacji **1.2**

Programy, których kod źródłowy można znaleźć w internecie, są oznaczane jako listingi, np.:

Listing 1.1. `inputTest/InputTest.java`

Przykłady kodu

W witrynie towarzyszącej tej książce, pod adresem www.helion.pl/ksiazki/javpd9.htm, opublikowane są w postaci skompresowanego archiwum wszystkie pliki z przykładami kodu źródłowego. Można je wypakować za pomocą dowolnego programu otwierającego paczki ZIP albo przy użyciu narzędzia `jar` dostępnego w zestawie Java Development Kit. Więcej informacji na temat tego pakietu i przykłady kodu można znaleźć w rozdziale 2.

Podziękowania

Pisanie książki to zawsze ogromny wysiłek, a pisanie kolejnego wydania nie wydaje się dużo łatwiejsze, zwłaszcza kiedy weźmie się pod uwagę ciągłe zmiany zachodzące w technologii Java. Aby książka mogła powstać, potrzeba zaangażowania wielu osób. Dlatego też z wielką przyjemnością chciałbym podziękować za współpracę całemu zespołowi Core Java.

Wiele cennych uwag pochodzi od osób z wydawnictwa Prentice Hall, którym udało się pozostać w cieniu. Chciałbym, aby wszystkie te osoby wiedziały, że bardzo doceniam ich pracę. Jak zawsze gorące podziękowania kieruję do mojego redaktora z wydawnictwa Prentice Hall — Grega Doencha — za przeprowadzenie tej książki przez proces pisania i produkcji oraz za to, że pozwolił mi pozostać w błędnej nieświadomości istnienia wszystkich osób pracujących w zapleczu. Jestem wdzięczny Julie Nahil za doskonałe wsparcie w dziedzinie produkcji, oraz Dmitry'emu i Alinie Kirsanovom za korektę i skład. Dziękuję również mojemu współautorowi poprzednich wydań tej książki — Gary'emu Cornellowi, który podjął inne wyzwania.

Dziękuję wszystkim Czytelnikom poprzednich wydań tej książki za informacje o żenujących błędach, które popełniłem, i komentarze dotyczące ulepszenia mojej książki. Jestem szczególnie wdzięczny znakomitemu zespołowi korektorów, którzy czytając wstępную wersję tej książki i wykazując niebywałą czułość na szczegóły, uratowali mnie przed popełnieniem jeszcze większej liczby błędów.

Do recenzentów tego wydania i poprzednich edycji książki należą: Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Alec Beaton (IBM), Cliff Berg, Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), dr Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (coreservlets.com, Inc.), Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Tim Kimmel (Walmart), Chris Laffra, Charlie Lai (Apple), Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (konsultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University),

Hao Pham, Paul Phlion, Blake Ragsdell, Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nicea, Francja), dr Paul Sanghera (San Jose State University, Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Paul Tyma (konsultant), Peter van der Linden (Motorola Mobile Devices), Burt Walsh, Dan Xu (Oracle) i John Zavgren (Oracle).

Cay Horstmann
San Francisco, Kalifornia
wrzesień 2012

1

Wstęp do Javy

W tym rozdziale:

- Java jako platforma programistyczna
- Słowa klucze białej księgi Javy
- Aplety Javy i internet
- Krótka historia Javy
- Najczęstsze nieporozumienia dotyczące Javy

Pierwsze wydanie Javy w 1996 roku wywołało ogromne emocje nie tylko w prasie komputerowej, ale także w takich mediach należących do głównego nurtu, jak „The New York Times”, „The Washington Post” czy „Business Week”. Język Java został jako pierwszy i do tej pory jedyny język programowania wyróżniony krótką, bo trwającą 10 minut, wzmianką w National Public Radio. Kapitał wysokiego ryzyka w wysokości 100 000 000 dolarów został zebrany wyłącznie dla produktów powstałych przy zastosowaniu określonego języka komputerowego. Wracanie dzisiaj do tych świetnych czasów jest bardzo zabawne, a więc w tym rozdziale krótko opisujemy historię języka Java.

1.1. Java jako platforma programistyczna

W pierwszym wydaniu tej książki napisaliśmy o Javie takie oto słowa:

„Ten cały szum wokół Javy jako języka programowania jest przesadzony. Java to z pewnością dobrzy język programowania. Nie ma wątpliwości, że jest to jedno z najlepszych narzędzi dostępnych dla poważnych programistów. Naszym zdaniem mogłaby być wspaniałym językiem programowania, ale na to jest już chyba zbyt późno. Kiedy przychodzi do rzeczywistych zastosowań, swoją głowę podnosi ohydna zmora zgodności z istniejącym już kodem”.

Za ten akapit na naszego redaktora posypały się gromy ze strony kogoś bardzo wysoko postawionego w firmie Sun Microsystems, kogo nazwiska wolimy nie ujawniać. Jednak z perspektywy czasu wydaje się, że nasze przewidywania były słuszne. Java ma mnóstwo bardzo pozytycznych cech, które opisujemy w dalszej części tego rozdziału. Ma też jednak pewne wady, a najnowsze dodatki do języka ze względu na zgodność nie są już tak eleganckie jak kiedyś.

Jak jednak napisaliśmy w pierwszym wydaniu tej książki, Java nigdy nie była tylko językiem. Istnieje bardzo dużo języków programowania, a tylko kilka z nich zrobiło furorę. Java to cała **platforma** z dużą biblioteką zawierającą ogromne ilości gotowego do wykorzystania kodu oraz środowisko wykonawcze, które zapewnia bezpieczeństwo, przenośność między różnymi systemami operacyjnymi oraz automatyczne usuwanie nieużytków (ang. *garbage collecting*).

Jako programiści żądamy języka o przyjaznej składni i zrozumiałej semantyce (a więc nie C++). Do tego opisu pasuje Java, jak również wiele innych dobrych języków programowania. Niektóre z nich oferują przenośność, zbieranie nieużytków itd., ale nie mają bogatych bibliotek, przez co zmuszeni jesteśmy pisać własne, kiedy chcemy wykonać obróbkę grafiki, stworzyć aplikację sieciową bądź łączącą się z bazą danych. Cóż, Java ma to wszystko — jest to dobry język, który oddaje do dyspozycji programisty wysokiej jakości środowisko wykonawcze wraz z ogromną biblioteką. To właśnie to połączenie sprawia, że tak wielu programistów nie potrafi oprzeć się urokowi Javy.

1.2. Słowa klucze białej księgi Javy

Twórcy języka Java napisali bardzo wpływową białą księgę, w której opisali swoje cele i osiągnięcia. Dodatkowo opublikowali krótkie streszczenie zorganizowane według następujących 11 słów kluczowych:

1. prosty,
2. obiektowy,
3. sieciowy,
4. niezawodny,
5. bezpieczny,
6. niezależny od architektury,
7. przenośny,
8. interpretowany,
9. wysokowydajny,
10. wielowątkowy,
11. dynamiczny.

W tej części rozdziału:

- Krótko podsumujemy, posługując się fragmentami z białej księgi, co projektanci Javy mają do powiedzenia na temat każdego ze słów kluczowych.
- Wyrazimy własne zdanie na temat każdego z tych słów kluczowych, opierając się na naszych doświadczeniach związanych z aktualną wersją Javy.



W trakcie pisania tej książki biała księga Javy była dostępna pod adresem <http://www.oracle.com/technetwork/java/langenv-140151.html>.

1.2.1. Prosty

Naszym celem było zbudowanie takiego systemu, który można zaprogramować bez ukończenia tajemnych szkoleń, a który podtrzymywałby obecne standardowe praktyki. W związku z tym — mimo że w naszym przekonaniu język C++ nie nadawał się do tego celu — Java pod względem projektowym jest do niego podobna, jak to tylko możliwe. Dzięki temu nasz system jest bardziej zrozumiały. Java jest pozbawiona wielu rzadko używanych, słabo poznanych i wywołujących zamieszanie funkcji, które zgodnie z naszymi doświadczeniami przynoszą więcej złego niż dobrego.

Składnia Javy rzeczywiście jest oczyszczoną wersją składni języka C++. Nie ma potrzeby dołączania plików nagłówkowych, posługiwania się arytmetyką wskaźnikową (a nawet składnią wskaźnikową), strukturami, uniami, przeciążaniem operatorów, wirtualnymi klasami bazowymi itd. (więcej różnic pomiędzy Javą a C++ można znaleźć w uwagach rozmieszczonej na kartach tej książki). Nie jest jednak tak, że projektanci pozbili się wszystkich właściwości języka C++, które nie były eleganckie. Na przykład nie zrobiono nic ze składnią instrukcji switch. Każdy, kto zna język C++, z łatwością przełączy się na składnię języka Java.

Osoby przyzwyczajone do środowisk wizualnych (jak Visual Basic) przy nauce Javy będą napotykać trudności. Trzeba pojąć mnóstwo dziwnych elementów składni (choć nie zabiera to zbyt dużo czasu). Większe znaczenie ma to, że w Javie trzeba znacznie więcej pisać. Piękno języka Visual Basic polega na tym, że duża część infrastruktury aplikacji jest automatycznie dostarczana przez środowisko programistyczne. Wszystko to w Javie trzeba napisać własnoręcznie, a to z reguły wymaga dość dużej ilości kodu. Istnieją jednak środowiska udostępniane przez niezależnych producentów, które umożliwiają programowanie w stylu „przeciągnij i upuść”.

Wyznacznikiem prostoty są także niewielkie rozmiary. Jednym z celów Javy jest umożliwienie tworzenia oprogramowania działającego niezależnie na małych urządzeniach. Rozmiar podstawowego interpretera i obsługi klas wynosi około 40 kilobajtów. Podstawowe standardowe biblioteki i obsługa wątków (w zasadzie jest to samodzielne mikrojadro) to dodatkowe 175 K.

W tamtych czasach było to niebywałe wręcz osiągnięcie. Oczywiście od tamtej pory biblioteka Javy rozrosła się do nieprawdopodobnych rozmiarów. W związku z tym powstała oddzielna wersja Javy o nazwie Java Micro Edition z mniejszą biblioteką, która nadaje się do stosowania na małych urządzeniach.

1.2.2. Obiektowy

Mówiąc krótko, projektowanie obiektowe to technika programowania, której punktem centralnym są dane (czyli obiekty) oraz interfejsy dające dostęp do tych obiektów. Przez analogię — obiektowy stolarz byłby przede wszystkim zainteresowany krzesłem, które ma zrobić, a potrzebne do tego narzędzia stawiałby na drugim miejscu. Nieobiektowy stolarz z kolei na pierwszym miejscu stawiałby swoje narzędzia. Narzędzia związane z programowaniem obiektowym w Javie są w zasadzie takie jak w C++.

Obiektowa metoda programowania udowodniła swoją wartość w ciągu ostatnich 30 lat. Jest nie do pomyślenia, aby jakikolwiek nowoczesny język z niej nie korzystał. Rzeczywiście właściwości Javy, dzięki którym można nazywać ją językiem obiektowym, są podobne do języka C++. Główna różnica pomiędzy tymi dwoma językami objawia się w wielodziedziczeniu, które w Javie zostało zastąpione prostszymi interfejsami, oraz w modelu metaklas Javy (który jest opisany w rozdziale 5.).



Osoby, które nie miały styczności z programowaniem obiektowym, powinny bardzo uważnie przeczytać rozdziały 4. – 6. Zawierają one opis technik programowania obiektowego oraz wyjaśnienie, czemu ta technika lepiej nadaje się do wyrafinowanych projektów niż tradycyjne języki proceduralne, takie jak C lub Basic.

1.2.3. Sieciowy

Java ma bogatą bibliotekę procedur wspomagających pracę z takimi protokołami TCP/IP jak HTTP i FTP. Aplikacje w tym języku mogą uzyskiwać dostęp poprzez sieć do obiektów z taką samą łatwością, jakby znajdowały się one w lokalnym systemie plików.

W naszej ocenie funkcje sieciowe Javy są zarówno solidne, jak i łatwe w użyciu. Każdy, kto kiedykolwiek spróbował programowania sieciowego w innym języku programowania, będzie zachwycony tym, jak proste są tak niegdyś uciążliwe zadania jak połączenia na poziomie gniazd (ang. *socket connection*); programowanie sieciowe opisaliśmy w drugim tomie. Mechanizm zdalnych wywołań metod umożliwia komunikację pomiędzy obiektami rozproszonymi (także opisany w drugim tomie).

1.2.4. Niezawodny

Java została stworzona do pisania programów, które muszą być niezawodne w rozmaitych sytuacjach. Dużo uwagi poświęcono wczesnemu sprawdzaniu możliwości wystąpienia ewentualnych problemów, późniejszemu sprawdzaniu dynamicznemu (w trakcie działania programu) oraz wyeliminowaniu sytuacji, w których łatwo popełnić błąd. Największa różnica pomiędzy Javą a C/C++ polega na tym, że model wskaźnikowy tego pierwszego języka jest tak zaprojektowany, aby nie było możliwości nadpisania pamięci i zniszczenia w ten sposób danych.

Jest to także bardzo przydatna funkcja. Kompilator Javy wykrywa wiele błędów, które w innych językach ujawniłyby się dopiero po uruchomieniu programu. Wracając do wskaźników, każdy, kto spędził wiele godzin na poszukiwaniu uszkodzenia w pamięci spowodowanego błędnym wskaźnikiem, będzie bardzo zadowolony z Javy.

Osoby programujące do tej pory w języku takim jak Visual Basic, w którym nie stosuje się jawne wskaźników, pewnie zastanawiają się, czemu są one takie ważne. Programiści języka C nie mają już tyle szczęścia. Im wskaźniki potrzebne są do uzyskiwania dostępu do łańcuchów, tablic, obiektów, a nawet plików. W języku Visual Basic w żadnej z wymienionych sytuacji nie stosuje się wskaźników ani nie trzeba zajmować się przydzielaniem dla nich pamięci. Z drugiej jednak strony w językach pozbawionych wskaźników trudniej jest zaimplementować wiele różnych struktur danych. Java łączy w sobie to, co najlepsze w obu tych podejściach. Wskaźniki nie są potrzebne do najczęściej używanych struktur, jak łańcuchy czy tablice. Możliwości stwarzane przez wskaźniki są jednak cały czas w zasięgu ręki — przydają się na przykład w przypadku list dwukierunkowych (ang. *linked lists*). Ponadto cały czas jesteśmy w pełni bezpieczni, ponieważ nie ma możliwości uzyskania dostępu do niewłaściwego wskaźnika, popełnienia błędu przydzielania pamięci ani konieczności wystrzegania się przed wyciekiemami pamięci.

1.2.5. Bezpieczny

Java jest przystosowana do zastosowań w środowiskach sieciowych i rozproszonych. W tej dziedzinie położono duży nacisk na bezpieczeństwo. Java umożliwia tworzenie systemów odpornych na wirusy i ingerencję.

W pierwszym wydaniu naszej książki napisaliśmy: „Nigdy nie mów nigdy” i okazało się, że mieliśmy rację. Niedługo po wydaniu pakietu JDK zespół ekspertów z Princeton University znalazł ukryte błędy w zabezpieczeniach Java 1.0. Firma Sun Microsystems zaprosiła programistów do zbadania zabezpieczeń Javy, udostępniając publicznie specyfikację i implementację wirtualnej maszyny Javy oraz biblioteki zabezpieczeń. Wszystkie znane błędy zabezpieczeń zostały szybko naprawione. Dzięki temu przechytrzenie zabezpieczeń Javy jest nie lada sztuką. Znalezione do tej pory błędy były ściśle związane z techniczną stroną języka i było ich niewiele.

Od samego początku przy projektowaniu Javy starano się uniemożliwić przeprowadzanie niektórych rodzajów ataków, takich jak:

- przepelenie stosu wykonywania — często stosowany atak przez robaki i wirusy;
- niszczenie pamięci poza swoją własną przestrzenią procesową;
- odczyt lub zapis plików bez zezwolenia.

Pewna liczba zabezpieczeń została dodana do Javy z biegiem czasu. Od wersji 1.1 istnieje pojęcie klasy podpisanej cyfrowo (ang. *digitally signed class*), które opisane jest w drugim tomie. Dzięki temu zawsze wiadomo, kto napisał daną klasę. Jeśli ufamy klasie napisanej przez kogoś innego, można dać jej większe przywileje.



W konkurencyjnej technologii firmy Microsoft, opartej na ActiveX, jako zabezpieczenie stosuje się tylko podpisy cyfrowe. To oczywiście za mało — każdy użytkownik produktów firmy Microsoft może potwierdzić, że programy od znanych dostawców psują się i mogą powodować szkody. Model zabezpieczeń Javy jest o niebo lepszy niż ten oparty na ActiveX, ponieważ kontroluje działającą aplikację i zapobiega spustoszeniom, które może ona wyrządzić.

1.2.6. Niezależny od architektury

Kompilator generuje niezależny od konkretnej architektury plik w formacie obiektowym. Tak skompilowany kod można uruchamiać na wielu procesorach, pod warunkiem że zainstalowano Java Runtime System. Kompilator dokonuje tego, generując kod bajtowy niemający nic wspólnego z żadnym konkretnym procesorem. W zamian kod ten jest tak konstruowany, aby był łatwy do interpretacji na każdym urządzeniu i aby można go było z łatwością przetłumaczyć na kod maszynowy w locie.

Nie jest to żadna nowość. Już ponad 30 lat temu Niklaus Wirth zastosował tę technikę w swoich oryginalnych implementacjach systemów Pascal i UCSD.

Oczywiście interpretowanie kodu bajtowego musi być wolniejsze niż działanie instrukcji maszynowych z pełną prędkością i nie wiadomo, czy jest to tak naprawdę dobry pomysł. Jednak maszyny wirtualne mogą tłumaczyć często wykonywany kod bajtowy na kod maszynowy w procesie nazywanym komplikacją w czasie rzeczywistym (ang. *just-in-time compilation*). Metoda ta okazała się tak efektywna, że nawet firma Microsoft zastosowała maszynę wirtualną na swojej platformie .NET.

Maszyna wirtualna Javy ma także inne zalety. Zwiększa bezpieczeństwo, ponieważ może kontrolować działanie sekwencji instrukcji. Niektóre programy tworzą nawet kod bajtowy w locie, tym samym dynamicznie zwiększając możliwości działającego programu.

1.2.7. Przenośny

W przeciwieństwie do języków C i C++ Java nie jest w żaden sposób uzależniona od implementacji. Rozmiary podstawowych typów danych są określone, podobnie jak wykonywane na nich działania arytmetyczne.

Na przykład typ `int` w Javie zawsze oznacza 32-bitową liczbę całkowitą. W C i C++ typ `int` może przechowywać liczbę całkowitą 16-, 32-bitową lub o dowolnym innym rozmiarze, jaki wymyśli sobie twórca kompilatora. Jedyne ograniczenie polega na tym, że typ `int` nie może być mniejszy niż typ `short` `int` i większy niż `long` `int`. Ustalenie rozmiarów typów liczbowych spowodowało zniknięcie głównego problemu z przenoszeniem programów. Dane binarne są przechowywane i przesyłane w ustalonym formacie, dzięki czemu unika się nieporozumień związanych z kolejnością bajtów. Łańcuchy są przechowywane w standardowym formacie Unicode.

Biblioteki wchodzące w skład systemu definiują przenośne interfejsy. Dostępna jest na przykład abstrakcyjna klasa Window i jej implementacje dla systemów Unix, Windows i Mac OS X.

Każdy, kto kiedykolwiek próbował napisać program, który miał wyglądać dobrze w systemie Windows, na komputerach Macintosh i w dziesięciu różnych odmianach Uniksa, wie, jak ogromny jest to wysiłek. Java 1.0 wykonała to heroiczne zadanie i udostępniła prosty zestaw narzędzi, które odwzorowywały elementy interfejsu użytkownika na kilku różnych platformach. Niestety, w wyniku tego powstała biblioteka, która przy dużym nakładzie pracy dawała ledwie akceptowalne w różnych systemach rezultaty (dodatkowo na różnych platformach występowały **różne** błędy). Ale to były dopiero początki. W wielu aplikacjach od pięknego interfejsu użytkownika ważniejsze są inne rzeczy — właśnie takie aplikacje korzystały na pierwszych wersjach Javy. Obecny zestaw narzędzi do tworzenia interfejsu użytkownika jest napisany od nowa i nie jest uzależniony od interfejsu użytkownika hosta. Wynikiem jest znacznie spójniejszy i w naszym odczuciu atrakcyjniejszy interfejs niż ten, który był dostępny we wczesnych wersjach Javy.

1.2.8. Interpretowany

Interpreter Javy może wykonać każdy kod bajtowy Javy bezpośrednio na urządzeniu, na którym interpreter ten zainstalowano. Jako że łączenie jest bardziej inkrementalnym i lekkim procesem, proces rozwoju może być znacznie szybszy i bardziej odkrywczy.

Łączenie narastające ma swoje zalety, ale opowieści o korzyściach płynących z jego stosowania w procesie rozwoju aplikacji są przesadzone. Pierwsze narzędzia Javy rzeczywiście były powolne. Obecnie kod bajtowy jest tłumaczony przez kompilator JIT (ang. *just-in-time compiler*) na kod maszynowy.

1.2.9. Wysokowydajny

Mimo że wydajność interpretowanego kodu bajtowego jest zazwyczaj więcej niż wystarczająca, zdarzają się sytuacje, w których potrzebna jest większa wydajność. Kod bajtowy może być tłumaczony w locie (w trakcie działania programu) na kod maszynowy przeznaczony dla określonego procesora, na którym działa aplikacja.

Na początku istnienia Javy wielu użytkowników nie zgadzało się ze stwierdzeniem, że jej wydajność jest więcej niż wystarczająca. Jednak najnowsze kompilatory JIT są tak dobre, że mogą konkurować z tradycyjnymi kompilatorami, a czasami nawet je prześcigać, ponieważ mają dostęp do większej ilości informacji. Na przykład kompilator JIT może sprawdzać, która część kodu jest najczęściej wykonywana, i zoptymalizować ją pod kątem szybkości. Bardziej zaawansowana technika optymalizacji polega na eliminacji wywołań funkcji (ang. *inlining*). Kompilator JIT wie, które klasy zostały załadowane. Może zastosować wstawianie kodu funkcji w miejsce ich wywołań, kiedy — biorąc pod uwagę aktualnie załadowane kolekcje klas — określona funkcja nie jest przesłonięta i możliwe jest cofnięcie tej optymalizacji w razie potrzeby.

1.2.10. Wielowątkowy

Korzyści płynące z wielowątkowości to lepsza interaktywność i działanie w czasie rzeczywistym.

Każdy, kto próbował programowania wielowątkowego w innym języku niż Java, będzie mile zaskoczony tym, jak łatwe jest to w Javie. Wątki w Javie mogą korzystać z systemów wieloprocesorowych, jeśli podstawowy system operacyjny to umożliwia. Problem w tym, że implementacje wątków na różnych platformach znacznie się różnią, a Java nic nie robi pod tym względem, aby zapewnić niezależność od platformy. Taki sam w różnych urządzeniach pozostaje tylko kod służący do wywoływania wielowątkowości. Implementacja wielowątkowości jest w Javie zrzucana na system operacyjny lub bibliotekę wątków. Niemniej łatwość, z jaką przychodzi korzystanie z niej w Javie, sprawia, że jest ona bardzo kuszącą propozycją, jeśli chodzi o programowanie po stronie serwera.

1.2.11. Dynamiczny

Java jest bardziej dynamicznym językiem niż C i C++ pod wieloma względami. Została zaprojektowana tak, aby dostosowywać się do ewoluującego środowiska. Do bibliotek można bez przeszkód dodawać nowe metody i zmienne egzemplarzy, nie wywierając żadnego wpływu na klienty. Sprawdzanie informacji o typach w Javie nie sprawia trudności.

Cecha ta jest ważna w sytuacjach, kiedy trzeba dodać kod do działającego programu. Najważniejszy przykład takiej sytuacji to pobieranie kodu z internetu w celu uruchomienia w przeglądarce. W Javie 1.0 sprawdzenie typu w czasie działania programu było proste, ale w aktualnej wersji Javy programista ma możliwość pełnego oglądu zarówno w strukturę, jak i działanie obiektów. Jest to niezwykle ważne, zwłaszcza dla systemów, w których konieczne jest analizowanie obiektów w czasie pracy, takich jak kreatory GUI Javy, inteligentne debugery, komponenty zdolne do podłączania się w czasie rzeczywistym oraz obiektowe bazy danych.



Niedługo po początkowym sukcesie Javy firma Microsoft wydała produkt o nazwie J++. Był to język programowania i maszyna wirtualna lądująca podobne do Javy. Obecnie Microsoft nie zajmuje się już tym projektem i zwrócił się w stronę innego języka — C#, który również przypomina Javę. Istnieje nawet język J# służący do migracji aplikacji napisanych w J++ na maszynę wirtualną używaną przez C#. W książce tej nie opisujemy języków J++, C# i J#.

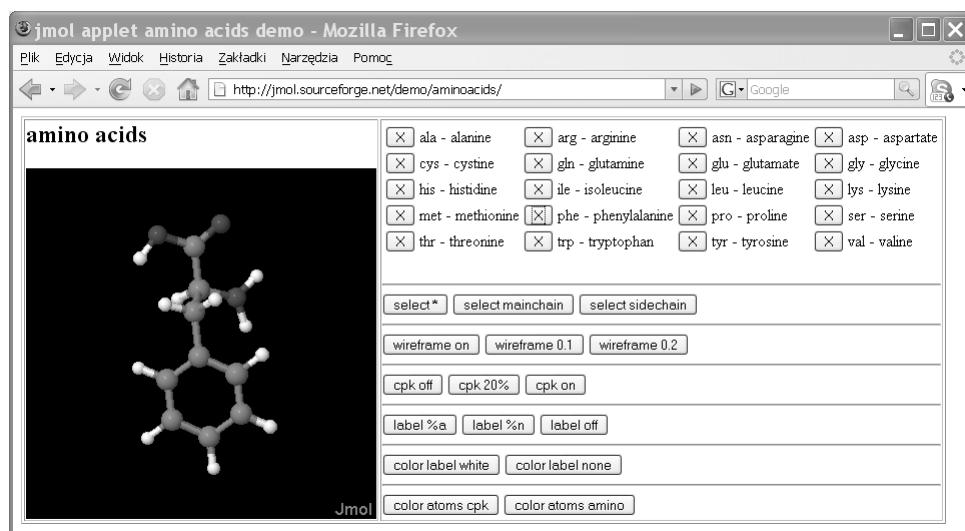
1.3. Apletty Javy i internet

Założenie jest proste: użytkownik pobiera kod bajtowy z internetu i uruchamia go na własnym urządzeniu. Programy napisane w Javie, które działają na stronach internetowych, noszą nazwę **apletów** Javy. Aby używać apletów, wystarczy mieć przeglądarkę obsługującą Javę, w której można uruchomić kod bajtowy tego języka. Nie trzeba niczego instalować. Dzięki temu,

że firma Sun udziela licencji na kod źródłowy Javy i nie zezwala na wprowadzanie żadnych zmian w języku i bibliotece standardowej, każdy aplet powinien działać w każdej przeglądarce reklamowanej jako obsługująca Javę. Najnowszą wersję oprogramowania pobiera się w trakcie odwiedzin strony internetowej zawierającej aplet. Najważniejsze jest jednak to, że dzięki zabezpieczeniom maszyny wirtualnej nie trzeba się obawiać ataków ze strony złośliwego kodu.

Pobieranie apletu odbywa się w podobny sposób jak wstawianie obrazu na stronę internetową. Aplet integruje się ze stroną, a tekst otacza go ze wszystkich stron jak obraz. Różnica polega na tym, że ten obraz jest **żywy**. Reaguje na polecenia użytkownika, zmienia wygląd oraz przesyła dane pomiędzy komputerem, na którym został uruchomiony, a komputerem, z którego pochodzi.

Rysunek 1.1 przedstawia dobry przykład dynamicznej strony internetowej, na której wykonywane są skomplikowane obliczenia. Aplet Jmol wyświetla budowę cząsteczek. Wyświetloną cząsteczkę można za pomocą myszy obracać w różne strony, co pozwala lepiej zrozumieć jej budowę. Tego typu bezpośrednią manipulację obiektytami nie jest możliwa na statycznych stronach WWW, ale w aplatach tak (aplet ten można znaleźć na stronie <http://jmol.sourceforge.net>).



Rysunek 1.1. Aplet Jmol

Kiedy aplaty pojawiły się na scenie, wywołały niemałe poruszenie. Wielu ludzi uważa, że to właśnie dzięki zaletom apletów Java zyskała tak dużą popularność. Jednak początkowe zaurocenie przemieniło się szybko w roczarowanie. Różne wersje przeglądarek Netscape i Internet Explorer działały z różnymi wersjami Javy. Niektóre z nich były przestarzałe. Ze względu na tę przykryą sytuację tworzenie aplatów przy wykorzystaniu najnowszych wersji Javy było coraz trudniejsze. Obecnie większość dynamicznych efektów na stronach internetowych jest realizowana za pomocą JavaScriptu i technologii Flash. Java natomiast stała się najpopularniejszym językiem do tworzenia aplikacji działających po stronie serwera, które generują strony internetowe i stanowią ich zaplecze logiczne.

1.4. Krótka historia Javy

Podrozdział ten krótko opisuje historię ewolucji Javy. Informacje tu zawarte pochodzą z różnych źródeł (najważniejsze z nich to wywiad z twórcami Javy opublikowany w internetowym magazynie „SunWorld” w 1995 roku).

Historia Javy sięga 1991 roku, kiedy zespół inżynierów z firmy Sun, którego przewodniczącymi byli Patrick Naughton i (wszędobylski geniusz komputerowy) James Gosling, piastujący jedno z najwyższych stanowisk w firmie o nazwie **Sun Fellow**, postanowił zaprojektować niewielki język programowania nadający się do użytku w takich urządzeniach konsumenckich jak tuner telewizji kablowej. Jako że urządzenia te nie dysponują dużą mocą ani pamięcią, założono, że język musi być bardzo niewielki i powinien generować zwięzły kod. Ponadto ze względu na fakt, że producenci mogą w swoich urządzeniach stosować różne procesory, język ten nie mógł być związany tylko z jedną architekturą. Projekt otrzymał kryptonim **Green**.

Chęć utworzenia kompaktowego i niezależnego od platformy kodu doprowadziła zespół do wskrzeszenia modelu znanego z implementacji Pascala z wczesnych dni istnienia komputerów osobistych. Pionierski projekt przenośnego języka generującego kod pośredni dla hipotetycznej maszyny należał do Niklusa Wirtha — wynalazcy Pascala (maszyny te nazywane są często **wirtualnymi**, stąd nazwa „wirtualna maszyna Javy”). Ten kod pośredni można było następnie uruchamiać na wszystkich urządzeniach, które miały odpowiedni interpreter. Inżynierowie skupieni wokół projektu Green także posłużyli się maszyną wirtualną, rozwiązując w ten sposób swój główny problem.

Jako że pracownicy firmy Sun obracali się w środowisku uniksowym, swój język oparli na C++, a nie na Pascalu. Stworzony przez nich język był obiektowy, a nie proceduralny. Jak jednak mówi w wywiadzie Gosling: „Przez cały czas język był tylko narzędziem, a nie celem”. Gosling zdecydował się nazwać swój język Oak (dąb), prawdopodobnie dlatego że lubił widok dębu stojącego za oknem jego biura w Sun. Później odkryto, że język programowania o tej nazwie już istniał, i zmieniono nazwę na Java. Okazało się to strzałem w dziesiątkę.

W 1992 roku inżynierowie skupieni wokół projektu Green przedstawili swoje pierwsze dzieło o nazwie *7. Był to niezwykle inteligentny pilot zdalnego sterowania (miał moc stacji SPARC zamkniętą w pudełku o wymiarach $15 \times 10 \times 10$ centymetrów). Niestety, nikt w firmie Sun nie był nim zainteresowany, przez co inżynierowie musieli znaleźć inny sposób na wypromowanie swojej technologii. Jednak żadna z typowych firm produkujących elektronikę użytkową nie wykazała zainteresowania. Następnym krokiem zespołu był udział w przetargu na utworzenie urządzenia TV Box obsługującego takie nowe usługi telewizji kablowej jak filmy na żądanie. Nie dostali jednak kontraktu (co zabawne, umowę podpisał ten sam Jim Clark, który założył firmę Netscape — firma ta miała duży wkład w sukces Javy).

Inżynierowie pracujący nad projektem Green (przechrzczonym na „First Person, Inc.”) spędzili cały rok 1993 i połowę 1994 na poszukiwaniu kupca dla ich technologii — nie znaleźli nikogo (Patrick Naughton, który był jednym z założycieli zespołu i zajmował się promocją jego produktów, twierdzi, że uzbierał 300 000 punktów Air Miles, próbując sprzedać ich technologię). Projekt First Person przestał istnieć w 1994 roku.

Podczas gdy w firmie Sun miały miejsce te wszystkie wydarzenia, sieć ogólnosłowiańska będąca częścią internetu cały czas się rozrastała. Kluczem do sieci jest przeglądarka, która interpretuje hipertekst i wyświetla wynik na ekranie monitora. W 1994 roku większość użytkowników internetu korzystała z niekomercyjnej przeglądarki o nazwie Mosaic, która powstała w 1993 roku w centrum komputerowym uniwersytetu Illinois (pracował nad nią między innymi Marc Andreessen, który był wtedy studentem tego uniwersytetu i dostawał 6,85 dolara za godzinę. Andreessen zdobył sławę i pieniądze jako jeden ze współzałożycieli i szef działu technologii firmy Netscape).

W wywiadzie dla „SunWorld” Gosling przyznał, że w połowie 1994 roku projektanci języka zdali sobie sprawę, iż „mogli stworzyć naprawdę dobrą przeglądarkę. Była to jedna z niewielu aplikacji klient-serwer należących do głównego nurtu, wymagającej tych dziwnych rzeczy, które zrobiliśmy, czyli niezależności od architektury, pracy w czasie rzeczywistym, niezawodności i bezpieczeństwa. W świecie stacji roboczych pojęcia te nie miały wielkiego znaczenia. Postanowiliśmy więc napisać przeglądarkę internetową”.

Budową przeglądarki, która przeobraziła się w przeglądarkę o nazwie HotJava, zajęli się Patrick Naughton i Jonathan Payne. Przeglądarkę HotJava naturalnie napisano w języku Java, ponieważ jej celem było zaprezentowanie ogromnych możliwości, które stwarzał ten język. Programiści pamiętali jednak też o czymś, co obecnie nazywamy appletami, i dodali możliwość uruchamiania kodu wbudowanego w strony internetowe. 23 maja 1995 roku owoc tej pracy, mającej na celu udowodnienie wartości Javy, ujrzał światło dzienne w magazynie „SunWorld”. Stał się on kamieniem węgielnym szalonej popularności Javy, która trwa do dzisiaj.

Pierwsze wydanie Javy firma Sun opublikowała na początku 1996 roku. Szybko zorientowano się, że Java 1.0 nie stanie się narzędziem wykorzystywanym do tworzenia poważnych aplikacji. Oczywiście można było za jej pomocą stworzyć nerwowo poruszający się tekst w obszarze roboczym przeglądarki, ale nie było już na przykład możliwości **drukowania**. Mówiąc szczerze, Java 1.0 nie była gotowa na wielkie rzeczy. W kolejnej wersji, Java 1.1, uzupełniono najbardziej oczywiste braki, znacznie ulepszono refleksję i dodano model zdarzeń dla programowania GUI. Jednak nadal możliwości były raczej ograniczone.

Wielkim wydarzeniem na konferencji JavaOne w 1998 roku było ogłoszenie, że niebawem pojawi się Java 1.2. Zastąpiono w niej dziecinne narzędzia do obróbki grafiki i tworzenia GUI wyrafinowanymi i skalowalnymi wersjami, które znacznie przybliżały spełnienie obietnicy: „Napisz raz, uruchamiaj wszędzie” w stosunku do poprzednich wersji. Trzy dni po jej wydaniu (!), w grudniu 1998 roku, dział marketingu firmy Sun zmienił nazwę Java 1.2 na bardziej chwytną Java 2 Standard Edition Software Development Kit Version 1.2.

Poza wydaniem standardowym opracowano jeszcze dwa inne: Micro Edition dla urządzeń takich jak telefony komórkowe oraz Enterprise Edition do przetwarzania po stronie serwera. Ta książka koncentruje się na wersji standardowej.

Kolejne wersje Java 1.3 i Java 1.4 to stopniowe ulepszenia w stosunku do początkowej wersji Java 2. Jednocześnie rozrastała się biblioteka standardowa, zwiększała się wydajność i oczywiście poprawiono wiele błędów. W tym samym czasie ucichła wrzawa wokół appletów i aplikacji działających po stronie klienta, a Java stała się najczęściej wybieraną platformą do tworzenia aplikacji działających po stronie serwera.

Pierwsza wersja Javy, w której wprowadzono znaczące zmiany w **języku programowania** Java w stosunku do wersji 1.1, miała numer 5 (pierwotnie był to numer 1.5, ale na konferencji JavaOne w 2004 roku podskoczył do piątki). Po wielu latach badań dodano typy sparametryzowane (ang. *generic types*), które można z grubsza porównać do szablonów w C++. Sztuka polegała na tym, aby przy dodawaniu tej funkcji nie zmieniać nic w maszynie wirtualnej. Niektóre z dodanych funkcji zostały zaczerpnięte z języka C#: pętla `for each`, możliwość automatycznej konwersji typów prostych na referencyjne i odwrotnie (ang. *autoboxing*) oraz metadane.

Wersja 6 (bez przyrostka .0) ujrzała świat pod koniec 2006 roku. Tym razem również nie wprowadzono żadnych zmian w języku, ale zastosowano wiele usprawnień związanych z wydajnością i rozszerzono bibliotekę.

W centrach danych zaczęto rzadziej korzystać ze specjalistycznego sprzętu serwerowego, przez co firma Sun Microsystems wpadła w tarapaty i w 2009 roku została wykupiona przez Oracle. Rozwój Javy został na dłuższy czas wstrzymany. Jednak w 2011 roku firma Oracle opublikowała kolejną wersję języka z drobnymi ulepszeniami o nazwie Java 7. Poważniejsze zmiany przełożono do wersji Java 8, której ukazanie się jest planowane na 2013 rok.

Tabela 1.1 przedstawia ewolucję języka Java i jego biblioteki. Jak widać, rozmiar interfejsu programistycznego (API) rósł w rekordowym tempie.

Tabela 1.1. Ewolucja języka Java

Wersja	Rok	Nowe funkcje języka	Liczba klas i interfejsów
1.0	1996	Powstanie języka	211
1.1	1997	Klasy wewnętrzne	477
1.2	1998	Brak	1524
1.3	2000	Brak	1840
1.4	2002	Asercje	2723
5.0	2004	Klasy sparametryzowane, pętla <code>for each</code> , atrybuty o zmiennej liczbie argumentów (varargs), enumeracje, statyczny import	3279
6	2006	Brak	3793
7	2011	Instrukcja <code>switch</code> z łańcuchami, operator diamentowy, literały binarne, udoskonalenia mechanizmu obsługi wyjątków	4024

1.5. Główne nieporozumienia dotyczące Javy

Kończymy ten rozdział kilkoma uwagami związanymi z nieporozumieniami dotyczącymi Javy.

Java jest rozszerzeniem języka HTML.

Java jest językiem programowania, a HTML to sposób opisu struktury stron internetowych. Nie mają ze sobą nic wspólnego z wyjątkiem tego, że w HTML są dostępne rozszerzenia umożliwiające wstawianie appletów Javy na strony HTML.

Używam XML, więc nie potrzebuję Javy.

Java to język programowania, a XML jest sposobem opisu danych. Dane w formacie XML można przetwarzanie za pomocą wielu języków programowania, ale API Javy ma doskonałe narzędzia do przetwarzania XML. Ponadto wiele znaczących narzędzi XML jest zaimplementowanych w Javie. Więcej informacji na ten temat znajduje się w drugiej części tej książki.

Java jest łatwa do nauki.

Żaden język programowania o tak dużych możliwościach jak Java nie jest łatwy do nauczenia się. Trzeba odróżnić, jak łatwo napisać program do zabawy i jak trudno napisać poważną aplikację. Warto zauważyc, że opisowi języka Java w tej książce poświęcone zostały tylko cztery rozdziały. Pozostałe rozdziały w obu częściach opisują sposoby wykorzystania tego języka przy użyciu **bibliotek** Javy. Biblioteki te zawierają tysiące klas i interfejsów oraz dziesiątki tysięcy funkcji. Na szczęście nie trzeba ich wszystkich znać, ale trzeba zapoznać się z zaskakującą dużą ich liczbą, aby móc zrobić cokolwiek dobrego w Javie.

Java stanie się uniwersalnym językiem programowania dla wszystkich platform.

Teoretycznie jest to możliwe i praktycznie wszyscy poza firmą Microsoft chcieliby, aby tak się stało. Jednak wiele aplikacji, które bardzo dobrze działają na komputerach biurowych, nie działały prawidłowo na innych urządzeniach lub w przeglądarkach. Ponadto aplikacje te zostały napisane w taki sposób, aby maksymalnie wykorzystać możliwości procesora i natywnej biblioteki interfejsowej, oraz zostały już przeniesione na wszystkie najważniejsze platformy. Do tego typu aplikacji należą procesory tekstu, edytory zdjęć i przeglądarki internetowe. Większość z nich została napisana w językach C i C++, a ponowne napisanie ich w Javie nie przyniosłoby użytkownikom żadnych korzyści.

Java jest tylko kolejnym językiem programowania.

Java to bardzo przyjazny język programowania. Większość programistów przedkłada go nad C, C++ czy C#. Jednak przyjaznych języków programowania jest bardzo dużo, a nigdy nie zyskały one dużej popularności, podczas gdy języki zawierające powszechnie znane wady, jak C++ i Visual Basic, cieszą się ogromnym powodzeniem.

Dlaczego? Powodzenie języka programowania jest bardziej uzależnione od przydatności jego **systemu wsparcia** niż od elegancji składni. Czy istnieją przydatne i wygodne standardowe biblioteki funkcji, które chcesz zaimplementować? Czy są firmy produkujące doskonałe środowiska programistyczne i wspomagające znajdywanie błędów? Czy język i jego narzędzia integrują się z resztą infrastruktury komputerowej? Sukcesu Javy należy upatrywać w tym, że można w niej robić z łatwością takie rzeczy, które kiedyś były bardzo trudne — można tu zaliczyć na przykład wielowątkowość i programowanie sieciowe. Dzięki zmniejszeniu liczby błędów wynikających z używania wskaźników programiści wydają się bardziej produktywni, co jest oczywiście zaletą, ale nie stanowi źródła sukcesu Javy.

Po pojawieniu się języka C# Java idzie w zapomnienie.

Język C# przejął wiele dobrych pomysłów od Javy, jak czystość języka programowania, maszyna wirtualna czy automatyczne usuwanie nieużytków. Jednak z niewiadomych przyczyn wielu dobrych rzeczy w tym języku brakuje, zwłaszcza zabezpieczeń i niezależności od platformy. Dla tych, którzy są związani z systemem Windows, język C# wydaje się dobrym wyborem. Sądząc jednak po ogłoszeniach dotyczących oferowanej pracy, Java nadal stanowi wybór większości deweloperów.

Java jest własnością jednej firmy i dlatego należy jej unikać.

Po utworzeniu Javy firma Sun Microsystems udzielała darmowych licencji na Javę dystrybutorom i użytkownikom końcowym. Mimo że firma ta sprawowała pełną kontrolę nad Javą, w proces tworzenia nowych wersji języka i projektowania nowych bibliotek zostało zaangażowanych wiele firm. Kod źródłowy maszyny wirtualnej i bibliotek był zawsze ogólnodostępny, ale tylko do wglądu. Nie można go było modyfikować ani ponownie rozdzielać. Do tej pory Java była zamknięta, ale dobrze się sprawowała.

Sytuacja uległa radykalnej zmianie w 2007 roku, kiedy firma Sun ogłosiła, że przyszłe wersje Javy będą dostępne na licencji GPL, tej samej otwartej licencji, na której dostępny jest system Linux. Firma Oracle zobowiązała się pozostawić Javę otwartą. Jest tylko jedna rysa na tej powierzchni — patenty. Na mocy licencji GPL każdy może używać Javy i ją modyfikować, ale dotyczy to tylko zastosowań desktopowych i serwerowych. Jeśli ktoś chce używać Javy w układach wbudowanych, musi mieć inną licencję, za którą najpewniej będzie musiał zapłacić. Jednak patenty te w ciągu najbliższych kilku lat wygasną i wówczas Java będzie całkowicie darmowa.

Java jest językiem interpretowanym, a więc jest zbyt powolna do poważnych zastosowań.

Na początku Java była interpretowana. Obecnie poza platformami skali mikro (jak telefony komórkowe) maszyna wirtualna Javy wykorzystuje kompilator czasu rzeczywistego. Najczęściej używane części kodu działają tak szybko, jakby były napisane w C++, a w niektórych przypadkach nawet szybciej.

Java ma pewien narzut w stosunku do C++. Uruchamianie maszyny wirtualnej zajmuje sporo czasu, poza tym GUI w Javie są wolniejsze od ich natywnych odpowiedników, ponieważ zostały przystosowane do pracy na różnych platformach.

Przez wiele lat ludzie skarzyli się, że Java jest powolna. Jednak dzisiejsze komputery są dużo szybsze od tych, które były dostępne w czasach, gdy zaczęto się na to skarzyć. Powolny program w Javie i tak działa nieco szybciej niż niewiarygodnie szybkie programy napisane kilka lat temu w C++. Obecnie te skargi brzmią jak echo dawnych czasów, a niektórzy zaczęli dla odmiany narzekać na to, że interfejsy użytkownika w Javie są brzydsze niż wolniejsze.

Wszystkie programy pisane w Javie działają na stronach internetowych.

Wszystkie **aplety** Javy działają wewnątrz przeglądarki. Takie są z założenia aplety — są to programy napisane w Javie, które działają wewnątrz okna przeglądarki. Jednak większość programów pisanych w Javie to samodzielne aplikacje, działające poza przeglądarką internetową. W rzeczywistości wiele programów w Javie działa po stronie serwera i generuje kod stron WWW.

Programy w Javie są zagrożeniem bezpieczeństwa.

Na początku istnienia Javy opublikowano kilka raportów opisujących błędy w systemie zabezpieczeń Javy. Większość z nich dotyczyło implementacji Javy w określonej przeglądarce. Badacze potraktowali zadanie znalezienia wyrw w murze obronnym Javy i złamania siły oraz wyrafinowania modelu zabezpieczeń appletów jako wyzwanie. Znalezione przez nich techniczne usterki zostały szybko naprawione i według naszej wiedzy żadne rzeczywiste systemy nie zostały jeszcze złamane. Spójrzmy na to z innej perspektywy — w systemie Windows miliony wirusów atakujących pliki wykonywalne i makra programu Word spowodowały bardzo dużo szkód, ale wywołyły niewiele krytyki na temat słabości atakowanej platformy. Także mechanizm ActiveX w przeglądarce Internet Explorer może być dobrą pozywką dla nadużyć, ale jest to tak oczywiste, że z nudów niewielu badaczy publikuje swoje odkrycia na ten temat.

Niektórzy administratorzy systemu wyłączyli nawet Javę w przeglądarkach firmowych, a pozostały możliwość pobierania plików wykonywalnych i dokumentów programu Word, które są o wiele bardziej groźne. Nawet 15 lat od momentu powstania Java jest znacznie bardziej bezpieczna niż jakakolwiek inna powszechnie używana platforma.

Język JavaScript to uproszczona wersja Javy.

JavaScript, skryptowy język stosowany na stronach internetowych, został opracowany przez firmę Netscape i początkowo jego nazwa brzmiała LiveScript. Składnią JavaScript przypomina Javę, ale poza tym języki te nie mają ze sobą nic wspólnego (oczywiście wyłączając nazwę). Podzbiór JavaScriptu jest opublikowany jako standard ECMA-262. Język ten jest ściślej zintegrowany z przeglądarkami niż applety Javy. Programy w JavaScriptie mogą wpływać na wygląd wyświetlanych dokumentów, podczas gdy applety mogą sterować zachowaniem tylko ograniczonej części okna.

Dzięki Javie mogę wymienić mój komputer na terminal internetowy za 1500 złotych.

Po pierwszym wydaniu Javy niektórzy ludzie gotowi byliby postawić duże pieniądze, że tak się stanie. Od pierwszego wydania tej książki utrzymujemy, że twierdzenie, iż użytkownicy domowi zechcą zastąpić wszechstronne komputery ograniczonymi urządzeniami pozbawionymi pamięci, jest absurdalne. Wyposażony w Javę komputer sieciowy mógłby być prawdopodobnym rozwiązaniem umożliwiającym wdrożenie strategii jednokrotnego ustawienia opcji konfiguracyjnych bez potrzeby późniejszego wracania do nich (ang. *zero administration initiative*). Umożliwiłoby to zmniejszenie kosztów ponoszonych na utrzymanie komputerów w firmach, ale jak na razie nie widać wielkiego ruchu w tym kierunku. W aktualnie dostępnych tabletach Java nie jest wykorzystywana.

2

Środowisko programistyczne Javy

W tym rozdziale:

- Instalacja oprogramowania Java Development Kit
- Wybór środowiska programistycznego
- Korzystanie z narzędzi wiersza poleceń
- Praca w zintegrowanym środowisku programistycznym
- Uruchamianie aplikacji graficznej
- Budowa i uruchamianie appletów

W tym rozdziale nauczysz się instalować oprogramowanie Java Development Kit (JDK) oraz kompilować i uruchamiać różne typy programów: programy konsolowe, aplikacje graficzne i applety. Narzędzia JDK są uruchamiane za pomocą poleceń wpisywanych w oknie interpretera poleceń. Wielu programistów woli jednak wygodę pracy w zintegrowanym środowisku programistycznym. Opisaliśmy jedno dostępne bezpłatnie środowisko, w którym można kompilować i uruchamiać programy napisane w Javie. Mimo niewątpliwych zalet, takich jak łatwość nauki, takie środowiska pochłaniają bardzo dużo zasobów i bywają nieporęczne przy pisaniu niewielkich aplikacji. Prezentujemy zatem kompromisowe rozwiązanie w postaci edytora tekstowego, który umożliwia uruchamianie kompilatora Javy i programów napisanych w tym języku. Jeśli opanujesz techniki opisywane w tym rozdziale i wybierzesz odpowiednie dla siebie narzędzia programistyczne, możesz przejść do rozdziału 3., od którego zaczyna się opis języka programowania Java.

2.1. Instalacja oprogramowania

Java Development Kit

Najpełniejsze i najnowsze wersje pakietu JDK dla systemów Linux, Mac OS X, Solaris i Windows są dostępne na stronach firmy Oracle. Istnieją też wersje w różnych fazach rozwoju dla wielu innych platform, ale podlegają one licencjom i są rozprowadzane przez firmy produkujące te platformy.

2.1.1. Pobieranie pakietu JDK

Aby pobrać odpowiedni dla siebie pakiet Java Development Kit, trzeba przejść na stronę internetową www.oracle.com/technetwork/java/javase/ i rozszyfrować całe mnóstwo żargonowych pojęć (zobacz zestawienie w tabeli 2.1).

Tabela 2.1. Pojęcia specyficzne dla Javy

Nazwa	Akronim	Objaśnienie
Java Development Kit	JDK	Oprogramowanie dla programistów, którzy chcą pisać programy w Javie.
Java Runtime Environment	JRE	Oprogramowanie dla klientów, którzy chcą uruchamiać programy napisane w Javie.
Standard Edition	SE	Platforma Javy do użytku na komputerach biurkowych i w przypadku prostych zastosowań serwerowych.
Enterprise Edition	EE	Platforma Javy przeznaczona do skomplikowanych zastosowań serwerowych.
Micro Edition	ME	Platforma Javy znajdująca zastosowanie w telefonach komórkowych i innych małych urządzeniach.
Java 2	J2	Przestarzały termin określający wersje Javy od 1998 do 2006 roku.
Software Development Kit	SDK	Przestarzały termin, który oznaczał pakiet JDK od 1998 do 2006 roku.
Update	u	Termin określający wydanie z poprawionym błędem.
NetBeans	—	Zintegrowane środowisko programistyczne firmy Oracle.

Znamy już skrót JDK oznaczający Java Development Kit. Żeby nie było za łatwo, informujemy, że wersje od 1.2 do 1.4 tego pakietu miały nazwę Java SDK (ang. *Software Development Kit*). Wciąż można znaleźć odwołania do tej starej nazwy. Jest też Java Runtime Environment (JRE), czyli oprogramowanie zawierające maszynę wirtualną bez kompilatora. Jako programiści nie jesteśmy tym zainteresowani. Ten program jest przeznaczony dla użytkowników końcowych, którym kompilator nie jest potrzebny.

Kolej na wszedobylski termin Java SE. Jest to Java Standard Edition, w odróżnieniu od Java EE (ang. *Enterprise Edition*) i Java ME (ang. *Micro Edition*).

Czasami można też spotkać termin Java 2, który został ukuty w 1998 roku przez dział marketingu w firmie Sun. Uważano, że zwiększenie numeru wersji o ułamek nie oddaje w pełni postępu, jakiego dokonano w JDK 1.2. Jednak — jako że później zmieniono zdanie — zdecydowano się zachować numer 1.2. Kolejne wydania miały numery 1.3, 1.4 i 5.0. Zmieniono jednak nazwę **platformy** z Java na Java 2. W ten sposób powstał pakiet Java 2 Standard Edition Software Development Kit Version 5.0, czyli J2SE SDK 5.0.

Inżynierowie mieli problemy z połapaniem się w tych nazwach, ale na szczęście w 2006 roku zwyciężył rozsądek. Bezużyteczny człon Java 2 został usunięty, a aktualna wersja Java Standard Edition została nazwana Java SE 6. Nadal można sporadycznie spotkać odwołania do wersji 1.5 i 1.6, ale są one synonimami wersji 5 i 6.

Na zakończenie trzeba dodać, że mniejsze zmiany wprowadzane w celu naprawienia usterek przez firmę Oracle nazywane są aktualizacjami (ang. *updates*). Na przykład pierwsza aktualizacja pakietu programistycznego dla Java SE 7 ma oficjalną nazwę JDK 7u1, ale jej wewnętrzny numer wersji to 1.7.0_01. Aktualizacje nie muszą być instalowane na bazie starszych wersji — zawierają najnowsze wersje całego pakietu JDK.

Czasami firma Oracle udostępnia paczki zawierające zarówno pakiet Java Development Kit, jak i zintegrowane środowisko programistyczne. Jego nazwy kilkakrotnie się zmieniały; do tej pory można się było spotkać z Forte, Sun ONE Studio, Sun Java Studio i NetBeans. Trudno zgadnąć, jaką nazwę nadadzą mu kolejnym razem nadgorliwcy z działu marketingu. Na razie zalecamy więc zainstalowanie jedynie pakietu JDK. Jeśli zdecydujesz się później na używanie środowiska firmy Sun, pobierz je ze strony <http://netbeans.org>.



W trakcie instalacji sugerowany jest domyślny katalog na pliki, w którego nazwie znajduje się numer wersji pakietu JDK, np. `jdk1.7.0`. Na pierwszy rzut oka wydaje się to niepotrzebną komplikacją, ale spodobało nam się to z tego względu, że w ten sposób o wiele łatwiej można zainstalować nowe wydanie JDK do testowania.

Użytkownikom systemu Windows odradzamy akceptację domyślnej ścieżki ze spacjami w nazwie, jak `C:\Program Files\jdk1.7.0`. Najlepiej usunąć z tej ścieżki część *Program Files*.

W tej książce katalog instalacji określamy mianem *jdk*. Kiedy na przykład piszemy o katalogu *jdk/bin*, mamy na myśli ścieżkę typu `/usr/local/jdk1.7.0/bin` lub `C:\jdk1.7.0\bin`.

2.1.2. Ustawianie ścieżki dostępu

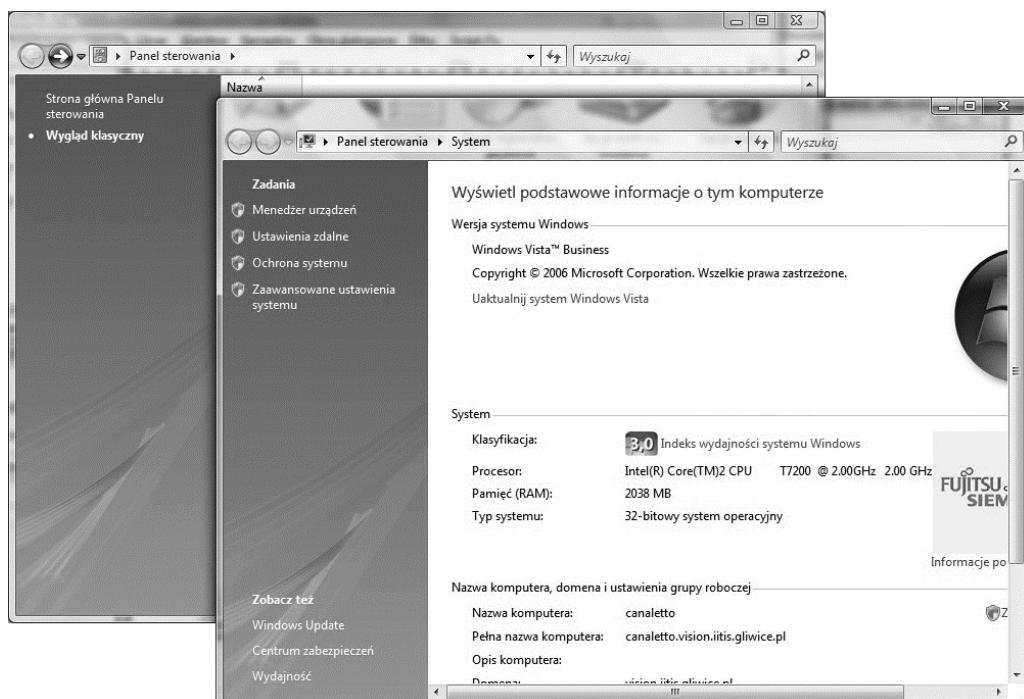
Po instalacji pakietu JDK trzeba wykonać jeszcze jedną czynność: dodać katalog *jdk/bin* do ścieżki dostępu, czyli listy katalogów, które przemierza system operacyjny w poszukiwaniu plików wykonywalnych. Postępowanie w tym przypadku jest inne w każdym systemie operacyjnym.

- W systemie Unix (wliczając Linux, Mac OS X i Solaris) sposób edycji ścieżki dostępu zależy od używanej powłoki. Użytkownicy powłoki Bourne Again (która jest domyślna dla systemu Linux) muszą na końcu pliku `~/.bashrc` lub `~/.bash.profile` dodać następujący wiersz:

```
export PATH=~/jdk/bin:$PATH
```

- W systemie Windows należy zalogować się jako administrator. Przejdź do *Panelu sterowania*, przełącz na widok klasyczny i kliknij dwukrotnie ikonę *System*. W systemie Windows XP od razu otworzy się okno *Właściwości systemu*. W systemie Windows Vista i Windows 7 należy kliknąć pozycję *Zaawansowane ustawienia systemu* (zobacz rysunek 2.1). W oknie dialogowym *Właściwości systemu* kliknij kartę *Zaawansowane*, a następnie przycisk *Zmienne środowiskowe*. W oknie *Zmienne środowiskowe* znajdź zmienną o nazwie *Path*. Kliknij przycisk *Edytuj* (zobacz rysunek 2.2). Dodaj katalog *jdk\bin* na początku ścieżki i wpis ten oddziel od reszty wpisów średnikiem, jak poniżej:

jdk\bin;inne wpisy



Rysunek 2.1. Otwieranie okna właściwości systemu w systemie Windows Vista

Słowo *jdk* należy zastąpić ścieżką do katalogu instalacyjnego Javy, np. *c:\jdk1.7.0_02*. Jeśli zainstalowałeś Javę w folderze *Program Files*, całą ścieżkę wpisz w cudzysłowie: "*c:\Program Files\jdk1.7.0_02\bin*"; *inne wpisy*.

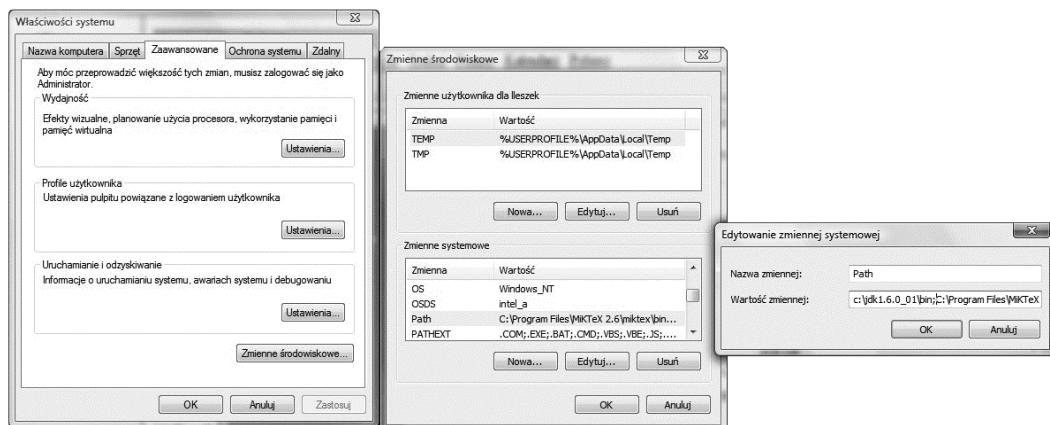
Zapisz ustawienia. Każde nowe okno konsoli będzie wykorzystywać prawidłową ścieżkę.

Oto jak można sprawdzić, czy powyższe czynności zostały wykonane prawidłowo: otwórz okno konsoli i wpisz poniższe polecenie:

```
javac -version
```

a następnie naciśnij klawisz *Enter*. Na ekranie powinien pojawić się następujący tekst:

```
javac 1.7.0_02
```



Rysunek 2.2. Ustawianie zmiennej środowiskowej Path w systemie Windows Vista

Jeśli zamiast tego ukaże się komunikat typu `javac`: polecenie nie zostało znalezione lub Nazwa nie jest rozpoznawana jako polecenie wewnętrzne lub zewnętrzne, program wykonywalny lub plik wsadowy, trzeba wrócić do początku i dokładnie sprawdzić swoją instalację.



Aby otworzyć okno konsoli w systemie Windows, należy postępować zgodnie z następującymi wskazówkami: w systemie Windows XP kliknij opcję *Uruchom* w menu *Start* i wpisz polecenie `cmd`. W systemach Windows Vista i 7 wystarczy wpisać `cmd` w polu *Rozpocznij wyszukiwanie* w menu *Start*. Następnie naciśnij klawisz *Enter*.

Osobom, które nigdy nie miały do czynienia z oknem konsoli, zalecamy zapoznanie się z kursem objaśniającym podstawy korzystania z tego narzędzia dostępnym pod adresem <http://www.horstmann.com/bigj/help/windows/tutorial.html>.

2.1.3. Instalacja bibliotek i dokumentacji

Kod źródłowy bibliotek w pakiecie JDK jest dostępny w postaci skompresowanego pliku o nazwie `src.zip`. Oczywiście, aby uzyskać dostęp do tego źródła, trzeba niniejszy plik rozpakować. Gorąco do tego zachęcamy. Wystarczy wykonać następujące czynności:

1. Upewnij się, że po zainstalowaniu pakietu JDK katalog `jdk/bin` znajduje się w ścieżce dostępu.
2. Otwórz okno konsoli.
3. Przejdz do katalogu `jdk` (np. `cd /usr/local/jdk1.7.0` lub `cd c:\jdk1.7.0`).
4. Utwórz podkatalog `src`.

```
mkdir src
cd src
```

5. Wykonaj polecenie:

```
jar xvf ../src.zip
```

albo `jar xvf ..\src.zip` w systemie Windows.



Plik *src.zip* zawiera kod źródłowy wszystkich bibliotek publicznych. Więcej źródeł (dla kompilatora, maszyny wirtualnej, metod rodzimych i prywatnych klas pomocniczych) można znaleźć na stronie <http://jdk7.java.net>.

Dokumentacja znajduje się w oddzielnym, skompresowanym pliku. Można ją pobrać ze strony www.oracle.com/technetwork/java/javase/downloads. Sprowadza się to do wykonania kilku prostych czynności:

1. Upewnij się, że po zainstalowaniu pakietu JDK katalog *jdk/bin* znajduje się w ścieżce dostępu.
2. Pobierz plik archiwum zip zawierający dokumentację i zapisz go w katalogu *jdk*. Plik ten ma nazwę *jdk-wersja-apidocs.zip*, gdzie *wersja* to numer wersji, np. 7.
3. Otwórz okno konsoli.
4. Przejdź do katalogu *jdk*.
5. Wykonaj poniższe polecenie:

```
jar xvf jdk-wersja-apidocs.zip
```

gdzie *wersja* to odpowiedni numer wersji.

2.1.4. Instalacja przykładowych programów

Należy też zainstalować przykładowe programy z tej książki. Można je pobrać ze strony <http://horstmann.com/corejava>. Programy te znajdują się w pliku archiwum ZIP o nazwie *corejava.zip*. Należy je wypakować do oddzielnego katalogu — polecamy utworzenie katalogu o nazwie *JavaPodstawy*. Oto zestawienie wymaganych czynności:

1. Upewnij się, że po zainstalowaniu pakietu JDK katalog *jdk/bin* znajduje się w ścieżce dostępu.
2. Utwórz katalog o nazwie *JavaPodstawy*.
3. Pobierz z internetu i zapisz w tym katalogu plik *corejava.zip*.
4. Otwórz okno konsoli.
5. Przejdź do katalogu *JavaPodstawy*.
6. Wykonaj poniższe polecenie:

```
jar xvf corejava.zip
```

2.1.5. Drzewo katalogów Javy

Zagłębiając się w Javę, zechcesz sporadycznie zatrzymać się do plików źródłowych. Będziesz też oczywiście zmuszony do pracy z dokumentacją techniczną. Rysunek 2.3 obrazuje drzewo katalogów JDK.

Struktura katalogów	Opis
<i>jdk</i>	Może to być jedna z kilku nazw, np. <i>jdk1.7.0_02</i>
<i>bin</i>	Kompilator i inne narzędzia
<i>demo</i>	Przykładowe programy
<i>docs</i>	Dokumentacja biblioteki w formacie HTML (po rozpakowaniu pliku <i>j2sdkwersja-doc.zip</i>)
<i>include</i>	Pliki potrzebne do kompilacji metod rodzimych (zobacz drugi tom)
<i>jre</i>	Pliki środowiska uruchomieniowego Javy
<i>lib</i>	Pliki biblioteki
<i>src</i>	Źródła biblioteki (po rozpakowaniu pliku <i>src.zip</i>)

Rysunek 2.3. Drzewo katalogów Javy

Dla uczących się Javy najważniejsze są katalogi *docs* i *src*. Katalog *docs* zawiera dokumentację biblioteki Javy w formacie HTML. Można ją przeglądać za pomocą dowolnej przeglądarki internetowej, jak chociażby Firefox.



W swojej przeglądarce dodaj do ulubionych stronę *docs/api/index.html*. W trakcie poznawania platformy Java będziesz do niej często zaglądać.

Katalog *src* zawiera kod źródłowy publicznych bibliotek Javy. W miarę zdobywania wiedzy na temat Javy być może będziesz chciał uzyskać więcej informacji, niż dostarcza niniejsza książka i dokumentacja. W takiej sytuacji najlepszym miejscem do rozpoczęcia poszukiwań jest kod źródłowy Javy. Świadomość, że zawsze można zajrzeć do kodu źródłowego, aby sprawdzić, jak faktycznie działa dana funkcja biblioteczna, ma w dużym stopniu działanie uspokajające. Jeśli chcemy na przykład zbadać wnętrze klasy *System*, możemy zajrzeć do pliku *src/java/Lang/System.java*.

2.2. Wybór środowiska programistycznego

Osoby, które do tej pory pracowały w środowisku Microsoft Visual Studio, są przyzwyczajone do środowiska z wbudowanym edytorem tekstu i menu, udostępniającymi opcje kompilacji i uruchamiania programu oraz zintegrowanego debugera. Podstawowy pakiet JDK nie oferuje nawet zbliżonych możliwości. **Wszystko** robi się poprzez wpisywanie odpowiednich poleceń w oknie konsoli. Brzmi strasznie, niemniej jest to nieodzowna umiejętność programisty. Po zainstalowaniu Javy może być konieczne usunięcie usterek dotyczących tej instalacji, a dopiero potem można zainstalować środowisko programistyczne. Ponadto dzięki wykonaniu podstawowych czynności we własnym zakresie można lepiej zrozumieć, co środowisko programistyczne „robi” za naszymi plecami.

Po opanowaniu podstawowych czynności kompilowania i uruchamiania programów w Javie zechcemy jednak przenieść się do profesjonalnego środowiska programistycznego. W ciągu

ostatnich lat środowiska te stały się tak wygodne i wszechstronne, że nie ma sensu mączyć się bez nich. Dwa z nich zasługują na wyróżnienie: *Eclipse* i *NetBeans*. Oba są dostępne bezpłatnie. W tym rozdziale opisujemy, jak rozpocząć pracę w środowisku *Eclipse*, jako że jest ono nieco lepsze od *NetBeans*, choć to drugie szybko dogania swojego konkurenta. Oczywiście do pracy z tą książką można użyć także dowolnego innego środowiska.

Kiedyś do pisania prostych programów polecaliśmy edytory tekstowe, takie jak *Emacs*, *JEdit* czy *TextPad*. Ze względu na fakt, że zintegrowane środowiska są już bardzo szybkie i wygodne, teraz zalecamy używanie właśnie nich.

Podsumowując, naszym zdaniem każdy powinien znać podstawy obsługi narzędzi JDK, a po ich opanowaniu przejść na zintegrowane środowisko programistyczne.

2.3. Używanie narzędzi wiersza poleceń

Zaczniemy od mocnego uderzenia: komplikacji i uruchomienia programu w Javie w wierszu poleceń.

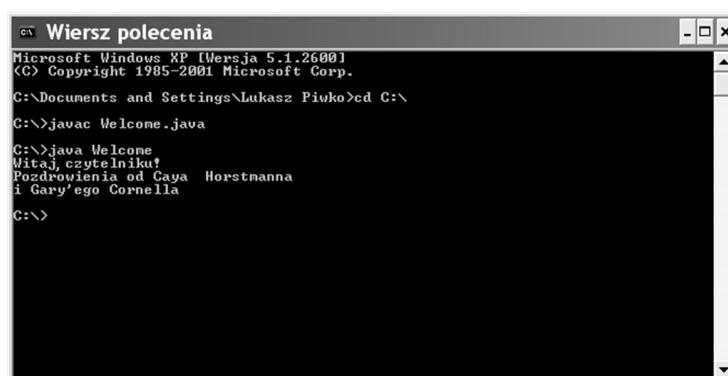
1. Otwórz okno konsoli.
2. Przejdź do katalogu *JavaPodstawy/t1/r02/Welcome* (katalog *JavaPodstawy* to ten, w którym zapisaliśmy kod źródłowy programów prezentowanych w tej książce, o czym była mowa w podrozdziale 2.1.4, „Instalacja przykładowych programów”).
3. Wpisz następujące polecenia:

```
javac Welcome.java  
java Welcome
```

Wynik w oknie konsoli powinien być taki jak na rysunku 2.4.

Rysunek 2.4.

Kompilacja i uruchamianie programu *Welcome.java*



```
Wiersz poleceń  
Microsoft Windows XP [Wersja 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
C:\Documents and Settings\Lukasz Piukko>cd C:\  
C:\>javac Welcome.java  
C:\>java Welcome  
Witaj, czytelniku!  
Pozdrowienia od Caya Horstmann  
i Gary'ego Cornellia  
C:\>
```

Gratulacje! Właśnie skompilowaliśmy i uruchomiliśmy nasz pierwszy program w Javie.

Co się wydarzyło? Program o nazwie *javac* to kompilator Javy. Skompilował plik o nazwie *Welcome.java* na plik *Welcome.class*. Program *java* uruchamia wirtualną maszynę Javy. Wykonuje kod bajtowy zapisany w pliku klasy przez kompilator.



Jeśli w poniższym wierszu pojawił się komunikat o błędzie:

```
for (String g : greeting)
```

to znaczy, że używasz bardzo starej wersji kompilatora Javy. Użytkownicy starszych wersji Javy muszą zastąpić powyższą pętlę następującą:

```
for (int i = 0; i < greeting.length; i++)
    System.out.println(greeting[i]);
```

Program *Welcome* jest niezwykle prosty. Wyświetla tylko wiadomość w konsoli. Jego kod źródłowy przedstawia listing 2.1 (sposób działania tego kodu opisujemy w następnym rozdziale).

Listing 2.1. Welcome/Welcome.java

```
/*
 * Program ten wyświetla wiadomość powitalną od autorów.
 * @version 1.20 2004-02-28
 * @author Cay Horstmann
 */
public class Welcome
{
    public static void main(String[] args)
    {
        String[] greeting = new String[3];
        greeting[0] = "Witaj, czytelniku!";
        greeting[1] = "Pozdrowienia od Caya Horstmanna";
        greeting[2] = "i Gary'ego Cornell'a";

        for (String g : greeting)
            System.out.println(g);
    }
}
```

2.3.1. Rozwiązywanie problemów

W erze wizualnych środowisk programistycznych wielu programistów nie potrafi uruchamiać programów w oknie konsoli. Wiele rzeczy może nie pójść zgodnie z planem, co prowadzi do rozczarowań.

Zwróć uwagę na następujące rzeczy:

- Jeśli wpisujesz program ręcznie, zwracaj baczną uwagę na wielkość liter. W szczególności pamiętaj, że nazwa klasy to *Welcome*, a nie *welcome* lub *WELCOME*.
- Kompilator wymaga **nazwy pliku** (*Welcome.java*). Aby uruchomić program, należy podać **nazwę klasy** (*Welcome*) bez rozszerzenia *.java* lub *.class*.

- Jeśli pojawił się komunikat typu złe polecenie lub zła nazwa pliku bądź javac: polecenie nie zostało znalezione, należy wrócić i dokładnie sprawdzić swoją instalację, zwłaszcza ustawienia ścieżki dostępu.
- Jeśli kompilator *javac* zgłosi błąd typu cannot read: *Welcome.java*, należy sprawdzić, czy plik ten znajduje się w odpowiednim katalogu.

W systemie Unix należy sprawdzić wielkość liter w nazwie pliku *Welcome.java*. W systemie Windows należy użyć polecenia *dir* w oknie konsoli, **nie** w Eksploratorze. Niektóre edytory tekstu (zwłaszcza Notatnik) dodają na końcu nazwy każdego pliku rozszerzenie *.txt*. Jeśli program *Welcome.java* był edytowany za pomocą Notatnika, to został zapisany jako *Welcome.java.txt*. Przy domyślnych ustawieniach systemowych Eksplorator działa w zasadzie z Notatnikiem i ukrywa rozszerzenie *.txt*, ponieważ należy ono do znanych typów plików. W takim przypadku trzeba zmienić nazwę pliku za pomocą polecenia *ren* lub zapisać go ponownie, ujmując nazwę w cudzysłowy: "*Welcome.java*".

- Jeśli po uruchomieniu programu pojawi się komunikat o błędzie *java.lang.→NoClassDefFoundError*, dokładnie sprawdź nazwę klasy, która sprawia problemy.

Jeśli błąd dotyczy nazwy *welcome* (pisanej małą literą), należy jeszcze raz wydać polecenie *java Welcome* z wielką literą *W*. Jak zawsze w Javie wielkość liter ma znaczenie.

Jeśli błąd dotyczy *Welcome/java*, oznacza to, że przypadkowo wpisano polecenie *java Welcome.java*. Należy jeszcze raz wpisać polecenie *java Welcome*.

- Jeśli po wpisaniu polecenia *java Welcome* maszyna wirtualna nie może znaleźć klasy *Welcome*, należy sprawdzić, czy ktoś nie ustał w systemie zmiennej środowiskowej **CLASSPATH** (ustawianie na poziomie globalnym nie jest dobrym pomysłem, ale niektóre słabej jakości instalatory oprogramowania w systemie Windows tak właśnie robią). Zmienną tę można usunąć tymczasowo w oknie konsoli za pomocą polecenia:

```
set CLASSPATH=
```

To polecenie działa w systemach Windows oraz Unix i Linux z powłoką C. W systemach Unix i Linux z powłoką Bourne/bash należy użyć polecenia:

```
export CLASSPATH=
```



W doskonałym kursie znajdującym się pod adresem <http://docs.oracle.com/javase/tutorial/getStarted/> można znaleźć opisy znacznie większej liczby pułapek, w które wpadają początkujący programiści.

2.4. Praca w zintegrowanym środowisku programistycznym

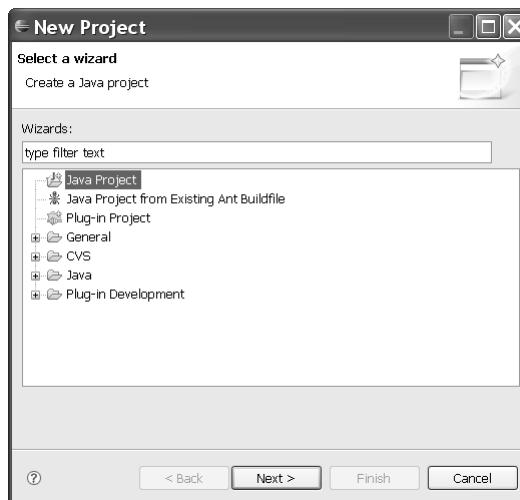
W tym podrozdziale nauczysz się kompilować programy w zintegrowanym środowisku programistycznym o nazwie Eclipse, które można nieodpłatnie pobrać ze strony <http://eclipse.org>. Program ten został napisany w Javie, ale ze względu na użyty w nim niestandardową bibliotekę okien nie jest on tak przenośny jak sama Java. Niemniej istnieją jego wersje dla systemów Linux, Mac OS X, Solaris i Windows.

Dostępnych jest jeszcze kilka innych IDE, ale Eclipse cieszy się obecnie największą popularnością. Oto podstawowe kroki poczatkującego:

1. Po uruchomieniu programu Eclipse kliknij opcję *File/New Project*.
2. W oknie kreatora wybierz pozycję *Java Project* (zobacz rysunek 2.5). Te zrzuty zostały zrobione w wersji 3.3 Eclipse. Nie jest to jednak wymóg i możesz używać innej wersji tego środowiska.

Rysunek 2.5.

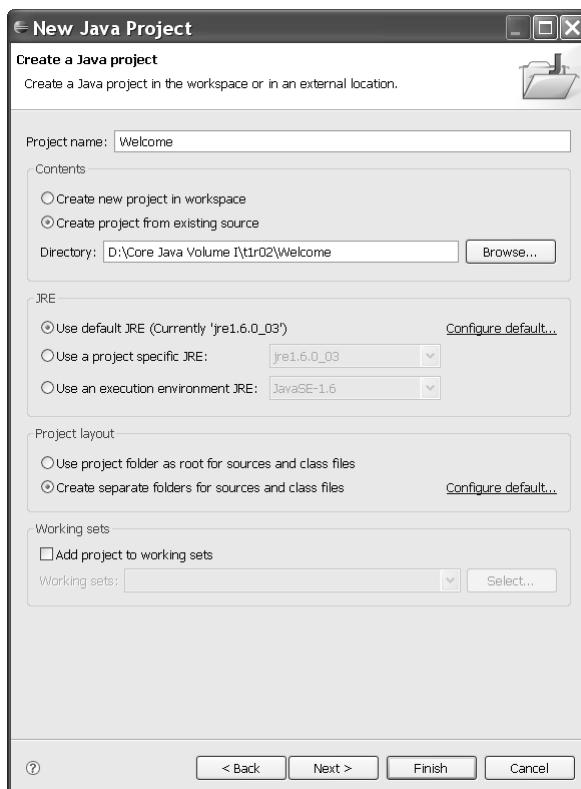
Okno dialogowe
New Project
w Eclipse



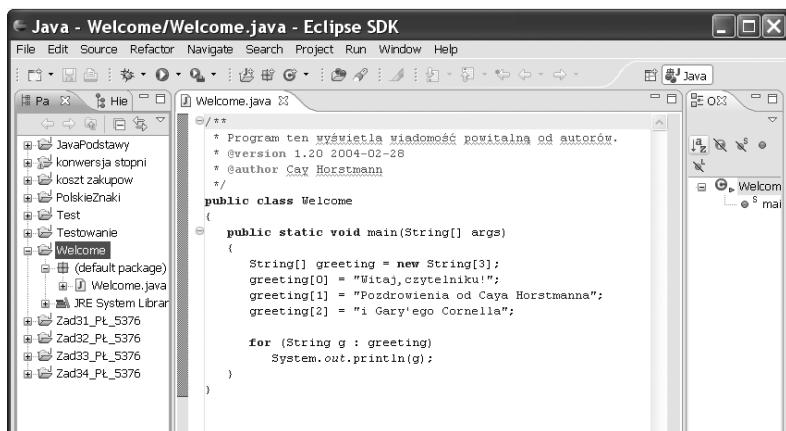
3. Kliknij przycisk *Next*. Wprowadź nazwę projektu *Welcome* i wpisz pełną ścieżkę katalogu, który zawiera plik *Welcome.java* (zobacz rysunek 2.6).
4. Zaznacz opcję *Create project from existing source* (utwórz projekt z istniejącego źródła).
5. Kliknij przycisk *Finish* (zakończ), aby utworzyć projekt.
6. Aby otworzyć projekt, kliknij znajdujący się w lewym panelu obok okna projektu symbol trójkąta. Następnie kliknij symbol trójkąta znajdujący się obok napisu *Default package* (domyślny pakiet). Kliknij dwukrotnie plik o nazwie *Welcome.java*. Powinno się pojawić okno z kodem źródłowym programu (zobacz rysunek 2.7).

Rysunek 2.6.

Konfiguracja projektu w Eclipse

**Rysunek 2.7.**

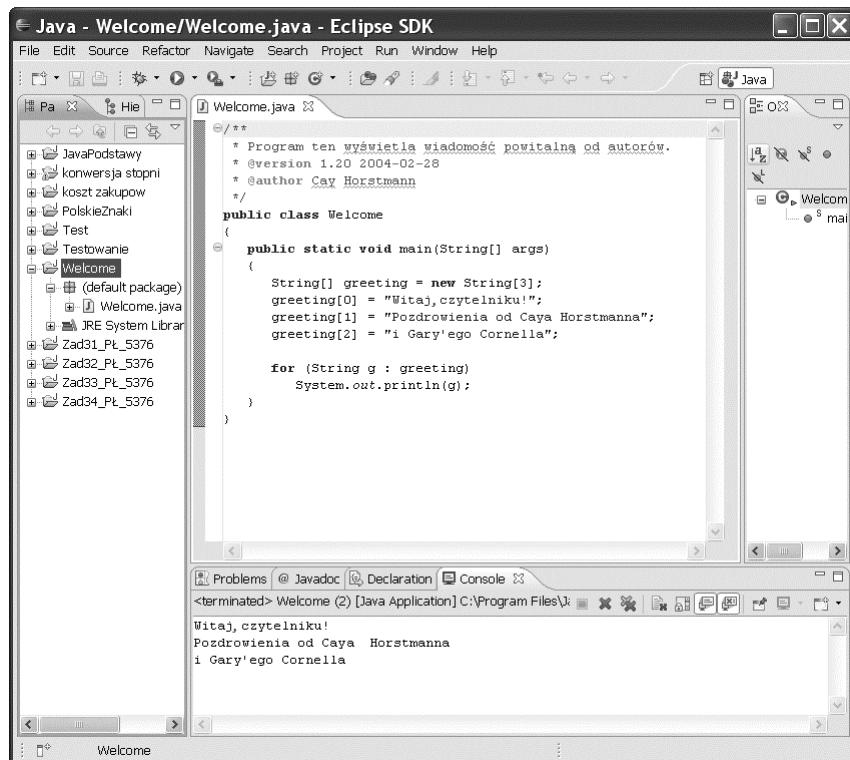
Edycja kodu źródłowego w Eclipse



7. W lewym panelu kliknij prawym przyciskiem myszy nazwę projektu (*Welcome*). Kliknij opcję *Run/Run As/Java Application*. Okno z wynikami programu znajduje się na dole okna Eclipse (zobacz rysunek 2.8).

Rysunek 2.8.

Uruchamianie programu w środowisku Eclipse



2.4.1. Znajdowanie błędów kompilacji

Ten program nie powinien zawierać żadnych literówek ani innych błędów (przecież to tylko kilka wierszy kodu). Założmy jednak na nasze potrzeby, że czasami zdarzy nam się zrobić w kodzie literówkę (a nawet błąd składniowy). Zobaczmy, co się stanie. Celowo zepsujemy nasz program, zmieniając wielką literę S w słowie String na małą:

```
String[] greeting = new string[3];
```

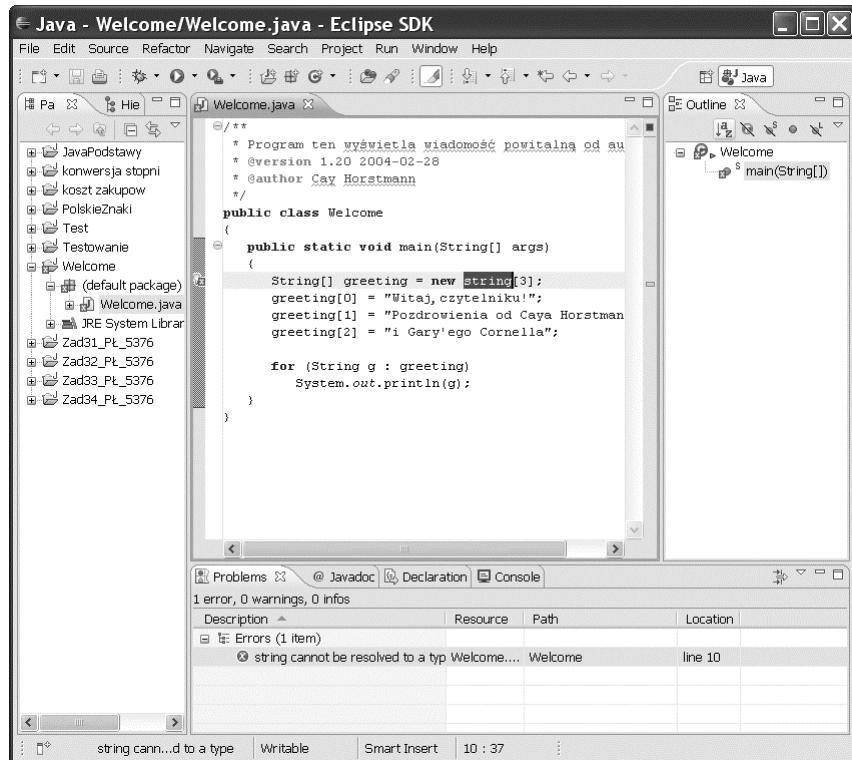
Ponownie uruchamiamy kompilator. Pojawia się komunikat o błędzie dotyczący nieznanego typu o nazwie `string` (zobacz rysunek 2.9). Wystarczy kliknąć komunikat błędu, aby kursor został przeniesiony do odpowiadającego mu wiersza w oknie edycji. Możemy poprawić nasz błąd. Takie działanie środowiska umożliwia szybkie poprawianie tego typu błędów.



Błędy w Eclipse są często oznaczane ikoną żarówki. Aby przejrzeć listę sugerowanych rozwiązań problemu, należy tę ikonę kliknąć.

Te krótkie instrukcje powinny wystarczyć na początek pracy w środowisku zintegrowanym. Opis debugera Eclipse znajduje się w rozdziale 11.

Rysunek 2.9.
Komunikaty
o błędach
w Eclipse



2.5. Uruchamianie aplikacji graficznej

Program powitalny nie należy do najbardziej ekscytujących. Kolejna aplikacja graficzna. Ten program jest przeglądarką plików graficznych, która ładuje i wyświetla obrazy. Najpierw skompilujmy i uruchomimy ją z poziomu wiersza poleceń.

- 1 Otwórz okno konsoli.
- 2 Przejdź do katalogu `JavaPodstawy/t1/r02/ImageViewer`.
- 3 Wpisz poniższe polecenia:

```
javac ImageViewer.java
java ImageViewer
```

Pojawi się nowe okno aplikacji `ImageViewer` (zobacz rysunek 2.10).

Następnie kliknij opcję *Plik/Otwórz*, aby otworzyć plik (kilka plików do otwarcia znajduje się w katalogu z klasą). Aby zamknąć program, należy kliknąć pozycję *Zakończ* w menu *Plik* albo krzyżek w prawym górnym rogu okna przeglądarki.

Rzućmy okiem na kod źródłowy tego programu. Jest on znacznie dłuższy od poprzedniego, ale biorąc pod uwagę to, ile wierszy kodu trzeba było napisać w językach C i C++, aby

Rysunek 2.10.

Działanie aplikacji ImageViewer



stworzyć podobną aplikację, trzeba przyznać, że nie jest zbyt skomplikowany. Oczywiście łatwo taki program napisać (a raczej przeciągnąć i upuścić) w Visual Basicu. JDK nie umożliwia wizualnego budowania interfejsów, a więc cały kod widoczny na listingu 2.2 trzeba napisać ręcznie. Pisaniem takich programów graficznych zajmiemy się w rozdziałach od 7. do 9.

Listing 2.2. ImageViewer/ImageViewer.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
 * Program do przeglądania obrazów.
 * @version 1.22 2007-05-21
 * @author Cay Horstmann
 */
public class ImageViewer
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ImageViewerFrame();
                frame.setTitle("ImageViewer");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka z etykietą wyświetlającą obraz.
     */
    class ImageViewerFrame extends JFrame
    {
```

```
private JLabel label;
private JFileChooser chooser;
private static final int DEFAULT_WIDTH = 300;
private static final int DEFAULT_HEIGHT = 400;
public ImageViewerFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    // Użycie etykiety do wyświetlenia obrazów.
    label = new JLabel();
    add(label);

    // Dodawanie opcji wyboru obrazu.
    chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    // Pasek menu.
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);

    JMenu menu = new JMenu("Plik");
    menuBar.add(menu);

    JMenuItem openItem = new JMenuItem("Otwórz");
    menu.add(openItem);
    openItem.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            // Wyświetlenie okna dialogowego wyboru pliku.
            int result = chooser.showOpenDialog(null);

            // Jeśli plik został wybrany, ustawiamy go jako ikonę etykiety.
            if (result == JFileChooser.APPROVE_OPTION)
            {
                String name = chooser.getSelectedFile().getPath();
                label.setIcon(new ImageIcon(name));
            }
        }
    });
}

JMenuItem exitItem = new JMenuItem("Zakończ");
menu.add(exitItem);
exitItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        System.exit(0);
    }
});
}
```

2.6. Tworzenie i uruchamianie apletów

Pierwsze dwa programy zaprezentowane w książce są samodzielnymi **aplikacjami**. Jak jednak pamiętamy z poprzedniego rozdziału, najwięcej szumu wokół Javy spowodowała możliwość uruchamiania apletów w oknie przeglądarki internetowej. Pokażemy, jak się kompiluje i uruchamia **aplety** z poziomu wiersza poleceń. Następnie załadujemy nasz aplet do dostępnej w JDK przeglądarki apletów. Na zakończenie wyświetlimy go w przeglądarce internetowej.

Otwórz okno konsoli, przejdź do katalogu *JavaPodstawy/t1/r02/WelcomeApplet* i wpisz następujące polecenia:

```
javac WelcomeApplet.java
appletviewer WelcomeApplet.html
```

Rysunek 2.11 przedstawia okno przeglądarki apletów.

Rysunek 2.11.

Aplet
WelcomeApplet
w oknie
przeglądarki
apletów



Pierwsze polecenie już znamy — służy do uruchamiania kompilatora Javy. W tym przypadku skompilowaliśmy plik z kodem źródłowym o nazwie *WelcomeApplet.java* na plik z kodem bajtowym o nazwie *WelcomeApplet.class*.

Jednak tym razem nie uruchamiamy programu *java*, tylko program *appletviewer*. Jest to specjalne narzędzie dostępne w pakiecie JDK, które umożliwia szybkie przetestowanie apletu. Program ten przyjmuje na wejściu pliki HTML, a nie pliki klas Javy. Zawartość pliku *WelcomeApplet.html* przedstawia listing 2.3.

Listing 2.3. WelcomeApplet.html

```
<html>
  <head>
    <title>WelcomeApplet</title>
  </head>
  <body>
    <hr/>
    <p>
      Ten aplet pochodzi z książki
      <a href="http://www.horstmann.com/corejava.html">Java. Podstawy</a>,
      której autorami są <em>Cay Horstmann</em> i <em>Gary Cornell</em>,
      wydanej przez wydawnictwo Helion.
    </p>
    <applet code="WelcomeApplet.class" width="400" height="200">
      <param name="greeting" value ="Witaj, czytelniku!"/>
    </applet>
  </body>
</html>
```

```

</applet>
<hr/>
<p><a href="WelcomeApplet.java">źródło</a></p>
</body>
</html>

```

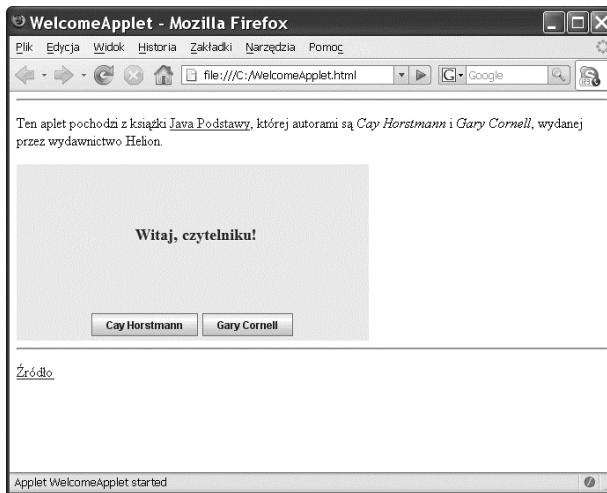
Osoby znające HTML rozpoznają kilka standardowych elementów tego języka oraz znacznik `applet`, nakazujący przeglądarce apletów, aby załadowała applet, którego kod znajduje się w pliku `WelcomeApplet.class`. Przeglądarka apletów bierze pod uwagę tylko znacznik `applet`.

Oczywiście aplety są przeznaczone do uruchamiania w przeglądarkach internetowych, ale nie stety w wielu z nich obsługa tych obiektów jest standardowo wyłączona. Informacje dotyczące konfiguracji najpopularniejszych przeglądarki do obsługi Javy można znaleźć pod adresem http://java.com/en/download/help/enable_browser.xml. Mając odpowiednio skonfigurowaną przeglądarkę, możesz w niej uruchomić nasz applet.

- 1 Uruchom przeglądarkę.
- 2 Z menu *Plik* wybierz opcję *Otwórz* (lub coś w tym rodzaju).
- 3 Przejdź do katalogu *JavaPodstawy/t1/r02/WelcomeApplet*. Załóż plik o nazwie *WelcomeApplet.html*.
- 4 Przeglądarka wyświetli applet wraz z dodatkowym tekstem. Rezultat będzie podobny do tego na rysunku 2.12.

Rysunek 2.12.

Działanie apletu *WelcomeApplet* w przeglądarce internetowej



Jak widać, aplikacja ta jest zdolna do interakcji z internetem. Kliknięcie przycisku *Cay Horstmann* powoduje przejście do strony internetowej Caya Horstmanna. Kliknięcie przycisku *Gary Cornell* powoduje wyświetlenie okna wysyłania poczty e-mail z adresem Gary'ego Cornella wstawionym w polu adresata.

Zauważ, że żaden z tych przycisków nie działa w przeglądarce apletów. Nie ma ona możliwości wysyłania poczty e-mail ani wyświetlania stron internetowych, więc ignoruje nasze

żądania w tym zakresie. Przeglądarka appletów nadaje się do testowania appletów w izolacji, ale do sprawdzenia, jak applety współpracują z przeglądarką internetową i internetem, potrzebna jest przeglądarka internetowa.



Aplet można także uruchamiać w edytorze lub zintegrowanym środowisku programistycznym. W Eclipse należy w tym celu użyć opcji *Run/Run As/Java Applet* (uruchom/uruchom jako/applet Java).

Kod appletu przedstawia listing 2.4. Na razie wystarczy rzucić tylko na niego okiem. Do pisania appletów wróćmy w rozdziale 10.

Listing 2.4. WelcomeApplet.java

```

import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;

/**
 * Aplet ten wyświetla powitanie autorów.
 * @version 1.22 2007-04-08
 * @author Cay Horstmann
 */
public class WelcomeApplet extends JApplet
{
    public void init()
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                setLayout(new BorderLayout());

                JLabel label = new JLabel(getParameter("greeting"), SwingConstants.
                CENTER);
                label.setFont(new Font("Serif", Font.BOLD, 18));
                add(label, BorderLayout.CENTER);

                JPanel panel = new JPanel();

                JButton cayButton = new JButton("Cay Horstmann");
                cayButton.addActionListener(makeAction("http://www.horstmann.com"));
                panel.add(cayButton);

                JButton garyButton = new JButton("Gary Cornell");
                garyButton.addActionListener(makeAction("mailto:gary_cornell@.
                apress.com"));
                panel.add(garyButton);

                add(panel, BorderLayout.SOUTH);
            }
        });
    }
}

```

```
private ActionListener makeAction(final String urlString)
{
    return new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            try
            {
                getAppletContext().showDocument(new URL(urlString));
            }
            catch (MalformedURLException e)
            {
                e.printStackTrace();
            }
        }
    };
}
```

3

Podstawowe elementy języka Java

W tym rozdziale:

- Prosty program w Javie
- Komentarze
- Typy danych
- Zmienne
- Operatory
- Łańcuchy
- Wejście i wyjście
- Kontrola przepływu sterowania
- Wielkie liczby
- Tablice

Do tego rozdziału należy przejść dopiero wtedy, gdy z powodzeniem zainstalowało się pakiet JDK i uruchomiło przykładowe programy z rozdziału 2. Ponieważ czas zacząć programowanie, w rozdziale tym zapoznasz się z podstawowymi pojęciami programistycznymi Javy, takimi jak typy danych, instrukcje warunkowe i pętle.

Niestety, napisanie w Javie programu z graficznym interfejsem użytkownika nie jest łatwe — wymaga dużej wiedzy na temat sposobów tworzenia okien, dodawania do nich pól tekstowych, przycisków, które reagują na zawartość tych pól itd. Jako że opis technik pisania programów GUI w Javie znacznie wykracza poza nasz cel przedstawienia podstaw programowania w tym języku, przykładowe programy w tym rozdziale są bardzo proste. Komunikują się za pośrednictwem okna konsoli, a ich przeznaczeniem jest tylko ilustracja omawianych pojęć.

Doświadczeni programiści języka C++ mogą tylko przejrzeć ten rozdział, koncentrując się na ramkach opisujących różnice pomiędzy Javą a C++. Programiści innych języków, jak Visual Basic, będą znać większość omawianych pojęć, ale odkryją, że składnia Javy jest całkiem inna od znanych im języków. Te osoby powinny bardzo uważnie przeczytać ten rozdział.

3.1. Prosty program w Javie

Przyjrzyjmy się uważnie najprostszemu programowi w Javie, jaki można napisać — takiemu, który tylko wyświetla komunikat w oknie konsoli:

```
public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println("Nie powiemy „Witaj, świecie!”");
    }
}
```

Warto poświęcić trochę czasu i nauczyć się tego fragmentu na pamięć, ponieważ wszystkie aplikacje są oparte na tym schemacie. Przede wszystkim w Javie **wielkość liter ma znaczenie**. Jeśli w programie będzie literówka (jak np. słowo `Main` zamiast `main`), to program nie zadziała.

Przestudiujmy powyższy kod wiersz po wierszu. Słowo kluczowe `public` nosi nazwę **modyfikatora dostępu** (ang. *access modifier*). Określa ono, jaki rodzaj dostępu do tego kodu mają inne części programu. Więcej informacji na temat modyfikatorów dostępu zawiąźmy w rozdziale 5. Słowo kluczowe `class` przypomina, że wszystko w Javie należy do jakiejś klasy. Ponieważ klasami bardziej szczegółowo zajmujemy się w kolejnym rozdziale, na razie będziemy je traktować jako zbiory mechanizmów programu, które są odpowiedzialne za jego działanie. Jak pisaliśmy w rozdziale 1., klasy to bloki, z których składają się wszystkie aplikacje i apłyty Javy. **Wszystko** w programie w Javie musi się znajdować wewnątrz jakiejś klasy.

Po słowie kluczowym `class` znajduje się nazwa klasy. Reguły dotyczące tworzenia nazw klas w Javie są dosyć liberalne. Nazwa klasy musi się zaczynać od litery, po której może znajdować się kombinacja dowolnych znaków i cyfr. Nie ma w zasadzie ograniczeń, jeśli chodzi o długość. Nie można stosować słów zarezerwowanych Javy (np. `public` lub `class`) — lista wszystkich słów zarezerwowanych znajduje się w dodatku.

Zgodnie ze standardową konwencją nazewnictwą (której przykładem jest nazwa klasy `FirstSample`) nazwy klas powinny się składać z rzeczowników pisanych wielką literą. Jeśli nazwa klasy składa się z kilku słów, każde z nich powinno być napisane wielką literą; notacja polegająca na stosowaniu wielkich liter wewnątrz nazw jest czasami nazywana notacją wielbłędzą (ang. *camel case* lub *CamelCase*).

Plik zawierający kod źródłowy musi mieć taką samą nazwę jak klasa publiczna oraz rozszerzenie `.java`. W związku z tym nasz przykładowy kod powinien zostać zapisany w pliku o nazwie `FirstSample.java` (przypominam, że wielkość liter ma znaczenie także tutaj — nie można napisać `firsstsample.java`).

Jeśli plik ma prawidłową nazwę i nie ma żadnych literówek w kodzie źródłowym, w wyniku jego komplikacji powstanie plik zawierający kod bajtowy tej klasy. Kompilator automatycznie nada skompilowanemu plikowi nazwę *FirstSample.class* i zapisze go w tym samym katalogu, w którym znajduje się plik źródłowy. Program uruchamiamy za pomocą następującego polecenia (nie zapomnij o pominięciu rozszerzenia *.class*):

```
java FirstSample
```

Po uruchomieniu program ten wyświetla w konsoli łańcuch *Nie powiemy „Witaj, świecie!”*.

Polecenie:

```
java NazwaKlasy
```

zastosowane do skompilowanego programu powoduje, że wirtualna maszyna Javy zaczyna wykonywanie od kodu zawartego w metodzie *main* wskazanej klasy (terminem „metoda” określa się to, co w innych językach jest funkcją). W związku z tym metoda *main* **musi** się znajdować w pliku źródłowym klasy, którą chcemy uruchomić. Można oczywiście dodać własne metody do klasy i wywoływać je w metodzie *main*. Pisanie metod omawiamy w następnym rozdziale.



Zgodnie ze specyfikacją języka Java (oficjalnym dokumentem opisującym ten język, który można pobrać lub przeglądać na stronie <http://docs.oracle.com/javase/specs>) metoda *main* musi być publiczna (*public*).

Jednak niektóre wersje maszyny wirtualnej Javy uruchamiały programy w Javie, których metoda *main* nie była publiczna. Pewien programista zgłosił ten błąd. Aby się o tym przekonać, wejdź na stronę <http://bugs.sun.com/bugdatabase/index.jsp> i wpisz numer identyfikacyjny błędu 4252539. Błąd ten został oznaczony jako zamknięty i nie do naprawy (ang. *closed, will not be fixed*). Jeden z inżynierów pracujących w firmie Sun wyjaśnił (<http://docs.oracle.com/javase/specs/jvms/se7/html>), że specyfikacja maszyny wirtualnej Javy nie wymaga, aby metoda *main* była publiczna, w związku z czym „naprawienie tego błędu może spowodować problemy”. Na szczęście sięgnięto po rozum do głowy i od wersji 1.4 Java SE metoda *main* jest publiczna.

Ta historia pozwala zwrócić uwagę na kilka rzeczy. Z jednej strony rozczarowuje nas sytuacja, że osoby odpowiadające za jakość są przepracowane i nie zawsze dysponują wystarczającą wiedzą specjalistyczną z zakresu najbardziej zaawansowanych zagadnień związanych z Javą. Przez to nie zawsze podejmują trafne decyzje. Z drugiej strony trzeba zauważyc, że firma Sun zamieszcza raporty o błędach na stronie internetowej, aby każdy mógł je zweryfikować. Taki spis błędów jest bardzo wartościowym źródłem wiedzy dla programistów. Można nawet głosować na swój ulubiony błąd. Błędy o największej liczbie głosów mają największą szansę poprawienia w kolejnej wersji pakietu JDK.

Zauważ, że w kodzie źródłowym użyto nawiasów klamrowych. W Javie, podobnie jak w C i C++, klamry oddzielają poszczególne części (zazwyczaj nazywane **blokami**) kodu programu. Kod każdej metody w Javie musi się zaczynać od otwierającej klamry {, a kończyć zamkającą klamrą }.

Styl stosowania nawiasów klamrowych wywołał niepotrzebną dyskusję. My stosujemy styl polegający na umieszczaniu dopełniających się klamer w tej samej kolumnie. Jako że kompilator ignoruje białe znaki, można stosować dowolny styl nawiasów klamrowych. Więcej do powiedzenia na temat stosowania klamer będziemy mieli przy okazji omawiania pętli.

Na razie nie będziemy się zajmować znaczeniem słów `static void` — traktuj je jako coś, czego potrzebujesz do kompilacji programu w Javie. Po rozdziale czwartym przestanie to być tajemnicą. Teraz trzeba tylko zapamiętać, że każdy program napisany w Javie musi zawierać metodę `main` zadeklarowaną w następujący sposób:

```
public class NazwaKlasy
{
    public static void main(String[] args)
    {
        instrukcje programu
    }
}
```

C++ Programiści języka C++ doskonale znają pojęcie „klasa”. Klasy w Javie są pod wieloma względami podobne do tych w C++, ale jest też kilka różnic, o których nie można zapominać. Na przykład w Javie **wszystkie** funkcje są metodami jakiejś klasy (w standardowej terminologii są one nazywane metodami, a nie funkcjami składowymi). W związku z tym w Javie konieczna jest obecność klasy zawierającej metodę `main`. Programiści C++ pewnie znają też **statyczne funkcje składowe**. Są to funkcje zdefiniowane wewnątrz klasy, które nie wykonują żadnych działań na obiektach. Metoda `main` w Javie jest zawsze statyczna. W końcu słowo kluczowe `void`, podobnie jak w C i C++, oznacza, że metoda nie zwraca wartości. W przeciwieństwie do języka C i C++ metoda `main` w Javie nie zwraca żadnego kodu wyjścia (ang. `exit code`) do systemu operacyjnego. Jeśli metoda `main` zakończy działanie w normalny sposób, program ma kod wyjścia 0, który oznacza pomyślne zakończenie. Aby zakończyć działanie programu innym kodem wyjścia, należy użyć metody `System.exit`.

Teraz kierujemy naszą uwagę na poniższy fragment:

```
{
    System.out.println("Nie powiemy „Witaj, świecie!”");
}
```

Klamry oznaczają początek i koniec **ciała** metody. Ta metoda zawiera tylko jedną instrukcję. Podobnie jak w większości języków programowania, instrukcje Javy można traktować jako zdania tego języka. Każda instrukcja musi być zakończona średnikiem. Przede wszystkim należy pamiętać, że znak powrotu karetki nie oznacza końca instrukcji, dzięki czemu mogą one obejmować nawet kilka wierszy.

W treści metody `main` znajduje się instrukcja wysyłająca jeden wiersz tekstu do konsoli.

W tym przypadku użyliśmy obiektu `System.out` i wywołaliśmy na jego rzecz metodę `println`. Zwróć uwagę na kropki zastosowane w wywołaniu metody. Ogólna składnia stosowana w Javie do wywołania jej odpowiedników funkcji jest następująca:

```
obiekt.metoda(parametry)
```

W tym przypadku wywoaliśmy metodę `println` i przekazaliśmy jej argument w postaci łańcucha. Metoda ta wyświetla zawartość parametru w konsoli. Następnie kończy wiersz wyjściowy, dzięki czemu każde wywołanie metody `println` wyświetla dane w oddzielnym wierszu. Zwróć uwagę, że w Javie, podobnie jak w C i C++, łańcuchy należy ujmować w cudzysłowy — więcej informacji na temat łańcuchów znajduje się w dalszej części tego rozdziału.

Metody w Javie, podobnie jak funkcje w innych językach programowania, przyjmują zero, jeden lub więcej **parametrów** (często nazywanych **argumentami**). Nawet jeśli metoda nie przyjmuje żadnych parametrów, nie można pominąć stojących po jej nazwie nawiasów. Na przykład metoda `println` bez żadnych argumentów drukuje pusty wiersz. Wywołuje się ją następująco:

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



Na rzecz obiektu `System.out` można także wywoływać metodę `print`, która nie dodaje do danych wyjściowych znaku nowego wiersza. Na przykład wywołanie `System.out.print("Witaj")` drukuje napis *Witaj* bez znaku nowego wiersza. Kolejne dane zostaną umieszczone bezpośrednio po słowie *Witaj*.

3.2. Komentarze

Komentarze w Javie, podobnie jak w większości języków programowania, nie są uwzględniane w programie wykonywalnym. Można zatem stosować je w dowolnej ilości bez obawy, że nadmiernie zwiększą rozmiary kodu. W Javie są trzy rodzaje komentarzy. Najczęściej stosowana metoda polega na użyciu znaków `//`. Ten rodzaj komentarza obejmuje obszar od znaków `//` do końca wiersza, w którym się znajdują.

```
System.out.println("Nie powiemy „Witaj, świecie!”"); // Czy to nie słodkie?
```

Dłuższe komentarze można tworzyć poprzez zastosowanie znaków `//` w wielu wierszach lub użycie komentarza w stylu `/* */`. W ten sposób w komentarzu można ująć cały blok treści programu.

Wreszcie, trzeci rodzaj komentarza służy do automatycznego generowania dokumentacji. Ten rodzaj komentarza zaczyna się znakami `/**` i kończy `*/`. Jego zastosowanie przedstawia listing 3.1.Więcej informacji na temat tego rodzaju komentarzy i automatycznego generowania dokumentacji znajduje się w rozdziale 4.

Listing 3.1. FirstSample.java

```
/**  
 * Jest to pierwszy przykładowy program w rozdziale 3.  
 * @version 1.01 1997-03-22  
 * @author Gary Cornell  
 */  
public class FirstSample  
{  
    public static void main(String[] args)  
    {
```

```

        System.out.println("Nie powiemy „Witaj, świecie!”");
    }
}

```



Komentarzy /* */ nie można zagnieździć. Oznacza to, że nie można dezaktywować fragmentu kodu programu, otaczając go po prostu znakami /* i */, ponieważ kod ten może zawierać znaki */.

3.3. Typy danych

Java jest językiem o **ścisłej kontroli typów**. Oznacza to, że każda zmienna musi mieć określony typ. W Javie istnieje osiem **podstawowych typów**. Cztery z nich reprezentują liczby całkowite, dwa — liczby rzeczywiste, jeden o nazwie char zarezerwowano dla znaków reprezentowanych przez kody liczbowe należące do systemu Unicode (patrz punkt 3.3.3), zaś ostatni jest logiczny (boolean) — przyjmuje on tylko dwie wartości: true albo false.



W Javie dostępny jest pakiet do obliczeń arytmetycznych na liczbach o dużej precyzyj. Jednak tak zwane „duże liczby” (ang. *big numbers*) są **obiektami**, a nie nowym typem w Javie. Sposób posługiwania się nimi został opisany w dalszej części tego rozdziału.

3.3.1. Typy całkowite

Typy całkowite to liczby pozbawione części ułamkowej. Zaliczają się do nich także wartości ujemne. Wszystkie cztery dostępne w Javie typy całkowite przedstawia tabela 3.1.

Tabela 3.1. Typy całkowite Javy

Typ	Liczba bajtów	Zakres (z uwzględnieniem wartości brzegowych)
int	4	od -2 147 483 648 do 2 147 483 647 (nieco ponad 2 miliardy)
short	2	od -32 768 do 32 767
long	8	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
byte	1	od -128 do 127

Do większości zastosowań najlepiej nadaje się typ int. Aby zapisać liczbę mieszkańców naszej planety, trzeba użyć typu long. Typy byte i short są używane do specjalnych zadań, jak niskopoziomowa praca nad plikami lub duże tablice, kiedy pamięć jest na wagę złota.

Zakres wartości typów całkowitych nie zależy od urządzenia, na którym uruchamiany jest kod Javy. Eliminuje to główny problem programisty, który chce przenieść swój program

z jednej platformy na inną lub nawet z jednego systemu operacyjnego do innego na tej samej platformie. W odróżnieniu od Javy, języki C i C++ używają najbardziej efektywnego typu całkowitego dla każdego procesora. W wyniku tego program prawidłowo działający na procesorze 32-bitowym może powodować błąd przekroczenia zakresu liczby całkowitej na procesorze 16-bitowym. Jako że programy w Javie muszą działać prawidłowo na wszystkich urządzeniach, zakresy wartości różnych typów są stałe.

Duże liczby całkowite (typu `long`) są opatrzone modyfikatorem `L` lub `l` (na przykład `4000000000L`). Liczby w formacie szesnastkowym mają przedrostek `0x` (na przykład `0xCAFE`). Liczby w formacie ósemkowym poprzedza przedrostek `0`. Na przykład liczba `010` w zapisie ósemkowym to `8` w zapisie dziesiętnym. Oczywiście zapis ten może wprowadzać w błąd, w związku z czym odradzamy jego stosowanie.

W Java 7 wprowadzono dodatkowo możliwość zapisu liczb w formacie binarnym, do czego służy przedrostek `0b`. Przykładowo `0b1001` to inaczej `9`. Ponadto również od tej wersji języka można w literałach liczbowych stosować znaki podkreślenia, np. `1_000_000` (albo `0b1111_0100_0010_0100_0000`) — milion. Znaki te mają za zadanie ułatwić czytanie kodu ludziom. Kompilator Javy je usuwa.



W językach C i C++ typ `int` to liczba całkowita, której rozmiar zależy od urządzenia docelowego. W procesorach 16-bitowych, jak 8086, typ `int` zajmuje 2 bajty pamięci. W procesorach 32-bitowych, jak Sun SPARC, są to wartości czterobajtowe. W przypadku procesorów Intel Pentium rozmiar typu `int` zależy od systemu operacyjnego: w DOS-ie i Windows 3.1 typ `int` zajmuje 2 bajty pamięci. W programach dla systemu Windows działających w trybie 32-bitowym typ `int` zajmuje 4 bajty. W Javie wszystkie typy numeryczne są niezależne od platformy.

Zauważ, że w Javie nie ma typu `unsigned`.

3.3.2. Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe służą do przechowywania liczb z częścią ułamkową. Dwa dostępne w Javie typy zmiennoprzecinkowe przedstawia tabela 3.2.

Tablica 3.2. Typy zmiennoprzecinkowe

Typ	Liczba bajtów	Zakres
<code>float</code>	4	około $\pm 3,40282347E+38F$ (6 – 7 znaczących cyfr dziesiętnych)
<code>double</code>	8	około $\pm 1,79769313486231570E+308$ (15 znaczących cyfr dziesiętnych)

Nazwa `double` (podwójny) wynika z tego, że typ ten ma dwa razy większą precyzję niż typ `float` (czasami liczby te nazywa się liczbami o **podwójnej precyzyji**). W większości przypadków do reprezentacji liczb zmiennoprzecinkowych wybierany jest typ `double`. Ograniczona precyzyja typu `float` często okazuje się niewystarczająca. Siedem znaczących (dziesiętnych) cyfr może wystarczyć do precyzyjnego przedstawienia naszej pensji w złotówkach i groszach, ale może być już to za mało precyzyjne do przechowywania liczby określającej zarobki

naszego szefa. W związku z tym powodów do stosowania typu `float` jest niewiele; może to być sytuacja, w której zależy nam na nieznacznym zwiększeniu szybkości poprzez zastosowanie liczb o pojedynczej precyzyji lub kiedy chcemy przechowywać bardzo dużą ich ilość.

Liczby typu `float` mają przyrostek `F` lub `f` (na przykład `3.14F`). Liczby zmiennoprzecinkowe pozbawione tego przyrostka (na przykład `3.14`) są zawsze traktowane jako typ `double`. Można też podać przyrostek `D` lub `d` (na przykład `3.14D`).



Liczby zmiennoprzecinkowe można podawać w zapisie szesnastkowym. Na przykład `0.125`, czyli 2^{-3} , można zapisać jako `0x1.0p-3`. Wykładnik potęgi w zapisie szesnastkowym to `p`, a nie `e` (`e` jest cyfrą szesnastkową). Zauważ, że mantysa jest w notacji szesnastkowej, a wykładnik w dziesiętnej. Podstawą wykładnika jest 2 , nie 10 .

Wszelkie obliczenia arytmetyczne wykonywane na liczbach zmiennoprzecinkowych są zgodne ze standardem IEEE 754. Istnieją trzy szczególne wartości pozwalające określić liczby, których wartości wykraczają poza dozwolony zakres błędu:

- dodatnia nieskończoność,
- ujemna nieskończoność,
- `NaN` — nie liczby (ang. *Not a Number*).

Na przykład wynikiem dzielenia dodatniej liczby przez zero jest dodatnia nieskończoność. Działanie dzielenia zero przez zero lub wyciągania pierwiastka kwadratowego z liczby ujemnej daje w wyniku `NaN`.



Stałe `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` i `Double.NaN` (oraz ich odpowiedniki typu `float`) reprezentują wymienione specjalne wartości, ale są rzadko używane. Nie można na przykład wykonać takiego sprawdzenia:

```
if (x == Double.NaN) // Nigdy nie jest true.
```

aby dowiedzieć się, czy dany wynik jest równy stałej `Double.NaN`. Wszystkie tego typu wartości są różne. Można za to używać metody `Double.isNaN`:

```
if (Double.isNaN(x)) // Sprawdzenie, czy x jest „nie liczbą”.
```



Liczby zmiennoprzecinkowe **nie** nadają się do obliczeń finansowych, w których nie dopuszczalny jest błąd zaokrąglania (ang. *roundoff error*). Na przykład instrukcja `System.out.println(2.0 - 1.1)` da wynik `0.8999999999999999` zamiast spodziewanego `0.9`. Tego typu błędy spowodowane są tym, że liczby zmiennoprzecinkowe są reprezentowane w systemie binarnym. W systemie tym nie ma dokładnej reprezentacji ułamka $1/10$, podobnie jak w systemie dziesiętnym nie istnieje dokładna reprezentacja ułamka $1/3$. Aby wykonywać precyzyjne obliczenia numeryczne bez błędu zaokrąglania, należy użyć klasy `BigDecimal`, która jest opisana w dalszej części tego rozdziału.

3.3.3. Typ char

Typ char służy do reprezentacji pojedynczych znaków. Najczęściej są to stałe znakowe. Na przykład 'A' jest stałą znakową o wartości 65. Nie jest tym samym co "A" — łańcuchem zawierającym jeden znak. Kody Unicode mogą być wyrażane w notacji szesnastkowej, a ich wartości mieszczą się w zakresie od \u0000 do \uFFFF. Na przykład kod \u2122 reprezentuje symbol ™, a \u03c0 to grecka litera Π.

Poza symbolem zastępczym \u oznaczającym zapis znaku w kodzie Unicode jest jeszcze kilka innych symboli zastępczych umożliwiających zapisywanie różnych znaków specjalnych. Zestawienie tych znaków przedstawia tabela 3.3. Można je stosować zarówno w stałych znakowych, jak i w łańcuchach, np. 'u\2122' albo "Witaj\n". Symbol zastępczy \u jest jedynym symbolem zastępczym, którego można używać także **poza** cudzysłowami otaczającymi znaki i łańcuchy. Na przykład zapis:

```
public static void main(String\u005B\u005D args)
```

jest w pełni poprawny — kody \u005B i \u005D oznaczają znaki [i].

Tabela 3.3. Symbole zastępcze znaków specjalnych

Symbol zastępczy	Nazwa	Wartość Unicode
\b	Backspace	\u0008
\t	Tabulacja	\u0009
\n	Przejście do nowego wiersza	\u000a
\r	Powrót karetki	\u000d
\"	Cudzysłów	\u0022
\'	Apostrof	\u0027
\\\	Lewy ukośnik	\u005c

Aby w pełni zrozumieć typ char, trzeba poznać system kodowania znaków Unicode. Unicode opracowano w celu pozbycia się ograniczeń tradycyjnych systemów kodowania. Przed powstaniem systemu Unicode istniało wiele różnych standardów: ASCII w USA, ISO 8859-1 dla języków krajów Europy Zachodniej, ISO-8859-2 dla języków środkowo- i wschodnio-europejskich (w tym polskiego), KOI-8 dla języka rosyjskiego, GB18030 i BIG-5 dla języka chińskiego itd. Powoduje to dwa problemy: jeden kod może oznaczać różne znaki w różnych systemach kodowania, a poza tym kody znaków w językach o dużej liczbie znaków mają różne rozmiary — niektóre często używane znaki zajmują jeden bajt, a inne potrzebują dwóch bajtów.

Unicode ma za zadanie rozwiązać te problemy. Kiedy w latach osiemdziesiątych XX wieku podjęto próby unifikacji, wydawało się, że dwubajtowy stały kod był więcej niż wystarczający do zakodowania znaków używanych we wszystkich językach świata. W 1991 roku światło dzienne ujrzał Unicode 1.0. Wykorzystywana w nim była prawie połowa wszystkich dostępnych 65 536 kodów. Java od samego początku używała znaków 16-bitowego systemu Unicode, co dawało jej dużą przewagę nad innymi językami programowania, które stosowały znaki ośmioróżkowe.

Niestety z czasem nastąpiło to, co było nieuchronne. Unicode przekroczył liczbę 65 536 znaków, głównie z powodu dodania bardzo dużych zbiorów ideogramów używanych w językach chińskim, japońskim i koreańskim. Obecnie 16-bitowy typ `char` nie wystarcza do opisu wszystkich znaków Unicode.

Aby wyjaśnić, jak ten problem został rozwiązany w Javie, zaczynając od Java SE 5.0, musimy wprowadzić nieco nowej terminologii. **Współrzędna kodowa znaku** (ang. *code point*) to wartość związana ze znakiem w systemie kodowania. W standardzie Unicode współrzędne kodowe znaków są zapisywane w notacji szesnastkowej i są poprzedzane lańcuchem `U+`, np. współrzędna kodowa litery A to `U+0041`. Współrzędne kodowe znaków systemu Unicode są pogrupowane w 17 **przestrzeniach numeracyjnych** (ang. *code planes*). Pierwsza z nich, nazywana podstawową przestrzenią wielojęzyczną (ang. *Basic Multilingual Plane — BMP*), zawiera **klasyczne** znaki Unicode o współrzędnych kodowych z przedziału od `U+0000` do `U+FFFF`. Pozostałe szesnaście przestrzeni o współrzędnych kodowych znaków z przedziału od `U+10000` do `U+10FFFF` zawiera **znaki dodatkowe** (ang. *supplementary characters*).

Kodowanie UTF-16 to sposób reprezentacji wszystkich współrzędnych kodowych znaków za pomocą kodów o różnej długości. Znaki w podstawowej przestrzeni są 16-bitowymi wartościami o nazwie **jednostek kodowych** (ang. *code units*). Znaki dodatkowe są kodowane jako kolejne pary jednostek kodowych. Każda z wartości należących do takiej pary należy do zakresu 2048 nieużywanych wartości BMP, zwanych **obszarem surogatów** (ang. *surrogates area*) — zakres pierwszej jednostki kodowej to `U+D800 – U+DBFF`, a drugiej `U+DC00 – U+DFFF`. Jest to bardzo sprytnie rozwiązanie, ponieważ od razu wiadomo, czy jednostka kodowa reprezentuje jeden znak, czy jest pierwszą lub drugą częścią znaku dodatkowego. Na przykład matematyczny symbol oznaczający zbiór liczb całkowitych \mathbb{Z} ma współrzędną kodową `U+1D56B` i jest kodowany przez dwie jednostki kodowe `U+D835` oraz `U+DD6B` (opis algorytmu kodowania UTF-16 można znaleźć na stronie <http://en.wikipedia.org/wiki/UTF-16>).

W Javie typ `char` opisuje **jednostkę kodową** UTF-16.

Zdecydowanie odradzamy posługiwania się w programach typem `char`, jeśli nie ma konieczności wykonywania działań na jednostkach kodowych UTF-16. Prawie zawsze lepszym rozwiązaniem jest traktowanie lańcuchów (które opisujemy w podrozdziale 3.6, „Lańcuchy”) jako abstrakcyjnych typów danych.

3.3.4. Typ `boolean`

Typ `boolean` (logiczny) może przechowywać dwie wartości: `true` i `false`. Służy do sprawdzania warunków logicznych. Wartości logiczne nie można konwertować na wartości całkowitoliczbowe.

3.4. Zmienne

W Javie każda zmienna musi mieć określony **typ**. Deklaracja zmiennej polega na napisaniu nazwy typu, a po nim nazwy zmiennej. Oto kilka przykładów deklaracji zmiennych:

 W języku C++ zamiast wartości logicznych można stosować liczby, a nawet wskaźniki. Wartość 0 jest odpowiednikiem wartości logicznej `false`, a wartość różna od zera odpowiada wartości `true`. W Javie tak **nie** jest. Dzięki temu programiści Javy mają ochronę przed popełnieniem błędu:

```
if (x = 0) // ups... miałem na myśl x == 0
```

W C++ test ten przejdzie komplikację i będzie można go uruchomić, a jego wartością zawsze będzie `false`. W Javie testu tego nie będzie można skompilować, ponieważ wyrażenia całkowitoliczbowego `x = 0` nie można przekonwertować na wartość logiczną.

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

Należy zauważać, że na końcu każdej deklaracji znajduje się średnik. Jest on wymagany z tego względu, że deklaracja zmiennej jest instrukcją w Javie.

Nazwa zmiennej musi się zaczynać literą oraz składać się z liter i cyfr. Zwróćmy uwagę, że pojęcia „litera” i „cyfra” w Javie mają znacznie szersze znaczenie niż w większości innych języków. Zgodnie z definicją litera to jeden ze znaków 'A' – 'Z', 'a' – 'z', '_' lub **każdy** znak Unicode będący literą jakiegoś języka. Na przykład polscy programiści mogą w nazwach zmiennych używać liter z ogonkami, takich jak ą. Grek może użyć litery Π. Podobnie cyfry należą do zbioru '0' – '9' oraz są nimi **wszystkie** znaki Unicode, które oznaczają cyfrę w jakimś języku. W nazwach zmiennych nie można stosować symboli typu '+' czy © ani spacji. **Wszystkie** znaki użyte w nazwie zmiennej oraz ich **wielkość mają znaczenie**. Długość nazwy zmiennej jest w zasadzie nieograniczona.



Aby sprawdzić, które znaki Unicode są w Javie literami, można użyć metod `isJavaIdentifierStart` i `isJavaIdentifierPart`, które są dostępne w klasie `Character`.



Mimo że znak \$ jest w Javie traktowany jak zwykła litera, nie należy go używać w swoim kodzie. Jest stosowany w nazwach generowanych przez kompilator i inne narzędzia Javy.

Dodatkowo nazwa zmiennej w Javie nie może być taka sama jak słowo zarezerwowane (listę słów zarezerwowanych zawiera dodatek).

Kilka deklaracji można umieścić w jednym wierszu:

```
int i, j; // Obie zmienne są typu int.
```

Nie polecamy jednak takiego stylu pisania kodu. Dzięki deklarowaniu każdej zmiennej oddzielnie programy są łatwiejsze do czytania.



Jak wiemy, w nazwach są rozróżniane małe i wielkie litery. Na przykład nazwy hireday i hireDay to dwie różne nazwy. W zasadzie nie powinno się stosować nazw zmiennych różniących się tylko wielkością liter, chociaż czasami trudno jest wymyślić dobrą nazwę. Wielu programistów w takich przypadkach nadaje zmiennej taką samą nazwę jak nazwa typu:

```
Box box; // Box to nazwa typu, a box to nazwa zmiennej.
```

Inni wolą stosować przedrostek a:

```
Box aBox;
```

3.4.1. Inicjacja zmiennych

Po zadeklarowaniu zmiennej trzeba ją zainicjować za pomocą instrukcji przypisania — nie można użyć wartości niezainicjowanej zmiennej. Na przykład poniższe instrukcje w Javie są błędne:

```
int vacationDays;
System.out.println(vacationDays); // Błąd — zmienność nie została zainicjowana.
```

Przypisanie wartości do zadeklarowanej zmiennej polega na napisaniu nazwy zmiennej po lewej stronie znaku równości (=) i wyrażenia o odpowiedniej wartości po jego prawej stronie.

```
int vacationDays;
vacationDays = 12;
```

Zmienną można zadeklarować i zainicjować w jednym wierszu. Na przykład:

```
int vacationDays = 12;
```

Wreszcie, deklaracje w Javie można umieszczać w dowolnym miejscu w kodzie. Na przykład poniższy kod jest poprawny:

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // Zmienna może być zadeklarowana w tym miejscu.
```

Do dobrego stylu programowania w Javie zalicza się deklarowanie zmiennych jak najbliżej miejsca ich pierwszego użycia.



W językach C i C++ rozróżnia się **deklarację** i **definicję** zmiennej. Na przykład:

```
int i = 10;
```

jest definicją zmiennej, podczas gdy:

```
extern int i;
```

to deklaracja. W Javie deklaracje nie są oddzielane od definicji.

3.4.2. Stałe

Stałe oznaczamy słowem kluczowym `final`. Na przykład:

```

public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Rozmiar papieru w centymetrach: "
            + paperWidth * CM_PER_INCH + " na " + paperHeight * CM_PER_INCH);
    }
}

```

Słowo kluczowe `final` oznacza, że można tylko jeden raz przypisać wartość i nie będzie można już jej zmienić w programie. Nazwy stałych piszemy zwyczajowo samymi wielkimi literami.

W Javie chyba najczęściej używa się stałych, które są dostępne dla wielu metod jednej klasy. Są to tak zwane **stałe klasowe**. Tego typu stałe definiujemy za pomocą słowa kluczowego `static final`. Oto przykład użycia takiej stałej:

```

public class Constants2
{
    public static final double CM_PER_INCH = 2.54;

    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Rozmiar papieru w centymetrach: "
            + paperWidth * CM_PER_INCH + " na " + paperHeight * CM_PER_INCH);
    }
}

```

Zauważmy, że definicja stałej klasowej znajduje się **na zewnątrz** metody `main`. W związku z tym stała ta może być używana także przez inne metody tej klasy. Ponadto, jeśli (jak w naszym przykładzie) stała jest zadeklarowana jako publiczna (`public`), dostęp do niej mają także metody innych klas — jak w naszym przypadku `Constants2.CM_PER_INCH`.



Słowo `const` jest słowem zarezerwowanym w Javie, ale obecnie nie jest do niczego używane. Do deklaracji stałych trzeba używać słowa kluczowego `final`.

3.5. Operatory

Znane wszystkim operatory arytmetyczne `+`, `-`, `*` i `/` służą w Javie odpowiednio do wykonywania operacji dodawania, odejmowania, mnożenia i dzielenia. Operator `/` oznacza dzielenie całkowitoliczbowe, jeśli obie liczby są typu całkowitoliczbowego, oraz dzielenie zmienno-przecinkowe w przeciwnym przypadku. Operatorem reszty z dzielenia (**dzielenia modulo**) jest symbol `%`. Na przykład wynikiem działania `15/2` jest `7`, a `15%2` jest `1`, podczas gdy `15.0/2 = 7.5`.

Pamiętajmy, że dzielenie całkowitoliczbowe przez zero powoduje wyjątek, podczas gdy wynikiem dzielenia zmiennoprzecinkowego przez zero jest nieskończoność lub wartość NaN.

Binarne operatory arytmetyczne w przypisaniach można wygodnie skracać. Na przykład zapis:

`x += 4;`

jest równoważny z zapisem:

`x = x + 4`

Ogólna zasada jest taka, że operator powinien się znajdować po lewej stronie znaku równości, np. `*=` czy `%=`.



Jednym z głównych celów, które postawili sobie projektanci Javy, jest przenośność. Wyniki obliczeń powinny być takie same bez względu na to, której maszyny wirtualnej użyto. Uzyskanie takiej przenośności jest zaskakująco trudne w przypadku działań na liczbach zmiennoprzecinkowych. Typ `double` przechowuje dane liczbowe w 64 bitach pamięci, ale niektóre procesory mają 80-bitowe rejestrów liczb zmiennoprzecinkowych. Rejestry te w swoich obliczeniach pośrednich stosują zwiększoną precyzję. Przyjrzyjmy się na przykład poniższemu działaniu:

```
double w = x * y/z;
```

Wiele procesorów Intel wartość wyrażenia `x * y` zapisuje w 80-bitowym rejestrze. Następnie wykonywane jest dzielenie przez `z`, a wynik z powrotem obcinany do 64 bitów. Tym sposobem otrzymujemy dokładniejsze wyniki i unikamy przekroczenia zakresu wykładnika. Ale wynik może być **inny**, niż gdyby obliczenia były cały czas wykonywane w 64 bitach. Z tego powodu w pierwszych specyfikacjach wirtualnej maszyny Javy był zapisany wymóg, aby wszystkie obliczenia pośrednie używały zmniejszonej precyzji. Nie przepadała za tym cała społeczność programistyczna. Obliczenia o zmniejszonej precyzji mogą nie tylko powodować przekroczenie zakresu, ale są też **wolniejsze** niż obliczenia o zwiększonej precyzji, ponieważ obcinanie bitów zajmowało czas. W związku z tym opracowano aktualizacje języka Java mająca na celu rozwiązać problem sprzecznych wymagań dotyczących optymalizacji wydajności i powtarzalności wyników. Projektanci maszyny wirtualnej mogą obecnie stosować zwiększoną precyzję w obliczeniach pośrednich. Jednak metody oznaczone słowem **kluczowym strictfp** muszą korzystać ze ścisłych działań zmiennoprzecinkowych, które dają powtarzalne wyniki.

Na przykład metodę `main` można oznaczyć następująco:

```
public static strictfp void main(String[] args)
```

W takim przypadku wszystkie instrukcje znajdujące się w metodzie `main` używają ograniczonych obliczeń zmiennoprzecinkowych. Jeśli oznaczymy w ten sposób klasę, wszystkie jej metody będą stosować obliczenia zmiennoprzecinkowe o zmniejszonej precyzji.

Sedno problemu leży w działaniu procesorów Intel. W trybie domyślnym obliczenia pośrednie mogą używać rozszerzonego wykładnika, ale nie rozszerzonej mantysy (chipy Intela umożliwiają obcinanie mantysy niepowodujące strat wydajności). W związku z tym główna różnica pomiędzy trybem domyślnym a ścisłym jest taka, że obliczenia ścisłe mogą przekroczyć zakres, a domyślne nie.

Muszę jednak uspokoić tych, u których na ciele wystąpiła gesia skórka w trakcie lektury tej uwagi. Przekroczenie zakresu liczby zmiennoprzecinkowej nie zdarza się na co dzień w zwykłych programach. W tej książce nie używamy słowa **kluczowego strictfp**.

3.5.1. Operatory inkrementacji i dekrementacji

Programiści doskonale wiedzą, że jednym z najczęściej wykonywanych działań na zmiennych liczbowych jest dodawanie lub odejmowanie jedynki. Java, podobnie jak C i C++, ma zarówno operator inkrementacji, jak i dekrementacji. Zapis `n++` powoduje zwiększenie wartości zmiennej `n` o jeden, a `n--` zmniejszenie jej o jeden. Na przykład kod:

```
int n = 12
n++;
```

zwiększa wartość przechowywaną w zmiennej `n` na 13. Jako że operatory te zmieniają wartość zmiennej, nie można ich stosować do samych liczb. Na przykład nie można napisać `4++`.

Operatory te występują w dwóch postaciach. Powyżej widzieliśmy postaci przyrostkowe, które — jak wskazuje nazwa — umieszcza się po operądzie. Druga postać to postać przedrostkowa — `++n`. Obie zwiększą wartość zmiennej o jeden. Różnica pomiędzy nimi ujawnia się, kiedy zostaną użyte w wyrażeniu. W przypadku zastosowania formy przedrostkowej wartość zmiennej jest zwiększana przed obliczeniem wartości wyrażenia, a w przypadku formy przyrostkowej wartość zmiennej zwiększa się po obliczeniu wartości wyrażenia.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // a ma wartość 16, a m — 8
int b = 2 * m++; // b ma wartość 14, a n — 8
```

Nie zalecamy stosowania operatora `++` w innych wyrażeniach, ponieważ zaciemnia to kod i często powoduje irytujące błędy.

(Jak powszechnie wiadomo, nazwa języka C++ pochodzi od operatora inkrementacji, który jest też „winowiącą” powstania pierwszego dowcipu o tym języku. Przeciwnicy C++ zauważają, że nawet nazwa tego języka jest błędna: „Powinna brzmieć `++C`, ponieważ języka tego chcielibyśmy używać tylko po wprowadzeniu do niego poprawek”.)

3.5.2. Operatory relacyjne i logiczne

Java ma pełny zestaw operatorów relacyjnych. Aby sprawdzić, czy dwa argumenty są równe, używamy dwóch znaków równości (`==`). Na przykład wyrażenie:

```
3 == 7
```

zwróci wartość `false`.

Operator nierówności ma postać `!=`. Na przykład wyrażenie:

```
3 != 7
```

zwróci wartość `true`.

Dodatkowo dostępne są operatory większości (`>`), mniejszości (`<`), mniejszy lub równy (`<=`) oraz większy lub równy (`>=`).

Operatorem koniunkcji logicznej w Javie, podobnie jak w C++, jest `&&`, a alternatywy logicznej `||`. Jak nietrudno się domyślić, znając operator `!=`, znak wykrzyknika `(!)` jest operatorem negacji. Wartości wyrażeń z użyciem operatorów `&&` i `||` są obliczane metodą na skróty. Wartość drugiego argumentu nie jest obliczana, jeśli ostateczny rezultat wynika już z pierwszego. Jeżeli między dwoma wyrażeniami postawimy operator `&&`:

`wyrażenie1 && wyrażenie2`

i wartość logiczna pierwszego z nich okaże się `false`, to wartość całego wyrażenia nie może być inna niż `false`. W związku z tym wartość drugiego wyrażenia **nie** jest obliczana. Można to wykorzystać do unikania błędów. Jeśli na przykład wartość zmiennej `x` w wyrażeniu:

`x != 0 && 1/x > x + y // Unikamy dzielenia przez zero.`

jest równa zero, druga jego część nie będzie obliczana. Zatem działanie `1/x` nie zostanie wykonane, jeśli `x = 0`, dzięki czemu nie wystąpi błąd dzielenia przez zero.

Podobnie wartość wyrażenia `wyrażenie1 || wyrażenie2` ma automatycznie wartość `true`, jeśli pierwsze wyrażenie ma wartość `true`. Wartość drugiego nie jest obliczana.

W Javie dostępny jest też czasami przydatny operator trójargumentowy w postaci `? :`. Wartością wyrażenia:

`warunek ? wyrażenie1 : wyrażenie2`

jest `wyrażenie1`, jeśli `warunek` ma wartość `true`, lub `wyrażenie2`, jeśli `warunek` ma wartość `false`. Na przykład wynikiem wyrażenia:

`x < y ? x : y`

jest `x` lub `y` — w zależności od tego, która wartość jest mniejsza.

3.5.3. Operatory bitowe

Do pracy na typach całkowitoliczbowych można używać operatorów dających dostęp bezpośrednio do bitów, z których się one składają. Oznacza to, że za pomocą techniki maskowania można dobrać się do poszczególnych bitów w liczbie. Operatory bitowe to:

`&` (bitowa koniunkcja) `|` (bitowa alternatywa) `^` (lub wykluczające) `~`(bitowa negacja)

Operatory te działają na bitach. Jeśli na przykład zmienna `n` jest typu `int`, to wyrażenie:

`int fourthBitFromRight = (n & 8) / 8;`

da wynik 1, jeśli czwarty bit od prawej w binarnej reprezentacji wartości zmiennej `n` jest jedynką, lub 0 w przeciwnym razie. Dzięki użyciu odpowiedniej potęgi liczby 2 można zamaskować wszystkie bity poza jednym.



Operatory `&` i `|` zastosowane do wartości logicznych zwracają wartości logiczne. Są one podobne do operatorów `&&` i `||`, tyle że do obliczania wartości wyrażeń z ich użyciem nie jest stosowana metoda na skróty. A zatem wartości obu argumentów są zawsze obliczane przed zwróceniem wyniku.

Można też używać tak zwanych operatorów przesunięcia, w postaci `>>` i `<<`, które przesuwają liczbę o jeden bit w prawo lub w lewo. Często przydatne są przy tworzeniu ciągów bitów używanych przy maskowaniu:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Ostatni z operatorów bitowych `>>>` odpowiada za przesunięcie bitowe w prawo z wypełnieniem zerami, podczas gdy operator `>>` przesuwa bity w prawo i do ich wypełnienia używa znaku liczby. Nie ma operatora `<<<`.



Argument znajdujący się po prawej stronie operatorów przesunięcia jest redukowany modulo do 32 bitów (chyba że argument po lewej stronie jest typu `long`; w takim przypadku argument z prawej strony jest redukowany modulo do 64 bitów). Na przykład wartość wyrażenia `1 << 35` jest taka sama jak `1 << 3`, czyli 8.



C++ W językach C i C++ nie ma gwarancji, że operator `>>` wykonuje przesunięcie arytmetyczne (wypełnienie bitem znaku), a nie przesunięcie logiczne (wypełnienie zerami). Implementatorzy mogą na własną rękę wybrać takie działanie, które jest bardziej efektywne. Oznacza to, że operator `>>` w C++ jest zdefiniowany tylko dla liczb nieujemnych. Java jest wolna od tej wieloznaczności.

3.5.4. Funkcje i stałe matematyczne

Klasa `Math` zawiera zestaw funkcji matematycznych, które mogą być bardzo przydatne przy pisaniu niektórych rodzajów programów.

Do wyciągania pierwiastka stopnia drugiego z liczby służy metoda `sqrt`:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // wynik 2.0
```



Między metodami `println` i `sqrt` jest pewna różnica. Pierwsza działa na obiekcie `System.out`, który jest zdefiniowany w klasie `System`. Druga natomiast nie działa na żadnym obiekcie. Tego typu metody noszą nazwę **metod statycznych**. Więcej na ich temat dowiesz się w rozdziale 4.

W Javie nie ma operatora podnoszącego liczbę do potęgi. Do tego celu trzeba użyć metody `pow` dostępnej w klasie `Math`. Wyrażenie:

```
double y = Math.pow(x, a);
```

ustawia wartość zmiennej `y` na liczbę `x` podniesioną do potęgi `a` (x^a). Metoda `pow` przyjmuje parametry typu `double` i zwraca wynik tego samego typu.

Klasa `Math` udostępnia także metody obliczające funkcje trygonometryczne:

```
Math.sin
Math.cos
```

```
Math.tan
Math.atan
Math.atan2
```

a także funkcję wykładniczą i jej odwrotność, czyli logarytm naturalny, oraz logarytm dziesiętny:

```
Math.exp
Math.log
Math.log10
```

Dostępne są też dwie stałe określające w maksymalnym przybliżeniu stałe matematyczne π i e :

```
Math.PI
Math.E
```



Można uniknąć stosowania przedrostka `Math` przed metodami i stałymi matematycznymi, umieszczając poniższy wiersz kodu na początku pliku źródłowego:

```
import static java.lang.Math.*;
```

Na przykład:

```
System.out.println("Pierwiastek kwadratowy z \u03c0 wynosi " + sqrt(PI));
```

Importy statyczne opisujemy w rozdziale 4.



Funkcje klasy `Math` używają procedur z jednostki liczb zmiennoprzecinkowych komputera w celu osiągnięcia jak najlepszej wydajności. Jeśli od prędkości ważniejsze są dokładne wyniki, należy posłużyć się klasą `StrictMath`. Implementuje ona algorytmy z biblioteki, którą można nieodpłatnie rozpowszechniać, o nazwie `fdlibm`, a która gwarantuje identyczne wyniki na wszystkich platformach. Kod źródłowy tych algorytmów można znaleźć na stronie <http://www.netlib.org/fdlibm> (dla każdej funkcji `fdlibm`, która ma więcej niż jedną definicję, klasa `StrictMath` używa wersji zgodnej ze standardem IEEE 754, której nazwa zaczyna się od litery `e`).

3.5.5. Konwersja typów numerycznych

Często konieczna jest konwersja z jednego typu liczbowego na inny. Rysunek 3.1 przedstawia dozwolone rodzaje konwersji.

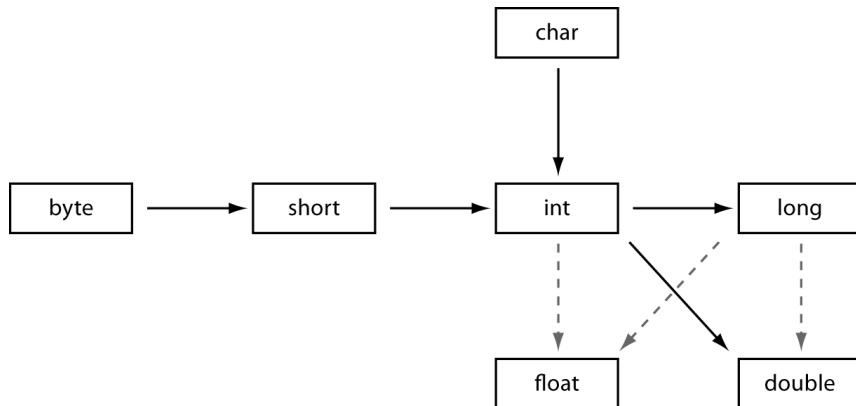
Sześć typów konwersji (rysunek 3.1) niepowodujących strat danych oznaczono strzałkami ciągłymi. Konwersje, które mogą spowodować utratę części danych, oznaczono strzałkami przerywanymi. Na przykład duża liczba całkowita, jak 123 456 789, składa się z większej liczby cyfr, niż może się zmieścić w typie `float`. Po konwersji tej liczby całkowitej na liczbę typu `float` stracimy nieco na precyzyji:

```
int n = 123456789;
float f = n; //fma wartość 1.23456792E8
```

Jeśli operatorem dwuargumentowym połączymy dwie wartości (np. `n + f`, gdzie `n` to liczba całkowita, a `f` liczba zmiennoprzecinkowa), zostaną one przekonwertowane na wspólny typ przed wykonaniem działania.

Rysunek 3.1.

Dozwolone konwersje pomiędzy typami liczbowymi



- Jeśli któryś z operandów jest typu `double`, drugi również zostanie przekonwertowany na typ `double`.
- W przeciwnym razie, jeśli któryś z operandów jest typu `float`, drugi zostanie przekonwertowany na typ `float`.
- W przeciwnym razie, jeśli któryś z operandów jest typu `long`, drugi zostanie przekonwertowany na typ `long`.
- W przeciwnym razie oba operandy zostaną przekonwertowane na typ `int`.

3.5.6. Rzutowanie

W poprzednim podrozdziale dowiedzieliśmy się, że wartości typu `int` są w razie potrzeby automatycznie konwertowane na typ `double`. Są jednak sytuacje, w których chcemy przekonwertować typ `double` na typ `int`. W Javie możliwe są takie konwersje, ale oczywiście mogą one pociągać za sobą utratę informacji. Konwersje, w których istnieje ryzyko utraty informacji, nazywają się **rzutowaniem** (ang. *casting*). Aby wykonać rzutowanie, należy przed nazwą rzutowanej zmiennej postawić nazwę typu docelowego w okrągłych nawiasach. Na przykład:

```
double x = 9.997;
int nx = (int) x;
```

W wyniku tego działania zmienna `nx` będzie miała wartość 9, ponieważ rzutowanie liczby zmennoprzecinkowej na całkowitą powoduje usunięcie części ułamkowej.

Aby zaokrąglić liczbę zmennoprzecinkową do **najbliższej** liczby całkowitej (co w większości przypadków bardziej się przydaje), należy użyć metody `Math.round`:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

Teraz zmienna `nx` ma wartość 10. Przy zaokrąglaniu za pomocą metody `round` nadal konieczne jest zastosowanie rzutowania, tutaj `(int)`. Jest to spowodowane tym, że metoda `round` zwraca wartość typu `long`, a tego typu wartość można przypisać zmiennej typu `int` wyłącznie na drodze jawnego rzutowania, ponieważ istnieje ryzyko utraty danych.



Wynikiem rzutowania na określony typ liczby, która nie mieści się w jego zakresie, jest obcięcie tej liczby i powstanie całkiem nowej wartości. Na przykład rzutowanie (byte) 300 da w wyniku liczbę 44.



C++ Nie można wykonać rzutowania pomiędzy wartościami liczbowymi i logicznymi. Zapobiega to powstawaniu wielu błędów. W nielicznych przypadkach, kiedy wymagana jest konwersja wartości logicznej na wartość liczbową, można użyć wyrażenia warunkowego, np. `b ? 1 : 0`.

3.5.7. Nawiasy i priorytety operatorów

Tabela 3.4 przedstawia zestawienie operatorów z uwzględnieniem ich priorytetów. Jeśli nie ma nawiasów, kolejność wykonywania działań jest taka jak kolejność operatorów w tabeli. Operatory o takim samym priorytecie są wykonywane od lewej do prawej, z wyjątkiem tych, które mają wiązanie prawostronne, podane w tabeli. Ponieważ operator `&&` ma wyższy priorytet od operatora `||`, wyrażenie:

`a && b || c`

jest równoznaczne z wyrażeniem:

`(a && b) || c`

Tabela 3.4. Priorytety operatorów

Operator	Wiązanie
<code>[] . ()</code> (wywołanie metody)	lewe
<code>! ~++ -- + (jednoargumentowy) () (rzutowanie) new</code>	prawe
<code>* / %</code>	lewe
<code>+ -</code>	lewe
<code><< >> >>></code>	lewe
<code>< <= > >= instanceof</code>	lewe
<code>== !=</code>	lewe
<code>&</code>	lewe
<code>^</code>	lewe
<code> </code>	lewe
<code>&&</code>	lewe
<code> </code>	lewe
<code>? :</code>	prawe
<code>= += -= *= /= %= &= ^= <<= >>= >>>=</code>	prawe

Ze względu na fakt, że operator `+=` ma wiązanie lewostronne, wyrażenie:

```
a += b += c
```

jest równoważne z wyrażeniem:

```
a += (b += c)
```

To znaczy, że wartość wyrażenia `b += c` (która wynosi tyle co `b` po dodawaniu) zostanie dodana do `a`.



W przeciwieństwie do języków C i C++ Java nie ma operatora przecinka. Jednak w pierwszym i trzecim argumencie instrukcji `for` można używać **list wyrażeń oddzielonych przecinkami**.

3.5.8. Typ wyliczeniowy

Czasami zmienna może przechowywać tylko ograniczoną liczbę wartości. Na przykład kiedy sprzedajemy pizzę albo ubrania, możemy mieć rozmiary mały, średni, duży i ekstra duży. Oczywiście można te rozmiary zakodować w postaci cyfr 1, 2, 3 i 4 albo liter M, S, D i X. To podejście jest jednak podatne na błędy. Zbyt łatwo można zapisać w zmiennej nieprawidłową wartość (jak 0 albo m).

Można też definiować własne **typy wyliczeniowe** (ang. *enumerated type*). Typ wyliczeniowy zawiera skończoną liczbę nazwanych wartości. Na przykład:

```
enum Rozmiar { MAŁY, ŚREDNI, DUŻY, EKSTRA_DUŻY };
```

Teraz możemy deklarować zmienne takiego typu:

```
Rozmiar s = Rozmiar.ŚREDNI;
```

Zmienna typu `Rozmiar` może przechowywać tylko jedną z wartości wymienionych w deklaracji typu lub specjalną wartość `null`, która oznacza, że zmienna nie ma w ogóle żadnej wartości.

Bardziej szczegółowy opis typów wyliczeniowych znajduje się w rozdziale 5.

3.6. Łańcuchy

W zasadzie łańcuchy w Javie składają się z szeregu znaków Unicode. Na przykład łańcuch "Java\u2122" składa się z pięciu znaków Unicode: J, a, v, a i ™. W Javie nie ma wbudowanego typu `String`. Zamiast tego standardowa biblioteka Javy zawiera predefiniowaną klasę o takiej właśnie nazwie. Każdy łańcuch w cudzysłowach jest obiektem klasy `String`:

```
String e = ""; //pusty łańcuch
String greeting = "Cześć!";
```

3.6.1. Podłańcuchy

Aby wydobyć z łańcucha podłańcuch, należy użyć metody `substring` klasy `String`. Na przykład:

```
String greeting = "Cześć!";
String s = greeting.substring(0, 3);
```

Powyższy kod zwróci łańcuch "Cze".

Drugi parametr metody `substring` określa położenie pierwszego znaku, którego **nie** chcemy skopiować. W powyższym przykładzie chcieliśmy skopiować znaki na pozycjach 0, 1 i 2 (od pozycji 0 do 2 włącznie). Z punktu widzenia metody `substring` nasz zapis oznacza: od pozycji zero włącznie do pozycji 3 **z wyłączeniem**.

Sposób działania metody `substring` ma jedną zaletę: łatwo można obliczyć długość podłańcucha. łańcuch `s.substring(a, b)` ma długość `b - a`. Na przykład łańcuch "Cze" ma długość $3 - 0 = 3$.

3.6.2. Konkatenacja

W Javie, podobnie jak w większości innych języków programowania, można łączyć (konkatenować) łańcuchy za pomocą znaku `+`.

```
String expletive = "brzydkie słowo";
String PG13 = "usunięto";
String message = expletive + PG13;
```

Powyższy kod ustawia wartość zmiennej `message` na łańcuch "brzydkiesłowousunięto" (zauważ brak spacji pomiędzy słowami). Znak `+` łączy dwa łańcuchy w takiej kolejności, w jakiej zostały podane, **nic** w nich nie zmieniając.

Jeśli z łańcuchem zostanie połączona wartość niebędąca łańcuchem, zostanie ona przekonwertowana na łańcuch (w rozdziale 5. przekonamy się, że każdy obiekt w Javie można przekonwertować na łańcuch). Na przykład kod:

```
int age = 13;
String rating = "PG" + age;
```

ustawia wartość zmiennej `rating` na łańcuch "PG13".

Funkcjonalność ta jest często wykorzystywana w instrukcjach wyjściowych. Na przykład kod:

```
System.out.println("Odpowiedź brzmi " + answer);
```

jest w pełni poprawny i wydrukowałby to, co potrzeba (przy zachowaniu odpowiednich odstępów, gdyż po słowie `brzmi` znajduje się spacja).

3.6.3. Łańcuchów nie można modyfikować

W klasie `String` brakuje metody, która umożliwiałaby **zmianę** znaków w łańcuchach. Aby zmienić komunikat w zmiennej `greeting` na "Czekaj", nie możemy bezpośrednio zamienić trzech ostatnich znaków na „kaj”. Programiści języka C są w takiej sytuacji zupełnie bezradni. Jak zmodyfikować łańcuch? W Javie okazuje się to bardzo proste. Należy połączyć podłańcuch, który chcemy zachować, ze znakami, które chcemy wstawić w miejsce tych wyrzuconych.

```
greeting = greeting.substring(0, 3) + "kaj";
```

Ta deklaracja zmienia wartość przechowywaną w zmiennej `greeting` na "Czekaj".

Jako że w łańcuchach nie można zmieniać znaków, obiekty klasy `String` w dokumentacji języka Java są określane jako **niezmienialne** (ang. *immutable*). Podobnie jak liczba 3 jest zawsze liczbą 3, łańcuch "Cześć!" zawsze będzie szeregiem jednostek kodowych odpowiadających znakom C, z, e, ś, ĺ i !. Nie można zmienić tych wartości. Można jednak, o czym się przekonaliśmy, zmienić zawartość **zmiennej** `greeting`, sprawiając, aby odwoływała się do innego łańcucha. Podobnie możemy zadecydować, że zmienna liczbowa przechowująca wartość 3 zmieni odwołanie na wartość 4.

Czy to nie odbija się na wydajności? Wydaje się, że zmiana jednostek kodowych byłaby prostsza niż tworzenie nowego łańcucha od początku. Odpowiedź brzmi: tak i nie. Rzeczywiście generowanie nowego łańcucha zawierającego połączone łańcuchy "Cze" i "kaj" jest nieefektywne, ale niezmienialność łańcuchów ma jedną zaletę: kompilator może traktować łańcuchy jako **współdzielone**.

Aby zrozumieć tę koncepcję, wyobraźmy sobie, że różne łańcuchy są umieszczone w jednym wspólnym zbiorniku. Zmienne łańcuchowe wskazują na określone lokalizacje w tym zbiorniku. Jeśli skopujemy taką zmienną, zarówno oryginalny łańcuch, jak i jego kopia współdzielą te same znaki.

Projektanci języka Java doszli do wniosku, że korzyści płynące ze współdzielenia są większe niż straty spowodowane edycją łańcuchów poprzez ekstrakcję podłańcuchów i konkatenację. Przyjrzyj się swoim własnym programom — zapewne w większości przypadków nie ma w nich modyfikacji łańcuchów, a głównie różne rodzaje porównań (jest tylko jeden dobrze znany wyjątek — składanie łańcuchów z pojedynczych znaków lub krótszych łańcuchów przychodzących z klawiatury bądź pliku; dla tego typu sytuacji w Javie przewidziano specjalną klasę, którą opisujemy w podrozdziale 3.6.9, „Składanie łańcuchów”).

3.6.4. Porównywanie łańcuchów

Do sprawdzania, czy dwa łańcuchy są identyczne, służy metoda `equals`. Wyrażenie:

```
s.equals(t)
```

zwróci wartość `true`, jeśli łańcuchy `s` i `t` są identyczne, lub `false` w przeciwnym przypadku. Zauważmy, że `s` i `t` mogą być zmiennymi łańcuchowymi lub stałymi łańcuchowymi. Na przykład wyrażenie:



Programiści języka C, którzy po raz pierwszy stykają się z łańcuchami w Javie, nie mogą ukryć zdumienia, ponieważ dla nich łańcuchy są tablicami znaków:

```
char greeting[] = "Cześć!";
```

Jest to nieprawidłowa analogia. Łańcuch w Javie można porównać ze wskaźnikiem `char*`:

```
char* greeting = "Cześć!";
```

Kiedy zastąpimy komunikat `greeting` jakimś innym łańcuchem, Java wykona z grubsza takie działania:

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "kaj", 3);
greeting = temp;
```

Teraz zmieniona `greeting` wskazuje na łańcuch "Czekaj". Nawet najbardziej zatwardziały wielbiciel języka C musi przyznać, że składnia Javy jest bardziej elegancka niż szereg wywołań funkcji `strncpy`. Co się stanie, jeśli wykonamy jeszcze jedno przypisanie do zmiennej `greeting`?

```
greeting = "Cześć!";
```

Czy to nie spowoduje wycieku pamięci? Przecież oryginalny łańcuch został umieszczony na stercie. Na szczęście Java automatycznie usuwa nieużywane obiekty. Jeśli dany blok pamięci nie jest już potrzebny, zostanie wyczyszczony.

Typ `String` Javy dużo łatwiej opanować programistom języka C++, którzy używają klasy `String` zdefiniowanej w standardzie ISO/ANSI tego języka. Obiekty klasy `String` w C++ także automatycznie przydzielają i czyszczą pamięć. Zarządzanie pamięcią odbywa się w sposób jawny za pośrednictwem konstruktorów, operatorów przypisania i destruktorów. Ponieważ w C++ łańcuchy są zmienialne (ang. *mutable*), można zmieniać w nich poszczególne znaki.

```
"Cześć!".equals(greeting)
```

jest poprawne. Aby sprawdzić, czy dwa łańcuchy są identyczne, z pominięciem wielkości liter, należy użyć metody `equalsIgnoreCase`.

```
"Cześć!".equalsIgnoreCase("cześć!"))
```

Do porównywania łańcuchów **nie** należy używać operatora `==`. Za jego pomocą można tylko stwierdzić, czy dwa łańcuchy są przechowywane w tej samej lokalizacji. Oczywiście skoro łańcuchy są przechowywane w tym samym miejscu, to muszą być równe. Możliwe jest jednak też przechowywanie wielu kopii jednego łańcucha w wielu różnych miejscach.

```
String greeting = "Cześć!"; // Inicjacja zmiennej greeting łańcuchem.
if (greeting == "Cześć!") . . .
// prawdopodobnie true
if (greeting.substring(0, 3) == "Cze") . . .
// prawdopodobnie false
```

Gdyby maszyna wirtualna zawsze traktowała równe łańcuchy jako współdzielone, można by było je porównywać za pomocą operatora `==`. Współdzielone są jednak tylko **stale** łańcuchowe. Łańcuchy będące na przykład wynikiem operacji wykonywanych za pomocą operatora `+` albo metody `substring` nie są współdzielone. W związku z tym **nigdy** nie używaj

operatora == do porównywania łańcuchów, chyba że chcesz stworzyć program zawierający najgorszy rodzaj błędu — pojawiający się od czasu do czasu i sprawiający wrażenie, że występuje losowo.



Osoby przyzwyczajone do klasy `string` w C++ muszą zachować szczególną ostrożność przy porównywaniu łańcuchów. Klasa C++ `string` przesłania operator == do porównywania łańcuchów. W Javie dość niefortunnie nadano łańcuchom takie same własności jak wartościom liczbowym, aby następnie nadać im właściwości wskaźników, jeśli chodzi o porównywanie. Projektanci tego języka mogli zmienić definicję operatora == dla łańcuchów, podobnie jak zrobili z operatorem +. Cóż, każdy język ma swoje wady.

Programiści języka C nigdy nie używają operatora == do porównywania łańcuchów. Do tego służy im funkcja `strcmp`. Metoda Javy `compareTo` jest dokładnym odpowiednikiem funkcji `strcmp`. Można napisać:

```
if (greeting.compareTo("Cześć!") == 0) . . .
```

ale użycie metody `equals` wydaje się bardziej przejrzystym rozwiązaniem.

3.6.5. Łańcuchy puste i łańcuchy null

Pusty łańcuch "" to łańcuch o zerowej długości. Aby sprawdzić, czy łańcuch jest pusty, można użyć instrukcji:

```
if (str.length() == 0)
```

lub

```
if (str.equals(""))
```

Pusty łańcuch jest w Javie obiektem zawierającym informację o swojej długości (0) i pustą treść. Ponadto zmienna typu `String` może też zawierać specjalną wartość o nazwie `null`, oznaczającą, że aktualnie ze zmienną nie jest powiązany żaden obiekt (więcej informacji na temat wartości `null` znajduje się w rozdziale 4.). Aby sprawdzić, czy wybrany łańcuch jest `null`, można użyć następującej instrukcji warunkowej:

```
if (str == null)
```

Czasami trzeba też sprawdzić, czy łańcuch nie jest ani pusty, ani `null`. Wówczas można się posłużyć poniższą instrukcją warunkową:

```
if (str != null && str.length() != 0)
```

Najpierw należy sprawdzić, czy łańcuch nie jest `null`, ponieważ wywołanie metody na wartości `null` jest błędem, o czym szerzej napisano w rozdziale 4.

3.6.6. Współrzędne kodowe znaków i jednostki kodowe

Łańcuchy w Javie są ciągami wartości typu `char`. Jak wiemy z podrozdziału 3.3.3, „Typ `char`”, typ danych `char` jest jednostką kodową reprezentującą współrzędne kodowe znaków Unicode w systemie UTF-16. Najczęściej używane znaki Unicode mają reprezentacje

składające się z jednej jednostki kodowej. Reprezentacje znaków dodatkowych składają się z par jednostek kodowych.

Metoda `length` zwraca liczbę jednostek kodowych, z których składa się podany łańcuch w systemie UTF-16. Na przykład:

```
String greeting = "Cześć!";
int n = greeting.length();           // wynik = 6
```

Aby sprawdzić rzeczywistą długość, to znaczy liczbę współrzędnych kodowych znaków, należy napisać:

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

Wywołanie `s.charAt(n)` zwraca jednostkę kodową znajdująca się na pozycji `n`, gdzie `n` ma wartość z zakresu pomiędzy `0` a `s.length() - 1`. Na przykład:

```
char first = greeting.charAt(0);    // Pierwsza jest litera 'C'.
char last = greeting.charAt(4);     // Piąty znak to 'é'.
```

Aby dostać się do *i*-tej współrzędnej kodowej znaku, należy użyć następujących instrukcji:

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```



W Javie, podobnie jak w C i C++, współrzędne i jednostki kodowe w łańcuchach są liczone od 0.

Dlaczego robimy tyle szumu wokół jednostek kodowych? Rozważmy poniższe zdanie:

Z oznacza zbiór liczb całkowitych

Znak **Z** wymaga dwóch jednostek kodowych w formacie UTF-16. Wywołanie:

```
char ch = sentence.charAt(1)
```

nie zwróci spacji, ale drugą jednostkę kodową znaku **Z**. Aby uniknąć tego problemu, nie należało używać typu `char`. Działa on na zbyt niskim poziomie.

Jeśli nasz kod przemierza łańcuch i chcemy zobaczyć każdą współrzędną kodową po kolei, należy użyć poniższych instrukcji:

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

Można też napisać kod działający w odwrotną stronę:

```
i--;
if (Character.isSurrogate(sentence.charAt(i))) i--;
int cp = sentence.codePointAt(i);
```

3.6.7. API String

Klasa String zawiera ponad 50 metod. Zaskakująco wiele z nich jest na tyle użytecznych, że możemy się spodziewać, iż będziemy ich często potrzebować. Poniższy wyciąg z API zawiera zestawienie metod, które w naszym odczuciu są najbardziej przydatne.



Takie wyciągi z API znajdują się w wielu miejscach książki. Ich celem jest przybliżenie czytelnikowi API Javy. Każdy wyciąg z API zaczyna się od nazwy klasy, np. `java.lang.String` — znaczenie nazwy **pakietu** `java.lang` jest wyjaśnione w rozdziale 4. Po nazwie klasy znajdują się nazwy, objaśnienia i opis parametrów jednej lub większej liczby metod.

Z reguły nie wymieniamy wszystkich metod należących do klasy, ale wybieramy te, które są najczęściej używane, i zamieszczamy ich zwięzłe opisy. Pełną listę metod można znaleźć w dokumentacji dostępnej w internecie (zobacz podrozdział 3.6.8, „Dokumentacja API w internecie”).

Dodatkowo podajemy numer wersji Javy, w której została wprowadzona dana klasa. Jeśli jakaś metoda została do niej dodana później, ma własny numer wersji.

java.lang.String 1.0

- `char charAt(int index)`

Zwraca jednostkę kodową znajdująca się w określonej lokalizacji. Metoda ta jest przydatna tylko w pracy na niskim poziomie nad jednostkami kodowymi.

- `int codePointAt(int index) 5.0`

Zwraca współrzedną kodową znaku, która zaczyna się lub kończy w określonej lokalizacji.

- `int offsetByCodePoints(int startIndex, int cpCount) 5.0`

Zwraca indeks współrzędnej kodowej, która znajduje się w odległości `cpCount` współrzędnych kodowych od współrzędnej kodowej `startIndex`.

- `int compareTo(String other)`

Zwraca wartość ujemną, jeśli łańcuch znajduje się przed innym (`other`) łańcuchem w kolejności słownikowej, wartość dodatnią, jeśli znajduje się za nim, lub 0, jeśli łańcuchy są identyczne.

- `boolean endsWith(String suffix)`

Zwraca wartość `true`, jeśli na końcu łańcucha znajduje się przyrostek `suffix`.

- `boolean equals(Object other)`

Zwraca wartość `true`, jeśli łańcuch jest identyczny z łańcuchem `other`.

- `boolean equalsIgnoreCase(String other)`

Zwraca wartość `true`, jeśli łańcuch jest identyczny z innym łańcuchem przy zignorowaniu wielkości liter.

- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int indexOf(int cp)
- int indexOf(int cp, int fromIndex)

Zwraca początek pierwszego podłańcucha podanego w argumencie str lub współrzędnej kodowej cp, szukanie zaczynać od indeksu 0, pozycji fromIndex czy też -1, jeśli napisu str nie ma w tym łańcuchu.

- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- int lastindex0f(int cp)
- int lastindex0f(int cp, int fromIndex)

Zwraca początek ostatniego podłańcucha podanego w argumencie str lub współrzędnej kodowej cp. Szukanie zaczyna od końca łańcucha albo pozycji fromIndex.

- int length()

Zwraca długość łańcucha.

- int codePointCount(int startIndex, int endIndex) 5.0

Zwraca liczbę współrzędnych kodowych znaków znajdujących się pomiędzy pozycjami startIndex i endIndex - 1. Surogaty niemające pary są traktowane jako współrzędne kodowe.

- String replace(CharSequence oldString, CharSequence newString)

Zwraca nowy łańcuch, w którym wszystkie łańcuchy oldString zostały zastąpione łańcuchami newString. Można podać obiekt String lub StringBuilder dla parametru CharSequence.

- boolean startsWith(String prefix)

Zwraca wartość true, jeśli łańcuch zaczyna się od podłańcucha prefix.

- String substring(int beginIndex)

- String substring(int beginIndex, int endIndex)

Zwraca nowy łańcuch składający się ze wszystkich jednostek kodowych znajdujących się na pozycjach od beginIndex do końca łańcucha albo do endIndex - 1.

- String toLowerCase()

Zwraca nowy łańcuch zawierający wszystkie znaki z oryginalnego ciągu przekonwertowane na małe litery.

- String toUpperCase()

Zwraca nowy łańcuch zawierający wszystkie znaki z oryginalnego ciągu przekonwertowane na duże litery.

■ `String trim()`

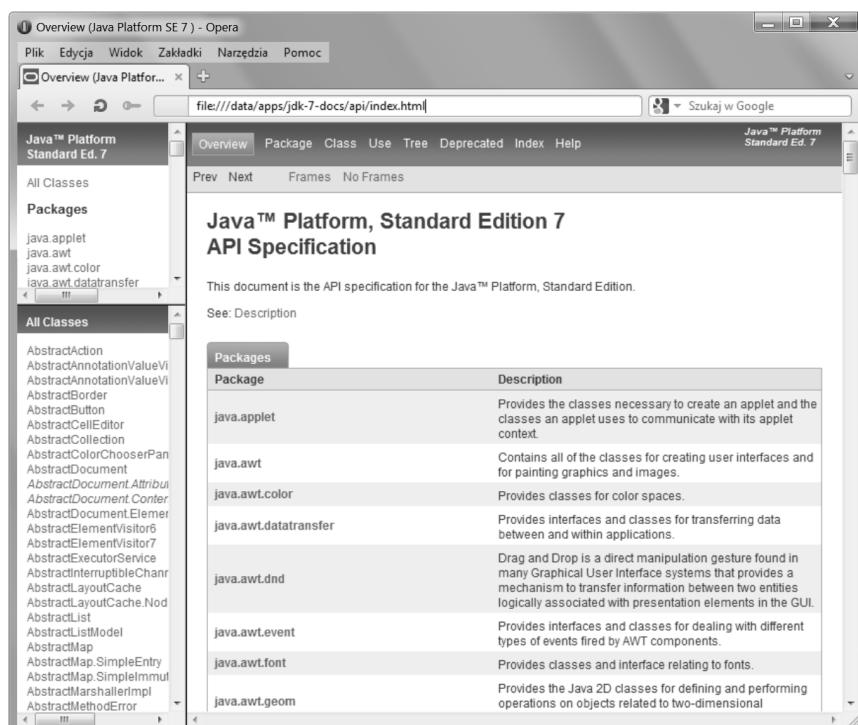
Usuwa wszystkie białe znaki z początku i końca łańcucha. Zwraca wynik jako nowy łańcuch.

3.6.8. Dokumentacja API w internecie

Jak się przed chwilą przekonaliśmy, klasa `String` zawiera mnóstwo metod. W bibliotekach standardowych jest kilka tysięcy klas, które zawierają dużo więcej metod. Zapamiętanie wszystkich przydatnych metod i klas jest niemożliwe. Z tego względu koniecznie trzeba się zapoznać z zamieszczoną w internecie dokumentacją API, w której można znaleźć informacje o każdej metodzie dostępnej w standardowej bibliotece. Dokumentacja API wchodzi też w skład pakietu JDK. Aby ją otworzyć, należy w przeglądarce wpisać adres pliku `docs/api/index.html` znajdującego się w katalogu instalacji JDK. Stronę tę przedstawia rysunek 3.2.

Rysunek 3.2.

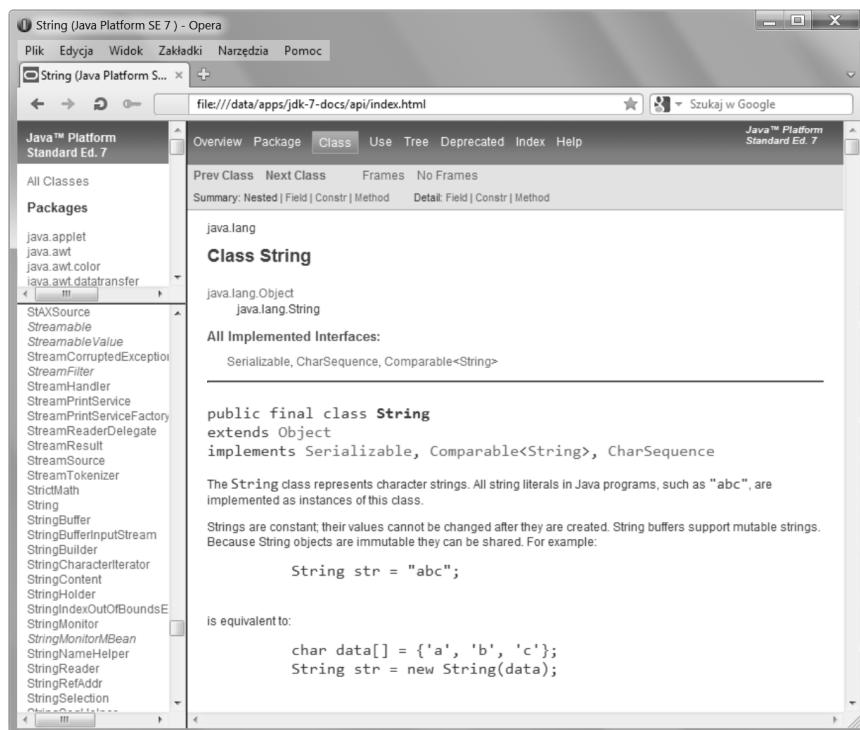
Trzyczęściowe okno dokumentacji API



Ekran jest podzielony na trzy części. W górnej ramce po lewej stronie okna znajduje się lista wszystkich dostępnych pakietów. Pod nią jest nieco większa ramka, która zawiera listy wszystkich klas. Kliknięcie nazwy jednej z klas powoduje wyświetlenie dokumentacji tej klasy w dużym oknie po prawej stronie (zobacz rysunek 3.3). Aby na przykład uzyskać dodatkowe informacje na temat metod dostępnych w klasie `String`, należy w drugiej ramce znaleźć odnośnik `String` i go kliknąć.

Rysunek 3.3.

Opis klasy String



Następnie za pomocą suwaka znajdujemy zestawienie wszystkich metod posortowanych w kolejności alfabetycznej (zobacz rysunek 3.4). Aby przeczytać dokładny opis wybranej metody, kliknij jej nazwę (rysunek 3.5). Jeśli na przykład klikniemy odnośnik *compareToIgnoreCase*, wyświetli się opis metody *compareToIgnoreCase*.



Dodaj stronę *docs/api/index.html* do ulubionych w swojej przeglądarce.

3.6.9. Składanie łańcuchów

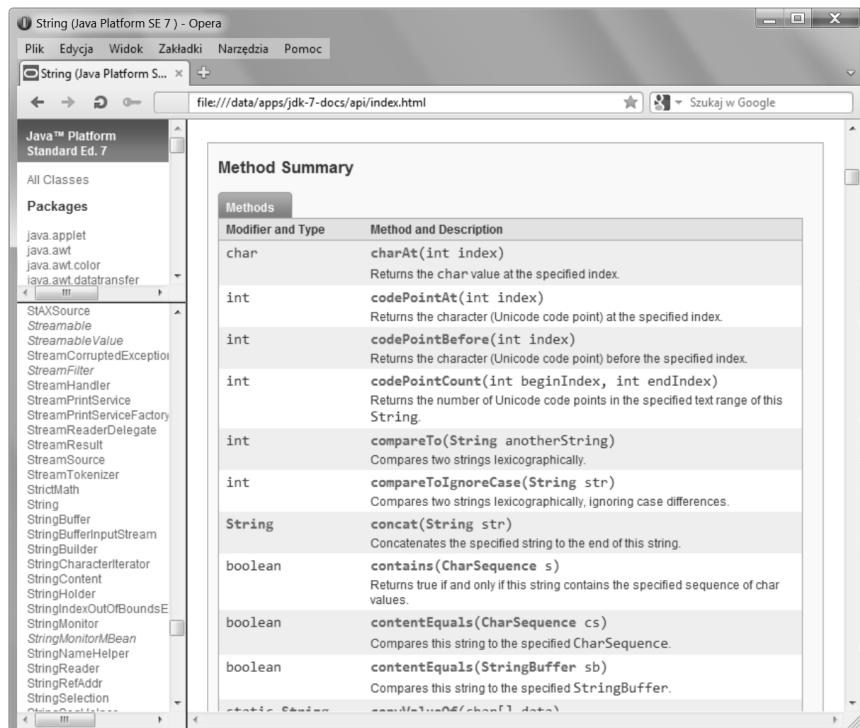
Czasami konieczne jest złożenie łańcucha z krótszych łańcuchów, takich jak znaki wprowadzane z klawiatury albo słowa zapisane w pliku. Zastosowanie konkatenacji do tego celu byłoby wyjściem bardzo nieefektywnym. Za każdym razem, gdy łączone są znaki, tworzony jest nowy obiekt klasy String. Zabiera to dużo czasu i pamięci. Klasa StringBuilder pozwala uniknąć tego problemu.

Aby złożyć łańcuch z wielu bardzo małych części, należy wykonać następujące czynności. Najpierw tworzymy pusty obiekt builder klasy StringBuilder (szczegółowy opis konstruktorów i operatora new znajduje się w rozdziale 4.):

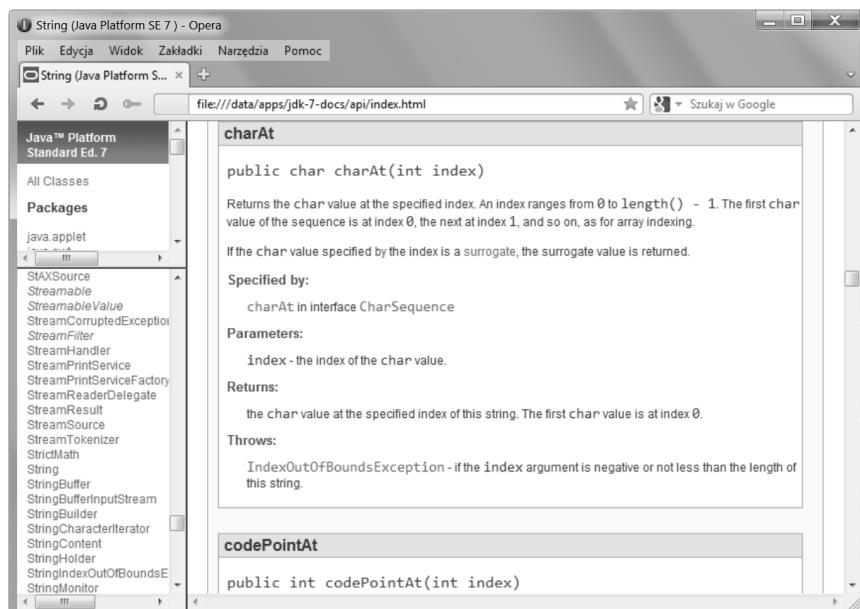
```
StringBuilder builder = new StringBuilder();
```

Rysunek 3.4.

Zestawienie metod klasy String

**Rysunek 3.5.**

Szczegółowy opis metod klasy String



Kolejne części dodajemy za pomocą metody append.

```
builder.append(ch);    // Dodaje jeden znak.  
builder.append(str);  // Dodaje łańcuch.
```

Po złożeniu łańcucha wywołujemy metodę `toString`. Zwróci ona obiekt klasy `String` zawierający sekwencję znaków znajdująca się w obiekcie `builder`.

```
String completedString = builder.toString();
```



Klasa `StringBuilder` została wprowadzona w JDK 5.0. Jej poprzedniczka o nazwie `StringBuffer` jest nieznacznie mniej wydajna, ale pozwala na dodawanie lub usuwanie znaków przez wiele wątków. Jeśli edycja łańcucha odbywa się w całości w jednym wątku (tak jest zazwyczaj), należy używać metody `StringBuilder`. API obu tych klas są identyczne.

Poniższy wyciąg z API przedstawia najczęściej używane metody dostępne w klasie `StringBuilder`.

java.lang.StringBuilder 5.0

- `StringBuilder()`
Tworzy pusty obiekt `builder`.
- `int length()`
Zwraca liczbę jednostek kodowych zawartych w obiekcie `builder` lub `buffer`.
- `StringBuilder append(String str)`
Dodaje łańcuch `c`.
- `StringBuilder append(char c)`
Dodaje jednostkę kodową `c`.
- `StringBuilder appendCodePoint(int cp)`
Dodaje współrzędną kodową, konwertując ją na jedną lub dwie jednostki kodowe.
- `void setCharAt(int i, char c)`
Ustawia *i*-tą jednostkę kodową na `c`.
- `StringBuilder insert(int offset, String str)`
Wstawia łańcuch, umieszczając jego początek na pozycji `offset`.
- `StringBuilder insert(int offset, char c)`
Wstawia jednostkę kodową na pozycji `offset`.
- `StringBuilder delete(int startIndex, int endIndex)`
Usuwa jednostki kodowe znajdujące się między pozycjami `startIndex` i `endIndex - 1`.
- `String toString()`
Zwraca łańcuch zawierający sekwencję znaków znajdująca się w obiekcie `builder` lub `buffer`.

3.7. Wejście i wyjście

Aby programy były bardziej interesujące, powinny przyjmować dane wejściowe i odpowiednio formatować dane wyjściowe. Oczywiście odbieranie danych od użytkownika w tworzonych obecnie programach odbywa się za pośrednictwem GUI. Jednak programowanie interfejsu wymaga znajomości wielu narzędzi i technik, które są nam jeszcze nieznane. Ponieważ naszym aktualnym priorytetem jest zapoznanie się z językiem programowania Java, poprzedzaniemy na razie na skromnych programach konsolowych. Programowanie GUI opisują rozdziały od 7. do 9.

3.7.1. Odbieranie danych wejściowych

Wiadomo już, że drukowanie danych do standardowego strumienia wyjściowego (tzn. do okna konsoli) jest łatwe. Wystarczy wywołać metodę `System.out.println`. Pobieranie danych ze standardowego strumienia wejściowego `System.in` nie jest już takie proste. Czytanie danych odbywa się za pomocą skanera będącego obiektem klasy `Scanner` przywiązanego do strumienia `System.in`:

```
Scanner in = new Scanner(System.in);
```

Operator `new` i konstruktory zostały szczegółowo omówione w rozdziale 4.

Następnie dane wejściowe odczytuje się za pomocą różnych metod klasy `Scanner`. Na przykład metoda `nextLine` czyta jeden wiersz danych:

```
System.out.print("Jak się nazywasz? ");
String name = in.nextLine();
```

W tym przypadku zastosowanie metody `nextLine` zostało podyktowane tym, że dane na wejściu mogą zawierać spacje. Aby odczytać jedno słowo (ograniczone spacjami), należy wywołać poniższą metodę:

```
String firstName = in.next();
```

Do wczytywania liczb całkowitych służy metoda `nextInt`:

```
System.out.print("Ile masz lat? ");
int age = in.nextInt();
```

Podobne działanie ma metoda `nextDouble`, z tym że dotyczy liczb zmiennoprzecinkowych.

Program przedstawiony na listingu 3.2 prosi użytkownika o przedstawienie się i podanie wieku, a następnie drukuje informację typu:

```
Witaj użytkowniku Łukasz. W przyszłym roku będziesz mieć 32 lata.
```

Listing 3.2. InputTest.java

```
import java.util.*;
/*

```

```
* Ten program demonstruje pobieranie danych z konsoli.
* @version 1.10 2004-02-10
* @author Cay Horstmann
*/
public class InputTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        // Pobranie pierwszej porcji danych.
        System.out.print("Jak się nazywasz? ");
        String name = in.nextLine();

        // Pobranie drugiej porcji danych.
        System.out.print("Ile masz lat? ");
        int age = in.nextInt();

        // Wydruk danych w konsoli.
        System.out.println("Witaj użytkowniku " + name + ". W przyszłym roku będziesz
        mieć " + (age + 1) + "lat.");
    }
}
```



Klasa Scanner nie nadaje się do odbioru haseł z konsoli, ponieważ wprowadzane dane są widoczne dla każdego. W Java SE 6 wprowadzono klasę Console, która służy właśnie do tego celu. Aby pobrać hasło, należy użyć poniższego kodu:

```
Console cons = System.console();
String username = cons.readLine("Nazwa użytkownika: ");
char[] passwd = cons.readPassword("Hasło: ");
```

Ze względów bezpieczeństwa hasło jest zwracane w tablicy znaków zamiast w postaci łańcucha. Po zakończeniu pracy z hasłem powinno się natychmiast nadpisać przechowującą je tablicę, zastępując obecne elementy jakimiś wartościami wypełniającymi (przetwarzanie tablic jest opisane w dalszej części tego rozdziału).

Przetwarzanie danych wejściowych za pomocą obiektu Console nie jest tak wygodne jak w przypadku klasy Scanner. Jednorazowo można wczytać tylko jeden wiersz danych. Nie ma metod umożliwiających odczyt pojedynczych słów lub liczb.

Należy także zwrócić uwagę na poniższy wiersz:

```
import java.util.*;
```

Znajduje się on na początku programu. Definicja klasy Scanner znajduje się w pakiecie `java.util`. Użycie jakiekolwiek klasy spoza podstawowego pakietu `java.lang` wymaga wykorzystania dyrektywy `import`. Pakiety i dyrektywy `import` zostały szczegółowo opisane w rozdziale 4.

java.util.Scanner 5.0

■ `Scanner(InputStream in)`

Tworzy obiekt klasy Scanner przy użyciu danych z podanego strumienia wejściowego.

- `String nextLine()`
Wczytuje kolejny wiersz danych.
- `String text()`
Wczytuje kolejne słowo (znakiem rozdzielającym jest spacja).
- `int nextInt()`
- `double nextDouble()`
Wczytuje i konwertuje kolejną liczbę całkowitą lub zmiennoprzecinkową.
- `boolean hasNext()`
Sprawdza, czy jest kolejne słowo.
- `boolean hasNextInt()`
- `boolean hasNextDouble()`
Sprawdza, czy dana sekwencja znaków jest liczbą całkowitą, czy liczbą zmiennoprzecinkową.

java.lang.System 1.0

- `static Console console()` **6**

Zwraca obiekt klasy `Console` umożliwiający interakcję z użytkownikiem za pośrednictwem okna konsoli, jeśli jest to możliwe, lub wartość `null` w przeciwnym przypadku. Obiekt `Console` jest dostępny dla wszystkich programów uruchomionych w oknie konsoli. W przeciwnym przypadku dostępność zależy od systemu.

java.io.Console 6

- `static char[] readPassword(String prompt, Object... args)`
- `static String readLine(String prompt, Object... args)`

Wyświetla łańcuch `prompt` i wczytuje wiersz danych z konsoli. Za pomocą parametrów `args` można podać argumenty formatowania, o czym mowa w następnym podrozdziale.

3.7.2. Formatowanie danych wyjściowych

Wartość zmiennej `x` można wydrukować w konsoli za pomocą instrukcji `System.out.print(x)`. Polecenie to wydrukuje wartość zmiennej `x` z największą liczbą cyfr niebędących zerami, które może pomieścić dany typ. Na przykład kod:

```
double x = 10000.0 / 3.0;  
System.out.print(x);
```

wydrukuje:

3333.333333333335

Problemy zaczynają się wtedy, gdy chcemy na przykład wyświetlić liczbę dolarów i centów.

W pierwotnych wersjach Javy formatowanie liczb sprawiało sporo problemów. Na szczęście w wersji Java SE 5 wprowadzono zasłużoną już metodę `printf` z biblioteki C. Na przykład wywołanie:

```
System.out.printf("%8.2f", x);
```

drukuję wartość zmiennej `x` w **polu o szerokości** 8 znaków i z dwoma miejscami po przecinku. To znaczy, że poniższy wydruk zawiera wiodącą spację i siedem widocznych znaków:

3333.33

Metoda `printf` może przyjmować kilka parametrów. Na przykład:

```
System.out.printf("Witaj. %s. W przyszłym roku będziesz mieć lat %d", name, age);
```

Każdy **specyfikator formatu**, który zaczyna się od znaku %, jest zastępowany odpowiadającym mu argumentem. **Znak konwersji** znajdujący się na końcu specyfikatora formatu określa typ wartości do sformatowania: `f` oznacza liczbę zmiennoprzecinkową, `s` łańcuch, a `d` liczbę całkowitą dziesiętną. Tabela 3.5 zawiera wszystkie znaki konwersji.

Dodatkowo można kontrolować wygląd sformatowanych danych wyjściowych za pomocą kilku **znaczników**. Tabela 3.6 przedstawia wszystkie znaczniki. Na przykład przecinek dodaje separator grup. To znaczy:

```
System.out.printf("%, .2f", 10000.0 / 3.0);
```

wydrukuje:

3 333.33

Można stosować po kilka znaczników naraz, na przykład zapis "%,(.2f" oznacza użycie separatorów grup i ujęcie liczb ujemnych w nawiasy.



Za pomocą znaku konwersji `s` można formatować dowolne obiekty. Jeśli obiekt taki implementuje interfejs `Formattable`, wywoływana jest jego metoda `formatTo`. W przeciwnym razie wywoływana jest metoda `toString` w celu przekonwertowania obiektu na łańcuch. Metoda `toString` opisana jest w rozdziale 5., a interfejsy w rozdziale 6.

Aby utworzyć sformatowany łańcuch, ale go nie drukować, należy użyć statycznej metody `String.format`:

```
String message = String.format("Witaj. %s. W przyszłym roku będziesz mieć lat %d",
    name, age);
```

Mimo że typ `Date` omawiamy szczegółowo dopiero w rozdziale 4., przedstawiamy krótki opis opcji metody `printf` do formatowania daty i godziny. Stosowany jest format dwuliterowy, w którym pierwsza litera to `t`, a druga jest jedną z liter znajdujących się w tabeli 3.7. Na przykład:

```
System.out.printf("%tc", new Date());
```

Wynikiem jest aktualna data i godzina w następującym formacie¹:

Pn lis 26 15:47:12 CET 2007

¹ Aby program zadziałał, na początku kodu źródłowego należy wstawić wiersz `import java.util.Date;` — przyp. tłum.

Tabela 3.5. Znaki konwersji polecenia printf

Znak konwersji	Typ	Przykład
d	Liczba całkowita dziesiętna	159
x	Liczba całkowita szesnastkowa	9f
o	Liczba całkowita ósemkowa	237
f	Liczba zmiennoprzecinkowa	15.9
e	Liczba zmiennoprzecinkowa w notacji wykładniczej	1.59e+01
g	Liczba zmiennoprzecinkowa (krótszy z formatów e i f)	–
a	Liczba zmiennoprzecinkowa szesnastkowa	0x1.fccdp3
s	Łańcuch	Witaj
c	Znak	H
b	Wartość logiczna	true
h	Kod mieszający	42628b2
tx	Data i godzina	Zobacz tabelę 3.7
%	Symbol procenta	%
n	Separator wiersza właściwy dla platformy	–

Tabela 3.6. Znaczniki polecenia printf

Flaga	Przeznaczenie	Przykład
+	Oznacza, że przed liczbami zawsze ma się znajdować znak.	+3333,33
spacja	Oznacza, że liczby nieujemne są poprzedzone dodatkową spacją.	3333,33
0	Oznacza dodanie początkowych zer.	003333,33
-	Oznacza, że pole ma być wyrównane do lewej.	3333,33
(Oznacza, że liczby ujemne mają być prezentowane w nawiasach.	(3333,33)
.	Oznacza, że poszczególne grupy mają być rozdzielane.	3 333,33
# (dla formatu f)	Oznacza, że zawsze ma być dodany przecinek dziesiętny.	3 333,
# (dla formatu x lub o)	Dodaje odpowiednio przedrostek 0x lub 0.	0xcafe
\$	Określa indeks argumentu do sformatowania. Na przykład %1\$d %1\$x drukuje tę samą liczbę w notacji dziesiętnej i szesnastkowej.	159 9F
<	Formatuje podobnie jak poprzednia specyfikacja. Na przykład zapis %d %<x wydrukuje liczbę w formacie dziesiętnym i szesnastkowym.	159 9F

Tabela 3.7. Znaki konwersji Date i Time

Znak konwersji	Typ	Przykład
C	Pełna data i godzina	Pn lis 26 15:47:12 CET 2007
F	Data w formacie ISO 8601	2007-11-26
D	Data w formacie stosowanym w USA (miesiąc/dzień/rok)	11/26/07
T	Godzina w formacie 24-godzinnym	15:25:10
r	Godzina w formacie 12-godzinnym	03:52:55 PM
R	Godzina w formacie 24-godzinnym, bez sekund	15:25
Y	Rok w formacie czterocyfrowym	2007
y	Dwie ostatnie cyfry roku (z wiodącymi zerami)	07
C	Dwie pierwsze cyfry roku (z wiodącymi zerami)	20
B	Pełna nazwa miesiąca	listopad
b lub h	Skrót nazwy miesiąca	lis
m	Dwie cyfry oznaczające numer miesiąca (z wiodącym zera)	02
d	Numer dnia miesiąca (z wiodącym zera)	09
e	Numer dnia miesiąca (bez wiodącego zera)	9
A	Pełna nazwa dnia	poniedziałek
a	Skrót nazwy dnia	Pn
j	Dzień roku w formacie trzycyfrowym (z wiodącymi zerami), od 001 do 366	069
H	Godzina w formacie dwucyfrowym (z wiodącym zera), od 00 do 23	18
k	Godzina w formacie dwucyfrowym (bez wiodącego zera), od 00 do 23	18
I	Godzina w formacie dwucyfrowym (z wiodącym zera), od 01 do 12	06
l	Godzina w formacie dwucyfrowym (bez wiodącego zera), od 1 do 12	6
M	Minuty w formacie dwucyfrowym (z wiodącym zera)	05
S	Sekundy w formacie dwucyfrowym (z wiodącym zera)	19
L	Milisekundy w formacie trzycyfrowym (z wiodącymi zerami)	046
N	Nanosekundy w formacie dziewięciocyfrowym (z wiodącymi zerami)	047000000

Tabela 3.7. Znaki konwersji Date i Time — ciąg dalszy

Znak konwersji	Typ	Przykład
P	Symbol oznaczający godziny przedpołudniowe i popołudniowe (wielkie litery)	PM
p	Symbol oznaczający godziny przedpołudniowe i popołudniowe (małe litery)	pm
z	Przesunięcie względem czasu GMT w standardzie RFC 822	+0100
Z	Strefa czasowa	CET
s	Liczba sekund, które upłynęły od daty 1970-01-01, 00:00:00 GMT	1196089646
Q	Liczba milisekund, które upłynęły od daty 1970-01-01, 00:00:00	1196089667265

Jak widać w tabeli 3.7, niektóre formaty zwracają tylko określona część daty, na przykład tylko dzień albo tylko miesiąc. Formatowanie każdej części daty oddzielnie byłoby nierozsądnym rozwiązaniem. Dlatego w łańcuchu formatującym można podać indeks argumentu, który ma być sformatowany. Indeks musi się znajdować bezpośrednio po symbolu % i kończyć się symbolem \$. Na przykład:

```
System.out.printf("%1$s %2$te %2$tB %2$tY", "Data:", new Date());
```

Wynik wykonania tego wyrażenia będzie następujący:

Data: luty 9. 2004

Ewentualnie można użyć flagi <. Oznacza ona, że ten sam argument co w poprzedniej specyfikacji formatu powinien zostać użyty ponownie. Poniższa instrukcja:

```
System.out.printf("%s %te %<tB %<tY", "Data:", new Date());
```

da taki sam wynik jak poprzedni fragment kodu.



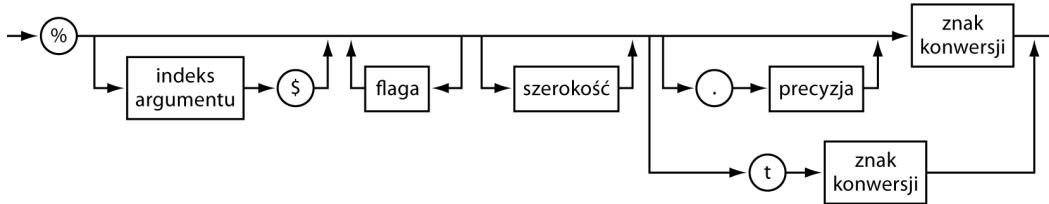
Wartości indeksów argumentów zaczynają się od 1, nie od 0; zapis %1\$... dotyczy pierwszego argumentu. W ten sposób zapobiegnięto mylению ich z flagą 0.

Przedstawione zostały wszystkie własności metody printf. Rysunek 3.6 prezentuje schemat opisujący składnię specyfikatorów formatu.



Niektóre z zasad formatowania są **związane z określona lokalizacją**. Na przykład w Niemczech separatorem dziesiętnym jest przecinek, a zamiast „Poniedziałek” wyświetla się „Montag”. Kontrola funkcji międzynarodowych programu została opisana w drugim tomie, w rozdziale 5.

specyfikator formatu:



Rysunek 3.6. Składnia specyfikatora format

3.7.3. Zapis i odczyt plików

Aby odczytać dane z pliku, należy utworzyć obiekt Scanner:

```
Scanner in = new Scanner(Paths.get("mojplik.txt"));
```

Jeśli nazwa pliku zawiera lewe ukośniki, należy pamiętać o zastosowaniu dla nich symboli zastępczych: "c:\\mojkatalog\\\\mojplik.txt".

Po wykonaniu tych czynności można odczytać zawartość pliku za pomocą metod klasy Scanner, które były opisywane wcześniej.

Aby zapisać dane do pliku, należy posłużyć się obiektem PrintWriter. Należy podać konstruktorem nazwę pliku:

```
PrintWriter out = new PrintWriter("mojplik.txt");
```

Jeśli plik nie istnieje, można użyć metod print, println lub printf, podobnie jak w przypadku drukowania do wyjścia System.out.



Obiekt Scanner można utworzyć przy użyciu parametru łańcuchowego, ale parametr ten zostanie zinterpretowany jako dane, a nie nazwa pliku. Jeśli na przykład napiszemy:

```
Scanner in = new Scanner("mojplik.txt"); // Błąd?
```

obiekt klasy Scanner będzie widział dane składające się z jedenastu znaków: 'm', 'o', 'j' itd. Istnieje duże prawdopodobieństwo, że autorowi kodu chodziło o coś innego.

Jasne jest zatem, że dostęp do plików jest równie łatwy jak używanie wejścia System.in oraz wyjścia System.out. Jest tylko jedno „ale”: jeśli obiekt klasy Scanner zostanie utworzony przy użyciu nazwy nieistniejącego pliku albo PrintWriter przy użyciu nazwy, której nie można utworzyć, wystąpi wyjątek. Dla kompilatora Javy wyjątki te mają większe znaczenie niż na przykład wyjątek dzielenia przez zero. Rozmaite techniki obsługi wyjątków zostały opisane w rozdziale 11. Na razie wystarczy, jeśli poinformujemy kompilator, że wiemy, iż istnieje możliwość wystąpienia wyjątku związanego z nieodnalezieniem pliku. Robimy to, dodając do metody main klauzulę throws:



Względne ścieżki do plików (np. *mojplik.txt*, *mojkatalog/mojplik.txt* lub *../mojplik.txt*) są lokalizowane względem katalogu, w którym uruchomiono maszynę wirtualną. Jeśli uruchomimy program z wiersza poleceń za pomocą polecenia:

```
java MyProg
```

katalogiem początkowym będzie aktualny katalog okna konsoli. W zintegrowanym środowisku programistycznym katalog początkowy jest określany przez IDE. Lokalizację tego katalogu można sprawdzić za pomocą poniższego wywołania:

```
String dir = System.getProperty("user.dir");
```

Jeśli nie możesz się połapać w lokalizacji plików, możesz zastosować ścieżki bezwzględne, takie jak "c:\\mojkatalog\\mojplik.txt" lub "/home/ja/mojkatalog/mojplik.txt".

```
public static void main(String[] args) throws FileNotFoundException
{
    Scanner in = new Scanner(Paths.get("mojplik.txt"));
    ...
}
```

Wiemy już, jak odczytywać i zapisywać pliki zawierające dane tekstowe. Bardziej zaawansowane zagadnienia, jak obsługa różnych standardów kodowania znaków, przetwarzanie danych binarnych, odczyt katalogów i zapis plików archiwum zip, zostały opisane w rozdziale 1. drugiego tomu.



Przy uruchamianiu programu w wierszu poleceń można użyć właściwej danemu systemowi składni przekierowywania w celu dodania dowolnego pliku do wejścia `System.in` i wyjścia `System.out`:

```
java MyProg < mojplik.txt > output.txt
```

Dzięki temu nie trzeba się zajmować obsługi wyjątku `FileNotFoundException`.

java.util.Scanner 5.0

■ `Scanner(Path p)`

Tworzy obiekt klasy `Scanner`, który wczytuje dane z podanej ścieżki.

■ `Scanner(String data)`

Tworzy obiekt klasy `Scanner`, który wczytuje dane z podanego łańcucha.

java.io.PrintWriter 1.1

■ `PrintWriter(String fileName)`

Tworzy obiekt `PrintWriter`, który zapisuje dane do pliku o podanej nazwie.

java.nio.file.Paths 7.0

■ `static Path get(String pathname)`

Tworzy obiekt `Path` ze ścieżki o podanej nazwie.

3.8. Przepływ sterowania

W Javie, podobnie jak w każdym języku programowania, do kontroli przepływu sterowania używa się instrukcji warunkowych i pętli. Zaczniemy od instrukcji warunkowych, aby później przejść do pętli. Na zakończenie omówimy nieco nieporęczną instrukcję switch, która może się przydać, gdy konieczne jest sprawdzenie wielu wartości jednego wyrażenia.



Instrukcje sterujące Javy są niemal identyczne z instrukcjami sterującymi w C++. Różnica polega na tym, że w Javie nie ma instrukcji go to, ale jest wersja instrukcji break z etykietą, której można użyć do przerwania działania zagnieżdżonej pętli (w takich sytuacjach, w których w C prawdopodobnie użylibyśmy instrukcji go to). Nareszcie dodano wersję pętli for, która nie ma odpowiednika w językach C i C++. Jest podobna do pętli foreach w C#.

3.8.1. Zasięg blokowy

Zanim przejdziemy do instrukcji sterujących, musimy poznać pojęcie **blok**.

Blok, czyli instrukcja złożona, to dowolna liczba instrukcji Javy ujętych w nawiasy klamrowe. Blok określa zasięg zmiennych. Bloki można **zagnieździć** w innych blokach. Poniżej znajduje się blok zagnieżdzony w bloku metody main:

```
public static void main(String[] args)
{
    int n;
    .
    .
    {
        int k;
        .
        .
    }           // Definicja zmiennej k jest dostępna tylko do tego miejsca.
}
```

Nie można zdefiniować dwóch zmiennych o takiej samej nazwie w dwóch zagnieżdzonych blokach. Na przykład poniższy kod jest błędny i nie można go skompilować:

```
public static void main(String[] args)
{
    int n;
    .
    .
    {
        int k;
        int n; // Błąd — nie można ponownie zdefiniować zmiennej n w bloku wewnętrznym.
        .
    }
}
```



W C++ można wewnątrz bloku ponownie zdefiniować zmienną wcześniejszą zdefiniowaną na zewnątrz tego bloku. Ta definicja wewnętrzna przesłania wtedy definicję zewnętrzna. Może to być jednak źródłem błędów i z tego powodu operacja taka nie jest dozwolona w Javie.

3.8.2. Instrukcje warunkowe

W Javie instrukcja warunkowa ma następującą postać:

```
if (warunek) instrukcja
```

Warunek musi być umieszczony w nawiasach okrągłych.

Podobnie jak w wielu językach, w Javie często po spełnieniu jednego warunku konieczne jest wykonanie wielu instrukcji. W takim przypadku należy zastosować **blok instrukcji** w następującej postaci:

```
{
    instrukcja1;
    instrukcja2;
}
```

Na przykład:

```
if (yourSales >= target)
{
    performance = "Średnio";
    bonus = 100;
}
```

Wszystkie instrukcje znajdujące się pomiędzy klamrami zostaną wykonane, jeśli wartość zmiennej `yourSales` będzie większa lub równa wartości zmiennej `target` (zobacz rysunek 3.7).



Blok (czasami nazywany **instrukcją złożoną**) umożliwia wykonanie więcej niż jednej instrukcji we wszystkich miejscach, gdzie przewiduje się użycie instrukcji.

Bardziej ogólna postać instrukcji warunkowej w Javie jest następująca (zobacz rysunek 3.8):

```
if (warunek) instrukcja1 else instrukcja2
```

Na przykład:

```
if (yourSales >= target)
{
    performance = "Średnio";
    bonus = 100 + 0.01 * (yourSales - target);
}
else
{
    performance = "Słabo";
    bonus = 0;
}
```

Rysunek 3.7.

Diagram przepływu sterowania instrukcji if

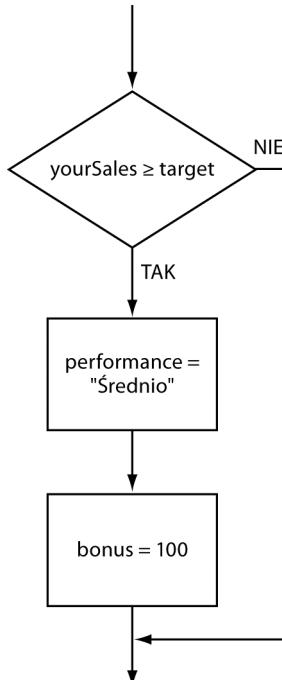
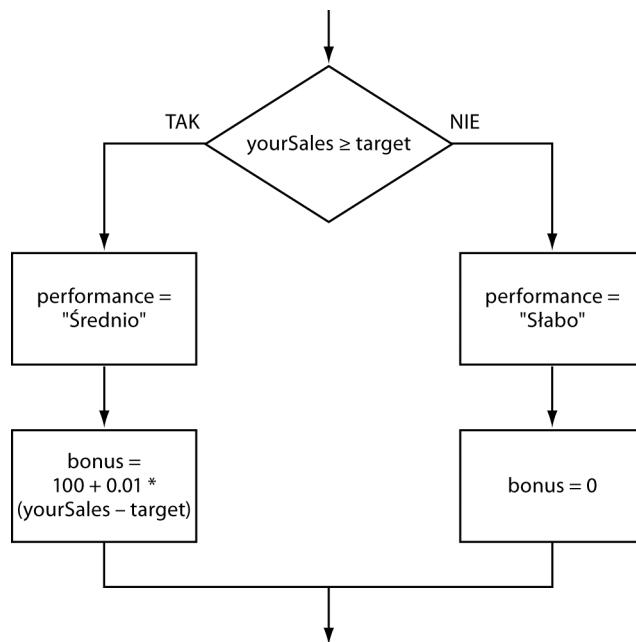
**Rysunek 3.8.**

Diagram przepływu sterowania instrukcji if-else



Stosowanie `else` jest opcjonalne. Dane `else` zawsze odpowiadają najbliższemu poprzedzającemu je `if`. W związku z tym w instrukcji:

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

`else` odpowiada drugiemu `if`. Oczywiście dobrze było zastosować klamry, aby kod był bardziej czytelny:

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Często stosuje się kilka instrukcji `else-if` jedna po drugiej (zobacz rysunek 3.9). Na przykład:

```
if (yourSales >= 2 * target)
{
    performance = "Znakomicie";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Nieźle";
    bonus = 500;
}
else if (yourSales >= target)
{
    performance = "Średnio";
    bonus = 100;
}
else
{
    System.out.println("Jesteś zwolniony");
}
```

3.8.3. Pętle

Pętla `while` wykonuje instrukcję (albo blok instrukcji) tak długo, jak długo warunek ma wartość `true`. Ogólna postać instrukcji `while` jest następująca:

```
while (warunek) instrukcja
```

Instrukcje pętli `while` nie zostaną nigdy wykonane, jeśli warunek ma wartość `false` na początku (zobacz rysunek 3.10).

Program z listingu 3.3 oblicza, ile czasu trzeba składać pieniędze, aby dostać określona emeryturę, przy założeniu, że każdego roku wpłacana jest taka sama kwota, i przy określonej stopie oprocentowania wpłaconych pieniędzy.

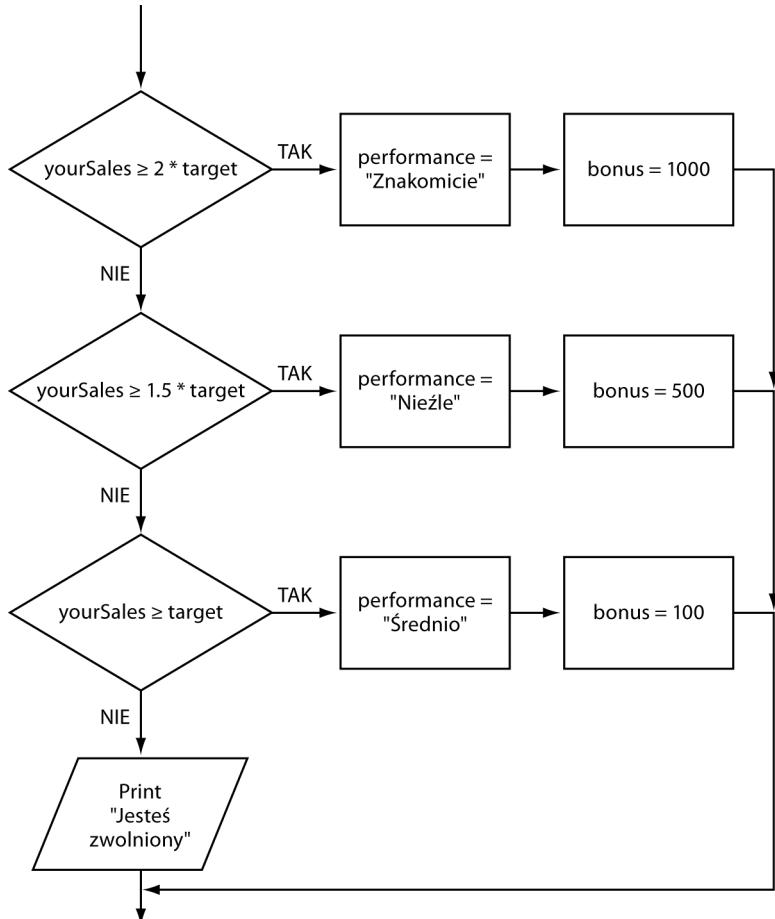
W ciele pętli zwiększamy licznik i aktualizujemy bieżącą kwotę uzbieranych pieniędzy, aż ich suma przekroczy wyznaczoną kwotę.

```
while (balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    years++;
}
System.out.println(years + "lat.");
```

(Nie należy ufać temu programowi przy planowaniu emerytury. Pominięto w nim kilka szczegółów, takich jak inflacja i przewidywana długość życia).

Rysunek 3.9.

Diagram przepływu sterowania instrukcji if-else if (wiele odgałęzień)



Pętla while sprawdza warunek na samym początku działania. W związku z tym jej instrukcje mogą nie zostać wykonane ani razu. Aby mieć pewność, że instrukcje zostaną wykonane co najmniej raz, sprawdzanie warunku trzeba przenieść na sam koniec. Do tego służy pętla do-while. Jej składnia jest następująca:

do instrukcja while (warunek)

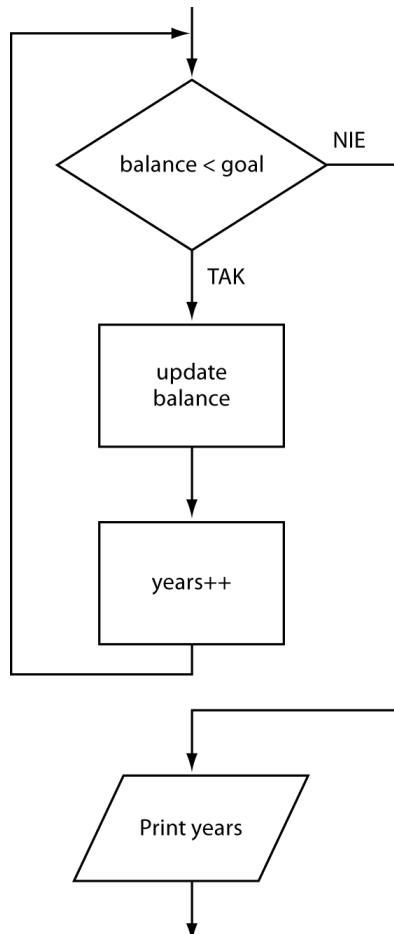
Ta instrukcja najpierw wykonuje instrukcję (która zazwyczaj jest blokiem instrukcji), a dopiero potem sprawdza warunek. Następnie znów wykonuje instrukcję i sprawdza warunek itd. Kod na listingu 3.4 oblicza nowe saldo na koncie emerytalnym, a następnie pyta, czy jesteśmy gotowi przejść na emeryturę:

```

do
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    year++;
    // Drukowanie aktualnego stanu konta.
    .
  
```

Rysunek 3.10.

Diagram przepływu sterowania instrukcji while



// Zapytanie o gotowość do przejścia na emeryturę i pobranie danych.

```

}
while (input.equals("N"));
```

Pętla jest powtarzana, dopóki użytkownik podaje odpowiedź N (zobacz rysunek 3.11). Ten program jest dobrym przykładem pętli, która musi być wykonana co najmniej jeden raz, ponieważ użytkownik musi zobaczyć stan konta, zanim podejmie decyzję o przejściu na emeryturę.

Listing 3.3. Retirement.java

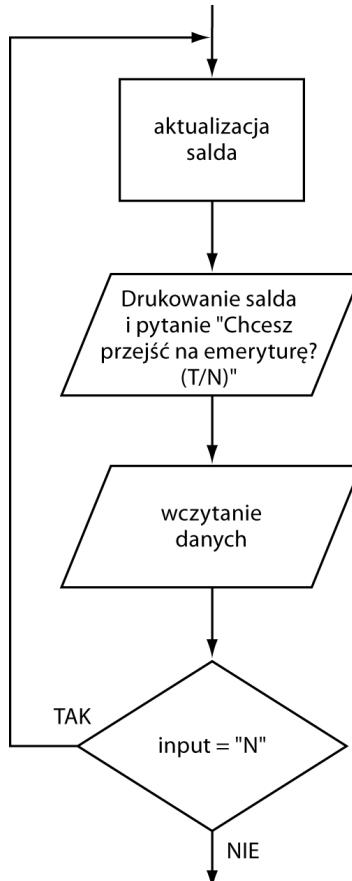
```

import java.util.*;

/**
 * Ten program demonstruje sposób użycia pętli <code>while</code>.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class Retirement
{
```

Rysunek 3.11.

Diagram
przepływu
sterowania
instrukcji do-while



```

public static void main(String[] args)
{
    // Wczytanie danych.
    Scanner in = new Scanner(System.in);

    System.out.print("Ile pieniędzy potrzebujesz, aby przejść na emeryturę? ");
    double goal = in.nextDouble();

    System.out.print("Ile pieniędzy rocznie będziesz wpłacać? ");
    double payment = in.nextDouble();

    System.out.print("Stopa procentowa w %: ");
    double interestRate = in.nextDouble();

    double balance = 0;
    int years = 0;

    // Aktualizacja salda konta, jeśli cel nie został osiągnięty.
    while (balance < goal)
    {
        // Dodanie tegorocznych płatności i odsetek.
        balance += payment;
        double interest = balance * interestRate / 100;
    }
}
  
```

```

        balance += interest;
        years++;
    }

    System.out.println("Możesz przejść na emeryturę za " + years + " lat.");
}
}

```

Listing 3.4. Retirement2.java

```

import java.util.*;

/**
 * Ten program demonstruje użycie pętli <code>do/while</code>.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class Retirement2
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Ile pieniędzy rocznie będziesz wpłacać? ");
        double payment = in.nextDouble();

        System.out.print("Stopa oprocentowania w %: ");
        double interestRate = in.nextDouble();

        double balance = 0;
        int year = 0;

        String input;

        // Aktualizacja stanu konta, kiedy użytkownik nie jest gotowy do przejścia na emeryturę.
        do
        {
            // Dodanie tegorocznych płatności i odsetek.
            balance += payment;
            double interest = balance * interestRate / 100;
            balance += interest;

            year++;

            // Drukowanie aktualnego stanu konta.
            System.out.printf("Po upływie %d lat stan twojego konta wyniesie %.2f%n",
                year, balance);

            // Zapytanie o gotowość do przejścia na emeryturę i pobranie danych.
            System.out.print("Chcesz przejść na emeryturę? (T/N) ");
            input = in.next();
        }
        while (input.equals("N"));
    }
}

```

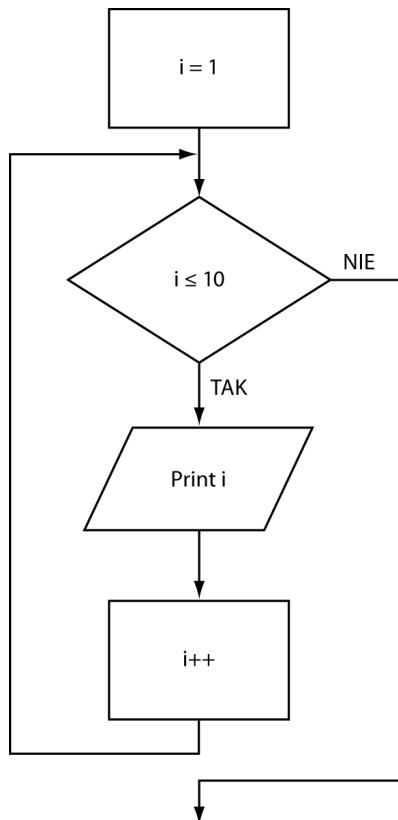
3.8.4. Pętle o określonej liczbie powtórzeń

Liczba iteracji instrukcji `for` jest kontrolowana za pomocą licznika lub jakieś innej zmiennej, której wartość zmienia się po każdym powtórzeniu. Z rysunku 3.12 wynika, że poniższa pętla drukuje na ekranie liczby od 1 do 10.

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

Rysunek 3.12.

Diagram
przepływu
sterowania
pętli `for`



Na pierwszym miejscu z reguły znajduje się inicjacja licznika. Drugie miejsce zajmuje warunek, który jest sprawdzany przed każdym powtórzeniem instrukcji pętli. Na trzeciej pozycji umieszczamy informację na temat sposobu zmiany wartości licznika.

Mimo iż w Javie, podobnie jak w C++, w różnych miejscach pętli `for` można wstawić prawie każde wyrażenie, niepisana zasada głosi, że do dobrego stylu należy, aby w tych miejscach inicjować, sprawdzać i zmieniać wartość jednej zmiennej. Nie stosując się do tej reguły, można napisać bardzo zagmatwane pętle.

Jednak nawet w granicach dobrego stylu programowania można sobie pozwolić na wiele. Można na przykład utworzyć pętlę zmniejszającą licznik:

```
for (int i = 10; i > 0; i--)
    System.out.println("Odliczanie . . . " + i);
System.out.println("Start!");
```



Należy zachować szczególną ostrożność przy porównywaniu w pętli liczb zmienno-przecinkowych. Pętla for w takiej postaci:

```
for (double x = 0; x != 10; x += 0.1) . . .
```

może się nigdy nie skończyć. Wartość końcowa nie zostanie osiągnięta ze względu na błąd związany z zaokrąglaniem. Na przykład w powyższej pętli wartość x przeskoczy z wartości 9.99999999999998 na 10.09999999999998, ponieważ liczba 0,1 nie ma dokładnej reprezentacji binarnej.

Zmienna zadeklarowana na pierwszej pozycji w pętli for ma zasięg do końca ciała tej pętli.

```
for (int i = 1; i <= 10; i++)
{
    .
}
// Tutaj zmienna i już nie jest dostępna.
```

Innymi słowy, wartość zmiennej zadeklarowanej w wyrażeniu pętli for nie jest dostępna poza tą pętlą. W związku z tym, aby móc użyć wartości licznika pętli poza tą pętlą, trzeba go zadeklarować poza jej nagłówkiem!

```
int i;
for (i = 1; i <= 10; i++)
{
    .
}
// Zmienna i tutaj też jest dostępna.
```

Z drugiej jednak strony w kilku pętlach for można zdefiniować zmienną o takiej samej nazwie:

```
for (int i = 1; i <= 10; i++)
{
    .
}
for (int i = 11; i <= 20; i++) // W tym miejscu dozwolona jest ponowna deklaracja zmiennej
i.
{
    .
}
```

Pętla for jest krótszym sposobem zapisu pętli while. Na przykład:

```
for (int i = 10; i > 0; i--)
    System.out.println("Odliczanie... " + i);
```

można zapisać następująco:

```
int i = 10;
while (i > 0)
{
    System.out.println("Odliczanie... " + i);
    i--;
}
```

Listing 3.5 przedstawia typowy przykład zastosowania pętli for.

Ten program oblicza szanse wygrania na loterii. Jeśli na przykład loteria polega na wybraniu sześciu liczb z przedziału 1 – 50, to istnieje $(50*49*48*47*46*45)/(1*2*3*4*5*6)$ możliwych kombinacji, co oznacza, że nasze szanse są jak 1 do 15 890 700. Powodzenia!

W ogólnym przypadku losowania k liczb ze zbioru n istnieje:

$$\frac{n * (n - 1) * (n - 2) * \dots * (n - k + 1)}{1 * 2 * 3 * \dots * k}$$

możliwych wyników. Poniższa pętla for oblicza tę wartość:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
    lotteryOdds = lotteryOdds * (n - i + 1) / i;
```



W sekcji 3.10.1 znajduje się opis uogólnionej pętli for (zwanej także pętlą typu for each), która została dodana w wersji Java SE 5.

Listing 3.5. LotteryOdds.java

```
import java.util.*;

/**
 * Ten program demonstruje zastosowanie pętli <code>for</code>.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryOdds
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Ile liczb ma być wylosowanych? ");
        int k = in.nextInt();

        System.out.print("Jaka jest górna granica przedziału losowanych liczb? ");
        int n = in.nextInt();

        /*
         * Obliczanie współczynnika dwumianowego  $n * (n - 1) * (n - 2) * \dots * (n - k + 1) / (1 * 2 * 3 * \dots * k)$ 
         */

        int lotteryOdds = 1;
        for (int i = 1; i <= k; i++)
            lotteryOdds = lotteryOdds * (n - i + 1) / i;

        System.out.println("Twoje szanse to 1 do " + lotteryOdds + ". Powodzenia!");
    }
}
```

3.8.5. Wybór wielokierunkowy — instrukcja switch

W sytuacjach gdy jest dużo opcji do wyboru, instrukcja warunkowa `if-else` może być mało efektywna. Dlatego w Javie udostępniono instrukcję `switch`, która niczym nie różni się od swojego pierwotnego w językach C i C++.

Na przykład do utworzenia systemu menu zawierającego cztery opcje, jak ten na rysunku 3.13, można użyć kodu podobnego do tego poniżej:

```
Scanner in = new Scanner(System.in);
System.out.print("Wybierz opcję (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        // Nieprawidłowe dane.
        . . .
        break;
}
```

Wykonywanie programu zaczyna się od etykiety `case`, która pasuje do wybranej opcji, i jest kontynuowane do napotkania instrukcji `break` lub końca instrukcji `switch`. Jeśli żadna z etykiet nie zostanie dopasowana, nastąpi wykonanie części oznaczonej przez etykietę `default` — jeśli taka istnieje.

Etykiety `case` mogą być:

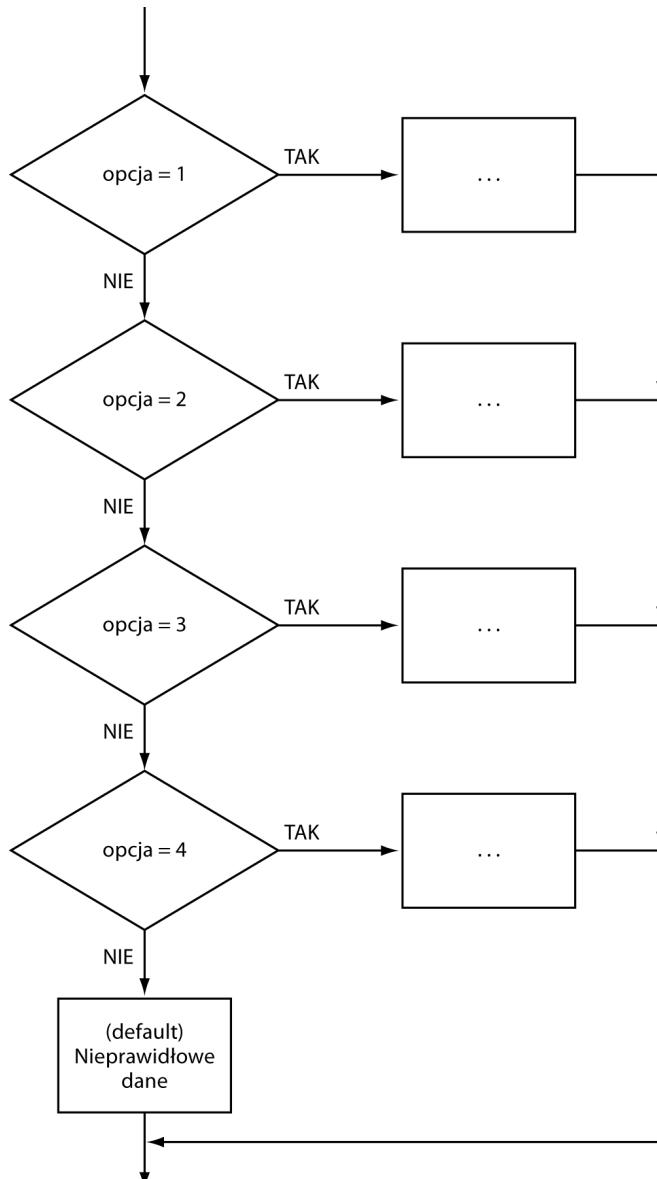
- wyrażeniami stałymi typu `char`, `byte`, `short` lub `int` (oraz odpowiednich klas opakowujących: `Character`, `Byte`, `Short` i `Integer` — ich opis znajduje się w rozdziale 4.);
- stałymi wyliczeniowymi;
- łańcuchami od Java SE 7.

Na przykład:

```
String input = . . .;
switch (input.toLowerCase())
{
    case "tak": // OK od Java SE 7
```

Rysunek 3.13.

Diagram
przepływu
sterowania
instrukcji switch



```

    break;
}

```

Używając instrukcji switch ze stałymi wyliczeniowymi, nie ma konieczności podawania nazwy wyliczenia w każdej etykiecie — jest ona pobierana domyślnie z wartości switch. Na przykład:



Istnieje ryzyko, że zostanie uruchomionych kilka opcji. Jeśli przez przypadek na końcu jednej z opcji nie znajdzie się instrukcja break, sterowanie zostanie przekazane do kolejnej etykiety case! Taki sposób działania jest niebezpieczny i często prowadzi do błędów. Z tego powodu nigdy nie używamy instrukcji case w swoich programach.

Jeśli jednak czujesz do instrukcji switch większą sympatię niż my, możesz kompilować swoje programy z użyciem opcji -Xlint:fallthrough:

```
javac -Xlint:fallthrough Test.java
```

Dzięki temu ustawieniu kompilator będzie zgłaszał wszystkie przypadki alternatyw niezawierających na końcu instrukcji break.

Gdy będziesz chciał wykonać bloki case po kolej, oznacz otaczającą je metodę anotacją @SuppressWarnings("fallthrough"). Dzięki temu dla tej metody nie będą zgłoszane ostrzeżenia. (Anotacje to technika przekazywania informacji do kompilatora lub innego narzędzia przetwarzającego kod źródłowy Java lub pliki klas. Ich szczegółowy opis znajduje się w rozdziale 13. drugiego tomu).

```
Size sz = . . .;
switch (sz)
{
    case SMALL: // Nie trzeba było pisać Size.SMALL.
        . . .
        break;
    . . .
}
```

3.8.6. Instrukcje przerywające przepływ sterowania

Mimo że projektanci języka Java zarezerwowali słowo goto, nie zdecydowali się wcielić go do języka. Instrukcje goto są uważane za wyznacznik słabego stylu programowania. Zdaniem niektórych programistów kampania skierowana przeciwko instrukcji goto jest przesadzona (zobacz słynny artykuł Donald'a E. Knutha pod tytułem *Structured Programming with goto statements*). Ich zdaniem stosowanie instrukcji goto bez ograniczeń może prowadzić do wielu błędów, ale użycie jej od czasu do czasu w celu **wyjścia z pętli** może być korzystne. Projektanci Javy przychylili się do tego stanowiska i dodali nową instrukcję break z etykietą.

Przyjrzyjmy się najpierw instrukcji break bez etykiety. Tej samej instrukcji break, za pomocą której wychodzi się z instrukcji switch, można użyć do przerwania działania pętli. Na przykład:

```
while (years <= 100)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance >= goal) break;
    years++;
}
```

Wyjście z pętli nastąpi, kiedy wartość znajdującej się na samej górze pętli zmiennej `years` przekroczy 100 albo znajdująca się w środku zmiennej `balance` będzie miała wartość większą lub równą `goal`. Oczywiście tę samą wartość zmiennej `years` można by było obliczyć bez użycia instrukcji `break`:

```
while (years <= 100 && balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance < goal)
        years++;
}
```

Należy jednak zauważyc, że wyrażenie sprawdzające `balance < goal` jest w tej wersji użyte dwukrotnie. Aby uniknąć tego powtórzenia, niektórzy programiści stosują instrukcję `break`.

W Javie dostępna jest też **instrukcja break z etykietą** (brak jej natomiast w języku C++), która umożliwia wyjście z kilku zagnieżdzonych pętli. Czasami w głęboko zagnieżdzonych pętlach dzieją się dziwne rzeczy. W takiej sytuacji najlepiej jest wyjść całkiem na zewnątrz. Zaprogramowanie takiego działania za pomocą dodatkowych warunków w różnych testach pętli jest rozwiązaniem mało wygodnym.

Poniżej znajduje się przykładowy kod prezentujący działanie instrukcji `break`. Należy zauważyc, że etykieta musi się znajdować przed najbardziej zewnętrzna pętlą, z której chcemy wyjść. Ponadto po etykiecie w tym miejscu musi się znajdować dwukropek.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .)      // Ta pętla jest opatrzona etykietą.
{
    .
    .
    for (. . .)      // Ta zagnieżdzona pętla nie ma etykiety.
    {
        System.out.print("Podaj liczbę >= 0: ");
        n = in.nextInt();
        if (n < 0)    // To nie powinno mieć miejsca — nie można kontynuować.
            break read_data;
        // Wyjście z pętli z etykietą read_data.

    }
}
// Ta instrukcja jest wykonywana bezpośrednio po przerwaniu pętli.
if (n < 0)          // Sprawdzenie, czy ma miejsce niepożądana sytuacja.
{
    // Obsługa niechcianej sytuacji.
}
else
{
    // Wykonywanie w normalnym toku.
}
```

Jeśli zostaną podane nieprawidłowe dane, instrukcja break z etykietą przeniesie sterowanie do miejsca bezpośrednio za blokiem opatrzonym tą etykietą. Następnie, tak jak w każdym przypadku użycia instrukcji break, trzeba sprawdzić, czy wyjście z pętli nastąpiło w toku normalnego działania, czy zostało spowodowane przez instrukcję break.



Co ciekawe, etykietę można dodać do każdej instrukcji, nawet instrukcji warunkowej if i instrukcji blokowej:

```
etykieta:
{
    ...
    if (warunek) break etykieta;    // Wychodzi z bloku.
    ...
}
// Przechodzi do tego miejsca, jeśli zostanie wykonana instrukcja break.
```

W związku z tym, jeśli tęsknisz za instrukcją goto i możesz umieścić blok bezpośrednio przed miejscem, do którego ma nastąpić przejście, możesz użyć instrukcji break! Oczywiście nie polecamy tej metody programowania. Zauważ też, że przejście jest możliwe tylko w jedną stronę — nie można **wskoczyć do bloku**.

Na zakończenie została jeszcze instrukcja continue, która podobnie jak instrukcja break zmienia normalny przepływ sterowania. Instrukcja continue przenosi sterowanie do nagłówka najgłębiej zagnieżdzonej pętli. Na przykład:

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
    System.out.print("Podaj liczbę: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n;    // Wyrażenie nie zostanie wykonane, jeśli n < 0.
}
```

Jeśli wartość zmiennej n jest mniejsza od 0, instrukcja continue powoduje natychmiastowe przejście do nagłówka pętli, nie dopuszczając do wykonania reszty instrukcji w bieżącej iteracji.

Instrukcja continue użыта w pętli for powoduje przejście do części aktualizującej wartość zmiennej w nagłówku tej pętli. Przyjrzyjmy się następującemu przykładowi:

```
for (count = 1; count <= 100; count++)
{
    System.out.print("Podaj liczbę (-1 kończy działanie programu): ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n;    // Wyrażenie nie zostanie wykonane, jeśli n < 0.
}
```

Jeśli n < 0, instrukcja continue powoduje przekazanie sterowania do instrukcji i++.

Istnieje także wersja instrukcji continue z etykietą, która powoduje przejście do nagłówka pętli z odpowiednią etykietą.



Wielu programistów myli instrukcje `break` i `continue`. Ich stosowanie nie jest obowiązkowe i to, co można osiągnąć przy ich użyciu, da się zawsze uzyskać w inny sposób. W tej książce nigdy nie używamy instrukcji `break` i `continue`.

3.9. Wielkie liczby

Jeśli precyza podstawowych typów całkowitoliczbowych i zmiennoprzecinkowych okaże się niezadowalająca, można zrobić użytek z klas dostępnych w pakietach `java.math`: `BigInteger` i `BigDecimal`. Klasy te umożliwiają działania na liczbach składających się z dowolnej liczby cyfr. Klasa `BigInteger` umożliwia wykonywanie działań arytmetycznych o dowolnej precyzyji na liczbach całkowitych, a klasa `BigDecimal` jest jej odpowiednikiem dla liczb zmiennoprzecinkowych.

Do konwersji zwykłych liczb na wielkie służy statyczna metoda `valueOf`:

```
BigInteger a = BigInteger.valueOf(100);
```

Niestety w działaniach na wielkich liczbach nie można używać dobrze nam znanych operatorów arytmetycznych, jak `+` czy `*`. Zamiast nich trzeba używać odpowiednich metod, jak `add` i `multiply`, dostępnych w klasach wielkich liczb:

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```



W przeciwieństwie do języka C++, Java nie umożliwia przeciążania operatorów. Nie da się z punktu widzenia programisty przeciążyć operatorów `+` i `*`, aby wykonywały działania właściwe metodom `add` i `multiply` dostępnym w klasie `BigInteger`. Projektanci Javy przeciążyli operator `+`, dzięki czemu można łączyć łańcuchy. Nie zdecydowali się jednak na przeciążenie pozostałych operatorów ani nie pozostawili takiej możliwości programistom.

Listing 3.6 przedstawia zmodyfikowaną wersję programu loteryjnego z listingu 3.5. W tej wersji działa ona także po podaniu bardzo dużych liczb. Jeśli na przykład loteria polega na wyborze 60 liczb ze zbioru 1 – 490, program ten poinformuje nas, że nasze szanse wynoszą 1 do 716 395 843 461 995 557 415 116 222 540 092 933 411 717 612 789 263 493 493 351 013 459 481 104 668 848. Powodzenia!

Program z listingu 3.5 obliczał wartość następującego wyrażenia:

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

Przy użyciu wielkich liczb odpowiednikiem tej instrukcji jest poniższa instrukcja:

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(BigInteger.
➥valueOf(i));
```

Listing 3.6. BigIntegerTest.java

```

import java.math.*;
import java.util.*;

/**
 * Ten program wykorzystuje wielkie liczby do obliczenia szans wygrania na loterii.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class BigIntegerTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Ile liczb ma być wylosowanych? ");
        int k = in.nextInt();

        System.out.print("Jaka jest górna granica przedziału losowanych liczb? ");
        int n = in.nextInt();

        /*
         * Obliczanie współczynnika dwumianowego  $n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)$ 
         */

        BigInteger lotteryOdds = BigInteger.valueOf(1);

        for (int i = 1; i <= k; i++)
            lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(
                BigInteger.valueOf(i));

        System.out.println("Twoje szanse to 1 do " + lotteryOdds + ". Powodzenia!");
    }
}

```

java.math.BigInteger **1.1**

- BigInteger add(BigInteger other)
- BigInteger subtract(BigInteger other)
- BigInteger multiply(BigInteger other)
- BigInteger divide(BigInteger other)
- BigInteger mod(BigInteger other)

Zwraca sumę, różnicę, iloczyn, iloraz i resztę liczb BigInteger i other.

- int compareTo(BigInteger other)

Zwraca wartość 0, jeśli liczba BigInteger jest równa liczbie other, wartość ujemną, jeśli liczba BigInteger jest mniejsza od liczby other, lub liczbę dodatnią w przeciwnym przypadku.

- static BigInteger valueOf(long x)

Zwraca wielką liczbę o wartości x.

java.math.BigDecimal **1.1**

- BigDecimal add(BigDecimal other)
- BigDecimal subtract(BigDecimal other)
- BigDecimal multiply(BigDecimal other)
- BigDecimal divide(BigDecimal other, RoundingMode mode) **5.0**

Zwraca sumę, różnicę, iloczyn, iloraz i resztę liczb BigDecimal i other. Obliczenie ilorazu jest możliwe tylko po podaniu **sposobu zaokrąglania**. Na przykład tryb RoundingMode.HALF_UP jest znany nam wszystkim ze szkoły (cyfry od 0 do 4 zaokrąglamy w dół, a od 5 do 9 w góre). Ten sposób zaokrąglania jest odpowiedni do typowych obliczeń. Opis pozostałych trybów zaokrąglania znajduje się w dokumentacji API.

- int compareTo(BigDecimal other)

Zwraca wartość 0, jeśli liczba BigDecimal jest równa liczbie other, wartość ujemną, jeśli liczba BigDecimal jest mniejsza od liczby other, lub liczbę dodatnią w przeciwnym przypadku.

- static BigDecimal valueOf(long x)
- static BigDecimal valueOf(long x, int scale)

Zwraca wielką liczbę, której wartość jest równa x lub $x/10^{scale}$.

3.10. Tablice

Tablica jest rodzajem struktury danych będącą zestawem elementów tego samego typu. Dostęp do każdego z tych elementów można uzyskać za pomocą jego indeksu w postaci liczby typu int. Jeśli na przykład a jest tablicą liczb całkowitych, to a[i] jest i-tym elementem tej tablicy.

Deklaracja zmiennej tablicowej polega na określeniu typu tablicy (czyli podaniu typu elementów i nawiasów kwadratowych []) i nazwy zmiennej. Poniżej znajduje się przykładowa deklaracja tablicy zdolnej do przechowywania liczb całkowitych:

```
int[] a;
```

Powyzsza instrukcja tylko deklaruje zmienną a. Nie inicjuje jej jednak tablicą. Do utworzenia tablicy potrzebny jest operator new.

```
int[] a = new int[100];
```

Powyzsza instrukcja tworzy i inicjuje tablicę, w której można zapisać 100 liczb całkowitych.

Długość tablicy nie musi być stała, np. instrukcja new int[n] tworzy tablicę o długości n.

Elementy tablicy są **numerowane od 0** (tu od 0 do 99). Tablicę można zapłnić wartościami na przykład za pomocą pętli:

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
    a[i] = i; // Zapewnia tablicę wartościami od 0 do 99.
```



Zmienną tablicową można zdefiniować na dwa sposoby:

```
int[] a;
lub
int a[];
```

Większość programistów stosuje ten pierwszy styl ze względu na eleganckie oddzielenie typu `int[]` (w przypadku tablicy liczb całkowitych) od nazwy zmiennej.

W nowych tablicach liczb wszystkie elementy są inicjowane zerami. W tablicach wartości logicznych elementom przypisywana jest wartość `false`, a w tablicach na obiekty elementom nadawana jest specjalna wartość `null`, oznaczająca, że nie zawierają one jeszcze żadnych obiektów. Może to być zaskakujące dla początkujących programistów, np.:

```
String[] names = new String[10];
```

Powyższa instrukcja tworzy tablicę dziesięciu łańcuchów, z których każdy jest `null`. Jeśli w tablicy mają być zapisane puste łańcuchy, należy je do niej przekazać:

```
for (int i = 0; i < 10; i++) names[i] = "";
```



Próba dostępu do elementu o indeksie 100 (lub jakimkolwiek innym większym od 99) w tablicy zawierającej 100 elementów zakończy się spowodowaniem wyjątku `ArrayIndexOutOfBoundsException` (indeks spoza przedziału tablicy).

Informację o liczbie elementów przechowywanych w tablicy można uzyskać za pomocą odwołania `nazwaTablicy.length`. Na przykład:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

Rozmiar tablicy nie można zmienić (ale można oczywiście zmieniać jej poszczególne elementy). Jeśli konieczne są częste zmiany rozmiaru tablicy w trakcie działania programu, należy użyć listy `ArrayList` (więcej informacji na ten temat znajduje się w rozdziale 5.).

3.10.1. Pętla typu for each

W języku Java dostępny jest bardzo użyteczny rodzaj pętli umożliwiającej przeglądanie tablic (jak również innych rodzajów kolekcji) bez stosowania indeksów.

Poniższa **udoskonalona** pętla `for`:

```
for (zmienna : kolekcja) instrukcja
```

ustawia podaną zmienną na każdy element kolekcji i wykonuje instrukcję (która oczywiście może być blokiem instrukcji). **Kolekcja** musi być tablicą lub obiektem klasy implementującej interfejs `Iterable`, jak np. `ArrayList`. Listy `ArrayList` omawiamy w rozdziale 5., a interfejs `Iterable` w drugim rozdziale drugiego tomu.

Na przykład poniższa pętla:

```
for (int element : a)
    System.out.println(element);
```

drukuje każdy element tablicy a w oddzielnym wierszu.

Pętlę tę należy czytać następująco: „Dla każdego elementu w a”. Projektanci rozważyli dodanie do Javy słów kluczowych, jak `foreach` (dla każdego) i `in` (w), ale spowodowałoby to uszkodzenie już napisanego kodu zawierającego metody lub zmienne o takich właśnie nazwach (np. `System.in`).

Ten sam efekt można oczywiście uzyskać za pomocą typowej pętli `for`:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

Pętla typu `for each` jest jednak bardziej zwięzła i mniej podatna na błędy (brak indeksów początkowego i końcowego).



Zmienna pętlowa pętli typu `for each` przemierza **elementy** tablicy, nie wartości indeksów.

Pętla typu `for each` jest bardzo miłym udoskonaleniem języka w stosunku do tradycyjnej formy, jeśli chcemy przetworzyć wszystkie elementy tablicy. Nadal jednak pętla `for` znajduje wiele zastosowań, na przykład kiedy nie chcemy przemierzać całej kolekcji danych lub musimy użyć indeksu w pętli.



Istnieje jeszcze prostsza metoda na wydrukowanie wszystkich elementów tablicy. Polega na użyciu metody `toString` klasy `Arrays`. Odwołanie `Arrays.toString(a)` zwróci łańcuch składający się ze wszystkich elementów tablicy ujętych w nawiasy kwadratowe i rozdzielonych przecinkami, np. `[2, 3, 5, 7, 11, 13]`. Poniższe wywołanie drukuje zawartość tej tablicy:

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

3.10.2. Inicjowanie tablic i tworzenie tablic anonimowych

Java umożliwia zastosowanie skróconego zapisu pozwalającego na jednoczesne utworzenie tablicy i zainicjowanie jej wartościami początkowymi. Oto przykład tej składni:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

Należy zauważyć, że w przypadku zastosowania tej składni nie używa się operatora `new`.

Można nawet zainicjować **tablicę anonimową**:

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

Powyższe wyrażenie przydziela pamięć dla nowej tablicy i zapełnia ją wartościami podanymi między klamrami. Sprawdza liczbę początkowych wartości i odpowiednio ustawia rozmiar tworzonej tablicy. Za pomocą tej metody można ponownie zainicjować tablicę, nie tworząc przy tym nowej zmiennej. Na przykład zapis:

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

jest skróconą wersją zapisu:

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };  
smallPrimes = anonymous;
```



Można tworzyć tablice o rozmiarze 0. Taka tablica może się okazać przydatna, kiedy napiszemy metodę zwracającą tablicę, której wynik jest pusty. Konstrukcja tablicy o rozmiarze 0 wygląda następująco:

```
new typElementu[0]
```

Zwrócmy uwagę, że tablica o rozmiarze 0 nie jest tym samym co null.

3.10.3. Kopiowanie tablicy

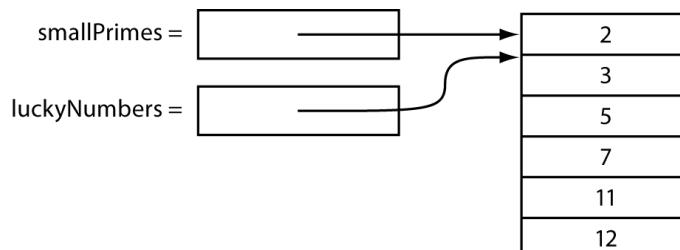
Jedną zmienną tablicową można skopiować do drugiej, ale w takim przypadku **obie zmienne wskazują na tę samą tablicę**:

```
int[] luckyNumbers = smallPrimes;  
luckyNumbers[5] = 12; // Teraz element smallPrimes[5] ma wartość 12.
```

Wynik przedstawia rysunek 3.14. Aby rzeczywiście skopiować wszystkie elementy jednej tablicy do innej, należy użyć metody `copyTo` dostępnej w klasie `Arrays`:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

Rysunek 3.14.
Kopiowanie zmiennej tablicowej



Drugi parametr określa rozmiar nowej tablicy. Metoda ta jest często wykorzystywana do zwiększenia rozmiaru tablicy:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

Dodatkowe elementy są zapełniane zerami, jeśli tablica przechowuje liczby, lub wartościami `false`, jeśli przechowywane są wartości logiczne. Jeśli rozmiar nowej tablicy jest mniejszy niż pierwotny, kopowane są elementy z początku tablicy.



Tablice w Javie nie są tym samym co tablice w C++ na stosie (ang. *stack*). Są natomiast w zasadzie odpowiednikiem wskaźników do tablic alokowanych na **stercie** (ang. *heap*). To znaczy:

```
int[] a = new int[100]; //Java
```

to nie to samo co:

```
int a[100]; //C++
```

ale to samo co:

```
int* a = new int[100]; //C++
```

W Javie operator [] zajmuje się **sprawdzaniem zakresu**. Ponadto nie można wykonywać działań arytmetycznych na wskaźnikach — nie można inkrementować zmiennej a, aby wskazywała na kolejny element tablicy.

3.10.4. Parametry wiersza poleceń

Do tej pory widzieliśmy jeden przykład tablicy w Javie, który został kilkakrotnie powtórzony. Każdy program w Javie składa się z metody `main` z parametrem `String[] args`. Oznacza to, że metoda `main` przyjmuje tablicę łańcuchów, czyli argumenty podawane w wierszu poleceń.

Przyjrzyjmy się poniższemu programowi:

```
public class Message
{
    public static void main(String[] args)
    {
        if (args[0].equals("-h"))
            System.out.print("Witaj. ");
        else if (args[0].equals("-g"))
            System.out.print("Żegnaj. ");
        // Wydruk pozostałych argumentów wiersza poleceń.
        for (int i = 1; i < args.length; i++)
            System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

Jeśli program ten uruchomimy w następujący sposób:

```
java Message -g okrutny świecie
```

tablica args będzie miała następującą zawartość:

```
args[0]: "-g"
args[1]: "okrutny"
args[2]: "świecie"
```

Program wydrukuje wiadomość:

```
Żegnaj, okrutny świecie!
```



W Javie nazwa programu nie jest przechowywana w tablicy args w metodzie main. Jeśli na przykład program zostanie uruchomiony następująco:

```
java Message -h świecie
```

element args[0] będzie zawierał wartość parametru "-h", a nie łańcuch "Message" czy "java".

3.10.5. Sortowanie tablicy

Do sortowania tablic przechowujących liczby służą metody sort dostępne w klasie Arrays:

```
int[] a = new int[10000];
...
Arrays.sort(a)
```

Ta metoda korzysta ze zoptymalizowanej wersji algorytmu QuickSort, który ma opinię bardzo efektywnego w sortowaniu większości zbiorów danych. W klasie Arrays dostępnych jest kilka innych metod usprawniających pracę z tablicami. Opisano je w uwadze o API na końcu tego podrozdziału.

Program widoczny na listingu 3.7 stanowi przykład praktycznego zastosowania tablic. Jego działanie polega na losowaniu kilku liczb na loterii. Jeśli na przykład zagramy w wybór sześciu liczb z 49, wynik może być następujący:

Postaw na następujące liczby. Dzięki nim zdobędziesz bogactwo!

```
1
10
23
29
31
34
```

Najpierw zapełniamy tablicę numbers liczbami 1, 2, 3..., n.

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;
```

Listing 3.7. LotteryDrawing.java

```
import java.util.*;

/**
 * Ten program demonstruje zastosowanie tablic.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryDrawing
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
```

```
System.out.print("Ile liczb musisz wylosować? ");
int k = in.nextInt();

System.out.print("Jaka jest największa liczba? ");
int n = in.nextInt();

// Zapelnienie tablicy liczbami 1 2 3 ... n.
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;

// Losowanie k liczb i zapisanie ich w drugiej tablicy.
int[] result = new int[k];
for (int i = 0; i < result.length; i++)
{
    // Losowanie indeksu z zakresu od 0 do n-1.
    int r = (int) (Math.random() * n);

    // Pobranie elementu z losowej lokalizacji.
    result[i] = numbers[r];

    // Przeniesienie ostatniego elementu do losowej lokalizacji.
    numbers[r] = numbers[n - 1];
    n--;
}

// Wydruk zapisanej tablicy.
Arrays.sort(result);
System.out.println("Postaw na następujące liczby. Dzięki nim zdobędziesz
    ↪ bogactwo!");
for (int r : result)
    System.out.println(r);
}
```

Druga tablica przechowuje liczby do wylosowania:

```
int[] result = new int[k];
```

Następnie losujemy k liczb. Metoda Math.random zwraca losową liczbę zmiennoprzecinkową z zamkniętego przedziału 0-1. Dzięki pomnożeniu jej wyniku przez n uzyskujemy losową liczbę z przedziału od 0 do n-1.

```
int r = (int) (Math.random() * n);
```

i-ty wynik będzie liczbą przechowywaną w indeksie i. Początkowo jest to i+1, ale niebawem się przekonamy, że zawartość tablicy numbers zmienia się po każdym losowaniu.

```
result[i] = numbers[r];
```

Trzeba się zabezpieczyć, aby nie wylosować tej samej liczby ponownie — wszystkie liczby na loterii muszą być inne. W tym celu zastępujemy element numbers[r] **ostatnią** liczbą w tablicy i zmniejszamy n o 1.

```
numbers[r] = numbers[n - 1];
n--;
```

Naszym celem jest to, aby za każdym razem był losowany **indeks**, a nie rzeczywiste wartości. Indeks ten wskazuje na element tablicy zawierającej wartości, które nie zostały jeszcze wylosowane.

Po wylosowaniu k liczb sortujemy zawartość tablicy result:

```
Arrays.sort(result);
for (int r : result)
    System.out.println(r);
```

java.util.Arrays 1.2

■ static String *toString*(*typ*[] a) **5.0**

Zwraca łańcuch złożony z elementów tablicy a ujętych w nawiasy kwadratowe i rozdzielonych przecinkami.

Parametry: a Tablica elementów typu int, long, short, char, byte, boolean, float lub double.

■ static *typ*[] *copyOf*(*typ*[] a, int *length*) **6**

■ static *typ*[] *copyOf*(*typ*[] a, int *start*, int *end*) **6**

Zwraca tablicę tego samego typu co tablica a, mającą rozmiar *length* albo *end-start* i zapełnioną wartościami z tablicy a.

Parametry: a Tablica elementów typu int, long, short, char, byte, boolean, float lub double.

start Indeks początkowy (włącznie).

end Indeks końcowy (włącznie). Może być większy od a.length — w takim przypadku puste miejsca są zapełniane zerami lub wartościami false.

length Rozmiar kopii. Jeśli *length* jest większa od a.length, puste miejsca są zapełniane zerami lub wartościami false. W przeciwnym przypadku kopiowanych jest *length* wartości początkowych.

■ static void *sort*(*typ*[] a)

Sortuje tablicę przy użyciu zoptymalizowanego algorytmu QuickSort.

Parametry: a Tablica elementów typu int, long, short, char, byte, boolean, float lub double.

■ static int *binarySearch*(*typ*[] a, *typ* v)

■ static int *binarySearch*(*typ*[] a, int *start*, int *end* *typ* v) **6**

Wyszukuje wartość v przy użyciu algorytmu wyszukiwania binarnego. W przypadku powodzenia zwraca indeks znalezionej wartości. W przeciwnym razie zwraca ujemną wartość r. -r - 1 to miejsce, w którym należy wstawić wartość v, aby tablica a pozostała posortowana.

Parametry:	a	Tablica elementów typu int, long, short, char, byte, boolean, float lub double.
	start	Indeks początkowy (włącznie).
	end	Indeks końcowy (wyłącznie).
	v	Wartość tego samego typu co elementy tablicy a.

■ static void fill(*typ*[] a, *typ* v)

Ustawia wszystkie elementy tablicy na wartość v.

Parametry:	a	Tablica elementów typu int, long, short, char, byte, boolean, float lub double.
	v	Wartość tego samego typu co elementy tablicy a.

■ static boolean equals(*typ*[] a, *typ*[] b)

Zwraca wartość true, jeśli tablice mają takie same rozmiary i jeśli wartości na odpowiadających sobie pozycjach pasują do siebie.

Parametry:	a, b	Tablice elementów typu int, long, short, char, byte, boolean, float lub double.
-------------------	------	---

3.10.6. Tablice wielowymiarowe

Tablice wielowymiarowe służą do reprezentacji tabel i innych bardziej złożonych struktur danych. Aby uzyskać dostęp do elementu tablicy wielowymiarowej, należy użyć więcej niż jednego indeksu. Można ten podrozdział pominać i wrócić do niego w razie potrzeby.

Przypuśćmy, że chcemy utworzyć tabelę liczb pokazującą, jaki będzie zwrot z inwestycji 10 000 zł przy różnych stopach procentowych składanych rocznie. Tabela 3.8 przedstawia taki scenariusz.

Powyższe informacje zapiszemy w tablicy dwuwymiarowej (czyli macierzy) o nazwie balances.

Deklaracja tablicy dwuwymiarowej w Javie jest bardzo prosta. Wystarczy napisać:

```
double[][] balances;
```

Jak zwykle tablicy nie można używać, dopóki się jej nie zainicjuje za pomocą wyrażenia new. W tym przypadku inicjacja może wyglądać następująco:

```
balances = new double[NYEARS][NRATES];
```

Jeśli znane są elementy tablicy, można użyć skróconej notacji inicjacji tablicy wielowymiarowej, która nie wymaga wywołania new. Na przykład:

```
int[][] magicSquare =  
{  
    {16, 3, 2, 13},  
    {5, 10, 11, 8},
```

Tabela 3.8. Wzrost dochodu z inwestycji przy różnych stopach oprocentowania

10%	11%	12%	13%	14%	15%
10 000,00	10 000,00	10 000,00	10 000,00	10 000,00	10 000,00
11 000,00	11 100,00	11 200,00	11 300,00	11 400,00	11 500,00
12 100,00	12 321,00	12 544,00	12 769,00	12 996,00	13 225,00
13 310,00	13 676,31	14 049,28	14 428,97	14 815,44	15 208,75
14 641,00	15 180,70	15 735,19	16 304,74	16 889,60	17 490,06
16 105,10	16 850,58	17 623,42	18 424,35	19 254,15	20 113,57
17 715,61	18 704,15	19 738,23	20 819,52	21 949,73	23 130,61
19 487,17	20 761,60	22 106,81	23 526,05	25 022,69	26 600,20
21 435,89	23 045,38	24 759,63	26 584,44	28 525,86	30 590,23
23 579,48	25 580,37	27 730,79	30 040,42	32 519,49	35 178,76

```
{9, 6, 7, 12},  
{4, 15, 14, 1}  
};
```

Dostęp do elementów takiej tablicy uzyskujemy za pomocą dwóch indeksów, np. `balances` \rightarrow `[i][j]`.

Przykładowy program zapisuje jednowymiarową tablicę o nazwie `interest` zawierającą stopy oprocentowania i dwuwymiarową tablicę o nazwie `balances` zawierającą stany środków dla każdego roku i każdej stopy procentowej. Pierwszy wiersz tablicy inicjujemy saldem początkowym:

```
for (int j = 0; j < balance[0].length; j++)  
    balances[0][j] = 10000;
```

Następnie obliczamy wartości w kolejnych wierszach:

```
for (int i = 1; i < balances.length; i++)  
{  
    for (int j = 0; j < balances[i].length; j++)  
    {  
        double oldBalance = balances[i - 1][j];  
        double interest = . . .;  
        balances[i][j] = oldBalance + interest;  
    }  
}
```

Listing 3.8 przedstawia pełny kod tego programu.

Listing 3.8. CompoundInterest.java

```
/**  
 * Ten program demonstruje przechowywanie danych tabelarycznych w tablicy dwuwymiarowej.  
 * @version 1.40 2004-02-10  
 * @author Cay Horstmann  
 */
```



Pętla typu `for each` nie sprawdza automatycznie wszystkich elementów tablicy dwuwymiarowej. Przechodzi tylko przez wiersze, które są tablicami jednowymiarowymi. Aby dotrzeć do wszystkich elementów tablicy dwuwymiarowej `a`, należy zagnieździć jedną pętlę w drugiej:

```
for (double[] row : a)
for (double value : row)
```

Działania na wartościach



Aby szybko wydrukować listę elementów tablicy dwuwymiarowej, należy wywołać:

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

Wynik będzie następujący:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 11]]
```

```
public class CompoundInterest
{
    public static void main(String[] args)
    {
        final double STARTRATE = 10;
        final int NRATES = 6;
        final int NYEARS = 10;

        // Ustawienie stóp oprocentowania na wartości w przedziale 10 – 15%.
        double[] interestRate = new double[NRATES];
        for (int j = 0; j < interestRate.length; j++)
            interestRate[j] = (STARTRATE + j) / 100.0;

        double[][] balances = new double[NYEARS][NRATES];

        // Ustawienie sald początkowych na 10 000.
        for (int j = 0; j < balances[0].length; j++)
            balances[0][j] = 10000;

        // Obliczenie odsetek dla przyszłych lat.
        for (int i = 1; i < balances.length; i++)
        {
            for (int j = 0; j < balances[i].length; j++)
            {
                // Pobranie sald z minionego roku z poprzedniego wiersza.
                double oldBalance = balances[i - 1][j];

                // Obliczenie odsetek.
                double interest = oldBalance * interestRate[j];

                // Obliczenie tegorocznego salda.
                balances[i][j] = oldBalance + interest;
            }
        }

        // Wydruk jednego wiersza stóp oprocentowania.
        for (int j = 0; j < interestRate.length; j++)
            System.out.printf("%9.0f%%", 100 * interestRate[j]);

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

```

// Wydruk tabeli sald.
for (double[] row : balances)
{
    // Wydruk wiersza tabeli.
    for (double b : row)
        System.out.printf("%10.2f", b);

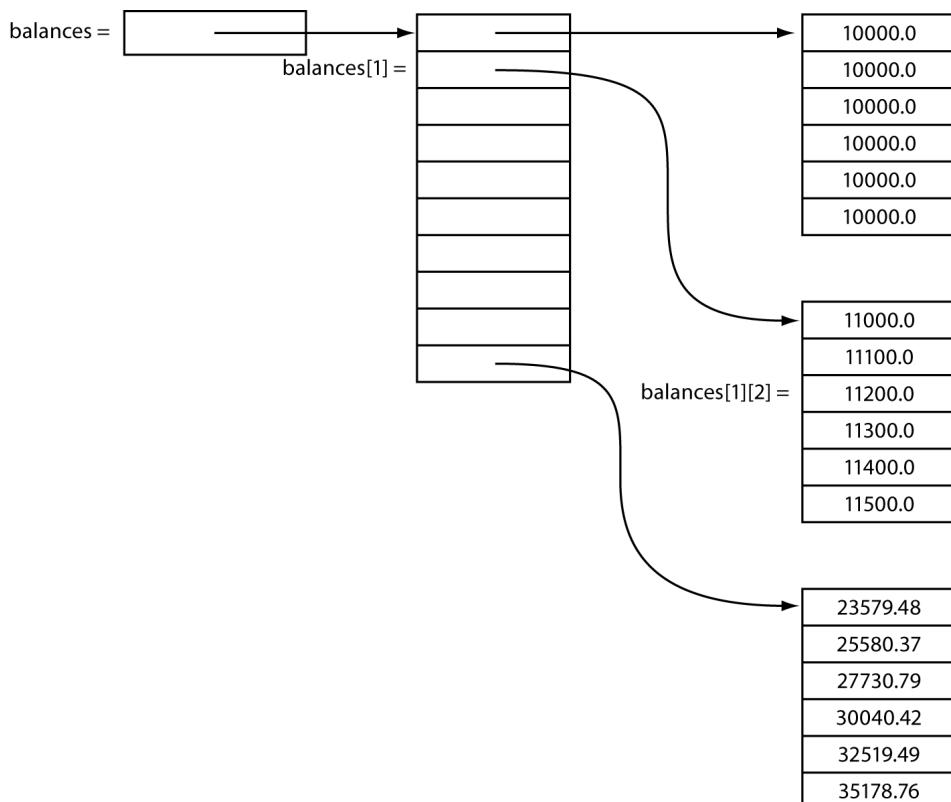
    System.out.println();
}
}
}

```

3.10.7. Tablice postrzępione

Opisane do tej pory rodzaje tablic nie różnią się niczym szczególnym od tych, które znamy z innych języków programowania. Jest jednak coś, o czym warto wiedzieć: w Javie **nie ma** prawdziwych tablic wielowymiarowych. Są one tylko symulowane przez „tablice tablic”.

Na przykład tablica `balances` utworzona w powyższym programie zawiera dziesięć elementów, z których każdy jest tablicą zawierającą sześć liczb zmiennoprzecinkowych (zobacz rysunek 3.15).



Rysunek 3.15. Tablica dwuwymiarowa

Wyrażenie `balances[i]` odwołuje się do i -tej podtablicy, która jest i -tym wierszem tablicy. Wiersz ten sam jest tablicą, a więc `balances[i][j]` odwołuje się do j -tego wiersza tej tablicy.

Jako że do poszczególnych wierszy tablic można uzyskać dostęp, można zamieniać je miejscami!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

Równie łatwe jest tworzenie tablic postrzepionych (ang. *ragged arrays*), czyli takich, w których wiersze mają różne długości. Oto typowy przykład. Utworzymy tablicę, w której element w i -tym wierszu i j -tej kolumnie jest równy liczbie możliwych wyników loterii polegającej na losowaniu j liczb spośród i liczb.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

Jako że j nie może być większe od i , powstaje macierz trójkątna. Wiersz i -ty ma $i+1$ elementów (zezwalamy na niewybranie żadnego elementu — można to zrobić tylko w jeden sposób). Aby utworzyć taką tablicę postrzepioną, należy najpierw alokować w pamięci tablicę przechowującą wiersze.

```
int[][] odds = new int[NMAX + 1][];
```

Następnie tworzymy wiersze:

```
for (int n = 0; n <= NMAX; n++)
    odds[n] = new int[n + 1];
```

Po utworzeniu tablicy można działać na jej elementach w normalny sposób, pod warunkiem że nie przekroczymy zakresu.

```
for (int n = 0; n < odds.length; n++)
    for (int k = 0; k < odds[n].length; k++)
    {
        // Obliczenie lotteryOdds
        .
        .
        odds[n][k] = lotteryOdds;
    }
```

Listing 3.9 przedstawia kompletny program.

Listing 3.9. LotteryArray.java

```
/*
 * Ten program demonstruje sposób tworzenia tablicy trójkątnej.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryArray
{
    public static void main(String[] args)
```



W C++ znana z Javy deklaracja:

```
double[][] balances = new double[10][6]; //Java
```

nie jest równoważna z:

```
double balances[10][6]; // C++
```

ani nawet z:

```
double (*balances)[6] = new double[10][6]; // C++
```

Zamiast tego tworzona jest tablica 10 wskaźników:

```
double** balances = new double*[10]; // C++
```

Następnie do każdego elementu w tablicy wskaźników wstawiana jest tablica 6 liczb:

```
for (i = 0; i < 10; i++)
    balances[i] = new double[6];
```

Na szczęście pętla ta działa automatycznie, kiedy tworzymy tablicę `new double[10][6]`. Aby utworzyć tablicę postrzępioną, każdy wiersz musimy tworzyć oddzielnie.

```
{
    final int NMAX = 10;

    // Tworzenie tablicy trójkątnej.
    int[][] odds = new int[NMAX + 1][];
    for (int n = 0; n <= NMAX; n++)
        odds[n] = new int[n + 1];

    // Zapelnienie tablicy trójkątnej.
    for (int n = 0; n < odds.length; n++)
        for (int k = 0; k < odds[n].length; k++)
    {
        /*
        * Obliczenie współczynnika dwumianowego n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k).
        */
        int lotteryOdds = 1;
        for (int i = 1; i <= k; i++)
            lotteryOdds = lotteryOdds * (n - i + 1) / i;

        odds[n][k] = lotteryOdds;
    }

    // Drukowanie tablicy trójkątnej.
    for (int[] row : odds)
    {
        for (int odd : row)
            System.out.printf("%4d", odd);
        System.out.println();
    }
}
```

Właśnie poznaliśmy podstawowe struktury programistyczne Javy. W kolejnym rozdziale zajmiemy się technikami obiektowymi.

4

Obiekty i klasy

W tym rozdziale:

- Wstęp do programowania obiektowego
- Używanie standardowych klas
- Definiowanie własnych klas
- Pola i metody statyczne
- Parametry metod
- Konstrukcja obiektów
- Pakiety
- Ścieżka klas
- Komentarze dokumentacyjne
- Porady dotyczące projektowania klas

Osoby, które do tej pory nie miały do czynienia z programowaniem obiektowym, powinny bardzo uważnie przeczytać ten rozdział. Programowanie obiektowe wymaga innego sposobu myślenia niż programowanie proceduralne. Przestawienie się bywa czasami trudne, ale kontynuacja nauki Javy bez znajomości technik obiektowych byłaby niemożliwa.

Programiści języka C++ odkryją w tym rozdziale (podobnie jak w poprzednim) dużo podobieństw między Javą i C++. Jednak Java i C++ różnią się na tyle, że także programiści C++ powinni przeczytać ten rozdział z uwagą. W przestawieniu się na nowy język będą pomocne uwagi dotyczące języka C++.

4.1. Wstęp do programowania obiektowego

Programowanie obiektowe (ang. *Object Oriented Programming — OOP*) jest obecnie najbardziej rozpowszechnionym paradygmatem programowania i zastąpiło techniki proceduralne opracowane jeszcze w latach 70. ubiegłego wieku. Java jest językiem w pełni obiektowym, a co za tym idzie — aby być efektywnym programistą Javy, trzeba znać techniki obiektowe.

Program obiektowy składa się z obiektów. Każdy obiekt udostępnia określony zestaw funkcji, a ich szczegóły implementacyjne są ukryte. Wiele obiektów używanych w programach pochodzi z biblioteki. Część z nich programista tworzy jednak własnoręcznie. To, czy programista zdecyduje się na budowę własnego obiektu, czy skorzysta z już istniejącego, zależy od jego czasu i możliwości. Dopóki obiekt spełnia wymagania, z reguły nie ma potrzeby zagłębiania się w tajniki jego implementacji. W programowaniu obiektowym implementacja obiektu nie ma znaczenia, dopóki działa on zgodnie z oczekiwaniemi.

Tradycyjne programowanie proceduralne polega na zaprojektowaniu zbioru procedur (czyli **algorytmów**) mających rozwiązać dany problem. Po utworzeniu procedur przychodzi kolej na zapisanie danych. Dlatego właśnie twórca języka Pascal, Niklaus Wirth, zatytułował swoją słynną książkę o programowaniu *Algorytmy + struktury danych = programy* (WNT, Warszawa 2004). Znamienne jest to, że w tytule tym na początku znajdują się algorytmy, a dopiero po nich struktury danych. Obrazuje to sposób, w jaki pracowali programiści w tamtych czasach. Najpierw opracowywali procedury przetwarzające dane, a następnie ujmowali te dane w takie struktury, które ułatwiały to przetwarzanie. W programowaniu obiektowym ta kolejność jest odwrócona — najpierw są dane, dopiero po nich algorytmy, które je przetwarzają.

W przypadku małych programów podział na procedury jest bardzo dobrym podejściem. Obiekty natomiast są najlepsze do pracy nad dużymi projektami. Weźmy prostą przeglądarkę internetową. Implementacja takiego programu mogłaby wymagać około 2000 procedur operujących na jakimś zbiorze danych globalnych. Przy zastosowaniu podejścia obiektowego byłoby około 100 klas, z których średnio każda zawierałaby 20 metod (zobacz rysunek 4.1). Ta druga struktura byłaby znacznie łatwiejsza do ogarnięcia przez programistę. Latwiej też znaleźć w niej błędy. Wyobraźmy sobie, że dane określonego obiektu znajdują się w nieprawidłowym stanie. Znalezienie problemu wśród 20 metod, które miały dostęp do tych danych, jest znacznie łatwiejsze niż wśród 2000 procedur.

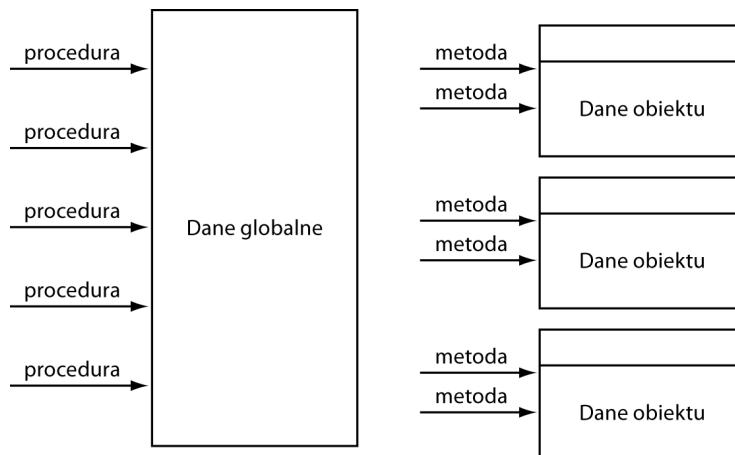
4.1.1. Klasa

Klasa jest szablonem, z którego tworzy się obiekty. Jeśli klasy są foremkami do robienia ciastek, to obiekty są samymi ciastkami. **Konstruując** obiekt, tworzymy **egzemplarz** klasy.

Wiemy już, że wszystko, co piszemy w Javie, znajduje się w jakiejś klasie. W standardowej bibliotece znajduje się kilka tysięcy klas o tak różnym przeznaczeniu jak wspomaganie projektowania interfejsu, obsługa dat i kalendarzy czy programowanie sieciowe. Niemniej konieczne jest tworzenie własnych klas do opisu obiektów rozwiązujących problemy związane z konkretnym programem.

Rysunek 4.1.

Programowanie proceduralne a programowanie obiektowe



Kluczowym pojęciem związanym z obiektami jest **hermetyzacja** (inaczej mówiąc, ukrywanie danych). Z formalnego punktu widzenia hermetyzacja polega na umieszczaniu danych i operacji w jednym pakuiecie oraz ukrywaniu szczegółów implementacyjnych przed użytkownikiem obiektu. Dane zawarte w obiekcie nazywają się **składowymi obiektu**, a procedury operujące tymi danymi to **metody**. Obiekt będący egzemplarzem danej klasy ma składowe o określonych wartościach. Zestaw tych wartości określa aktualny **stan** obiektu. Za każdym razem, gdy wywoływana jest metoda na rzecz obiektu, jego stan może się zmienić.

Aby hermetyzacja spełniała swoje zadanie, metody **nie mogą** być bezpośrednio wywoływanie na rzecz składowych obiektów klas innych niż ich własna. Dane obiektowe powinny być używane w programie **tylko** za pośrednictwem metod obiektów zawierających te dane. Hermetyzacja nadaje obiektowi charakter „czarnej skrzynki”, co jest kluczowe dla koncepcji wielokrotnego użycia kodu, jak i jego niezawodności. Oznacza to, że sposób przechowywania danych w klasie może się diametralnie zmienić, ale dopóki udostępnia ona te same metody do manipulacji tymi danymi, żaden obiekt nie zostanie tym dotknięty.

Budowę klas w Javie ułatwia jeszcze jedna cecha programowania obiektowego: klasy można budować poprzez **rozszerzanie** (ang. *extending*) innych klas. Wszystkie klasy w Javie dziedziczą po jednej klasie bazowej o nazwie `Object`. Więcej informacji na temat tej klasy znajduje się w rozdziale 5.

Kiedy rozszerzamy istniejącą klasę, nowo powstała klasa ma wszystkie cechy i metody klasy rozszerzanej. Nowe metody i pola są dostępne tylko w nowej klasie. Proces rozszerzania klasy w celu utworzenia nowej klasy nazywa się **dziedziczeniem** (ang. *inheritance*). Szczegółowe informacje na temat dziedziczenia znajdują się w kolejnym rozdziale.

4.1.2. Obiekty

Aby sprawnie poruszać się w świecie programowania obiektowego, należy znać trzy podstawowe cechy obiektu:

- **Zachowanie** obiektu — co można z obiektem zrobić i jakie metody można wywoływać na jego rzecz.

- **Stan** obiektu — jak obiekt reaguje w odpowiedzi na wywoływanie na jego rzecz metody.
- **Tożsamość** obiektu — jak odróżnić obiekt od innych obiektów, które mogą mieć te same zachowanie i stan.

Wszystkie obiekty będące egzemplarzami tej samej klasy są do siebie podobne pod tym względem, że charakteryzują się takim samym **zachowaniem**. Zachowanie obiektu definiują metody, które można wywoływać.

Każdy obiekt przechowuje informacje o tym, jak aktualnie wygląda. Jest to **stan** obiektu. Stan obiektu może się zmieniać w czasie, ale nie samoczynnie. Zmiana stanu obiektu musi być spowodowana wywołaniem metod (jeśli stan obiektu zmieni się, mimo że nie wywołano na jego rzecz żadnej metody, oznacza to, że została złamana zasada hermetyzacji).

Stan obiektu nie wystarczy jednak, aby ten obiekt w pełni opisać, ponieważ istnieje jeszcze **tożsamość obiektu**. Na przykład w systemie przetwarzania zamówień dwa zamówienia są odreborne, mimo iż dotyczą zakupu tego samego produktu. Należy zauważyc, że poszczególne obiekty będące egzemplarzami tej samej klasy **zawsze** mają inną tożsamość i **zazwyczaj** różnią się stanami.

Te kluczowe cechy mogą między sobą oddziaływać. Na przykład stan obiektu może mieć wpływ na jego zachowanie (jeśli zamówienie zostało wysłane lub opłacone, obiekt może odmówić wykonania metody, która dodaje lub usuwa elementy; podobnie jest w przypadku, gdy zamówienie jest puste, to znaczy żadne produkty nie zostały jeszcze dodane — obiekt nie powinien wówczas zezwolić na jego wysłanie).

4.1.3. Identyfikacja klas

Tradycyjny program napisany w technice proceduralnej zaczyna się na samej górze pliku funkcją `main`. W systemie obiektowym nie ma „góry”, przez co nowicjusze często mają problem, od czego zacząć. Odpowiedź jest taka, że najpierw trzeba utworzyć klasy, a potem dodać do nich metody.

Prosta zasada dotycząca nadawania nazw klasom nakazuje tworzenie nazw z rzeczowników obecnych w analizie problemu. Metody natomiast odpowiadają czasownikom.

Na przykład w systemie przetwarzania zamówień mogą się znaleźć następujące rzeczowniki:

- Item (produkt),
- Order (zamówienie),
- Shipping address (adres dostawy),
- Payment (płatność),
- Account (konto)¹.

¹ Ze względu na to, że także polscy programiści zazwyczaj stosują angielskie nazwy w swoich programach, nie tłumaczę żadnych nazw, tylko podaję ich polskie odpowiedniki, gdy jest to uzasadnione — *przyp. tłum.*

Z tych rzeczowników można utworzyć następujące nazwy klas: Item, Order, Shipping
→Address itd.

Następnie szukamy czasowników. Do zamówienia **dodajemy** (ang. *add*) produkty. Zamówienie można **wysłać** (ang. *ship*) albo **anulować** (ang. *cancel*). Płatności są **dokonywane** (ang. *apply*) na rzecz zamówień. Dla każdego z tych czasowników trzeba znaleźć obiekt, który jest odpowiedzialny za wykonywanie tych działań. Jeśli na przykład do zamówienia dodawany jest nowy produkt, to powinien w tę operację zaangażować się obiekt klasy Order, ponieważ ma informacje na temat zapisywania i sortowania produktów. To znaczy, że metoda *add* powinna być metodą klasy Order i przyjmować jako parametr obiekt klasy Item.

Oczywiście reguła „rzeczownika i czasownika” jest tylko praktyczną zasadą. W podjęciu decyzji, które rzeczowniki i czasowniki należy wykorzystać w nazwach przy budowie klasy, może pomóc tylko doświadczenie.

4.1.4. Relacje między klasami

Najczęściej spotykane relacje między klasami to:

- **zależność** (używa),
- **agregacja** (zawiera),
- **dziedziczenie** (jest).

Związek **zależności** (czyli „używa”) jest najbardziej oczywisty, a zarazem ogólny. Na przykład klasa Order używa klasy Account, ponieważ obiekty klasy Order potrzebują dostępu do obiektów Account w celu sprawdzenia wypłacalności klienta. Natomiast klasa Item nie jest zależna od klasy Account, ponieważ obiekty klasy Item nie potrzebują informacji o kontach klientów. Zatem klasa zależy od innej klasy, jeśli metody tej pierwszej używają obiektów tej drugiej lub na nich operują.

Liczę klas wzajemnie zależnych należy ograniczać do minimum. Jeśli klasa A nie wie nic o istnieniu klasy B, to nie mają dla niej znaczenia żadne zmiany w klasie B (a to oznacza, że zmiany wprowadzone w klasie B nie powodują powstawania błędów w klasie A)! W terminologii inżynierii oprogramowania określa się to mianem skojarzenia, czyli **stopniem powiązania między klasami** (ang. *coupling*).

Agregacja (czyli związek „zawiera”) jest łatwa do zrozumienia, ponieważ opisuje konkretne zjawisko. Na przykład obiekt klasy Order zawiera obiekty klasy Item. Innymi słowy, obiekty klasy A zawierają obiekty klasy B.



Niektórzy badacze metod programowania traktują pojęcie agregacji pogardliwie i preferują bardziej ogólny związek **asocjacji**. Z punktu widzenia modelowania jest to zrozumiałe, ale dla programistów związek „zawiera” jest bardzo adekwatnym pojęciem. Jest jeszcze jeden powód, dla którego wolimy agregację — standardowa notacja oznaczania asocjacji jest mniej jasna (zobacz tabelę 4.1).

Tabela 4.1. Oznaczenia powiązań między klasami języka UML

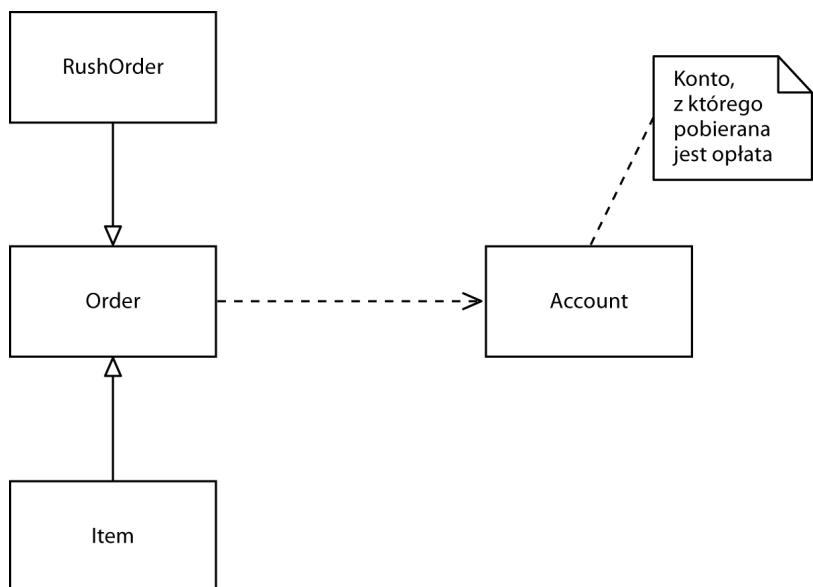
Relacja	Konektor UML
Dziedziczenie	—→
Dziedziczenie interfejsu	—·—·—·—·—·—→
Zależność	—·—·—·—·—·—>
Agregacja	◇—→
Asocjacja	—→
Asocjacja skierowana	—→

Dziedziczenie (czyli związek „jest”) wyraża związek pomiędzy klasą ogólną i klasą specjalną. Na przykład klasa RushOrder (szybkie zamówienie) dziedziczy po klasie Order. Wyspecjalizowana klasa RushOrder ma specjalne metody do obsługi priorytetów i inną metodę do obliczania opłat za transport, ale pozostałe jej metody, jak dodawanie produktów i pobieranie opłat, są odziedziczone po klasie Order. Ogólnie rzecz biorąc, jeśli klasa A rozszerza klasę B, klasa A dziedziczy metody po klasie B, ale ma większe możliwości od klasy B (dziedziczenie jest bardzo ważnym zagadnieniem i zostało szczegółowo opisane w następnym rozdziale).

Wielu programistów rysuje **diagramy klas** języka UML (ang. *Unified Modeling Language*) obrazujące powiązania między klasami. Przykład takiego diagramu przedstawia rysunek 4.2. Klasy są reprezentowane przez prostokąty, a powiązania mają postać strzałek z różnymi dodatkami. Tabela 4.1 przedstawia najczęściej używane w UML typy strzałek.

Rysunek 4.2.

Diagram klas



4.2. Używanie klas predefiniowanych

Ponieważ w Javie nie można nic zrobić bez klas, do tej pory użyliśmy już kilku z nich. Nie wszystkie one jednak są typowymi przedstawicielkami programowania obiektowego. Weźmy na przykład klasę Math. Wiemy, że można używać metod tej klasy jak Math.random i że nie jest nam do tego potrzebna wiedza na temat szczegółów implementacyjnych tych metod. Potrzebujemy tylko nazwy metody i informacji o jej parametrach. Jest to wynikiem hermetyzacji i z pewnością dotyczy wszystkich klas. Ale klasa Math hermetyzuje **tylko** funkcjonalność. Nie potrzebuje ani nie ukrywa danych. Ponieważ nie ma żadnych danych, nie trzeba się zajmować tworzeniem jej obiektów ani inicjacją zmiennych składowych egzemplarzy — ponieważ ich nie ma!

W kolejnym podrozdziale przyjrzymy się bardziej typowej klasie o nazwie Date. Dowiemy się, jak tworzyć obiekty tej klasy i wywoływać jej metody.

4.2.1. Obiekty i zmienne obiektów

Aby móc użyć obiektu, trzeba go najpierw utworzyć i określić jego stan początkowy. Potem można wywoływać na jego rzecz różne metody.

W Javie nowe egzemplarze klas tworzy się za pomocą **konstruktorów**. Konstruktor to specjalna metoda, której przeznaczeniem jest tworzenie i inicjacja obiektów. Weźmy na przykład klasę Date, która jest zdefiniowana w bibliotece standardowej. Jej obiekty określają punkty w czasie, np. "31 Grudzień 2007, 23:59:59 GMT".

Konstruktor ma zawsze taką samą nazwę jak klasa. Zatem konstruktor klasy Date ma nazwę Date. Aby utworzyć obiekt klasy Date, należy użyć konstruktora tej klasy i operatora new:

```
new Date()
```

To wyrażenie tworzy nowy obiekt. Obiekt ten jest inicjowany aktualną datą i godziną.



Niektórzy mogą się zastanawiać, czemu do reprezentacji dat używa się klas zamiast (jak w niektórych językach programowania) typu wbudowanego. Na przykład język Visual Basic ma typ wbudowany, dzięki czemu programista może napisać datę w następującym formacie: #6/1/1995#. Na pierwszy rzut oka wydaje się to całkiem dobrym rozwiązaniem — zamiast przejmować się klasami, programista może użyć typu wbudowanego. Należy jednak zadać pytanie, czy rozwiązanie zastosowane w języku Visual Basic jest dobre. W niektórych krajach format daty to miesiąc/dzień/rok, a w innych dzień/miesiąc/rok. Czy projektanci języka przewidzieli taką ewentualność? Jeśli nie wykonają swojej pracy dobrze, język będzie pozostawał w nieładzie, a nieszczęśliwi programiści nie będą mogli nic z tym zrobić. Przy zastosowaniu klas obowiązek projektowania zostaje przerzucony na twórcę biblioteki. Jeśli klasa ma wady, inni programiści mogą z łatwością napisać własną klasę rozszerzającą lub zastępującą klasę systemową (dowód: biblioteka dat w Javie ma kilka wad i trwają prace nad jej poprawą — zobacz <http://jcp.org/en/jsr-detail?id=310>).

Obiekt można przekazać do metody:

```
System.out.println(new Date());
```

Metodę można też wywołać na rzecz tworzonego obiektu. Jedną z metod klasy Date jest `toString`. Zwraca ona reprezentację łańcuchową daty. Poniżej przedstawiono sposób wywołania metody `toString` na rzecz tworzonego obiektu klasy Date:

```
String s = new Date().toString();
```

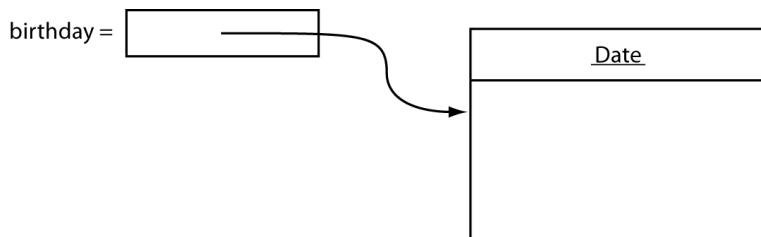
W przedstawionych przykładach utworzony obiekt był używany tylko jeden raz. Zazwyczaj jednak tworzone obiekty są potrzebne do wielokrotnego użytku. Wtedy trzeba zapisać je w zmiennych:

```
Date birthday = new Date();
```

Rysunek 4.3 przedstawia zmienną obiektową `birthday`, która jest referencją do nowo utworzonego obiektu.

Rysunek 4.3.

Tworzenie nowego obiektu



Między obiektami a zmiennymi obiektowymi istnieje bardzo istotna różnica. Na przykład poniższa instrukcja:

```
Date deadline; // Zmienna deadline nie odwołuje się do żadnego obiektu.
```

definiuje zmienną obiektową o nazwie `deadline`, która może się odwoływać do obiektów typu Date. Koniecznie trzeba pamiętać, że zmienna `deadline` **nie jest obiektem** ani nawet nie odwołuje się jeszcze do żadnego obiektu. Obecnie nie można na jej rzecz wywoływać żadnych metod klasy Date. Poniższa instrukcja:

```
s = deadline.toString(); // jeszcze nie
```

spowodowałaby błąd komilacji.

Konieczna jest uprzednia inicjacja zmiennej `deadline`. Są dwie możliwości. Można oczywiście inicjacji tej dokonać za pomocą nowo utworzonego obiektu:

```
deadline = new Date();
```

albo zmienną `deadline` ustawić na istniejący obiekt:

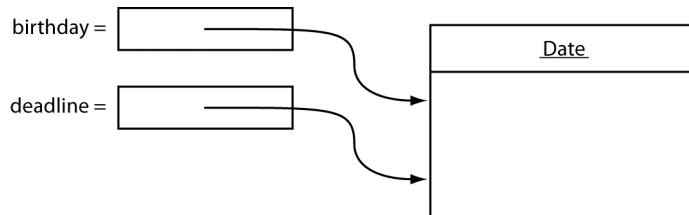
```
deadline = birthday;
```

W tej chwili obie zmienne są referencjami (odwołującymi się) do **tego samego** obiektu (zobacz rysunek 4.4).

Trzeba sobie uświadomić, że zmienna obiektowa nie jest obiektem, tylko **referencją** do obiektu.

Rysunek 4.4.

Zmienne obiektowe odwołujące się do tego samego obiektu



Wartość każdej zmiennej obiektowej jest referencją do obiektu, który jest przechowywany gdzieś indziej. Wartość zwracana przez operator `new` też jest referencją. Instrukcja typu:

```
Date deadline = new Date();
```

składa się z dwóch części. Wyrażenie `new Date()` tworzy obiekt typu `Date`, a jego wartością jest referencja do tego nowo utworzonego obiektu. Referencja ta zostaje zapisana w zmiennej `deadline`.

Aby zaznaczyć, że zmienna obiektowa nie odwołuje się do żadnego obiektu, należy jej wartość ustawić na `null`.

```
deadline = null;
...
if (deadline != null)
    System.out.println(deadline);
```

Wywołanie metody na rzecz zmiennej zawierającej wartość `null` spowoduje błąd działania programu.

```
birthday = null;
String s = birthday.toString(); // Błąd wykonania programu!
```

Zmienne obiektowe nie są automatycznie inicjowane wartością `null`. Trzeba je własnoręcznie inicjować za pomocą operatora `new` lub ustawiać ich wartość na `null`.

4.2.2. Klasa `GregorianCalendar`

W powyższych przykładach używaliśmy klasy `Date`, która należy do standardowej biblioteki. Egzemplarz tej klasy ma wyznaczony stan, którym jest **określony punkt w czasie**.

Chociaż nie musimy tego wiedzieć, by używać klasy `Date`, czas jest reprezentowany przez liczbę milisekund (ujemną lub dodatnią), który upłynął od ustalonego momentu (tak zwanej **epoki**). Tym momentem jest 1 stycznia 1970 o godzinie 00:00:00 UTC. UTC oznacza Coordinated Universal Time — naukowy standard wyrażania czasu, który z przyczyn praktycznych jest równoznaczny z lepiej znany GMT, czyli Greenwich Mean Time.

Okazuje się jednak, że klasa `Date` nie jest zbyt użyteczna przy manipulacji datami. Projektanci języka Java ustalili, że zapis daty w formacie **31 grudnia 1999, 23:59:59** jest z góry przyjętą konwencją, określana przez **kalendarz**. Ten konkretny zapis jest zgodny z najbardziej rozpowszechnionym na świecie kalendarzem gregoriańskim. Ten sam moment w czasie zostałby całkiem inaczej przedstawiony w chińskim lub hebrajskim kalendarzu księżyckowym, nie mówiąc już kalendarzu marsjańskim.



Wielu programistów tkwi w błędym przekonaniu, że zmienne obiektowe w Javie są odpowiednikami referencji w C++. Jednak w C++ nie ma referencji `null` i referencji nie można przypisywać. Zmienne obiektowe Javy należy porównywać ze **wskaźnikami do obiektów** w C++. Na przykład:

Date birthday; // Java

jest równoznaczne z:

Date* birthday; // C++

Oczywiście wskaźnik `Date*` nie jest zainicjowany, dopóki nie użyjemy operatora `new`. Składnia w Javie jest prawie taka sama jak w C++.

Date* birthday = new Date(); // C++

Jeśli skopiujemy jedną zmienną do innej zmiennej, obie zmienne będą się odwoływały do tej samej daty — będą wskazywać ten sam obiekt. Odpowiednikiem referencji `null` Javy jest wskaźnik `NULL` w C++.

Wszystkie obiekty w Javie znajdują się na stercie (ang. *heap*). Jeśli obiekt zawiera jakąś zmienną obiektową, zmienna ta zawiera tylko wskaźnik do innego obiektu na stercie.

Wskaźniki w C++ są źródłem mnóstwa błędów. Łatwo można utworzyć błędny wskaźnik albo nieodpowiednio przydzielić pamięć. W Javie tych problemów nie ma. Jeśli użyjemy niezainicjowanego wskaźnika, interpreter z pewnością zgłosi błąd, zamiast generować losowe wyniki. Zarządzaniem pamięcią zajmuje się natomiast system zbierania nieużytków.

C++ umożliwia implementację obiektów, które automatycznie tworzą kopie samych siebie. Służą do tego konstruktory kopiące i operatory przypisania. Na przykład kopią listy dowiązań jest nowa lista dowiązań o takiej samej zawartości, lecz osobnym zbiorze dowiązań. Dzięki temu można projektować klasy zachowujące się podobnie jak typy wbudowane. W Javie konieczne jest użyciu metody `clone`, aby utworzyć kopię obiektu.



W historii cywilizacji używano rozmaitych kalendarzy, przypisujących datom nazwy i porządkujących cykle słoneczne i księżycowe. Książka *Calendrical Calculations* autorstwa Nachuma Dershowitza i Edwarda M. Reingolda (Cambridge University Press, 2001) w fascynujący sposób opisuje najprzecieżniejsze kalendarze świata, od kalendarza francuskiej rewolucji po kalendarz Majów.

Projektanci biblioteki postanowili rozdzielić kwestie zapisywania informacji o czasie od nadawania nazw momentom czasu. W tym celu biblioteka standardowa Javy zawiera dwie osobne klasy: klasę `Date`, która reprezentuje moment w czasie, i klasę `GregorianCalendar`, która wyraża daty w znanej notacji kalendarzowej. W rzeczywistości klasa `GregorianCalendar` dziedziczy po bardziej ogólnej klasie `Calendar`. Standardowa biblioteka Javy zawiera też implementacje kalendarza buddyjskiego tajskiego i japońskiego imperialnego.

Oddzielenie pomiaru czasu od kalendarzy jest przykładem dobrego podejścia obiektowego. Uogólniając, dobrze jest przedstawiać różne koncepcje za pomocą osobnych klas.

Klasa `Date` zawiera kilka metod do porównywania dwóch momentów w czasie. Na przykład metody `before` i `after` informują, czy dany moment w czasie jest wcześniejszy, czy późniejszy niż inny moment:

```
if (today.before(birthday))
System.out.println("Jest jeszcze czas, aby kupić prezent.");
```



Klasa Date udostępnia też metody `getDay`, `getMonth` i `getYear`, ale ich stosowanie jest **odradzane**. Metoda jest odradzana, jeśli twórca biblioteki dojdzie do wniosku, że metoda ta w ogóle nie powinna była powstać.

Metody te należały do klasy Date, zanim twórcy biblioteki zdali sobie sprawę, że lepiej będzie utworzyć osobne klasy dla kalendarzy. Po wprowadzeniu klas kalendarzy metody klasy Date zostały oznaczone jako odradzane (ang. *deprecated*). Można ich nadal używać, ale kompilator będzie zgłaszał niezbyt eleganckie ostrzeżenia. Dobre jest trzymać się z dala od odradzanych metod, ponieważ mogą one zostać w przyszłości usunięte z biblioteki.

Klasa `GregorianCalendar` zawiera dużo więcej metod niż klasa Date. Przede wszystkim ma kilka przydatnych konstruktorów. Wyrażenie:

```
new GregorianCalendar()
```

tworzy nowy obiekt reprezentujący datę i godzinę w chwili jego utworzenia.

Można utworzyć obiekt ustawiony na północ określonej daty, podając rok, miesiąc i dzień:

```
new GregorianCalendar(1999, 11, 31)
```

Co ciekawe, miesiące są numerowane od 0. A zatem grudzień ma numer 11. Aby uniknąć nieporozumień, wprowadzono stałe typu `Calendar.DECEMBER`:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

Można także ustawić godzinę:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Oczywiście w większości przypadków utworzony obiekt powinien być przechowywany w zmiennej obiektowej:

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

Obiekt klasy `GregorianCalendar` zawiera pola przechowujące datę, na którą obiekt ten zostanie ustawiony. Dzięki hermetyzacji nie sposób odgadnąć, jakiej reprezentacji używa ta klasa, nie zaglądając do jej kodu źródłowego, ale oczywiście dzięki hermetyzacji nie ma to znaczenia. Znaczenie mają metody udostępniane przez klasę.

4.2.3. Metody udostępniające i zmieniające wartość elementu

W tym miejscu należy sobie zadać pytanie: jak dostać się do aktualnego dnia lub miesiąca daty zamkniętej w obiekcie klasy `GregorianCalendar`? A także jak zmienić niektóre wartości? Odpowiedzi na te pytania można znaleźć w dokumentacji internetowej i wyciągach z API znajdujących się na końcu tego podrozdziału. Omówimy tu tylko najważniejsze metody.

Zadaniem kalendarza jest obliczanie atrybutów, takich jak data, dzień tygodnia, miesiąca lub roku, dla określonego punktu w czasie. Aby sprawdzić któryś z tych ustawień, należy posłużyć się metodą akcesora `get` klasy `GregorianCalendar`. Dostęp do wybranych elementów można uzyskać za pomocą stałych zdefiniowanych w klasie `Calendar`, takich jak `Calendar.MONTH` czy `Calendar.DAY_OF_WEEK`.

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

W wyciągu z API znajduje się pełna lista tych stałych.

Stan obiektu można zmienić za pomocą metody mutatora set:

```
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_MONTH, 15);
```

Rok, miesiąc i dzień można też ustawić za pomocą jednego wygodnego wywołania:

```
deadline.set(2001, Calendar.APRIL, 15);
```

Dodatkowo do obiektu kalendarza można dodać dowolną liczbę dni, tygodni, miesięcy itd.:

```
deadline.add(Calendar.MONTH, 3); // Przeniesienie terminu o 3 miesiące.
```

Dodanie liczby ujemnej spowoduje przesunięcie czasu kalendarza do tyłu.

Pomiędzy metodami get oraz set i add jest zasadnicza różnica. Pierwsza z nich tylko sprawdza stan obiektu i zwraca informacje o nim. Metody set i add zmieniają stan obiektu. Metody, które modyfikują pola egzemplarza, nazywają się **mutatorami** (ang. *mutator method*), a te, które dają do nich tylko dostęp, noszą nazwę **akcesorów** (ang. *accessor method*).



W języku C++ metody akcesora są oznaczane przyrostkiem const. Metoda, która nie została zadeklarowana jako const, jest z założenia mutatorem. W Javie nie ma specjalnej notacji odróżniającej metody akcesorów od mutatorów.

Ogólnie przyjęta konwencja głosi, aby metody akcesora poprzedzać przedrostkiem get, a mutatora przedrostkiem set. Na przykład klasa GregorianCalendar zawiera metody getTime i setTime, które odpowiednio pobierają i ustawiają punkt w czasie reprezentowany przez obiekt:

```
Date time = calendar.getTime();
calendar.setTime(time);
```

Metody te mają szczególne znaczenie przy konwersji pomiędzy klasami GregorianCalendar i Calendar. Oto przykład: mając dany rok, miesiąc i dzień, chcemy utworzyć obiekt klasy Date. Ponieważ klasa Date nie ma żadnych danych na temat kalendarzy, najpierw tworzymy obiekt GregorianCalendar, a następnie pobieramy datę za pomocą wywołania metody getTime:

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

Podobnie, aby sprawdzić rok, miesiąc lub dzień obiektu klasy Date, należy utworzyć obiekt klasy GregorianCalendar, ustawić czas i wywołać metodę get:

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

Na zakończenie tego podrozdziału prezentujemy program, który demonstruje praktyczne zastosowanie klasy GregorianCalendar. Ten program wyświetla kalendarz bieżącego miesiąca, podobny do poniższego:

Pn	Wt	Śr	Cz	Pt	So	N
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30*		

Bieżący dzień jest oznaczony gwiazdką (*). Jak widać, program musi wiedzieć, jak obliczyć długość miesiąca oraz dzień tygodnia.

Przeanalizujmy najważniejsze części tego programu. Najpierw tworzymy obiekt kalendarza, który inicjujemy aktualną datą:

```
GregorianCalendar d = new GregorianCalendar();
```

Pobieramy aktualny dzień i miesiąc za pomocą dwóch wywołań metody get:

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

Następnie ustawiamy zmienną d na pierwszy dzień miesiąca i pobieramy dzień tygodnia dla tej daty:

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

Zmienna weekday jest ustawiona na wartość Calendar.NIEDZIELA, jeśli pierwszym dniem miesiąca jest niedziela, Calendar.PONIEDZIAŁEK, jeśli jest to poniedziałek itd. (wartości te są w rzeczywistości liczbami całkowitymi: 1, 2, ..., 7).

Zwróć uwagę, że pierwszy wiersz kalendarza jest odpowiednio wcięty, aby pierwszy dzień miesiąca był przyporządkowany do odpowiedniego dnia tygodnia. Trudności mogą wynikać z tego, że w USA tydzień zaczyna się od niedzieli i kończy w sobotę, a w Europie od poniedziałku i kończy w niedziele.

Maszyna wirtualna Javy ma dane o **lokalizacji** użytkownika. Dane te dotyczą stosowanych konwencji, wliczając początek tygodnia i nazwy dni tygodnia.



Aby zobaczyć wynik tego programu dla innej lokalizacji, należy w pierwszej linijce metody main dodać poniższy wiersz kodu:

```
Locale.setDefault(Locale.ITALY);
```

Metoda getFirstDayOfWeek pobiera pierwszy dzień tygodnia w bieżącej lokalizacji. Odpowiednie wcięcie uzyskujemy poprzez odjęcie 1 od dnia w obiekcie kalendarza, aż dojdziemy do pierwszego dnia tygodnia.

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
```

```
{  
    indent++;  
    d.add(Calendar.DAY_OF_MONTH, -1);  
    weekday = d.get(Calendar.DAY_OF_WEEK);  
}
```

Następnie drukujemy nagłówek kalendarza przedstawiający nazwy dni tygodnia. Są one dostępne w klasie `DateFormatSymbols`:

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

Metoda `getShortWeekdays` zwraca łańcuch złożony ze skrótów nazw dni tygodnia w języku użytkownika (np. Pn, Wt itd. po polsku). Indeksami w tej tablicy są numery dni tygodnia. Poniższa pętla drukuje nagłówek:

```
do  
{  
    System.out.printf("%4s", weekdayNames[weekday]);  
    d.add(Calendar.DAY_OF_MONTH, 1);  
    weekday = d.get(Calendar.DAY_OF_WEEK);  
}  
while (weekday != firstDayOfWeek);  
System.out.println();
```

Możemy teraz przejść do drukowania pozostały części kalendarza. Robimy wcięcie pierwszego wiersza i ustawiamy obiekt daty z powrotem na początek miesiąca. Wprowadzamy pętlę, w której zmienna `d` przemierza wszystkie dni miesiąca.

W każdym powtórzeniu drukowana jest liczba. Jeśli wartość `d` odpowiada dzisiejszej dacie, dodawana jest gwiazdka. Po dojściu do początku kolejnego tygodnia najpierw drukujemy nowy wiersz. Następnie wartość `d` ustawiamy na kolejny dzień:

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

Kiedy się zatrzymamy? Nie wiadomo, czy miesiąc ma 31, 30, 29, czy 28 dni. Powtarzamy pętlę, dopóki `d` mieści się w bieżącym miesiącu.

```
do  
{  
    . . .  
}  
while (d.get(Calendar.MONTH) == month);
```

Kiedy `d` przejdzie do następnego miesiąca, program zostaje zakończony.

Listing 4.1 przedstawia pełny kod tego programu.

Listing 4.1. CalendarTest.java

```
import java.text.DateFormatSymbols;  
import java.util.*;  
  
/**  
 * @version 1.4 2007-04-07  
 * @author Cay Horstmann  
 */
```

```

public class CalendarTest
{
    public static void main(String[] args)
    {
        // Konstrukcja i ustawienie obiektu d oraz jego inicjacja aktualną datą.
        GregorianCalendar d = new GregorianCalendar();

        int today = d.get(Calendar.DAY_OF_MONTH);
        int month = d.get(Calendar.MONTH);

        // Ustawienie d na początek miesiąca.
        d.set(Calendar.DAY_OF_MONTH, 1);

        int weekday = d.get(Calendar.DAY_OF_WEEK);

        // Pobranie pierwszego dnia tygodnia (poniedziałek w Polsce).
        int firstDayOfWeek = d.getFirstDayOfWeek();

        // Określenie odpowiedniego wcięcia pierwszego wiersza.
        int indent = 0;
        while (weekday != firstDayOfWeek)
        {
            indent++;
            d.add(Calendar.DAY_OF_MONTH, -1);
            weekday = d.get(Calendar.DAY_OF_WEEK);
        }

        // Drukowanie nazw dni tygodnia.
        String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
        do
        {
            System.out.printf("%4s", weekdayNames[weekday]);
            d.add(Calendar.DAY_OF_MONTH, 1);
            weekday = d.get(Calendar.DAY_OF_WEEK);
        }
        while (weekday != firstDayOfWeek);
        System.out.println();

        for (int i = 1; i <= indent; i++)
            System.out.print("    ");

        d.set(Calendar.DAY_OF_MONTH, 1);
        do
        {
            // Drukowanie dnia.
            int day = d.get(Calendar.DAY_OF_MONTH);
            System.out.printf("%3d", day);

            // Oznaczenie bieżącego dnia znakiem *.
            if (day == today) System.out.print("*");
            else System.out.print(" ");

            // Ustawienie d na kolejny dzień.
            d.add(Calendar.DAY_OF_MONTH, 1);
            weekday = d.get(Calendar.DAY_OF_WEEK);

            // Rozpoczęcie nowego wiersza na początku tygodnia.
            if (weekday == firstDayOfWeek) System.out.println();
        }
    }
}

```

```

        }
        while (d.get(Calendar.MONTH) == month);
        // Pętla kończy działanie, jeśli d jest pierwszym dniem następnego miesiąca.

        // Wydruk końcowego znaku nowego wiersza w razie potrzeby.
        if (weekday != firstDayOfWeek) System.out.println();
    }
}

```

Przekonaliśmy się, że klasa GregorianCalendar umożliwia pisanie programów kalendarzy z uwzględnieniem skomplikowanych problemów, jak dni tygodnia i różne długości miesięcy. Nie trzeba wiedzieć, **jak** klasa ta oblicza miesiące i dni tygodnia. Programista używa tylko **interfejsu** klas — metod get, set i add.

Ten program ma na celu pokazanie, w jaki sposób można wykorzystać interfejs klasę do bardzo złożonych zadań bez znajomości szczegółów implementacyjnych.

java.util.GregorianCalendar 1.1

■ **GregorianCalendar()**

Tworzy obiekt kalendarza reprezentujący bieżącą datę i godzinę w domyślnej strefie czasowej i lokalizacji.

- **GregorianCalendar(int year, int month, int day)**
- **GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)**

Tworzy kalendarz gregoriański z podanej daty i godziny.

Parametry: year rok

month miesiąc — wartości liczone są od zera
(np. styczeń ma numer 0)

day dzień

hour godzina (od 0 do 23)

minutes minuta (od 0 do 59)

seconds sekunda (od 0 do 59)

■ **int get(int field)**

Pobiera wartość określonego elementu.

Parametry: field Calendar.ERA, Calendar.YEAR, Calendar.MONTH,

Calendar.WEEK_OF_YEAR,
Calendar.WEEK_OF_MONTH,

Calendar.DAY_OF_MONTH, Calendar.DAY_OF_YEAR,

Calendar.DAY_OF_WEEK,

Calendar.DAY_OF_WEEK_IN_MONTH.

```
Calendar.AM_PM, Calendar.HOUR,
Calendar.HOUR_OF_DAY,
Calendar.MINUTE, Calendar.SECOND,
Calendar.MILLISECOND,
Calendar.ZONE_OFFSET, Calendar.DST_OFFSET
```

■ **void set(int field, int value)**

Ustawia wartość określonego elementu.

Parametry: **field** Jedna ze stałych przyjmowanych przez metodę get
value Nowa wartość

■ **void set(int year, int month, int day)**

■ **void set(int year, int month, int day, int hour, int minutes, int seconds)**

Ustawia nowe wartości elementów.

Parametry: **year** rok
month miesiąc — wartości liczone są od zera
(np. styczeń ma numer 0)
day dzień
hour godzina (od 0 do 23)
minutes minuty (od 0 do 59)
seconds sekundy (od 0 do 59)

■ **void add(int field, int amount)**

Metoda wykonująca działania arytmetyczne na datach. Dodaje określoną ilość czasu do określonego pola czasu. Aby na przykład dodać 7 dni do bieżącej daty w kalendarzu, należy zastosować wywołanie: c.add(Calendar.DAY_OF_MONTH, 7).

Parametry: **field** Element, który ma być zmodyfikowany
(za pomocą jednej ze stałych metody get).
amount Liczba, o jaką ma być zmieniona wartość elementu
(może być ujemna).

■ **int getFirstDayOfWeek()**

Pobiera pierwszy dzień tygodnia dla lokalizacji użytkownika, na przykład
Calendar.SUNDAY w USA.

■ **void setTime(Date time)**

Ustawia kalendarz na podany moment w czasie.

Parametr: **time** punkt w czasie

■ **Date getTime()**

Pobiera punkt w czasie reprezentowany przez aktualną wartość obiektu kalendarza.

```
java.text.DateFormatSymbols 1.1
```

- `String[] getShortWeekdays()`
- `String[] getShortMonths()`
- `String[] getWeekdays()`
- `String[] getMonths()`

Pobiera nazwy dni tygodnia lub miesięcy dla obecnej lokalizacji. Jako indeksy wykorzystuje stałe dnia tygodnia i miesiąca klasy `Calendar`.

4.3. Definiowanie własnych klas

W rozdziale 3. pisaliśmy już proste klasy. Wszystkie one jednak zawierały tylko jedną metodę `main`. Teraz nauczysz się pisać klasy z prawdziwego zdarzenia, które będzie można wykorzystać do bardziej wyszukanych zadań. Klasy te z reguły nie mają metody `main`. W zamian mają własne pola i metody. Kompletny program składa się z kilku klas. Jedna z nich musi zawierać metodę `main`.

4.3.1. Klasa Employee

Konstrukcja najprostszej klasy w Javie wygląda następująco:

```
class NazwaKlasy
{
    pole1
    pole2
    .
    .
    .
    konstruktor1
    konstruktor2
    .
    .
    .
    metoda1
    metoda2
    .
    .
    .
}
```

Przyjrzyjmy się poniższej, bardzo uproszczonej wersji klasy `Employee`, która można wykorzystać w firmie do napisania systemu obsługi listy płac.

```
class Employee
{
    //pola
    private String name;
    private double salary;
    private Date hireDay;

    //konstruktor
    public Employee(String n, double s, int year, int month, int day)
```

```

{
    name = n;
    salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
}
// metoda
public String getName()
{
    return name;
}
// kolejne metody
.
.
.
}

```

Zanim przejdziemy do dokładnej analizy klasy `Employee`, pokażemy jej zastosowanie w programie, który przedstawia listing 4.2.

Listing 4.2. EmployeeTest/EmployeeTest.java

```

import java.util.*;

/**
 * Ten program sprawdza działanie klasy Employee.
 * @version 1.11 2004-02-19
 * @author Cay Horstmann
 */
public class EmployeeTest
{
    public static void main(String[] args)
    {
        // Wstawienie trzech obiektów pracowników do tablicy staff.
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Jarosław Rybiński", 75000, 1987, 12, 15);
        staff[1] = new Employee("Katarzyna Remiszewska ", 50000, 1989, 10, 1);
        staff[2] = new Employee("Krystyna Kuczyńska ", 40000, 1990, 3, 15);

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

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

    class Employee
    {
        private String name;
        private double salary;
        private Date hireDay;

        public Employee(String n, double s, int year, int month, int day)

```

```
{  
    name = n;  
    salary = s;  
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);  
    // W klasie GregorianCalendar styczeń ma numer 0.  
    hireDay = calendar.getTime();  
}  
  
public String getName()  
{  
    return name;  
}  
  
public double getSalary()  
{  
    return salary;  
}  
  
public Date getHireDay()  
{  
    return hireDay;  
}  
  
public void raiseSalary(double byPercent)  
{  
    double raise = salary * byPercent / 100;  
    salary += raise;  
}  
}
```

W programie tym tworzymy tablicę o nazwie Employee i wstawiamy do niej trzy obiekty reprezentujące pracowników:

```
Employee[] staff = new Employee[3];  
  
staff[0] = new Employee("Jarosław Rybiński", . . .);  
staff[1] = new Employee("Katarzyna Remiszewska", . . .);  
staff[2] = new Employee("Krystyna Kuczyńska", . . .);
```

Następnie podnosimy zarobki każdego z pracowników o 5% za pomocą metody raiseSalary:

```
for (Employee e : staff)  
    e.raiseSalary(5);
```

Ostatecznie drukujemy informacje o każdym pracowniku, wywołując metody getName, getSalary i getHireDay:

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

Zauważ, że ten przykładowy program składa się z **dowóch** klas: Employee i EmployeeTest ze specyfikatorem dostępu public. Metoda main zawierająca opisane wcześniej instrukcje znajduje się w klasie EmployeeTest.

Plik źródłowy ma nazwę *EmployeeTest.java*, ponieważ nazwa pliku musi być taka sama jak nazwa klasy publicznej. W jednym pliku może być tylko jedna klasa publiczna i dowolna liczba klas niepublicznych.

W trakcie kompilacji tego pliku kompilator utworzy dwa pliki klas: *EmployeeTest.class* i *Employee.class*.

Aby uruchomić program, należy interpreterowi kodu bajtowego podać nazwę klasy zawierającej metodę `main`:

```
java EmployeeTest
```

Interpreter zaczyna wykonywanie kodu od metody `main` w klasie *EmployeeTest*. Program ten z kolei tworzy trzy nowe obiekty *Employee* i pokazuje ich stan.

4.3.2. Używanie wielu plików źródłowych

Program z listingu 4.2 zawiera dwie klasy w jednym pliku. Wielu programistów woli jednak każdą klasę umieścić w oddzielnym pliku. Na przykład klasa *Employee* mogłaby się znaleźć w pliku o nazwie *Employee.java*, a *EmployeeTest* w pliku *EmployeeTest.java*.

W przypadku tego stylu programowania są dwa sposoby kompilacji programu. Można wywołać kompilator Javy z symbolem wieloznacznym:

```
javac Employee*.java
```

Wszystkie pliki źródłowe pasujące do symbolu wieloznacznego zostaną skompilowane do plików klas. Można też po prostu napisać:

```
javac EmployeeTest.java
```

Ta druga opcja może się wydawać nieco zaskakująca, biorąc pod uwagę, że plik *Employee.java* nie jest w ogóle jawnie kompilowany. Kiedy kompilator napotyka klasę *Employee* w pliku *EmployeeTest.java*, szuka pliku o nazwie *Employee.class*. Jeśli go nie znajdzie, pobiera plik o nazwie *Employee.java* i kompiluje go. Kompilator robi nawet coś więcej: jeśli znacznik czasu pliku *Employee.java* jest nowszy niż pliku *Employee.class*, kompilator **automatycznie** dokona jego ponownej kompilacji.



Osoby znające uniksowe narzędzie *make* (lub jego odpowiednik w Windowsie, jak np. *nmake*), mogą traktować kompilator Javy tak, jakby miał wbudowaną funkcjonalność tego narzędzia.

4.3.3. Analiza klasy Employee

Zajmiemy się teraz szczegółową analizą klasy *Employee*. Zaczniemy od jej metod. W kodzie źródłowym widać, że klasa ta ma jeden konstruktor i cztery metody:

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
```

```
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Wszystkie metody tej klasy są publiczne. Słowo kluczowe `public` oznacza, że daną metodę może wywołać każda inna metoda z każdej klasy (cztery dostępne specyfikatory dostępu są opisane w tym i kolejnym rozdziale).

Dane, którymi będziemy manipulować w obiekcie klasy `Employee`, są przechowywane w trzech polach:

```
private String name;
private double salary;
private Date hireDay;
```

Słowo kluczowe `private` oznacza, że do pól klasy `Employee` mają dostęp **tylko** metody tej klasy. Żadna metoda zewnętrzna nie może odczytać ani zmodyfikować tych wartości.



Pola w klasie można oznać słowem kluczowym `public`, ale nie jest to zalecane. Ich wartości mogłyby być odczytane i zmodyfikowane z każdego miejsca programu, a to jest całkowicie sprzeczne z ideą hermetyzacji. Każda metoda z każdej klasy może zmodyfikować publiczne pole — z naszego doświadczenia wynika, że tak się dzieje zawsze w najmniej oczekiwany momencie. Zalecamy stosowanie zawsze specyfikatora `private` dla pól klas.

Zauważmy także, że dwa z pól same są obiektami: pola `name` i `hireDay` są referencjami do obiektów klas `String` i `Date`. Jest to typowa sytuacja — klasy często zawierają pola typów innych klas.

4.3.4. Pierwsze kroki w tworzeniu konstruktorów

Przyjrzymy się konstruktorowi klasy `Employee`:

```
public Employee(String n, double s, int year, int month, int day)
{
    name = n;
    salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
}
```

Jak widać, konstruktor ma taką samą nazwę jak klasa. Konstruktor ten działa, kiedy tworzony jest obiekt klasy `Employee`, i nadaje polom określone przez programistę początkowe wartości.

Jeśli na przykład utworzymy obiekt klasy `Employee` w poniższy sposób:

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

wartości pól będą następujące:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

Między konstruktorami a pozostałymi metodami jest zasadnicza różnica. Konstruktor można wywołać tylko przy użyciu operatora new. Nie można za pomocą konstruktora zmienić wartości pól. Na przykład poniższy zapis spowoduje błąd komilacji:

```
james.Employee("James Bond", 250000, 1950, 1, 1); // błąd
```

Więcej informacji na temat konstruktorów znajduje się w dalszej części tego rozdziału. Na razie należy zapamiętać, że:

- konstruktor musi mieć taką samą nazwę jak klasa;
- klasa może mieć więcej niż jeden konstruktor;
- konstruktor może przyjmować zero lub więcej parametrów;
- konstruktor nie zwraca wartości;
- konstruktor jest zawsze wywoływany przy użyciu operatora new.



Konstruktory w Javie działają tak samo jak w C++. Nie należy jednak zapominać, że obiekty w Javie są przechowywane na stercie i że konstruktor musi być wywoływany przy użyciu operatora new. Programiści C++ często zapominają o tym operatorze, programując w Javie:

```
Employee number007("James Bond", 100000, 1950, 1, 1);  
// C++, nie Java.
```

Powyższy kod zadziała w języku C++, ale nie w Javie.



Należy pamiętać, aby nie utworzyć zmiennej lokalnej o takiej samej nazwie jak pole klasy. Na przykład w poniższym kodzie wysokość pensji nie zostanie ustawiona:

```
public Employee(String n, double s, . . .)  
{  
    String name = n; // błąd  
    double salary = s; // błąd  
    . . .  
}
```

Konstruktor deklaruje dwie zmienne **lokalne** o nazwach name i salary. Są one dostępne wyłącznie w tym konstruktorze. **Przesłaniają** pola klasy o takich samych nazwach. Niektórzy programiści — wliczając autorów niniejszej książki — piszą szybciej, niż myślą, ponieważ mają nawyk dodawania nazwy typu. Jest to bardzo nieprzyjemny rodzaj błędu, który trudno wytropić. Trzeba pamiętać, aby nie nadać żadnej zmiennej w metodzie takiej samej nazwy jak zmiennej pola klasy.

4.3.5. Parametry jawne i niewjawne

Metody działają na obiektach i mają dostęp do zmiennych składowych tych obiektów. Na przykład poniższa metoda:

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

```

    salary += raise;
}

```

ustawia nową wartość zmiennej składowej `salary` obiektu, na rzecz którego zostanie wywołana. Spójrzmy na poniższe wywołanie:

```
number007.raiseSalary(5);
```

Spowoduje ono zwiększenie o 5% wartości zmiennej składowej `number007.salary`. A dokładniej powyższe wywołanie wykonuje następujące instrukcje:

```

double raise = number007.salary * 5 / 100;
number007.salary += raise;

```

Metoda `raiseSalary` pobiera dwa parametry. Pierwszy z nich, zwany parametrem **niejawnym** (ang. *implicit parameter*), to obiekt klasy `Employee`, który znajduje się przed nazwą metody. Drugi parametr, liczba w nawiasach za nazwą metody, to parametr **jawny** (ang. *explicit parameter*).

Jak widać, parametry jawne są wypisane w deklaracji metody, na przykład `double byPercent`. Parametr niejawnym nie pojawia się w deklaracji metody.

Słowo kluczowe `this` odnosi się we wszystkich metodach do parametru niejawnego. Metodę `raiseSalary` można by było napisać następująco:

```

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

```

Niektórzy programiści wolą ten styl pisania kodu, ponieważ wyraźnie odróżnia on zmienne składowe od zmiennych lokalnych.



W języku C++ metody z reguły deklaruje się poza klasą:

```

void Employee::raiseSalary(double byPercent) // C++, nie Java.
{
    .
}

```

Metoda zdefiniowana w klasie staje się automatycznie metodą wstawianą (ang. *inline method*).

```

class Employee
{
    .
    int getName() { return name; } // inline w C++
}

```

W Javie wszystkie metody są zdefiniowane w klasie. Nie są jednak z tego powodu metodami wstawianymi. Znalezienie okazji do wstawienia treści metody jest zadaniem maszyny wirtualnej. Kompilator JIT szuka wywołań krótkich metod, które są często używane, ale nie są przesłaniane, i optymalizuje je.

4.3.6. Korzyści z hermetyzacji

Przyjrzyjmy się teraz uważniej niezbyt skomplikowanym metodom `getName`, `getSalary` i `getHireDay`:

```
public String getName()
{
    return name;
}
public double getSalary()
{
    return salary;
}
public Date getHireDay()
{
    return hireDay;
}
```

Są to oczywiście przykłady metod akcesora. Ponieważ zwracają wartości składowych obiektów, czasami nazywane są **metodami dostępu do pól** (ang. *field accessor*).

Czy nie byłoby prościej, gdyby pola `name`, `salary` i `hireDay` były publiczne? Wtedy nie trzeba by było tworzyć dla nich oddzielnich metod.

Chodzi o to, że pole `name` jest tylko do odczytu. Po ustawieniu jego wartości za pomocą konstruktora nie ma sposobu, aby tę wartość zmienić. W ten sposób zyskujemy gwarancję, że pole `name` nigdy nie zostanie uszkodzone.

Pole `salary` nie jest tylko do odczytu, ale jego wartość można zmienić wyłącznie przy użyciu metody `raiseSalary`. Jeśli wartość tego pola jest nieprawidłowa, wiadomo, że trzeba poszukać błędu w tej metodzie. Gdyby pole `salary` było publiczne, źródło problemów z nim mogłoby się znajdować wszędzie.

Czasami konieczne jest pobranie i ustawienie wartości składowej obiektu. Do tego celu potrzebne są **trzy** rzeczy:

- prywatne pole,
- publiczna metoda akcesora,
- publiczna metoda mutatora.

To oznacza więcej pracy niż w przypadku utworzenia publicznego pola danych, ale metoda ta ma też zalety.

Po pierwsze, przy wprowadzaniu zmian wewnętrz implementacji wystarczy tylko zmiana w kodzie metod klasy. Jeśli na przykład sposób przechowywania nazwisk zmieni się na następujący:

```
String firstName;
String lastName;
```

metodę `getName` można zmodyfikować w taki sposób, aby zwracała:

```
firstName + " " + lastName
```

Zmiana ta jest niewidoczna dla reszty programu.

Oczywiście metody akcesora i mutatora mogą mieć dużo pracy z konwersją ze starej reprezentacji danych na nową. Prowadzi to jednak do drugiej korzyści: metody mutatora mogą sprawdzać błędy, podczas gdy kod, który po prostu przypisuje wartość polu, takiej możliwości nie ma. Na przykład metoda `setSalary` może pilnować, aby pensja pracownika nie była niższa od zera.



Pamiętaj, aby nie pisać metod akcesora, które zwracają referencje do obiektów zmienialnych. Złamaliśmy tę zasadę w klasie `Employee`, w której metoda `getHireDay` zwraca obiekt klasy `Date`:

```
class Employee
{
    private Date hireDay;
    ...
    public Date getHireDay()
    {
        return hireDay;
    }
    ...
}
```

W ten sposób łamiemy zasadę hermetyzacji! Spójrzmy na poniższy niesforny fragment kodu:

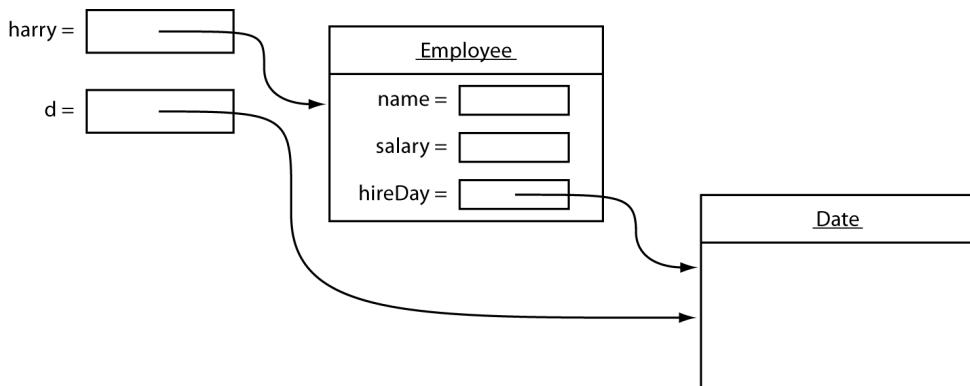
```
Employee harry = ...;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);
// Dodajmy Harry'emu 10 lat stażu pracy.
```

Powód nie jest oczywisty. Zarówno zmienna `d`, jak i `harry.hireDay` odwołują się do tego samego obiektu (zobacz rysunek 4.5). Wywołanie metody mutatora na rzecz obiektu `d` automatycznie powoduje modyfikację prywatnego stanu obiektu klasy `Employee`!

Jeśli konieczne jest zwrócenie referencji do modyfikowanego obiektu, należy najpierw **sklonować** ten obiekt. Klon jest wierną kopią obiektu i jest przechowywany w osobnej lokalizacji. Szczegółowy opis technik klonowania znajduje się w rozdziale 6. Poniżej znajduje się poprawny kod:

```
class Employee
{
    ...
    public Date getHireDay()
    {
        return hireDay.clone();
    }
    ...
}
```

Należy pamiętać, aby do tworzenia kopii modyfikowalnych obiektów używać metody `clone`.



Rysunek 4.5. Zwracanie referencji do zmienialnej składowej

4.3.7. Przywileje klasowe

Wiadomo, że każda metoda ma dostęp do prywatnych danych obiektu, na rzecz którego została wywołana. Jednak dla wielu osób zaskakujące jest to, że każda metoda ma dostęp do prywatnych danych **wszystkich obiektów swojej klasy**. Weźmy na przykład metodę `equals`, za pomocą której porównamy dwóch pracowników:

```
class Employee
{
    boolean equals(Employee other)
    {
        return name.equals(other.name);
    }
}
```

Typowe wywołanie tej metody wygląda następująco:

```
if (harry.equals(boss)) . . .
```

Metoda ta ma dostęp do prywatnych składowych obiektu `harry`, co nie jest żadnym zaskoczeniem. Uzyskuje jednak też dostęp do prywatnych składowych obiektu `boss`. Jest to dozwolone, ponieważ obiekt `boss` jest typu `Employee`, a metody klasy `Employee` mają dostęp do prywatnych pól **wszystkich** obiektów tej klasy.



W C++ obowiązuje ta sama zasada. Każda metoda danej klasy ma dostęp do prywatnych składowych tej samej klasy, nie tylko parametru niejawnego.

4.3.8. Metody prywatne

Implementując klasę, wszystkie jej pola oznaczamy słowem kluczowym `private`, ponieważ dane publiczne mogą być niebezpieczne. Ale co z metodami? Podczas gdy w większości metody są publiczne, w określonych warunkach używa się metod prywatnych. Czasami

korzystne może być rozbicie kodu wykonującego obliczenia na kilka osobnych metod pomocniczych. Zazwyczaj nie powinny one wchodzić w skład interfejsu publicznego — mogą być zbyt blisko bieżącej implementacji lub wymagać specjalnego protokołu albo specjalnej kolejności wywoływania. Takie metody najlepiej implementować jako prywatne.

Aby utworzyć prywatną metodę, należy zamiast słowa kluczowego `public` użyć słowa `private`.

Jeśli metoda jest prywatna, nie ma obowiązku dbać o jej dostępność w przypadku zmiany implementacji. Po wprowadzeniu zmian w sposobie reprezentacji danych jej implementacja może się okazać **trudniejsza** lub zupełnie **niepotrzebna**. Chodzi o to, że jeśli metoda jest prywatna, wiadomo, że nikt jej nie używa poza klasą, i można się jej bez obawy pozbyć. Jeśli metoda jest publiczna, nie można jej usunąć, ponieważ może z niej korzystać jakiś inny fragment programu.

4.3.9. Stałe jako pola klasy

W deklaracji pola klasy można użyć słowa kluczowego `final`. Tego typu pole musi być zainicjowane przy tworzeniu obiektu. To znaczy, że przed zakończeniem działania konstruktora wartość takiego pola musi zostać ustawiona. Po utworzeniu obiektu wartość tej składowej nie może być zmieniana. Na przykład pole `name` klasy `Employee` można zadeklarować przy użyciu słowa kluczowego `final`, ponieważ jego wartość po utworzeniu obiektu nigdy się nie zmienia — nie ma metody `setName`.

```
class Employee
{
    .
    .
    private final String name;
}
```

Modyfikator `final` jest szczególnie przydatny do deklaracji pól o typach podstawowych lub **klas niezmiennych** (ang. *immutable class*) — klasa niezmienna to taka, której metody nie zmieniają stanu swoich obiektów. Przykładem takiej klasy jest klasa `String`. Modyfikator `final` zastosowany do pól klasy zmiennej (ang. *mutable class*) może wprowadzać w błąd. Na przykład zapis:

```
private final Date hiredate;
```

oznacza tylko, że referencja do obiektu przechowywana w zmiennej `hiredate` nie może się zmienić po utworzeniu tego obiektu. Nie oznacza to, że obiekt `hiredate` jest stały. Każda metoda może wywołać mutator `setTime` na rzecz obiektu, do którego odwołuje się zmienna `hiredate`.

4.4. Pola i metody statyczne

We wszystkich oglądanych do tej pory przykładach metoda `main` jest opatrzona modyfikatorem `static`. Nadszedł czas na wyjaśnienie, do czego służy ten modyfikator.

4.4.1. Pola statyczne

W klasie może być tylko jeden egzemplarz danego pola, które jest określone jako statyczne. W przeciwnieństwie do tego każdy obiekt ma swoją własną kopię każdego pola klasy. Założymy na przykład, że każdemu pracownikowi chcemy przypisać unikalny numer identyfikacyjny. W tym celu dodajemy do klasy `Employee` pole niestatyczne `id` i pole statyczne `nextId`:

```
class Employee
{
    private static int nextId = 1;

    private int id;
    ...
}
```

Każdy obiekt klasy `Employee` będzie miał własne pole `id`, ale pole `nextId` będzie współdzielone przez wszystkie obiekty tej klasy. Innymi słowy, jeśli zostanie utworzonych 1000 obiektów klasy `Employee`, powstanie 1000 składowych obiektu o nazwie `id` — po jednej dla każdego obiektu, ale pole statyczne o nazwie `nextId` będzie tylko jedno. Pole to będzie istniało, nawet jeśli nie będzie ani jednego obiektu klasy `Employee`. Pole to należy do klasy, a nie do konkretnego obiektu.



W niektórych obiektowych językach programowania pola statyczne są nazywane **polami klasowymi** (ang. *class field*). Termin statyczny jest niezbyt udaną pozostępstwą po języku C++.

Przeanalizujmy implementację prostej metody:

```
public void setId()
{
    id = nextId;
    nextId++;
}
```

Ustawmy numer identyfikacyjny pracownika dla obiektu `harry`:

```
harry.setId();
```

Pole `id` obiektu `harry` zostaje ustawione na aktualną wartość pola statycznego `nextId`, po czym wartość tego pola jest zwiększana o 1:

```
harry.id = Employee.nextId;
Employee.nextId++;
```

4.4.2. Stałe statyczne

Zmienne statyczne spotyka się dosyć rzadko. Znacznie częściej zdarzają się stałe statyczne. Na przykład w klasie `Math` znajduje się definicja poniższej stałej statycznej:

```
public class Math
{
    ...
}
```

```
public static final double PI = 3.14159265358979323846;
}
```

Dostęp do tej stałej można uzyskać, pisząc Math.PI.

Gdyby słowo kluczowe static zostało pominięte, PI byłoby zwykłym polem klasy Math. To znaczy, że dostęp do niego prowadziłby poprzez obiekt klasy Math i każdy obiekt tej klasy miałby składową PI.

Inną często używaną stałą statyczną jest System.out. Jej deklaracja w klasie System wygląda następująco:

```
public class System
{
    ...
    public static final PrintStream out = . . .;
    ...
}
```

Wielokrotnie już sygnalizowaliśmy, że nie należy deklarować pól jako publicznych, ponieważ każdy może je zmodyfikować. Natomiast nie mamy nic przeciwko stałym publicznym (tzn. polem opatrzonych słowem kluczowym final). Ze względu na to, że out to stała, nie można przypisać do niej innego strumienia:

```
System.out = new PrintStream(. . .); // Błąd — out to stała.
```



W klasie System dostępna jest metoda setOut, która umożliwia ustawienie stałej System.out na inny strumień. Jak to możliwe? Metoda setOut jest metodą **rodzimą**, której implementacja została napisana w innym niż Java języku programowania. Metody rodzime mogą obchodzić mechanizmy kontrolne języka Java. Rozwiązanie to jest jednak bardzo rzadko stosowane i nie należy się nim posługiwać.

4.4.3. Metody statyczne

Metody statyczne nie działają na obiektach. Przykładem takiej metody jest metoda pow dostępna w klasie Math. Wyrażenie:

```
Math.pow(x, a)
```

oblicza wartość działania x^a . Nie jest do tego potrzebny żaden obiekt klasy Math. Innymi słowy, nie ma parametru niejawnego.

Metody statyczne można zapamiętać jako takie, które nie mają parametru this (w metodzie niestatycznej parametr this odwołuje się do parametru niejawnego metody — zobacz podrozdział 4.3.5, „Parametry jawne i niejawnne”).

Ponieważ metody statyczne nie działają na obiektach, za ich pomocą nie można operować na składowych obiektów. Mają natomiast dostęp do pól statycznych swoich klas. Poniżej znajduje się przykład takiej metody statycznej:

```
public static int getNextId()
{
    return nextId; // Zwraca wartość pola statycznego.
}
```

Wywołanie tej metody wymaga podania nazwy jej klasy:

```
int n = Employee.getNextId();
```

Czy można w tej metodzie pominąć słowo kluczowe `static`? Tak, ale wtedy do jej wywołania potrzebna by była referencja do obiektu typu `Employee`.



Do wywołania metody statycznej można użyć obiektu. Jeśli na przykład `harry` jest obiektem klasy `Employee`, zamiast wywołania `Employee.getNextId()` trzeba by było użyć `harry.getNextId()`. Taki sposób zapisu wydaje nam się mylący. Metoda `getNextId` w ogóle nie bierze pod uwagę obiektu `harry` przy obliczaniu wyniku. Zalecamy wywoływanie metod statycznych przy użyciu nazwy klasy, a nie nazwy obiektu.

Metody statyczne mają dwojakie zastosowanie:

- kiedy metoda nie wymaga dostępu do stanu obiektu, ponieważ wszystkie potrzebne jej parametry są dostarczane w postaci parametrów jawnych (na przykład `Math.pow`);
- kiedy metoda potrzebuje dostępu tylko do pól statycznych (na przykład `Employee.getNextId`).



C++ Pola i metody statyczne w Javie mają takie same przeznaczenie jak w języku C++. Różnica pomiędzy tymi językami polega w tym przypadku na składni. W C++ dostęp do pola statycznego lub metody statycznej poza jej zakresem uzyskuje się przy użyciu operatora `::`, np. `Math::PI`.

Termin „statyczny” (ang. `static`) ma ciekawą historię. Został on po raz pierwszy użyty w języku C do określenia zmiennej lokalnej, która nie znikła po wyjściu z bloku. Wtedy nazwa ta miała sens — zmienna pozostawała w pamięci i była dostępna po ponownym wejściu do bloku. Drugie znaczenie słowa kluczowego `static` dotyczyło zmiennych i funkcji globalnych, do których nie było dostępu z innych plików. Powodem użycia tego słowa była chęć uniknięcia wprowadzania nowego słowa kluczowego. W końcu w języku C++ słowo `static` zyskało swoje trzecie znaczenie — oznacza zmienne i funkcje, które należą do danej klasy, ale nie należą do jej obiektów. To samo znaczenie ma niniejsze słowo w Javie.

4.4.4. Metody fabryczne

Oto jeszcze jeden często spotykany sposób użycia metod statycznych. Klasa `NumberFormat` tworzy obiekty formatujące dla różnych stylów formatowania przy użyciu **metod fabrycznych** (ang. *factory method*).

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // Drukuję 0.10 dol.
System.out.println(percentFormatter.format(x)); // Drukuję 10%
```

Dlaczego klasa `NumberFormat` nie wykorzystuje konstruktora? Są ku temu dwa powody:

- Konstruktorom nie można zmieniać nazw. Konstruktor zawsze nazywa się tak samo jak klasa. W tym przypadku jednak były potrzebne dwie nazwy — dla przypadku stylu walutowego (`currency`) i procentowego (`percent`).
- Typ obiektu tworzonego za pomocą konstruktora nie może być zmieniony. Natomiast metody fabryczne zwracają obiekty klasy `DecimalPoint` — podklasy, która dziedziczy po klasie `NumberFormat` (więcej informacji na temat dziedziczenia znajduje się w rozdziale 5.).

4.4.5. Metoda main

Zauważmy, że metody statyczne można wywoływać, nie mając żadnych obiektów. W ten sposób na przykład wywołujemy metodę `Math.pow`.

Z tego właśnie powodu metoda `main` jest statyczna.

```
public class Application
{
    public static void main(String[] args)
    {
        // Konstruowanie obiektów.
        .
        .
    }
}
```

Metoda `main` nie działa na żadnym obiekcie — kiedy program rozpoczyna działanie, nie ma w nim jeszcze żadnych obiektów. Jej zadaniem jest tworzenie i uruchamianie obiektów wymaganych przez program.

Listing 4.3 przedstawia prostą wersję klasy `Employee` zawierającą pole statyczne `nextId` i metodę statyczną `getNextId`. Program ten wstawia do tablicy trzy obiekty typu `Employee` i drukuje informacje o reprezentowanych przez nie pracownikach. Na końcu drukuje kolejny numer identyfikacyjny, aby zademonstrować działanie metody statycznej.

Listing 4.3. StaticTest/StaticTest.java

```
/*
 * Ten program demonstruje użycie metod statycznych.
 * @version 1.01 2004-02-19
 * @author Cay Horstmann
 */
public class StaticTest
{
    public static void main(String[] args)
    {
        // Wstawienie do tablicy staff trzech obiektów reprezentujących pracowników.
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Tomasz", 40000);
        staff[1] = new Employee("Dariusz", 60000);
        staff[2] = new Employee("Grzegorz", 65000);
    }
}
```



Każda klasa może zawierać metodę `main`, co jest bardzo przydatne przy przeprowadzaniu testów jednostkowych klas. Możemy na przykład wstawić metodę `main` do klasy `Employee`:

```
class Employee
{
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }
    .
    .
    public static void main(String[] args) // test jednostkowy
    {
        Employee e = new Employee("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    .
}
```

Aby przetestować działanie klasy `Employee` w odosobnieniu, należy użyć następującego polecenia:

```
java Employee
```

Jeśli klasa `Employee` wchodzi w skład większej aplikacji, uruchomienie tego programu za pomocą poniższego polecenia:

```
java Aplikacja
```

spowoduje, że metoda `main` klasy `Employee` nie zostanie wykonana.

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

int n = Employee.getNextId(); // Wywołanie metody statycznej.
System.out.println("Następny dostępny identyfikator=" + n);
}

class Employee
{
    private static int nextId = 1;

    private String name;
    private double salary;
    private int id;

    public Employee(String n, double s)
    {
```

```
        name = n;
        salary = s;
        id = 0;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public int getId()
    {
        return id;
    }

    public void setId()
    {
        id = nextId;           // Ustawienie identyfikatora na kolejny dostępny numer.
        nextId++;
    }

    public static int getNextId()
    {
        return nextId;         // Zwrócenie pola statycznego.
    }

    public static void main(String[] args) // test jednostkowy
    {
        Employee e = new Employee("Grzegorz", 50000);
        System.out.println(e.getName() + " " + e.getSalary());
    }
}
```

Klasa Employee zawiera statyczną metodę main przeznaczoną do testów jednostkowych. Aby uruchomić obie metody main, należy użyć poniższych poleceń:

```
java Employee
i
java StaticTest
```

4.5. Parametry metod

Zaczniemy od przeglądu terminów opisujących sposoby przekazywania parametrów do metod (lub funkcji) w różnych językach programowania. Termin **wywołanie przez wartość** (ang. *call by value*) oznacza, że metoda odbiera tylko wartość dostarczoną przez wywołującego. Natomiast **wywołanie przez referencję** (ang. *call by reference*) oznacza, że metoda odbiera

lokalizację zmiennej dostarczonej przez wywołującego. W związku z tym metoda może **zmodyfikować** wartość zmiennej przekazanej przez referencję, ale nie może tego zrobić ze zmienną przekazaną przez wartość. Określenia „wywołanie przez...” są standardowo używane w terminologii programistycznej do opisu parametrów metod i dotyczą nie tylko Javy; jest jeszcze jeden termin tego typu — **wywołanie przez nazwę** (ang. *call by name*), ale ma on już tylko znaczenie historyczne, ponieważ był stosowany w języku Algol — jednym z najstarszych języków programowania wysokiego poziomu.

W Javie **zawsze** stosowane są wywołania przez wartość. Oznacza to, że metoda otrzymuje kopię wartości wszystkich parametrów, a więc nie może zmodyfikować wartości przekazanych do niej zmiennych.

Przeanalizujmy na przykład poniższe wywołanie:

```
double percent = 10;
harry.raiseSalary(percent);
```

Bez względu na to, jaka jest implementacja tej metody, wiadomo, że po jej wywołaniu wartość zmiennej percent nadal będzie wynosiła 10.

Przyjrzyjmy się tej sytuacji nieco uważniej. Niech nasza metoda spróbuje potroić wartość swojego parametru:

```
public static void tripleValue(double x) // nie działa
{
    x = 3 * x;
}
```

Wywołajmy tę metodę:

```
double percent = 10;
tripleValue(percent);
```

To jednak nie działa. Po wywołaniu metody wartość zmiennej percent nadal wynosi 10. Oto opis zdarzeń:

- 1 Zmienna x jest inicjowana kopią wartości zmiennej percent (tzn. 10).
- 2 Wartość zmiennej x jest potrojona — teraz wynosi 30. Ale zmienna percent ma nadal wartość 10 (zobacz rysunek 4.6).
- 3 Metoda kończy działanie, a zmienna x nie jest już używana.

Są jednak dwa rodzaje parametrów metod:

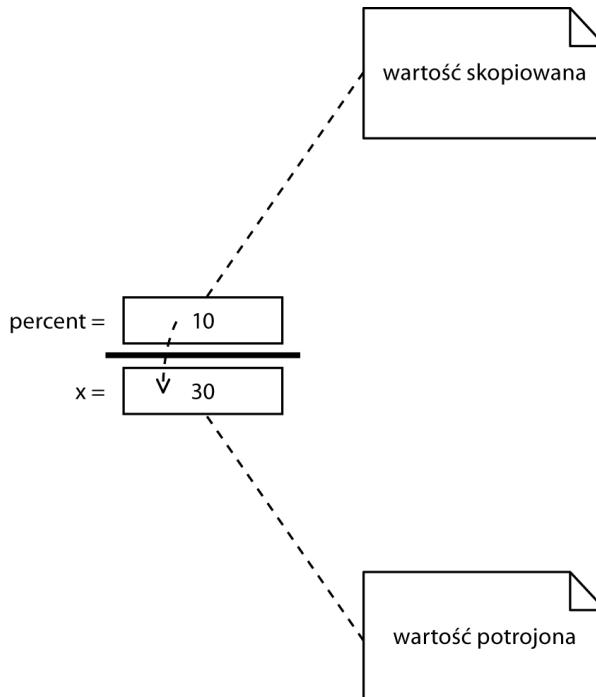
- typy podstawowe (liczby i wartości logiczne),
- referencje do obiektów.

Wiemy już, że metoda nie może zmienić wartości parametru typu podstawowego. Z parametrami obiektowymi jest inaczej. Można z łatwością utworzyć metodę, która potrafią pensję pracownika:

```
public static void tripleSalary(Employee x) // działa
{
    x.raiseSalary(200);
}
```

Rysunek 4.6.

Modyfikacja parametru liczbowego nie ma stałego efektu



Po uruchomieniu poniższego kodu:

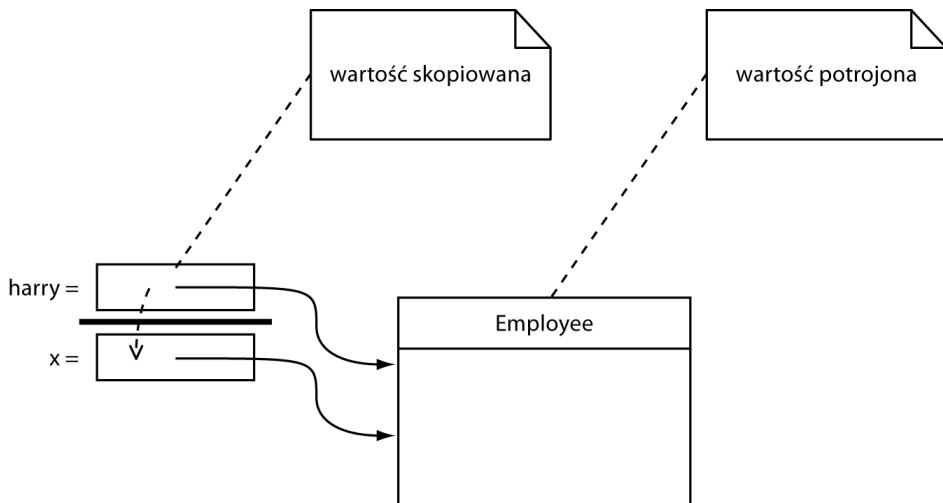
```
harry = new Employee(. . .);
tripleSalary(harry);
```

mają miejsce następujące zdarzenia:

1. Zmienna `x` jest inicjowana kopią wartości obiektu `harry`, to znaczy referencją do obiektu.
2. Metoda `raiseSalary` jest wywoływana na rzecz tej referencji. Pensja obiektu klasy `Employee`, do którego odwołuje się zarówno zmienna `x`, jak i `harry`, jest zwiększana o 200 procent.
3. Metoda kończy działanie i zmienna `x` nie jest dalej używana. Oczywiście zmienna obiektowa `harry` nadal odwołuje się do obiektu, którego pensja została potrojona (zobacz rysunek 4.7).

Jak widać, implementacja metody, która zmienia stan parametru w postaci obiektu, jest łatwą i często stosowaną metodą programowania. Prostota bierze się stąd, że metoda odbiera kopię referencji do obiektu i zarówno oryginał, jak i kopia odwołują się do tego samego obiektu.

W wielu językach programowania (zwłaszcza w C++ i Pascalu) parametry można przekazywać do metod na dwa sposoby: za pomocą wywołania przez wartość i przez referencję. Niektórzy programiści (i niestety niektórzy autorzy książek) uważają, że w Javie dla obiektów stosowane jest wywołanie przez referencję. To nieprawda. Ponieważ to błędne przekonanie jest bardzo powszechnne, warto szczegółowo przeanalizować przeciwny do niego przykład.



Rysunek 4.7. Modyfikacja parametru obiektowego ma stały efekt

Spróbowajmy napisać metodę zamieniającą dwa obiekty klasy Employee:

```
public static void swap(Employee x, Employee y) //nie działa
{
    Employee temp = x;
    x = y;
    y = temp;
}
```

Gdyby w Javie stosowane były wywołania przez referencję dla obiektów, ta metoda działałaby:

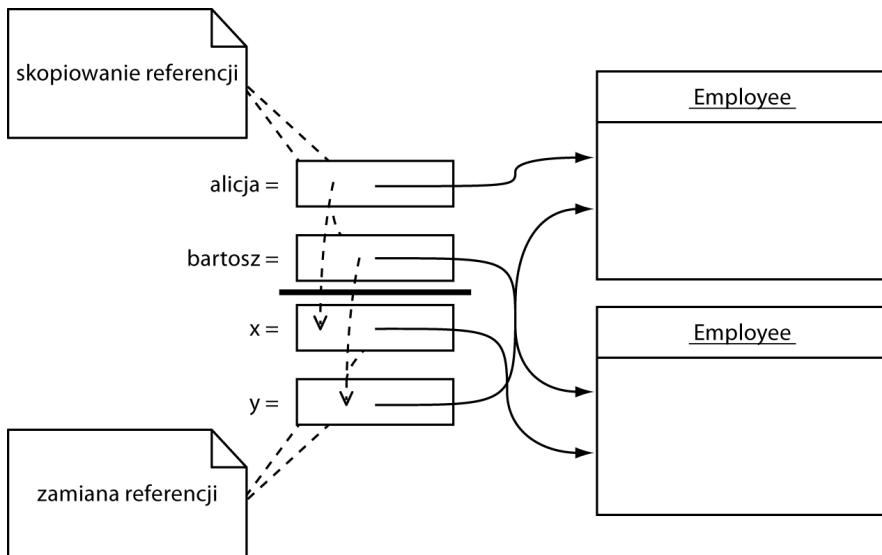
```
Employee a = new Employee("Alicja", . . .);
Employee b = new Employee("Bartosz", . . .);
swap(a, b);
// Czy a odwołuje się teraz do Bartosza, czy Alicji?
```

Jednak metoda ta nie zmienia referencji do obiektów przechowywanych w zmiennych a i b. Parametry x i y metody swap są inicjowane **kopiami** tych referencji. Następnie metoda przystępuje do zamiany tych kopii.

```
//x odwołuje się do Alicji, a y do Bartosza.
Employee temp = x;
x = y;
y = temp;
//Teraz x odwołuje się do Bartosza, a y do Alicji.
```

Wysiłek ten idzie jednak na marne. Zmienne parametrowe x i y wychodzą z użycia po zakończeniu metody. Oryginalne zmienne a i b nadal odwołują się do tych samych obiektów, do których odwoływały się przed wywołaniem metody (zobacz rysunek 4.8).

Powyższy opis problemu stanowi dowód na to, że język programowania Java nie używa wywołań przez referencję dla obiektów. W zamian **referencje do obiektów są przekazywane przez wartość**.



Rysunek 4.8. Zamiana parametrów obiektowych nie ma trwałego rezultatu

Oto zestawienie zasad dotyczących tego, co można, a czego nie można robić z parametrami metod w Javie:

- Metoda nie może zmodyfikować parametru typu podstawowego (czyli będącego liczbą lub wartością logiczną).
- Metoda może zmienić **stan** obiektu przekazanego jako parametr.
- Metoda nie może sprawić, aby parametr obiektowy zaczął się odwoływać do nowego obiektu.

Powyższe twierdzenia prezentuje program z listingu 4.4. Najpierw próbuje potroić wartość parametru liczbowego, co kończy się niepowodzeniem:

```
Testowanie tripleValue:
Przed: percent=10.0
Koniec metody: x=30.0
Po: percent=10.0
```

Następnie udaje się potroić pensję pracownika:

```
Testowanie tripleSalary:
Przed: salary=50000.0
Koniec metody: salary=150000.0
Po: salary=150000.0
```

Po zakończeniu działania metody stan obiektu, do którego odwołuje się zmienna `harry`, jest zmieniony. Jest to możliwe dzięki temu, że metoda ta zmodyfikowała stan obiektu poprzez kopię referencji do niego.

Na zakończenie program prezentuje niepowodzenie metody `swap`:

```
Testowanie swap:
Przed: a=Alicja
Przed: b=Grzegorz
Koniec metody: x=Grzegorz
Koniec metody: y=Alicja
Po: a=Alicja
Po: b=Grzegorz
```

Jak widać, parametry x i y zostały zamienione, ale zmienne a i b pozostały bez zmian.

C++ W C++ możliwe jest zarówno wywołanie przez wartość, jak i referencję. Parametry będące referencjami oznaczane są symbolem &. Na przykład metody void triple \rightarrow Value(double& x) czy void swap(Employee& x, Employee& y), które modyfikują swoje parametry, można z łatwością zaimplementować.

Listing 4.4. ParamTest/ParamTest.java

```
/*
 * Ten program demonstruje przekazywanie parametrów w Javie.
 * @version 1.00 2000-01-27
 * @author Cay Horstmann
 */
public class ParamTest
{
    public static void main(String[] args)
    {
        /*
         * Test 1. Metody nie mogą modyfikować parametrów liczbowych.
         */
        System.out.println("Testowanie tripleValue:");
        double percent = 10;
        System.out.println("Przed: percent=" + percent);
        tripleValue(percent);
        System.out.println("Po: percent=" + percent);

        /*
         * Test 2. Metody mogą zmieniać stan parametrów będących obiektami.
         */
        System.out.println("\nTestowanie tripleSalary:");
        Employee harry = new Employee("Grzegorz", 50000);
        System.out.println("Przed: salary=" + harry.getSalary());
        tripleSalary(harry);
        System.out.println("Po: salary=" + harry.getSalary());

        /*
         * Test 3. Metody nie mogą dodawać nowych obiektów do parametrów obiektowych.
         */
        System.out.println("\nTestowanie swap:");
        Employee a = new Employee("Alicja", 70000);
        Employee b = new Employee("Grzegorz", 60000);
        System.out.println("Przed: a=" + a.getName());
        System.out.println("Przed: b=" + b.getName());
        swap(a, b);
        System.out.println("Po: a=" + a.getName());
        System.out.println("Po: b=" + b.getName());
```

```
}

public static void tripleValue(double x)      //nie dziala
{
    x = 3 * x;
    System.out.println("Koniec metody: x=" + x);
}

public static void tripleSalary(Employee x)    //dziala
{
    x.raiseSalary(200);
    System.out.println("Koniec metody: salary=" + x.getSalary());
}

public static void swap(Employee x, Employee y)
{
    Employee temp = x;
    x = y;
    y = temp;
    System.out.println("Koniec metody: x=" + x.getName());
    System.out.println("Koniec metody: y=" + y.getName());
}

class Employee //Uproszczona klasa Employee.
{
    private String name;
    private double salary;

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

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

4.6. Konstruowanie obiektów

Umiemy już pisać proste konstruktory definiujące początkowy stan obiektów. Ponieważ jednak obiekty są niezwykle ważnym elementem języka, ich konstrukcję wspiera kilka różnych mechanizmów. Opisujemy je w poniższych podrozdziałach.

4.6.1. Przeciążanie

Przypomnijmy, że klasa GregorianCalendar miała więcej niż jeden konstruktor. Do wyboru były dwa:

GregorianCalendar today = new GregorianCalendar();

lub

GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);

Taka sytuacja nazywa się **przeciążaniem**. Przeciążanie to sytuacja, w której kilka metod ma taką samą nazwę (w tym przypadku konstruktor GregorianCalendar), ale różne parametry. Kompilator musi zdecydować, którą wersję wywoła. Decyzję podejmuje na podstawie dopasowania typów parametrów w nagłówkach różnych metod do typów wartości przekazanych w konkretnym wywołaniu. Jeśli niemożliwe jest dopasowanie parametrów lub istnieje więcej niż jedno dopasowanie, występuje błąd kompilacji (proces ten nazywa się **rozstrzyganiem przeciążania** — ang. *overloading resolution*).



W Javie można przeciążyć dowolną metodę. W związku z tym pełny opis metody składa się z nazwy i typów argumentów. Informacje te nazywane są **sygnaturą** metody. Na przykład klasa String zawiera cztery metody publiczne o nazwie indexOf. Oto ich sygnatury:

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Określenie typu zwrotnego nie wchodzi w skład sygnatury metody. Oznacza to, że nie można utworzyć dwóch metod o takich samych nazwach i typach parametrów, ale różnych typach zwrotnych.

4.6.2. Inicjacja pól wartością domyślnym

Jeśli wartość pola nie zostanie jawnie ustawiona w konstruktorze, pole to automatycznie przyjmie wartość domyślną — pola typów liczbowych są ustawiane na 0, wartości logicznych na false, a referencji do obiektów na null. Taki styl programowania jest jednak uważany za niewłaściwy. Z pewnością kod taki jest trudniejszy do zrozumienia, jeśli pola są inicjowane niewidocznie.



Na tym polega podstawowa różnica pomiędzy polami i zmiennymi lokalnymi. Zmienne lokalne muszą być jawnie inicjowane w metodzie. Natomiast jeśli pole klasy nie zostanie zainicjowane, zostanie automatycznie ustawione na wartość domyślną (0, false lub null).

Weźmy jako przykład klasę `Employee`. Wyobraźmy sobie, że nie określiliśmy w konstruktorze sposobu inicjacji niektórych jej pól. Domyślnie pole `salary` miałoby wartość 0, a pola `name` i `hireDay` miałyby wartości `null`.

Nie jest to jednak dobre rozwiązanie, ponieważ w wyniku wywołania metody `getName` lub `getHireDay` otrzymalibyśmy wartość `null`, której raczej nie oczekiwaliśmy:

```
Date h = harry.getHireDay();
calendar.setTime(h); // Jeżeli h ma wartość null, zostanie zgłoszony wyjątek.
```

4.6.3. Konstruktor bezargumentowy

Wiele klas zawiera konstruktor bezargumentowy tworzący obiekty o określonym domyślnym stanie początkowym. Poniżej znajduje się przykładowy konstruktor domyślny klasy `Employee`:

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = new Date();
}
```

Konstruktor domyślny jest stosowany, w przypadku gdy programista nie utworzy żadnego konstruktora. Konstruktor ten ustawia **wszystkie** pola na wartości domyślne. W związku z tym wszystkie dane liczbowe będące składowymi obiektu miałyby wartość 0, wartości logiczne byłyby ustawione na `false`, a zmienne obiektowe na `null`.

Jeśli klasa ma przynajmniej jeden konstruktor, ale nie ma konstruktora domyślnego, nie można tworzyć jej obiektów bez podania odpowiednich parametrów konstrukcyjnych. Na przykład pierwsza wersja klasy `Employee` na listingu 4.2 zawierała jeden konstruktor:

```
Employee(String name, double salary, int y, int m, int d)
```

W przypadku tej klasy utworzenie obiektu z wartościami domyślnymi nie byłoby możliwe. To znaczy, że poniższe wywołanie spowodowałoby błąd:

```
e = new Employee();
```

4.6.4. Jawna inicjacja pól

Dzięki możliwości przeciążania konstruktorów początkowy stan obiektu klasy może być ustawiany na wiele sposobów. Bez względu na wywoływany konstruktor nigdy nie zaszkodzi, jeśli każda składowa obiektu będzie miała jakąś sensowną wartość.



Należy pamiętać, że konstruktor domyślny jest dostępny **tylko** wtedy, gdy klasa nie ma żadnego innego konstruktora. Aby umożliwić tworzenie obiektów klasy, która ma już konstruktor, za pomocą widocznego poniżej wywołania:

```
new NazwaKlasy()
```

trzeba dostarczyć konstruktor domyślny (bezparametrowy). Oczywiście, jeśli wartości wszystkich pól mogą być domyślne, można napisać następujący konstruktor:

```
public NazwaKlasy()
{
}
```

Wystarczy przypisać każdemu polu w definicji klasy jakąś wartość. Na przykład:

```
class Employee
{
    .
    .
    private String name = "";
}
```

To przypisanie następuje przed wywołaniem konstruktora. Składnia ta jest szczególnie przydatna, jeśli wszystkie konstruktory klasy muszą ustawić określoną składową na tę samą wartość.

Wartość inicjująca nie musi być stała. W poniższym przykładzie pole jest inicjowane wywołaniem metody. Weźmy pod uwagę klasę Employee, w której każdy pracownik ma swój identyfikator id. Pole to może być inicjowane następująco:

```
class Employee
{
    private static int nextId;
    private int id = assignId();

    private static int assignId()
    {
        int r = nextId;
        nextId++;
        return r;
    }
    .
}
```



W C++ nie można bezpośrednio zainicjować pól klasy. Wszystkie pola muszą być ustawione w konstruktorze. W języku tym jednak można posługiwać się specjalną składnią w postaci listy inicjującej:

```
Employee::Employee(String n, double s, int y, int m, int d) //C++
: name(n),
  salary(s),
  hireDay(y, m, d)
{}
```

W Javie składnia taka nie jest potrzebna, ponieważ obiekty nie mają podobiektów, tylko wskaźniki do innych obiektów.

4.6.5. Nazywanie parametrów

Przy pisaniu bardzo prostych konstruktorów (a pisze się ich bardzo dużo) problemem może się okazać wymyślanie nazw dla parametrów.

My z reguły jesteśmy za stosowaniem nazw jednoliterowych:

```
public Employee(String n, double s)
{
    name = n;
    salary = s;
}
```

Wadą tej metody jest to, że stosowane w niej nazwy nic nie mówią o przeznaczeniu parametrów.

Niektórzy programiści przed nazwą każdego parametru stawiają przedrostek `a`:

```
public Employee(String aName, double aSalary)
{
    name = aName;
    salary = aSalary;
}
```

Jest to całkiem dobre rozwiązanie. Już na pierwszy rzut oka wiadomo, jakie jest przeznaczenie każdego z parametrów.

Inna często stosowana sztuczka wykorzystuje fakt, że zmienne parametryczne **przesłaniają** składowe obiektów o tej samej nazwie. Jeśli na przykład zostanie wywołany parametr o nazwie `salary`, to `salary` będzie się odnosić do parametru, a nie składowej obiektu. Aby uzyskać dostęp do tej składowej, trzeba wtedy napisać `this.salary`. Przypomnijmy sobie, że `this` oznacza parametr niejawnego, to znaczy obiekt, który jest konstruowany. Poniżej znajduje się przykład:

```
public Employee(String name, double salary)
{
    this.name = name;
    this.salary = salary;
}
```



W języku C++ często stosowaną praktyką jest poprzedzanie nazw składowych obiektów znakiem podkreślenia lub ustaloną literą (często wybór pada na litery `m` i `x`). Na przykład pole `salary` mogłoby mieć nazwę `salary`, `mSalary` lub `xSalary`. Technika ta jest mało rozpowszechniona wśród programistów Javy.

4.6.6. Wywoływanie innego konstruktora

Słowo kluczowe `this` odwołuje się do parametru niejawnego metody. Ma ono jednak jeszcze jedno zastosowanie.

Jeśli **pierwsza instrukcja konstruktora** ma postać `this(...)`, to konstruktor ten wywołuje inny konstruktor tej samej klasy. Oto typowy przykład takiej sytuacji:

```
public Employee(double s)
{
    // Wywołuje Employee(String, double)
    this("Employee #" + nextId, s);
    nextId++;
}
```

Kiedy wywołamy `new Employee(6000)`, konstruktor `Employee(double)` wywoła konstruktor `Employee(String, double)`.

Słowo kluczowe użyte w takim przypadku jest bardzo przydatne. Wspólny kod konstruktorów wystarczy napisać tylko jeden raz.



Referencja `this` w Javie jest identyczna ze wskaźnikiem `this` w C++. Jednak w tym drugim języku jeden konstruktor nie może wywołać innego konstruktora. Aby wydzielić wspólny kod inicjujący w C++, konieczne jest napisanie osobnej metody.

4.6.7. Bloki inicjujące

Znamy już dwa sposoby inicjacji pól danych:

- poprzez ustawienie wartości w konstruktorze;
- poprzez przypisanie wartości w deklaracji.

Jest jeszcze trzeci sposób polegający na zastosowaniu **bloku inicjującego**. W deklaracji klasy mogą się znajdować dowolne bloki kodu. Zawarte w nich instrukcje są wykonywane za każdym razem, gdy konstruowany jest obiekt danej klasy. Na przykład:

```
class Employee
{
    private static int nextId;

    private int id;
    private String name;
    private double salary;

    // Blok inicjujący obiektu.
    {
        id = nextId;
        nextId++;
    }

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }
}
```

```

public Employee()
{
    name = "";
    salary = 0;
}

...
}

```

Najpierw zostanie zainicjowane pole `id` w bloku inicjującym, bez względu na to, który konstruktor zostanie wywołany. Blok inicjujący jest wykonywany na początku, a po nim instrukcje zawarte w konstruktorze.

Sposób ten nigdy nie jest niezbędny i nie jest zbyt powszechnie stosowany. Zazwyczaj prościej jest umieścić kod inicjujący wewnątrz konstruktora.



W bloku inicjującym można ustawać wartości pól, mimo że ich definicje znajdują się dopiero w dalszej części klasy. Aby jednak uniknąć cyklicznych definicji, nie można odczytywać wartości pól, które są inicjowane później. Zasady te zostały szczegółowo opisane w sekcji 8.3.2.3 specyfikacji języka Java (<http://docs.oracle.com/javase/specs>). Ponieważ reguły te są na tyle skomplikowane, że nawet programiści kompilatorów mieli z nimi problemy — wczesne wersje Javy zawierały subtelne błędy w ich implementacji — zalecamy umieszczenie bloków inicjujących za definicjami pól.

Zastosowanie wszystkich możliwych sposobów inicjacji pól danych może ujemnie wpływać na czytelność kodu. Kiedy wywoływany jest konstruktor, mają miejsce następujące zdarzenia:

1. Wszystkie pola są inicjowane wartościami domyślnymi (`0`, `false` lub `null`).
2. Wszystkie inicjatory i bloki inicjujące są wykonywane w takiej kolejności, w jakiej znajdują się w klasie.
3. Jeśli w pierwszym wierszu konstruktora znajduje się wywołanie innego konstruktora, wykonywane są instrukcje innego konstruktora.
4. Wykonywane jest ciało konstruktora.

Oczywiście dobrze jest tak zorganizować swój kod inicjujący, aby inny programista mógł go bez większych problemów zrozumieć. Na przykład klasa, w której konstruktory są uzależnione od kolejności deklaracji pól danych, byłaby nieco dziwna i podatna na błędy.

Pole statyczne można zainicjować, podając wartość początkową lub korzystając ze statycznego bloku inicjującego. Pierwszy z tych sposobów już znamy:

```
private static int nextId = 1;
```

Jeśli inicjacja pól statycznych klasy odbywa się za pomocą bardziej złożonego kodu, można się posłużyć statycznym blokiem inicjującym.

Kod należy umieścić w bloku opatrzonym etykietą `static`. Oto przykład: chcemy, aby numery identyfikacyjne pracowników zaczynały się od losowej liczby całkowitej mniejszej od 10 000.

```
// statyczny blok inicjujący
static
{
    Random generator = new Random();
    nextId = generator.nextInt(10000);
}
```

Inicjacja statyczna następuje w chwili pierwszego załadowania klasy. Pola statyczne, podobnie jak zmienne składowe, przybierają wartości domyślne 0, false lub null, jeśli nie zostaną im nadane jawnie inne wartości. Inicjatory pól statycznych i statyczne bloki inicjujące są wykonywane w takiej kolejności, w jakiej znajdują się w deklaracji klasy.



Oto czarodziejska sztuczka w języku Java, dzięki której zaimponujesz swoim współpracownikom. Można napisać program „Witaj, świecie!” bez metody main.

```
public class Hello
{
    static
    {
        System.out.println("Witaj, świecie");
    }
}
```

W wyniku wywołania powyższej klasy za pomocą polecenia `java Hello` statyczny blok inicjujący wydrukuje napis „Witaj, świecie”, a potem pojawi się paskudny komunikat o błędzie informujący o braku metody main. Można uniknąć tego połajania, umieszczając na końcu bloku inicjującego wywołanie `System.exit(0)`.

Program na listingu 4.5 demonstruje w praktyce zagadnienia, które zostały omówione w tym podrozdziale:

- przeciążanie konstruktorów,
- wywołanie innego konstruktora za pomocą słowa kluczowego `this(...)`,
- konstruktor domyślny,
- zastosowanie bloku inicjującego obiektów,
- statyczny blok inicjujący,
- inicjacja zmiennych składowych.

Listing 4.5. ConstructorTest/ConstructorTest.java

```
import java.util.*;

/**
 * Ten program demonstruje techniki konstrukcji obiektów.
 * @version 1.01 2004-02-19
 * @author Cay Horstmann
 */
public class ConstructorTest
{
    public static void main(String[] args)
    {
        // Wstawienie do tablic staff trzech obiektów klasy Employee.
```

```

Employee[] staff = new Employee[3];

staff[0] = new Employee("Hubert", 40000);
staff[1] = new Employee(60000);
staff[2] = new Employee();

// Wydruk informacji o wszystkich obiektach klasy Employee.
for (Employee e : staff)
    System.out.println("name=" + e.getName() + ", id=" + e.getId() + ", salary=" +
        e.getSalary());
}

class Employee
{
    private static int nextId;

    private int id;
    private String name = ""; // Inicjacja zmiennej składowej obiektu.
    private double salary;

    // Statyczny blok inicjujący.
    static
    {
        Random generator = new Random();
        // Ustawienie zmiennej nextId na losową liczbę całkowitą z przedziału 0 – 9999.
        nextId = generator.nextInt(10000);
    }

    // Blok inicjujący obiektów.
    {
        id = nextId;
        nextId++;
    }

    // Trzy konstruktory przeciążone.
    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public Employee(double s)
    {
        // Wywołanie konstruktora Employee(String, double).
        this("Employee #" + nextId, s);
    }

    // Konstruktor domyślny.
    public Employee()
    {
        // Zmienna name zainicjowana wartością "" — patrz niżej.
        // Zmienna salary nie jest jawnie ustawiona — inicjacja wartością 0.
        // Zmienna id jest inicjowana w bloku inicjującym.
    }

    public String getName()
    {

```

```

        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public int getId()
    {
        return id;
    }
}

```

java.util.Random 1.0

■ Random()

Tworzy nowy generator liczb losowych.

■ int nextInt(int n) 1.2

Zwraca losową liczbę z przedziału od 0 do n-1.

4.6.8. Niszczenie obiektów i metoda finalize

W niektórych językach programowania, zwłaszcza w C++, dostępne są tak zwane destruktory. Metody te wykonują pewne operacje porządkowe, kiedy dany obiekt wyjdzie z użytku. Ich najczęstszą czynnością jest przywracanie pamięci przydzielonej obiektom. Ponieważ w Javie zastosowano mechanizm automatycznego usuwania nieużytków, nie trzeba tego robić ręcznie. Dlatego w języku Java nie ma destruktatorów.

Oczywiście niektóre obiekty korzystają z innych zasobów niż pamięć, jak np. pliki lub uchwyty do innych obiektów, które wykorzystują zasoby systemowe. W takiej sytuacji koniecznie trzeba zwrócić do ponownego użytku wykorzystywany zasób, kiedy przestanie być potrzebny.

Do każdej klasy można dodać metodę finalize. Jest ona wywoływana przed usunięciem obiektu przez system zbierania nieużytków. **Nie należy jednak polegać na metodzie finalize** do przywracania zasobów, których jest mało, ponieważ nigdy nie wiadomo, kiedy nastąpi jej wywołanie.



Wywołanie metody `System.runFinalizersOnExit(true)` gwarantuje, że metody finalizujące zostaną wywołane przed zakończeniem programu. Metoda ta nie jest jednak bezpieczna i jej stosowanie nie jest zalecane. Alternatywne rozwiązanie polega na wykonaniu pewnych czynności w momencie zamknięcia programu za pomocą metody `Runtime.addShutdownHook` — szczegółowe informacje na ten temat można znaleźć w dokumentacji API.

Jeśli dany zasób musi być zamknięty natychmiast po zakończeniu jego używania, trzeba o to zadbać we własnym zakresie. Do tego celu służy metoda `close`, którą programista

wywołuje w celu skasowania określonych zasobów i gdy skończy pracę z obiektem. W części 11.2.4, „Instrukcja try z zasobami”, dowiesz się, jak sprawić, aby metoda ta była wywoływana automatycznie.

4.7. Pakiety

Wygodnym sposobem na organizację pracy i oddzielenie własnych klas od pozostałych jest umieszczanie klas w tak zwanych **pakietach** (ang. *packages*).

Standardowa biblioteka Javy składa się z wielu pakietów, do których należą `java.lang`, `java.util`, `java.net` itd. Standardowe pakiety Javy mają strukturę hierarchiczną. Można je umieszczać jedne w drugich, podobnie jak w przypadku katalogów i podkatalogów. Wszystkie standardowe pakiety Javy znajdują się w pakietach `java` i `javax`.

Głównym powodem stosowania pakietów jest chęć uniknięcia kolizji nazw klas. Przypuśćmy, że dwóch niezależnych programistów wpadnie na świetny pomysł napisania klasy o nazwie `Employee`. Dopóki obie te klasy znajdują się w osobnych pakietach, nie ma żadnego konfliktu. Aby zachować unikatowość nazw pakietów, firma Sun zaleca stosowanie w tych nazwach odwróconych domen internetowych (które są unikatowe). W obrębie takiego pakietu można następnie tworzyć kolejne podpakiety. Na przykład jeden z programistów zarejestrował domenę `horstmann.com`. Po odwróceniu otrzymujemy `com.horstmann`. Pakiet ten można następnie podzielić na podpakiety o nazwach typu `com.horstmann.core.java`.

Kompilator nie rozpoznaje żadnych powiązań pomiędzy pakietami i podpakietami. Na przykład pakiety `java.util` i `java.util.jar` nie mają ze sobą nic wspólnego. Każdy z nich jest niezależnym pakietem klas.

4.7.1. Importowanie klas

Każda klasa może używać wszystkich klas ze swojego pakietu i wszystkich klas **publicznych** z innych pakietów.

Dostęp do klasy publicznej z innego pakietu można uzyskać na dwa sposoby. Pierwszy z nich polega na dodaniu pełnej nazwy pakietu przed nazwą **każdej** klasy. Na przykład:

```
java.util.Date today = new java.util.Date();
```

Ta metoda jest oczywiście bardzo pracochłonna. Prostszy i częściej stosowany sposób polega na użyciu instrukcji `import`. Instrukcja ta umożliwia stosowanie skróconego zapisu odwołań do klas w pakiecie. Dzięki jej zastosowaniu nie trzeba pisać pełnych nazw klas.

Można zainportować cały pakiet lub tylko jedną klasę, a instrukcje `import` powinny się znajdować na samym początku pliku źródłowego (ale pod instrukcjami `package`). Na przykład poniższa instrukcja importuje wszystkie klasy znajdujące się w pakiecie `java.util`:

```
import java.util.*;
```

Dzięki temu w zapisie:

```
    Date today = new Date();
```

nie jest potrzebny przedrostek określający pakiet. Można także zaimportować tylko określona klasę z pakietu:

```
    import java.util.Date;
```

Zapis `java.util.*` jest prostszy i nie wywiera żadnego wpływu na rozmiar kodu. Jednak dzięki importowi poszczególnych klas oddziennie osoba czytająca kod może się łatwiej zorientować, które klasy są w użyciu.



W edytorze Eclipse dostępna jest opcja *Organize Imports*, którą można znaleźć w menu *Source*. Po jej użyciu takie importy pakietów jak `java.util.*` są automatycznie zastępowane listą importów konkretnych klas, np.:

```
    import java.util.ArrayList;
    import java.util.Date;
```

Funkcja ta jest niezwykle przydatnym narzędziem.

Należy jednak pamiętać, że symbolu `*` można użyć do importu tylko jednego pakietu. Zapis `java.*` lub `java.*.*` do importu wszystkich pakietów z przedrostkiem `java` jest niedozwolony.

W większości przypadków programista nie interesuje się zbytnio importowanymi pakietami. Jedyna sytuacja, w której taka uwaga jest potrzebna, występuje wtedy, gdy dochodzi do konfliktu nazw. Na przykład zarówno pakiet `java.util`, jak i `java.sql` mają klasę `Date`. Wyobraźmy sobie, że napisaliśmy program importujący oba te pakiety.

```
    import java.util.*;
    import java.sql.*;
```

Użycie klasy `Date` w takiej sytuacji spowoduje błąd komplikacji:

```
    Date today; // Błąd - java.util.Date czy java.sql.Date?
```

Kompilator nie wie, której klasy o nazwie `Date` użyć. Problem ten można rozwiązać, dodając instrukcję importu konkretnej klasy:

```
    import java.util.*;
    import java.sql.*;
    import java.util.Date;
```

Co zrobić w sytuacji, w której potrzebne są obie te klasy? Konieczne jest używanie za każdym razem pełnej nazwy pakietu z nazwą klasy.

```
    java.util.Date deadline = new java.util.Date();
    java.sql.Date today = new java.sql.Date(...);
```

Lokalizacja klas w pakietach należy do **kompilatora**. Kod bajtowy w plikach klas zawsze zawiera pełne nazwy pakietów w odwołaniach do innych klas.



Programiści języka C++ często mylą instrukcję `import` z dyrektywą `#include`. Nie mają one ze sobą nic wspólnego. W C++ konieczne jest użycie dyrektywy `#include`, aby dołączyć zewnętrzne deklaracje, ponieważ kompilator C++ nie przeszukuje żadnych plików z wyjątkiem tego, który kompiluje, i dołączonych plików nagłówkowych. Kompilator Java otworzy każdy plik, jeśli wskaże mu się jego lokalizację.

W Javie można całkowicie pominać mechanizm `import`, ale to wymagałoby podawania pełnych nazw klas, jak `java.util.Date`. W języku C++ dyrektywy `#include` nie da się uniknąć.

Jedyna korzyść, jaka płynie z używania instrukcji `import`, to wygoda. Można się odwołać do klasy za pomocą nazwy, która jest krótsza niż pełna nazwa pakietu. Na przykład po dodaniu instrukcji `import java.util.*` (albo `import.java.util.Date`) do klasy `java.util.Date` można odwoływać się, pisząc tylko `Date`.

Odpowiednikami pakietów w języku C++ są przestrzenie nazw. Instrukcje Javy `package` i `import` można traktować jako odpowiedniki dyrektyw `namespace` i `using` w C++.

4.7.2. Importy statyczne

W Java SE 5.0 wprowadzono możliwość importowania za pomocą instrukcji `import` metod i pól statycznych, nie tylko klas.

Jeśli na przykład na początku pliku źródłowego zostanie wstawiona poniższa instrukcja:

```
import static java.lang.System.*;
```

metod i pól statycznych klasy `System` będzie można używać bez przedrostka w postaci nazwy tej klasy:

```
out.println("Zegnaj, świecie!"); // tj. System.out
exit(); // tj. System.exit
```

Można też zaimportować konkretną metodę lub pole:

```
import static java.lang.System.out;
```

Wątpliwe jest, aby programiści chcieli skracać nazwy `System.out` i `System.exit`, ponieważ wtedy kod byłby mniej przejrzysty. Z drugiej strony kod:

```
sqrt(pow(x, 2) + pow(y, 2))
```

wydaje się bardziej przejrzysty niż:

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

4.7.3. Dodawanie klasy do pakietu

Aby umieścić klasę w pakiecie, należy na początku pliku źródłowego, **przed** kodem definiującym klasy w tym pakiecie, umieścić nazwę wybranego pakietu. Na przykład początek pliku `Employee.java` z listingu 4.7 wygląda następująco:

```
package com.horstmann.corejava;
public class Employee
{
    .
}
```

Jeśli na początku pliku nie ma instrukcji `package`, klasy znajdujące się w tym pliku należą do **pakietu domyślnego** (ang. *default package*). Pakiet domyślny nie ma nazwy. Wszystkie prezentowane do tej pory klasy należały do tego pakietu.

Pliki źródłowe należy umieszczać w podkatalogu odpowiadającym pełnej nazwie pakietu. Na przykład wszystkie pliki pakietu `com.horstmann.corejava` powinny się znaleźć w podkatalogu `com\horstmann\corejava` (w systemie Windows `com\horstmann\corejava`). Komplilator umieszcza pliki klas w takiej samej strukturze katalogów.

Program na listingach 4.6 i 4.7 jest rozłożony na dwa pakiety: klasa `PackageTest` należy do pakietu domyślnego, a klasa `Employee` do pakietu `com.horstmann.corejava`. W związku z tym plik `Employee.java` musi się znajdować w podkatalogu `com\horstmann\corejava`. Struktura katalogów jest następująca:

```
(katalog bazowy)
└── PackageTest.java
└── PackageTest.class
└── com/
    └── horstmann/
        └── corejava/
            └── Employee.java
            └── Employee.class
```

Aby skompilować ten program, należy przejść do katalogu bazowego i wykonać polecenie:

```
javac PackageTest.java
```

Kompilator automatycznie odnajdzie plik `com\horstmann\corejava\Employee.java` i skompiluje go.

Przeanalizujmy bardziej realistyczny przykład, w którym nie ma pakietu domyślnego, a klasy są rozłożone na kilka różnych pakietów (`com.horstmann.corejava` i `com.mycompany`).

```
(katalog bazowy)
└── com/
    └── horstmann/
        └── corejava/
            └── Employee.java
            └── Employee.class
    └── mycompany/
        └── PayrollApp.java
        └── PayrollApp.class
```

W tym przypadku komplikację i uruchomienie klas należy przeprowadzić w katalogu **bazowym**, czyli tym, który zawiera katalog `com`:

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Zauważ, że kompilator działa na **plikach** (z rozszerzeniem *.java*), podczas gdy interpreter Javy uruchamia **klasy**.

Listing 4.6. PackageTest/PackageTest.java

```
import com.horstmann.corejava.*;
// W powyższym pakiecie znajduje się definicja klasy Employee.

import static java.lang.System.*;

/**
 * Ten program demonstruje użycie pakietów.
 * @author cay
 * @version 1.11 2004-02-19
 * @author Cay Horstmann
 */
public class PackageTest
{
    public static void main(String[] args)
    {
        // Dzięki instrukcji import nie ma konieczności stosowania pełnej nazwy
        // com.horstmann.corejava.Employee.
        Employee harry = new Employee("Hubert Kowalski", 50000, 1989, 10, 1);

        harry.raiseSalary(5);

        // Dzięki instrukcji import static nie ma konieczności pisać System.out.
        out.println("name=" + harry.getName() + ",salary=" + harry.getSalary());
    }
}
```

Listing 4.7. com.horstmann.corejava/Employee.java

```
package com.horstmann.corejava;

// Klasy znajdujące się w tym pliku należą do powyższego pakietu.

import java.util.*;

// Instrukcje import następują po instrukcji package.

/**
 * @version 1.10 1999-12-18
 * @author Cay Horstmann
 */
public class Employee
{
    private String name;
    private double salary;
    private Date hireDay;

    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        // W klasie GregorianCalendar styczeń ma numer 0.
        hireDay = calendar.getTime();
    }
}
```

```

    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public Date getHireDay()
    {
        return hireDay;
    }

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

```



Od następnego rozdziału przykłady kodu będziemy umieszczać w pakietach. Dzięki temu będzie można utworzyć projekt w IDE dla każdego rozdziału zamiast dla każdego podrozdziału.



Kompilator nie sprawdza struktury katalogów w czasie kompilacji plików źródłowych. Jeśli mamy na przykład plik źródłowy, na początku którego znajduje się poniższa dyrektywa:

```
package com.mycompany;
```

możemy go skompilować nawet poza podkatalogiem *com/mycompany*. Plik ten zostanie skompilowany bezbłędnie, **jeśli nie jest uzależniony od żadnych innych pakietów**. Tak powstałego programu nie da się jednak uruchomić. **Maszyna wirtualna** nie odnajdzie powstalych klas, kiedy spróbujemy uruchomić ten program.

4.7.4. Zasięg pakietów

Wiemy już, do czego służą modyfikatory dostępu *public* i *private*. Obiekty oznaczone pierwszym z tych dwóch modyfikatorów są dostępne we wszystkich klasach, a drugim — tylko w klasie, w której są zdefiniowane. Jeśli nie ma żadnego modyfikatora dostępu, obiekt (tzn. klasa, metoda lub zmienna) jest dostępny dla wszystkich metod w **pakiecie**.

Przypomnijmy sobie program z listingu 4.2. Klasa *Employee* nie jest tam zdefiniowana jako publiczna. W związku z tym dostęp do niej mają tylko inne klasy (np. *EmployeeTest*) znajdujące się w tym samym pakiecie (tu domyślnym). W przypadku klas to domyślne działanie jest korzystne, ale jeśli chodzi o zmienne, to wybór ten nie był trafny. Zmienne muszą być jawnie

oznaczone jako `private`, gdyż w przeciwnym przypadku będą widoczne w całym pakiecie. To oczywiście oznacza łamanie zasad hermetyzacji. Niestety bardzo łatwo można zapomnieć o wpisaniu słowa kluczowego `private`. Poniższy przykład pochodzi z klasy `Window` dostępnej w pakiecie `java.awt` wchodzący w skład JDK:

```
public class Window extends Container
{
    String warningString;
    ...
}
```

Należy zauważać, że zmienna `warningString` nie jest prywatna! Oznacza to, że metody wszystkich klas z pakietu `java.awt` mają do niej dostęp i mogą modyfikować jej wartość (np. ustawić na łańcuch `Zaufaj mi!`). Ze zmiennej tej korzystają jednak tylko metody należące do klasy `Window`, w związku z czym najlepszym rozwiązaniem byłoby oznaczyć ją słowem kluczowym `private`. Prawdopodobnie programista pisał kod w pośpiechu i najzwyczajniej zapomniał o modyfikatorze dostępu (nie podajemy nazwiska tego programisty — każdy może sobie sam zająrzyć do omawianego pliku źródłowego).



Zadziwiające jest to, że nigdy nie naprawiono tego błędu, mimo iż pisaliśmy o nim we wszystkich dziewięciu wydaniach tej książki. Najwidoczniej osoby zajmujące się implementacją Javy nie czytają naszych publikacji. Co więcej, w klasie przybyło z czasem sporo nowych pól, ale tylko około połowa z nich ma modyfikator dostępu `private`.

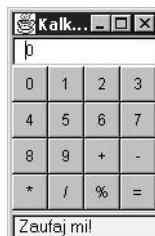
Czy jest to jakiś problem? To zależy, ponieważ pakiety nie są zamkniętymi jednostkami. Oznacza to, że każdy może dodać do pakietu swoje klasy. Oczywiście złośliwi lub niedouczni programiści mogą napisać kod, który będzie modyfikował zmienne o zasięgu pakietowym. Na przykład w pierwszych wersjach Javy można było z łatwością przemycić dodatkową klasę do pakietu `java.awt`. Wystarczyło na początku jej kodu napisać:

```
package java.awt;
```

Następnie plik z taką klasą trzeba było umieścić w podkatalogu `java.awt` na ścieżce klas i już dostęp do wewnętrzności pakietu `java.awt` stawał otworem. Ta sztuczka umożliwiała ustawienie łańcucha z ostrzeżeniem (zobacz rysunek 4.9).

Rysunek 4.9.

Zmiana łańcucha ostrzeżenia w oknie apletu



W JDK 1.2 wprowadzono zmiany w mechanizmie ładowającym klasy (ang. *class loader*), aby jawnie zabraniał ładowania klas użytkownika, których pakiety mają nazwy zaczynające się od słowa `java`. Oczywiście klasy niestandardowe z tej ochrony nie skorzystają. W zamian udostępniono **mechanizm pieczętowania pakietów** (ang. *package sealing*), którego zadaniem

jest rozwiązywanie problemów z różnymi sposobami dostępu do pakietów. Jeśli pakiet jest zapieczętowany, nie można do niego dodać żadnych nowych klas. W rozdziale 10. opisujemy techniki tworzenia plików JAR zawierających pakiety zapieczętowane.

4.8. Ścieżka klas

Jak wiadomo, klasy są przechowywane w podkatalogach systemu plików. Ścieżka do klasy musi odpowiadać nazwie jej pakietu.

Pliki klas można także przechowywać w plikach JAR (ang. *Java archive*). Plik JAR zawiera skompresowane pliki klas i katalogi oraz pozwala zaoszczędzić miejsce i polepszyć wydajność. Większość niestandardowych bibliotek używanych w programach ma postać co najmniej jednego pliku JAR. W JDK także jest dostępna pewna liczba takich plików, np. *jre/lib/rt.jar*, który zawiera tysiące klas bibliotecznych. Techniki tworzenia plików JAR zostały opisane w rozdziale 10.



Struktura plików JAR jest zgodna z formatem ZIP. W związku z tym do pliku *rt.jar* i każdego innego o takim rozszerzeniu można zajrzeć przy użyciu dowolnego narzędzia ZIP.

Aby móc używać określonych klas w różnych programach, należy wykonać następujące czynności:

1. Umieść pliki klas w wybranym katalogu, na przykład */home/user/classdir*. Pamiętaj, że jest to katalog **bazowy** drzewa pakietu. Jeśli dodasz klasę *com.horstmann.corejava.Employee*, plik klasy *Employee.class* musi się znaleźć w podkatalogu */home/user/classdir/com/horstmann/corejava*.
2. Umieść wszystkie pliki JAR w wybranym katalogu, na przykład */home/user/archives*.
3. Ustaw **ścieżkę klas**. Ścieżka klas to zbiór lokalizacji, które mogą zawierać pliki klas.

W systemie UNIX poszczególne elementy ścieżki klas są rozdzielane dwukropkiem:

/home/user/classdir::/home/user/archives/archive.jar

W systemie Windows separatorem jest średnik:

c:\classdir;.;c:\archives\archive.jar

W obu przypadkach kropka oznacza bieżący katalog.

Niniejsza ścieżka zawiera następujące elementy:

- katalog bazowy *home/user/classdir* lub *c:\classdir*;
- bieżący katalog (.)
- plik JAR */home/user/archives/archive.jar* lub *c:\archives\archive.jar*.

Od Java SE 6 do określenia katalogu z plikami JAR można użyć symbolu wieloznacznego:

`/home/user/classdir:./home/user/archives/*`

lub

`c:\classdir;.;c:\archives*`

W systemie UNIX trzeba zastosować symbol zastępczy dla *, aby uniknąć rozwinięcia powłoki.

Wszystkie pliki JAR (ale nie `.class`) znajdujące się w katalogu `archives` są dodawane do ścieżki klas.

Pliki biblioteczne wykonawcze (plik `rt.jar` i inne pliki JAR, które znajdują się w katalogach `jre/lib` i `jre/lib/ext`) są zawsze przeszukiwane w celu znalezienia klas. Nie dodaje się ich bezpośrednio do ścieżki klas.



Kompilator `javac` zawsze szuka plików w bieżącym katalogu, natomiast program uruchamiający Java Virtual Machine przeszukuje bieżący katalog tylko wtedy, kiedy w ścieżce klas znajduje się katalog .. Problemu nie ma, jeśli ścieżka klas nie została ustawiona, ponieważ wtedy zawiera tylko katalog .. Jeśli ścieżka klas zostanie ustawiona bez katalogu .., programy będą komplikowały się bez problemów, ale nie będą chciały działać.

Ścieżka klas zawiera listę wszystkich katalogów i plików JAR, od których należy **zacząć** szukanie klas. Przeanalizujmy naszą przykładową ścieżkę klas:

`/home/user/classdir:./home/user/archives/archive.jar`

Przypuśćmy, że maszyna wirtualna szuka pliku klasy `com.horstmann.corejava.Employee`. Szukanie zaczyna od systemowych plików klas, które znajdują się w archiwach w katalogach `jre/lib` i `jre/lib/ext`. Tam wspomnianej klasy nie ma, więc kontynuuje szukanie na ścieżce klas. Poszukiwane są następujące pliki:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`,
- `com/horstmann/corejava/Employee.class`, zaczynając od bieżącego katalogu,
- `com/horstmann/corejava/Employee.class` wewnętrz pliku `/home/user/archives/archive.jar`.

Kompilator ma trudniejsze zadanie dotyczące wyszukiwania plików niż maszyna wirtualna. Jeśli odwołamy się do klasy, nie podając jej pakietu, kompilator musi najpierw znaleźć pakiet, który tę klasę zawiera. W tym celu sprawdza wszystkie dyrektywy `import`, które są potencjalnym źródłem klas. Wyobraźmy sobie na przykład, że plik źródłowy zawiera poniższe dyrektywy:

```
import java.util.*;
import com.horstmann.corejava.*;
```

a w kodzie źródłowym użyto klasy `Employee`. Najpierw kompilator próbuje kolejno znaleźć klasy `java.lang.Employee` (ponieważ pakiet `java.lang` jest zawsze importowany domyślnie), `java.util.Employee`, `com.horstmann.Employee` i `Employee` w bieżącym pakiecie. Szuka

wszystkich tych klas we wszystkich lokalizacjach wymienionych na ścieżce klas. Jeśli kompilator znajdzie więcej niż jedną z tych klas, zgłasza błąd komilacji (ponieważ klasy muszą być unikatowe, kolejność instrukcji import nie ma znaczenia).

Kompilator idzie nawet o krok dalej i sprawdza, czy **plik źródłowy** klasy nie jest nowszy niż skompilowany plik tej klasy. Jeśli plik źródłowy jest nowszy, jest on automatycznie kompliowany jeszcze raz. Przypomnijmy, że z innych pakietów można importować tylko klasy publiczne. Jeden plik źródłowy może zawierać tylko jedną klasę publiczną, a nazwa tego pliku i nazwa zawartej w nim klasy publicznej muszą się ze sobą zgadzać. Dzięki temu kompilator nie ma problemów z lokalizacją plików źródłowych klas publicznych. Klasy niepubliczne można importować z bieżącego pakietu. Ich definicje mogą się znajdować w plikach o innych nazwach. Jeśli zostanie zaimportowana klasa z bieżącego pakietu, kompilator przeszuka **wszystkie** pliki źródłowe znajdujące się w tym pakiecie w celu znalezienia definicji tej klasy.

4.8.1. Ustawianie ścieżki klas

Ścieżkę klas najlepiej ustawiać za pomocą opcji **-classpath** (lub **-cp**):

```
java -classpath /home/user/classdir:./home/user/archives/archive.jar MyProg.java
```

lub

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

Całe polecenie musi się znajdować w jednym wierszu. Długie polecenia tego typu najlepiej wstawać do skryptów powłoki lub plików wsadowych.

Ustawianie ścieżki za pomocą opcji **-classpath** jest metodą preferowaną, ale nie jedyną. Inny sposób polega na ustawieniu zmiennej środowiskowej **CLASSPATH**. Dokładna procedura zależy od konkretnej powłoki. W powłoce Bourne Again (bash) należy użyć następującego polecenia:

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

W powłoce C:

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

W systemie Windows:

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

Ścieżka klas jest dostępna do wyjścia z powłoki.



Są osoby, które zalecają ustawienie zmiennej środowiskowej **CLASSPATH** na stałe. Ogólnie nie jest to dobry pomysł. Ludzie często zapominają o ustawieniach globalnych, a potem dziwią się, że mają problemy z ładowaniem klas. Szczególnie naganym przykładem jest w tym przypadku instalator dla systemu Windows programu QuickTime firmy Apple. Ustawia on globalnie zmienną **CLASSPATH** na plik JAR, którego używa, ale nie dodaje bieżącego katalogu do ścieżki klas. Z tego powodu mnóstwo programistów straciło trochę nerwów, kiedy ich programy przechodziły komplikację, ale nie można było ich uruchomić.



Niektórzy zalecają całkowite pominięcie ścieżki klas poprzez umieszczenie wszystkich plików JAR w katalogu `jar/lib/ext`. Jest to bardzo zła rada, i to z dwóch powodów. Archiwa ręcznie ładowające inne klasy nie działają poprawnie, kiedy znajdują się w katalogu rozszerzeń (więcej informacji na temat programów ładowających klasy znajduje się w rozdziale 9. drugiego tomu). Ponadto programiści często zapominają o plikach, które umieścili tam kilka miesięcy wcześniej. Potem zachodzą w głowę, czemu loader klas ignoruje ich wspaniałe klasy, podczas gdy ten po prostu ładuje dawno zapomniane klasy z katalogu rozszerzeń.

4.9. Komentarze dokumentacyjne

JDK zawiera bardzo przydatne narzędzie o nazwie `javadoc`, które generuje dokumentację w formie plików HTML z plików źródłowych. Dokumentacja API opisana w rozdziale 3. powstała w wyniku uruchomienia narzędzia `javadoc` na kodzie źródłowym standardowej biblioteki Javy.

Profesjonalną dokumentację może stworzyć każdy, kto doda do kodu źródłowego komentarze zaczynające się od specjalnej sekwencji znaków `/**`. Jest to bardzo wygodne rozwiązanie, ponieważ umożliwia przechowywanie kodu i dokumentacji do niego w jednym miejscu. Jeśli kod i dokumentacja znajdują się w osobnych plikach, z czasem mogą się pojawić między nimi rozbieżności. Jednak dzięki temu, że komentarze dokumentacyjne znajdują się w tym samym pliku co kod źródłowy, aktualizacja obu jest znacznie ułatwiona.

4.9.1. Wstawianie komentarzy

Narzędzie `javadoc` pobiera informacje dotyczące następujących elementów:

- pakietów,
- klas i interfejsów publicznych,
- publicznych i chronionych (`protected`) metod i konstruktorów,
- pól publicznych i chronionych.

Znaczenie słowa kluczowego `protected` zostało opisane w rozdziale 5., a interfejsy w rozdziale 6.

Każda z wymienionych konstrukcji może (i powinna) być opatrzona komentarzem. Komentarz powinien się znajdować **bezpośrednio nad tym**, czego dotyczy. Początek komentarza określa sekwencja znaków `/**`, a koniec `*/`.

W komentarzu można umieścić **dowolny** tekst oraz specjalne **znaczniki dokumentacyjne**. Znaczniki dokumentacyjne rozpoczynają się od znaku `@`, np. `@author` czy `@param`.

Pierwsze zdanie komentarza powinno być **streszczeniem**. Narzędzie `javadoc` automatycznie generuje strony streszczeń zawierające te zdania.

W tekście komentarza można używać znaczników HTML, takich jak `...` (sługiacy do emfazy), `<code>...</code>` (sługiacy do oznaczania fragmentów kodu), `...` (dajacy silne wyróżnienie) czy nawet `<img... />` (do wstawiania obrazów). Powinno się jednak unikać nagłówków `<h1>` i poziomych kresek `<hr>`, poniewaz mogą wchodzić w interakcje z formatowaniem dokumentu.



Jeśli w komentarzach znajdują się odnośniki do innych plików, takich jak obrazy (moga to byc wykresy lub rysunki przedstawiające elementy interfejsu użytkownika), należy pliki te umieścić w podkatalogu folderu zawierajcego plik źródłowy o nazwie `doc-files`. Narzędzie `javadoc` kopiuje katalogi `doc-files` i ich zawartość z katalogów źródłowych do katalogów dokumentacji. Nazwa katalogu `doc-files` musi się znaleźć w ścieżce odnośnika, np. ``.

4.9.2. Komentarze do klas

Komentarz klasy musi się znajdować **za** instrukcjami `import`, bezpośrednio przed definicją klasy.

Przykład komentarza do klasy:

```
/**  
 * Obiekt <code>Card</code> reprezentuje kartę do gry, np.  
 * „dama kier”. Karta ma kolor (karo, kier, trefl lub pik)  
 * i wartość (1 = as, 2 . . . 10, 11 = walet,  
 * 12 = dama, 13 = król)  
 */  
public class Card  
{  
    . . .  
}
```



Znak `*` nie musi się znajdować na początku każdej linijki. Na przykład poniższy komentarz jest tak samo poprawny:

```
/**  
 Obiekt <code>Card</code> reprezentuje kartę do gry, np.  
 „dama kier”. Karta ma kolor (karo, kier, trefl lub pik)  
 i wartość (1 = as, 2 . . . 10, 11 = walet,  
 12 = dama, 13 = król)  
 */
```

Jednak większość IDE automatycznie dodaje gwiazdki i zmienia ich ustawienie w odpowiedzi na zmiany w łamaniu wierszy.

4.9.3. Komentarze do metod

Komentarz do metody musi się znajdować bezpośrednio przed metodą, której dotyczy. Poza znacznikami ogólnego przeznaczenia można stosować dodatkowe znaczniki:

■ @param opis zmiennej

Dodaje pozycję do sekcji *Parameters* metody. Opis może zajmować kilka wierszy i zawierać znaczniki HTML. Wszystkie znaczniki `@param` dotyczące jednej metody powinny się znajdować w jednym miejscu.

■ @return opis

Dodaje sekcję *Returns*. Opis może zajmować kilka wierszy i zawierać znaczniki HTML.

■ @throws opis klasy

Dodaje informację, że dana metoda może spowodować wyjątek. Wyjątki są tematem rozdziału 11.

Przykład komentarza do metody:

```
/**  
 * Podnosi pensję pracownika.  
 * @param byPercent wartość określająca, o ile procent podnieść pensję (np. 10 = 10%).  
 * @return kwota podwyżki  
 */  
public double raiseSalary(double byPercent)  
{  
    double raise = salary * byPercent / 100;  
    salary += raise;  
    return raise;  
}
```

4.9.4. Komentarze do pól

Komentarze są potrzebne tylko do pól publicznych, co na ogół oznacza zmienne statyczne. Na przykład:

```
/**  
 * Kolor „karo”.  
 */  
public static final int HEARTS = 1;
```

4.9.5. Komentarze ogólne

W komentarzach dokumentacji klas można używać poniższych znaczników:

■ @author imię i nazwisko

Dodaje pozycję *Author*. Jeśli jest kilku autorów, można zastosować kilka znaczników `@author`.

■ @version tekst

Dodaje pozycję *Version*. Tekst może być opisem aktualnej wersji.

Następujących znaczników można używać we wszystkich komentarzach dokumentacyjnych:

■ **@since tekst**

Dodaje pozycję *Since*. Tekst to opis wersji, w której wprowadzono daną funkcję. Na przykład @since version 1.7.1.

■ **@deprecated tekst**

Dodaje komentarz informujący, że dana klasa, metoda lub zmienna nie powinny być używane. Tekst powinien zawierać informację o zamienniku. Na przykład:

@deprecated W zamian należy używać <code>setVisible(true)</code>

Do innych części dokumentacji lub dokumentów zewnętrznych można się odwoływać za pomocą hiperłączy. Do tego celu służą znaczniki @see i @link.

■ **@see odwołanie**

Dodaje hiperłącze w sekcji *See also*. Może być używany do klas i metod. **Odwołanie** może mieć jedną z poniższych form:

```
pakiet.kslas#struktura etykieta
<a href="...">etykieta</a>
"tekst"
```

Najbardziej użyteczna jest pierwsza wersja. Narzędzie *javadoc* automatycznie wstawia odnośnik do dokumentacji z podanej nazwy klasy, metody lub zmiennej. Na przykład:

@see com.horstmann.corejava.Employee#raiseSalary(double)

Powyższy znacznik tworzy odnośnik do metody `raiseSalary(double)` w klasie `com.horstmann.corejava.Employee`. Można pominąć nazwę pakietu lub nazwę pakietu i klasy. W takiej sytuacji struktura będzie zlokalizowana w bieżącym pakiecie lub klasie.

Należy zauważyć, że znakiem rozdzielającym klasę i nazwę metody lub zmiennej jest znak `#`, a nie kropka. Kompilator Javy jest bardzo inteligentny i bez problemu rozpoznaje różne zastosowania kropki jako separatora pakietów, podpakietów, klas, klas wewnętrznych, metod i zmiennych. Niestety narzędzie *javadoc* nie jest tak inteligentne i trzeba mu pomóc.

Jeśli po znaczniku @see znajduje się znak `<`, oznacza to, że trzeba podać hiperłącze. Może ono prowadzić pod dowolny adres URL. Na przykład:

@see Strona internetowa książki

W każdym z tych przypadków można określić opcjonalną **etykietę**, która będzie kotwicą odnośnika. W przypadku pominięcia etykiety kotwicą jest nazwa kodu docelowego lub adres URL.

Jeśli po znaczniku @see znajduje się znak `"`, tekst zostanie wyświetlony w sekcji *See also*. Na przykład:

@see "Core Java 2. Tom 2"

Znaczników @see można wstawić kilka, ale wszystkie muszą być w jednym miejscu.

- Odnośniki do klas lub metod można wstawić w dowolnych miejscach we wszystkich komentarzach dokumentacyjnych. Służy do tego znacznik w specjalnej formie:

```
{@link pakiet.klasa#struktura etykieta}
```

Wszystkie zasady dotyczące znacznika @see mają zastosowanie także do tego znacznika.

4.9.6. Komentarze do pakietów i ogólne

Komentarze do klas, metod i zmiennych znajdują się bezpośrednio w plikach źródłowych Javy pomiędzy znakami `/**` i `*/`. Natomiast generowanie komentarzy do **pakietów** wymaga dodania osobnego pliku do każdego katalogu pakietu. Są dwie możliwości:

- 1 Utworzenie pliku HTML o nazwie `package.html`. Zostanie pobrane wszystko, co znajduje się pomiędzy znacznikami `<body>` i `</body>`.
- 2 Utworzenie pliku Java o nazwie `package-info.java`. Na początku tego pliku muszą się znajdować komentarz `/** */` i instrukcja `package`. Nie powinno w nim być żadnych dodatkowych komentarzy ani kodu.

Istnieje także możliwość utworzenia ogólnego komentarza do wszystkich plików źródłowych. Powinien się znajdować w pliku o nazwie `overview.html`, zlokalizowanym w katalogu macierzystym wszystkich plików źródłowych. Zostanie pobrane wszystko, co znajduje się pomiędzy znacznikami `<body>` i `</body>`. Komentarz ten wyświetla się, gdy użytkownik kliknie opcję `Overview` na pasku nawigacyjnym.

4.9.7. Generowanie dokumentacji

Poniższe punkty opisują procedurę generowania dokumentacji, która w tym przypadku zostanie umieszczona w katalogu o nazwie `docDirectory`.

- 1 Przejdz do katalogu z plikami źródłowymi, których dokumentację chcesz wygenerować. Jeśli program zawiera zagnieżdżone pakiety, jak `com.horstmann.corejava`, należy otworzyć katalog zawierający podkatalog `com` (w tym samym katalogu powinien się znajdować plik `overview.html`, jeśli został utworzony).
- 2 Aby wygenerować dokumentację jednego pakietu, należy wydać następujące polecenie:

```
javadoc -d docDirectory nazwaPakietu
```

W przypadku kilku pakietów polecenie wygląda tak:

```
javadoc -d docDirectory NazwaPakietu1 nazwaPakietu2...
```

Jeśli pliki znajdują się w pakiecie domyślnym, powyższe polecenie ma następującą formę:

```
javadoc -d docDirectory *.java
```

W przypadku braku opcji `-d docDirectory` pliki HTML zostaną umieszczone w bieżącym katalogu. Nie zalecamy jednak takiego rozwiązania, gdyż powoduje ono niemały bałagan.

Działaniem programu *javadoc* można sterować za pomocą rozmaitych opcji wiersza polecen. Na przykład opcje `-author` i `-version` dodają do dokumentacji to, co oznaczono znacznikami `@author` i `@version` (przy domyślnych ustawieniach znaczniki te są pomijane). Inna przydatna opcja to `-link`, która dodaje łącza do klas standardowych. Na przykład poniższe polecenie:

```
javadoc -link http://docs.oracle.com/javase/7/docs/api *.java
```

utworzy odnośniki do wszystkich standardowych klas bibliotecznych w dokumentacji na stronie internetowej firmy Oracle.

Opcja `-linksource` konwertuje wszystkie pliki źródłowe na pliki HTML (brak kolorowania składni, ale są numery wierszy). Nazwa każdej klasy i metody jest zamieniana na odnośnik do źródła.

Informacje na temat pozostałych opcji można znaleźć w internetowej dokumentacji narzędzia *javadoc* pod adresem <http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc>.



Więcej możliwości konfiguracyjnych dają tak zwane doclety (ang. *doclets*). Można napisać doclet umożliwiający generowanie dokumentacji w dowolnym formacie innym niż HTML. Ponieważ niewiele osób potrzebuje takiej możliwości, po szczegółowe informacje na temat docletów odsyłamy do dokumentacji internetowej, która znajduje się pod adresem <http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/doclet/overview.html>.

4.10. Porady dotyczące projektowania klas

Na zakończenie tego rozdziału przedstawiamy kilka wskazówek, których stosowanie pomaga w tworzeniu lepszych klas z punktu widzenia dobrego stylu programowania obiektowego. Niniejsza lista nie jest bynajmniej ostatecznym źródłem wiedzy na ten temat.

1. Dane powinny być prywatne.

To jest najważniejsza ze wszystkich zasad — niestosowanie jej powoduje naruszenie zasad hermetyzacji. Niewykluczone, że z tego powodu będzie konieczne napisanie kilku mutatorów lub metod dostępowych, ale i tak lepiej, aby pola danych pozostały prywatne. Przekonaliśmy się na własnej skórze, że sposób reprezentacji danych może się zmienić, ale sposób ich używania ulega zmianom znacznie rzadziej. Jeśli dane są prywatne, zmiany w ich reprezentacji nie mają wpływu na użytkowników klas, a błędy są łatwiejsze do wykrycia.

2. Dane powinny być zawsze zainicjowane.

Java nie inicjuje zmiennych lokalnych, ale zmienne składowe obiektów. Nie należy pozwalać na inicjację zmiennych wartościami domyślnymi, tylko inicjować je jawnie, podając wartość domyślną lub wartości domyślne we wszystkich konstruktorach.

3. Nie należy stosować zbyt wielu różnych podstawowych typów danych w jednej klasie.

Jeśli klasa zawiera kilka **powiązanych** ze sobą zmiennych tego samego typu, należy je zastąpić nową klasą. Dzięki temu kod klas jest bardziej zrozumiały i łatwiejszy w modyfikacji. Na przykład poniższe pola klasy Customer można zastąpić nową klasą o nazwie Address:

```
private String street;  
private String city;  
private String state;  
private int zip;
```

Dzięki temu znacznie prościej jest wprowadzać zmiany w adresach, jak na przykład w przypadku konieczności dodania obsługi adresów międzynarodowych.

4. Nie wszystkie pola wymagają własnych metod dostępu i zmiany.

Po utworzeniu obiektu zmiany może wymagać na przykład wysokość pensji pracownika, ale z pewnością nie data zatrudnienia. Ponadto obiekty często zawierają składowe, do których nikt spoza klasy nie powinien mieć dostępu. Może to być na przykład tablica skrótów nazw województw w klasie Address.

5. Klasa o zbyt dużej funkcjonalności powinny być dzielone.

Oczywiście ta wskazówka nie jest precyzyjna, ponieważ każdy ma inne zdanie na temat tego, ile to jest za dużo funkcji. Jeśli jednak istnieje możliwość podzielenia złożonej klasy na dwie prostsze, należy z tej możliwości skorzystać (z drugiej strony nie należy przesadzać — klasy zawierające po jednej metodzie to odchylenie w drugą stronę).

Poniższa klasa jest przykładem złego stylu projektowania:

```
public class CardDeck //zły styl  
{  
    private int[] value;  
    private int[] suit;  
  
    public CardDeck() { . . . }  
    public void shuffle() { . . . }  
    public int getTopValue() { . . . }  
    public int getTopSuit() { . . . }  
    public void draw() { . . . }  
}
```

Klasa ta implementuje dwie odrębne koncepcje: talię kart (CardDeck) i związane z nią metody shuffle (tasuj) i draw (pobierz) oraz metody sprawdzające wartość i kolor karty. Należałoby utworzyć oddzielną klasę o nazwie Card reprezentującą kartę. W ten sposób powinny powstać dwie klasy, z których każda ma własny zakres działań:

```
public class CardDeck  
{  
    private Card[] cards;  
  
    public CardDeck() { . . . }  
    public void shuffle() { . . . }
```

```
    public Card getTop() { . . . }
    public void draw() { . . . }
}

public class Card
{
    private int value;
    private int suit;

    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }
}
```

6. Nazwy metod i klas powinny odpowiadać ich przeznaczeniu.

Podobnie jak zmiennym, klasom należy nadawać nazwy odzwierciedlające ich przeznaczenie (w bibliotece standardowej jest kilka klas, których nazwy budzą wątpliwości, np. klasa Date, która opisuje godzine).

Zgodnie z konwencją nazwa klasy powinna być rzeczownikiem (np. Zamówienie) lub składać się z przymiotnika i rzeczownika (np. SzybkieZamówienie). Nazwy akcesorów powinny się zaczynać od pisanego małymi literami słowa get (np. getSalary), a mutatorów od słowa set (np. setSalary).

W tym rozdziale opisaliśmy podstawowe informacje dotyczące obiektów i klas, dzięki którym Java jest językiem obiektowym. Aby jednak język był w pełni obiektowy, musi obsługiwać dziedziczenie i polimorfizm. Tym własnościom Javy został poświęcony następny rozdział.

5

Dziedziczenie

W tym rozdziale:

- Klasy, nadklasy i podklasy
- Klasa bazowa `Object`
- Klasa `ArrayList`
- Obiekty osłonowe i automatyczne opakowywanie typów
- Metody ze zmienną liczbą parametrów
- Klasy wyliczeniowe
- Refleksja
- 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 pobiędźnie 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, abytrzymać 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.

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 **nadklassą** (ang. *superclass*), **klassą bazową** (ang. *base class*) lub **klassą macierzystą** (ang. *parent class*). Nowo utworzona klasa nazywa się **podklassą** (ang. *subclass*), **klassą pochodną** (ang. *derived class*) lub **klassą potomną** (ang. *child class*). Większość programistów Javy używa terminów „nadklaśa” i „podklaśa”, ale niektórym bardziej odpowiada analogia klasy macierzystej i potomnej, która bardzo dobrze pasuje do koncepcji dziedziczenia.

Klasa `Employee` jest nadklassą, ale nie ze względu na to, że jest wyższa rangą albo ma większą funkcjonalność. **W rzeczywistości jest odwrotnie:** podklaśa ma **większą** funkcjonalność niż nadklaśa. Będzie można się o tym przekonać, kiedy przejdziemy do analizy klasy `Manager`, która zawiera więcej danych i ma większą funkcjonalność niż jej nadklaśa `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** zbiuru kierowników. Albo w drugą stronę — zbiór kierowników jest **podzbiorem** zbiuru pracowników.

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

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

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

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 getHireDay, mimo że nie zostały one zdefiniowane bezpośrednio w tej klasie. Są dostępne, ponieważ zostały odziedziczone po klasie Employee.

Podobnie zostały odziedziczone pola name, salary i hireDay. Każdy obiekt klasy Manager ma cztery pola: name, salary, hireDay i bonus.

W definicji klasy rozszerzającej inną klasę konieczne jest podanie tylko **różnic** pomiędzy tymi klasami. Metody ogólnego przeznaczenia należy umieszczać w nadkласie, a metody bardziej wyspecjalizowane — w podkласie. Wydzielanie wspólnego kodu i umieszczenie go w nadklasie jest często stosowaną techniką programowania obiektowego.

Niektóre metody obecne w klasie nadzędnej 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 nadklasą:

```
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. Metoda `getSalary` klasy `Manager` **nie ma bezpośredniego dostępu do prywatnych pól nadklasy** `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`. Tylko metody klasy `Employee` mają dostęp do tych prywatnych pól. 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 znajdująemy. 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 Java 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`.

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(n, s, 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 zainicjować 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 konstruktor podklasy nie wywołuje jawnie konstruktora nadklasy, wywoływany jest konstruktor domyślny (bezparametryowy) nadklasy. Jeśli nadklasa nie ma konstruktora domyślnego, a konstruktor podklasy nie wywołuje jawnie żadnego innego konstruktora nadklasy, kompilator zgłosi błąd.



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 inicjującej 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 ustawiśmy 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 Kwiatak", 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 Kwiatak 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 wywoływaną 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 Java nie ma potrzeby deklarowania metody jako wirtualnej. Wiązanie dynamiczne jest działaniem domyślnym. 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.
        Manager boss = new Manager("Karol Parol", 80000, 1987, 12, 15);
        boss.setBonus(5000);

        Employee[] 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.util.Date;
import java.util.GregorianCalendar;

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

    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public Date getHireDay()
    {
        return hireDay;
    }
}

```

```
    }

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

Listing 5.3. inheritance/Manager.java

```
package inheritance;

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;
    }
}
```

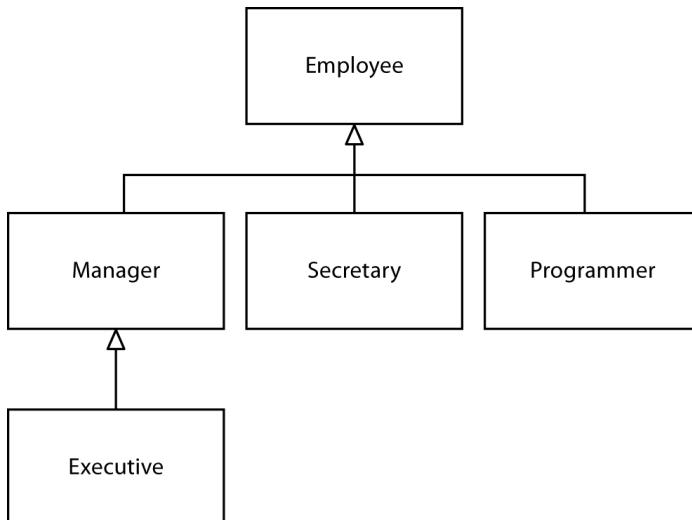
5.1.1. Hierarchia dziedziczenia

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*).

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.

Rysunek 5.1.

Hierarchia dziedziczenia klasy Employee



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

5.1.2. 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 podklassą 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 Java 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 komplikacji tego przypisania, ale zmienne `staff[0]` i `manager[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`.

5.1.3. Wiązanie dynamiczne

Ważne jest, aby zrozumieć, co się dzieje w momencie wywołania metody na rzecz obiektu. Wygląda to tak:

- 1 Kompilator sprawdza zadeklarowany typ obiektu i nazwę metody. Powiedzmy, że wywołujemy `x.f(param)`, a niejawnym parametrem `x` jest zadeklarowany jako obiekt klasy `C`. Pamiętajmy, że metoda 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 metod publicznych 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 **rozstrzyganiem 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.



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 nadzędnej, 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
```

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

- 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.

- 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. 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)*, kompilator przeszukuje tabelę metod nadklasy parametru niejawnego.

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 rozstrzyganiem 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 nadklaś *Object*, po której dziedziczy kilka metod. Na razie ignorujemy te dodatkowe metody.

Tabela metod klasy *Manager* wygląda nieco inaczej. Trzy metody są odziedziczone, jedna przedefiniowana, 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.

5.1.4. 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*. Założymy na przykład, że nie chcemy, aby klasa Executive była rozszerzana. W tym celu należy w jej definicji użyć modyfikatora *final*:

```
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:

```
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 konwersję pomiędzy klasą *Date* a stanem kalendarza. Żadna podklasa nie powinna mieć możliwości mieszanego się w to. Klasa *String* też jest finalna. Oznacza to, że nie można utworzyć jej podklasy. 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 wywoływanie poprzez proces polegający na wstawieniu jej kodu w **miejsce wywołania** (ang. *inlining*). Na przykład wywoływanie metody `e.getName()` zostało 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 zawsze 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.

5.1.5. 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. 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 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[1]` 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 umożliwia 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 niedorzecze 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 podklassą 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 rozwiążaniem 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];
    . . .
}
```

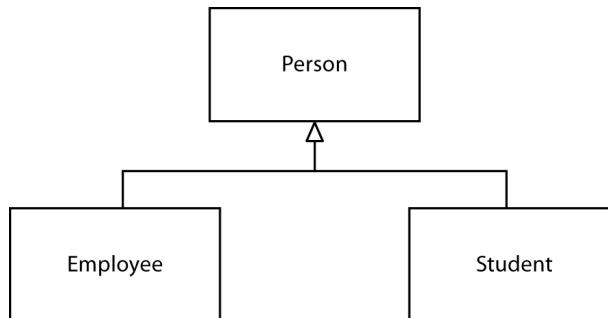
5.1.6. Klasa abstrakcyjne

Im bliżej wierzchołka hierarchii dziedziczenia, tym klasy są bardziej ogólne i często bardziej abstrakcyjne. W pewnym momencie klasa nadzędna staje się tak abstrakcyjna, że zaczyna być traktowana nie jako klasa do tworzenia obiektów o określonym przeznaczeniu, a jako podstawa do tworzenia innych klas. Przeanalizujmy 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.

Po co w ogóle zaprzątać 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.

Rysunek 5.2.

Diagram dziedziczenia klas Person i jej podklas



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.

```
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.

```
abstract class Person
{
    private String name;

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

Warto zauważyć, że można tworzyć **zmienne obiektowe** klas abstrakcyjnych, ale muszą się one odwoływać do obiektów nieabstrakcyjnych podklas. Na przykład:

```
Person p = new Student("Wincenty Witos", "Ekonomia");
```

W tym przypadku `p` jest zmienną abstrakcyjnego typu `Person` odwołującą 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`:

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

    public Student(String n, String m)
    {
        super(n);
        major = m;
    }
    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 5.4, 5.5, 5.6 i 5.7 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:

```

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

```

Listing 5.4. abstractClasses/PersonTest.java

```

import java.util.*;

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

        // Wstawienie do tablicy people obiektów Student i Employee.
        people[0] = new Employee("Henryk Kwiątek", 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 5.5. abstractClasses/Person.java

```

package abstractClasses;

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

    public Person(String n)
    {
        name = n;
    }

    public String getName()
    {

```

```
        return name;  
    }  
}
```

Listing 5.6. abstractClasses/Employee.java

```
package abstractClasses;  
  
import java.util.Date;  
import java.util.GregorianCalendar;  
  
public class Employee extends Person  
{  
    private double salary;  
    private Date hireDay;  
  
    public Employee(String n, double s, int year, int month, int day)  
    {  
        super(n);  
        salary = s;  
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);  
        hireDay = calendar.getTime();  
    }  
  
    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 5.7. 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 n, String m)
{
    // Przekazanie n do konstruktora nadklasy.
    super(n);
    major = m;
}

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, nie możliwe 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.

5.1.7. Ochrona 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 — podklaś 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.

Jednak metody klasy Manager mają dostęp tylko do pola hireDay obiektów Manager, a nie innych obiektów 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. Założymy, 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. Klasa bazowa `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:

```
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 14.).

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

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

Oczywiście zmienna typu `Object` jest jedynie generycznym kontenerem dla dowolnych wartości. Aby zrobić z niej użytk, 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*`.

5.2.1. 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`).

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

        // Musi zwrócić false, jeśli parametr jawnie 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.

```

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;
    }
}

```

5.2.2. Porównywanie a dziedziczenie

Jak powinna się zachować metoda `equals`, jeśli parametry jawni i niejawni 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:

- Zwrotność:** `x.equals(x)` powinno zwracać `true`, jeśli `x` nie ma wartości `null`.
- 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`.
- 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.
- Niezmienność:** jeśli obiekty, do których odwołują się zmienne `x` i `y`, nie zmieniły się, kolejne wywołania `x.equals(y)` zwracają tę samą wartość.
- 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` zmuszą 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.

Założymy 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 jawnemu 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.
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;
```

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

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

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 przedefiniujesz metodę equals w podklasie, użyj odwołania super.equals(other).



Do porównania elementów dwóch tablic można użyć statycznej metody `Arrays.equals`.



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

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

Ta metoda deklaruje parametr jawnego typu `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(*typ*[] a, *typ*[] b) 5.0

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.

5.2.3. Metoda `hashCode`

Kod mieszący (ang. *hash code*) to skrót do obiektu w postaci pochodzącej od niego liczby całkowitej. Kody mieszące powinny mieć różne wartości. To znaczy, że jeśli x i y to dwa różne obiekty, powinno istnieć wysokie prawdopodobieństwo, że `x.hashCode() == y.hashCode()`

to dwie różne liczby. Tabela 5.1 przedstawia trzy przykładowe kody mieszające zwrócone przez metodę hashCode klasy String.

Tabela 5.1. Kody mieszające zwrócone przez metodę hashCode

Łańcuch	Kod mieszający
Witaj	83588971
Henryk	-2137002381
Kwiatek	1350142454

Klasa String oblicza kody mieszające 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 kod mieszający, który jest derywowany od adresu obiektu w pamięci. Przeanalizujmy poniższy kod:

```
String s = "OK";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("OK");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Wynik przedstawia tabela 5.2.

Tabela 5.2. Kody mieszające łańcuchów i obiektów klasy StringBuilder

Obiekt	Kod mieszający
s	2556
sb	20526976
t	2556
tb	205271144

Należy zauważyć, że łańcuchy s i t mają takie same kody, ponieważ kody mieszające łańcuchów są pochodnymi ich **zawartości**. Obiekty sb i tb mają różne kody, ponieważ dla klasy StringBuilder nie zdefiniowano metody hashCode. W związku z tym domyślna metoda hashCode klasy Object utworzyła ich kody mieszające na podstawie adresów w pamięci.

W przypadku przedefiniowania metody equals należy także przedefiniować metodę hashCode dla obiektów, które użytkownicy mogą wstawiać do tablicy mieszającej (ang. *hash table*) — tablice mieszające zostały opisane w rozdziale 13.

Metoda hashCode powinna zwracać liczbę całkowitą (może być ujemna). Aby kody mieszające różnych obiektów były różne, wystarczy użyć kombinacji kodów mieszających pól tych obiektów.

Poniżej znajduje się przykładowa metoda hashCode klasy Employee:

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

Od Java 7 można tu dokonać dwóch udoskonaleń. Po pierwsze, lepiej jest 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.

```
public int hashCode()
{
    return 7 * Objects.hashCode(name)
    + 11 * new Double(salary).hashCode()
    + 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 mieścić identyfikatory, a nie imiona i nazwiska pracowników lub adresy w pamięci.



Jeśli pola są typu tablicowego, można użyć metody `Arrays.hashCode`, która oblicza kod mieszający złożony z kodów mieszających elementów tablicy.

java.lang.Object **1.0**

■ `int hashCode()`

Zwraca kod mieszający obiektu. Kod ten może być każdą dodatnią lub ujemną liczbą całkowitą. Identyczne obiekty powinny mieć takie same kody mieszające.

java.lang.Object **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.util.Arrays **1.2**

- static int hashCode(*typ*[] a) **5.0**

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

5.2.4. 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`, uwzględniającą 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:

```
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:

```
Point 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 przeciwnieństwie 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 powstający w ten sposób łańcuch.

Klasa `Object` zawiera metodę `toString`, która drukuje nazwę klasy i kod mieszący 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`.

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 11., jeszcze lepszym rozwiązaniem jest użycie obiektu klasy `Logger` i zastosowanie następującego wywołania:

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

Program na listingu 5.8 zawiera implementacje metod `equals`, `hashCode` oraz `toString` w klasach `Employee` (listing 5.9) i `Manager` (listing 5.10).



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`.



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

Listing 5.8. 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)
    {
        Employee alicel = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
        Employee alice2 = alicel;
        Employee alice3 = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
        Employee bob = new Employee("Bartosz Borkowski", 50000, 1989, 10, 1);

        System.out.println("alicel == alice2: " + (alicel == alice2));
        System.out.println("alicel == alice3: " + (alicel == alice3));
        System.out.println("alicel.equals(alice3): " + alicel.equals(alice3));
        System.out.println("alicel.equals(bob): " + alicel.equals(bob));
        System.out.println("bob.toString(): " + bob);

        Manager carl = new Manager("Karol Krakowski", 80000, 1987, 12, 15);
        Manager 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("alicel.hashCode(): " + alicel.hashCode());
        System.out.println("alice3.hashCode(): " + alice3.hashCode());
```

```

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

```

Listing 5.9. equals/Employee.java

```

package equals;

import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Objects;

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

    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public Date 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 jawnym parametrem 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
Employee 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 5.10. 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)
    {
        bonus = b;
    }

    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        Manager other = (Manager) otherObject;
        // Młoda 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 super.toString() + "[bonus=" + 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 wielu językach programowania, zwłaszcza w C, rozmiary wszystkich tablic muszą być ustalone w czasie komplikacji. Nie podoba się to programistom, ponieważ zmusza ich to do niewygodnych kompromisów. Ile 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 znacznie lepiej. Rozmiar tablicy można ustawić w trakcie działania programu.

```

int actualSize = . . .;
Employee[] 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 13., choć znajomość tych technik nie jest konieczna, aby móc używać typu `ArrayList`.

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

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

Wpiswanie parametru typu po obu stronach jest trochę żmudne. Dlatego w Java 7 parametr ten można po prawej stronie 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 ogólny 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 ogólny to `Employee`.



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 użycia starej klasy `Vector`.

Nowe elementy do listy są dodawane za pomocą metody `add`. Poniższy fragment kodu zapełnia 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 zapelnieniem 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 zbiegania 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 przeciągać 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<T> **1.2**

■ `ArrayList<T>()`

Tworzy pustą listę tablicową.

■ `ArrayList<T>(int initialCapacity)`

Tworzy pustą listę tablicową o podanej pojemności.

Parametry: `initialCapacity` Początkowa pojemność listy

■ `boolean add(T obj)`

Dodaje element na końcu listy. Zawsze zwraca wartość true.

Parametry: `obj` Element do dodania

■ `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.

Parametry: `capacity` Docelowa pojemność listy

■ `void trimToSize()`

Redukuje pojemność listy do jej aktualnego rozmiaru.

5.3.1. 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:

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

Do zapełniania 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, new Date());
```

w czasie komplikacji spowoduje tylko ostrzeżenie, a problemy zaczynają 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:

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

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

```
X[] 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 13.

Od Java 5.0 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 5.11 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 5.11. 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.  
        ArrayList<Employee> staff = new ArrayList<>();
```

```

staff.add(new Employee("Karol Krakowski", 75000, 1987, 12, 15));
staff.add(new Employee("Henryk Kwiątek", 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<T> 1.2

■ **void set(int index, T obj)**

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

Parametry: **index** Pozycja (wartość od 0 do `size() - 1`)
obj Nowa wartość

■ **T get(int index)**

Pobiera wartość zapisaną w określonym indeksie.

Parametry: **index** Indeks elementu, który ma zostać pobrany
(wartość od 0 do `size() - 1`)

■ **void add(int index, T obj)**

Dodaje element i przesuwa pozostałe do góry.

Parametry: **index** Indeks wstawianego elementu (wartość od 0
do `size() - 1`)
obj Nowy element

■ **T remove(int index)**

Usuwa element i przesuwa w dół wszystkie elementy, które znajdują się nad nim.
Usunięty element jest zwracany.

Parametry: **index** Indeks elementu, który ma zostać usunięty
(wartość od 0 do `size() - 1`)

5.3.2. 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 komplikacji.

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 zgłoszonym 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. Osłony obiektów i autoboxing

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 **osłonowymi** (ang. *wrapper*). Nazwy klas osłonowych są oczywiste: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` i `Boolean` (sześć pierwszych dziedziczy po wspólnej nadklasie `Number`). Klasy osłonowe są niezmienialne, tzn. nie można zmienić opakowanej wartości po utworzeniu osłony. Ponadto są finalne, co oznacza, że nie można tworzyć ich podklas.

Założymy, 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 osłonowa. Listę obiektów klasy `Integer` można zadeklarować bez problemu.

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



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.

Kolejna innowacja wprowadzona w Java SE 5.0 ułatwia 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 osłonowego operator inkrementacji:

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

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

W większości przypadków wydaje się, że typy podstawowe i ich osłony są jednym i tym samym. Jest między nimi tylko jedna znacząca różnica: tożsamość. Jak wiadomo, operator `==` zastosowany do obiektów osłonowych 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 osłonowych za pomocą metody `equals`.



Specyfikacja automatycznego opakowywania wymaga, aby typy `boolean`, `char ≤ 127` oraz `short` i `int` w przedziale `-128 do 127` były opakowywane w ustalone obiekty. Jeśli na przykład w powyższym fragmencie kodu `a` i `b` zostałyby zainicjowane wartością 100, porównywanie musiałoby zakończyć się powodzeniem.

Należy również zaznaczyć, że opakowywanie i odpakowywanie zawsze zaznaczamy „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.

Obiekty osłonowe typów liczbowych są także często używane do innego celu. Projektanci Javy odkryli, że obiekty osłonowe 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 osłonowych 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 **niezmienialne** — informacje zawarte w obiekcie osłonowym nie mogą być zmieniane. Nie można użyć tych klas osłonowych do tworzenia metod modyfikujących parametry liczbowe.

Aby napisać metodę zmieniającą parametry liczbowe, należy użyć jednego z typów Holder zdefiniowanych w pakiecie `org.omg.CORBA`. Dostępne są typy `IntHolder`, `BooleanHolder` itd. Każdy taki typ ma publiczne (!) pole o nazwie `value`, poprzez które można uzyskać dostęp do wartości.

```
public static void triple(IntHolder x)
{
    x.value = 3 * x.value;
}
```

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 zainicjowany 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

Przed wersją 5.0 Javy każda metoda miała stałą liczbę parametrów. Obecnie jednak można 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 trzykropek (...) 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[] { new Integer(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.MIN_VALUE;
    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[] { new Integer(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. Klasa 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;

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

```
    public String getAbbreviation() { return abbreviation; }

}
```

Wszystkie typy wyliczeniowe są podklasami klasy `Enum`. Dziedziczą po niej kilka metod. Do najbardziej przydatnych należy metoda `toString`, która zwraca nazwę stałej wyliczeniowej. Na przykład wywołanie `Size.SMALL.ToString()` zwracałańcu `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 pomineliśmy. 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 12.).

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)
    {
        Scanner 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> **5.0**

- 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.

5.7. Refleksja

Biblioteka refleksyjna dostarcza bogaty i zaawansowany zestaw narzędzi do pisania programów, które dynamicznie zarządzają kodem Javy. Mechanizm ten jest często wykorzystywany w JavaBeans — składniku architektonicznym Javy (więcej informacji na temat JavaBeans znajduje się w drugim tomie). Dzięki refleksji Java obsługuje narzędzia, do których przyzwyczajeni są użytkownicy języka Visual Basic. Narzędzia służące do szybkiej budowy aplikacji mogą dynamicznie uzyskiwać informacje o funkcjonalności dodawanych klas, zwalniając kiedy w trakcie projektowania lub działania programu dodawane są nowe klasy.

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 pominiąć resztę rozdziału i wrócić do niego kiedy indziej.

5.7.1. 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 c1 = 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());
```

drukuję:

Employee Henryk Kwiątek

jeśli `e` jest zwykłym pracownikiem, lub

Manager Henryk Kwiątek

jeśli `e` jest kierownikiem.

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

```
Date d = new Date();  
Class c1 = d.getClass();  
String name = c1.getName(); // Zmienna name jest ustawiana na java.util.Date.
```

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 podrozdziale 5.7.2, „Podstawy przechwytywania wyjątków”.



Przy uruchamianiu programu najpierw ładowana jest klasa zawierająca metodę `main`. Ładuje ona wszystkie klasy, których potrzebuje. Każda z tych załadowanych klas ładuje kolejne klasy, których potrzebuje itd. W przypadku dużej aplikacji proces ten może zajmować dużo czasu, co denerwowałoby użytkownika. Można jednak zastosować pewną sztuczkę, która da użytkownikowi wrażenie, że program uruchamia się szybciej. Należy sprawić, aby klasa zawierająca metodę `main` nie odwoływała się bezpośrednio do innych klas. Najpierw należy wyświetlić ekran powitalny, a potem ręcznie wymusić załadowanie pozostałych klas za pomocą wywołania `Class.forName`.

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

```
Class c11 = Date.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`.



Od Java SE 5.0 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 13.



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) . . .
```

Inna przydatna metoda umożliwia tworzenie w locie egzemplarzy klas. Nazywa się `newInstance()`. Na przykład:

```
e.getClass().newInstance();
```

Powyzsza instrukcja tworzy egzemplarz tego samego typu co `e`. Metoda `newInstance` wywołuje konstruktor domyślny (ten, który nie przyjmuje żadnych parametrów). Jeśli klasa nie ma konstruktora domyślnego, powodowany jest wyjątek.

Przy użyciu metod `forName` i `newInstance` można utworzyć obiekt z nazwy klasy przechowywanej w łańcuchu.

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```



Jeśli konieczne jest podanie parametrów dla konstruktora klasy, której obiekt jest tworzony w ten sposób, nie można użyć powyższych instrukcji. W zamian trzeba użyć metody `newInstance` klasy `Constructor`.



C++ 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 `type_info` w C++, a metoda `getClass` jest odpowiednikiem operatora `typeid`. Klasa `Class` Java jest jednak nieco bardziej wszechstronna niż `type_info`. Ta druga potrafi tylko zwrócić łańcuch z nazwą typu. Nie tworzy nowych obiektów tego typu.

5.7.2. Podstawy przechwytywania wyjątków

Techniki przechwytywania wyjątków zostały opisane w rozdziale 11., 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 napisano procedurę do ich obsługi. Jednak wiele często spotykanych wyjątków, jak dostęp do referencji `null`, jest niekontrolowanymi. Kompilator nie sprawdza, czy zadano o procedurę obsługi dla tych błędów — czas należy poświęcić na ich unikanie, a nie na pisanie procedur do ich obsługi.

Nie wszystkich błędów można jednak uniknąć. Jeśli wyjątek może wystąpić mimo najlepszych starań programisty, kompilator nalega na napisanie procedury do jego obsługi. Przykładem metody powodującej kontrolowany wyjątek jest `Class.forName`. W rozdziale 11. opisano kilka technik obsługi wyjątków. W tym miejscu prezentujemy tylko najprostszą implementację procedury obsługi wyjątku.

Instrukcje, które mogą spowodować wyjątek kontrolowany, należy umieścić w bloku `try`. W klauzuli `catch` należy wpisać kod obsługujący wyjątek.

```
try
{
    Instrukcje, które mogą powodować wyjątki
```

```

}
catch(Exception e)
{
    Procedura obsługi wyjątku
}

```

Na przykład:

```

try
{
    String name = . . .;           // Pobranie nazwy klasy.
    Class c1 = Class.forName(name); // Może spowodować wyjątek.
    Działania związane z c1
}
catch(Exception e)
{
    e.printStackTrace();
}

```

Jeśli klasa o podanej nazwie nie istnieje, reszta kodu w bloku try jest pomijana i następuje przejście do bloku catch (w tym miejscu drukujemy dane ze śledzenia stosu za pomocą metody printStackTrace klasy Throwable, która jest nadklassą klasy Exception). Jeśli żadna z metod w bloku try nie spowoduje wyjątku, kod w bloku catch zostaje pominięty.

Konieczne jest dostarczanie procedur tylko dla wyjątków kontrolowanych. Można łatwo sprawdzić, które metody powodują kontrolowane wyjątki. Kompilator zgłasza problem za każdym razem, kiedy wywoływana jest metoda mogąca spowodować wyjątek kontrolowany, a nie napisane dla niej procedury obsługi wyjątków.

java.lang.Class 1.0

- static Class forName(string className)

Zwraca obiekt klasy Class reprezentujący klasę o nazwie className.

- Object newInstance()

Zwraca nowy egzemplarz klasy.

java.lang.reflect.Constructor 1.1

- Object newInstance(Object[] args)

Tworzy nowy egzemplarz klasy przy użyciu konstruktora.

Parametry: args

Parametry konstruktora. Więcej informacji na temat podawania parametrów znajduje się w podrozdziale dotyczącym refleksji.

java.lang.Throwable 1.0

- void printStackTrace()

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

5.7.3. 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. Klasa `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 i chronione, ale nie nadklasy.

Listing 5.13 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 class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
    public static boolean isInfinite(double);
    public boolean isInfinite();
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
```

```

public static double parseDouble(java.lang.String);
public static native long doubleToLongBits(double);
public static native long doubleToRawLongBits(double);
public static native double longBitsToDouble(long);

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_VALUE;
public static final java.lang.Class TYPE;
private 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 komplikacji. Ten program będziemy wykorzystywać w kolejnym rozdziale, w którym będziemy zaglądać do wnętrza klas wewnętrznych generowanych automatycznie przez kompilator.

Listing 5.13. reflection/ReflectionTest.java

```

package reflection;

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

/**
 * Ten program wykorzystuje technikę refleksji do wydrukowania pełnych informacji o klasie.
 * @version 1.1 2004-02-21
 * @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
        {
            Scanner in = new Scanner(System.in);
            System.out.println("Podaj nazwę klasy (np. java.util.Date): ");
            name = in.next();
        }

        try
        {
            // Drukowanie nazwy klasy i nadklasy (jeśli != Object).
            Class cl = Class.forName(name);
            Class supercl = cl.getSuperclass();
            String modifiers = Modifier.toString(cl.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.print("klasa " + name);
            if (supercl != null && supercl != Object.class) System.out.print(" rozszerza
            ↩ klasę ");
            + supercl.getName());
        }
    }
}

```

```
        System.out.print("\n{\n");
        printConstructors(c1);
        System.out.println();
        printMethods(c1);
        System.out.println();
        printFields(c1);
        System.out.println("}\n");
    }
    catch (ClassNotFoundException e)
    {
        e.printStackTrace();
    }
    System.exit(0);
}

/**
 * Drukowanie wszystkich konstruktorów klasy.
 * @param cl klasa
 */
public static void printConstructors(Class c1)
{
    Constructor[] constructors = c1.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 c1)
{
    Method[] methods = c1.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 + ";");
    }
}
}

```

java.lang.Class 1.0

- **Field[] getFields() 1.1**
- **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 dziedziczone. Metoda `getDeclaredMethods` zwraca wszystkie metody klasy lub interfejsu, ale nie uwzględnia metod dziedziczonych.

- **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`.

`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 zwrotny.

`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.

5.7.4. Refleksja w analizie obiektów w czasie działania programu

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

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

Teraz pójdziemy o krok dalej i dobierzymy się do **zawartości** pól danych. Oczywiście zawartość określonego pola obiektu o znanych w trakcie pisania programu typie i nazwie można podejrzeć bez trudu. Jednak refleksja umożliwia uzyskanie informacji o polach obiektów, które w czasie komplikacji 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.

```
Employee harry = new Employee("Henryk Kwiątek", 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 Kwiątek".
```

Ten kod sprawia jednak jeden problem. Ponieważ pole name jest prywatne, metoda get spowoduje wyjątek IllegalAccessException. Za pomocą tej metody można sprawdzić tylko 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. Jeśli jednak program nie działa pod kontrolą menedżera zabezpieczeń, można 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 nadkласą 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.

Metoda `get` sprawia jeszcze jeden problem, z którym musimy sobie poradzić. Pole `name` jest typu `String`, a więc nie ma problemu, żeby zwrócić jego wartość jako typ `Object`, ale pole `salary` jest typu `double`, a w Javie typy liczbowe nie są obiektami. W tym przypadku można użyć metody `getDouble` z klasy `Field` lub wywołać metodę `get`. W tym drugim przypadku mechanizm refleksji automatycznie opakuje wartość pola w obiekt odpowiedniego typu, tutaj `Double`.

Oczywiście wartości, które można sprawdzić, można też ustawić. Wywołanie `f.set(obj, value)` ustawia pole `f` obiektu `obj` na wartość `value`.

Listing 5.14 przedstawia generyczną metodę `toString`, która działa z **każdą** klasą. Wszystkie pola danych są pobierane za pomocą metody `getDeclaredFields`. Następnie metoda `setAccessible(true)` umożliwia dostęp do wszystkich tych pól. Pobierane są nazwa i wartość każdego pola. Program na listingu 5.14 rekurencyjnie wywołuje metodę `toString`, zamieniając każdą wartość na łańcuch.

```
class ObjectAnalyzer
{
    public String toString(Object obj)
    {
        Class cl = obj.getClass();
        . .
        String r = cl.getName();
        // Przegląd pól tej klasy i wszystkich jej nadklaś.
        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() + "=";
                    try
                    {
                        Object val = f.get(obj);
                        r += toString(val);
                    }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
            r += "]";
            cl = cl.getSuperclass();
        }
        while (cl != null);
        return r;
    }
    . .
}
```

W pełnej wersji kodu na listingu 5.14 konieczne było rozwiązywanie kilku skomplikowanych problemów. Cykliczne odwołania mogą spowodować nieskończoną rekursję. Dlatego klasa

ObjectAnalyzer (listing 5.15) zapamiętuje obiekty, które były już odwiedzane. Aby zatrzymać do tablic, potrzebne jest zastosowanie innej metody. Więcej szczegółów na ten temat znajduje się w kolejnym podrozdziale.

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

```
ArrayList<Integer> squares = new ArrayList<>();
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 metody `toString`, która może się przydać w wielu programach.

Listing 5.14. objectAnalyzerTest/ObjectAnalyzerTest.java

```
package objectAnalyzer;

import java.util.ArrayList;

/**
 * Ten program analizuje obiekty za pomocą refleksji.
 * @version 1.12 2012-01-26
 * @author Cay Horstmann
 */
public class ObjectAnalyzerTest
{
    public static void main(String[] args)
    {
        ArrayList<Integer> squares = new ArrayList<>();
        for (int i = 1; i <= 5; i++)
            squares.add(i * i);
        System.out.println(new ObjectAnalyzer().toString(squares));
    }
}
```

Listing 5.15. 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)
    {
        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 nadkla.
        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() + "=";
                    try
                    {
                        Class t = f.getType();
                        Object val = f.get(obj);
                        if (t.isPrimitive()) r += val;
                        else r += toString(val);
                    }
                    catch (Exception e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
r += "]";
c1 = c1.getSuperclass();
}
while (c1 != null);

return r;
}
}

```

java.lang.reflect.AccessibleObject 1.2

- void setAccessible(boolean flag)

Ustawia znacznik dostępności obiektu refleksyjnego. Wartość true oznacza wyłączenie mechanizmu kontroli dostępu Javy, dzięki czemu można sprawdzać i ustawiać prywatne pola obiektu.

- boolean isAccessible()

Sprawdza znacznik dostępności obiektu refleksyjnego.

- 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.

5.7.5. 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 zapełniona.

```
Employee[] 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
{
    Object[] 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 żądaną 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`.

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 c1 = a.getClass();
    if (!c1.isArray()) return null;
```

```

Class componentType = c1.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);

```

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 5.16 demonstruje obie metody powiększania tablicy. Zauważ, że rzutowanie typu zwrotnego metody `badCopyOf` spowoduje wyjątek.

Listing 5.16. 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);
        System.out.println(Arrays.toString(a));

        String[] b = { "Tomek", "Daniel", "Henryk" };
        b = (String[]) goodCopyOf(b);
        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
    {
        Object[] 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.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.

5.7.6. Wywoływanie dowolnych metod

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 — umożliwiają 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** (opisane w kolejnym rozdziale). Jednak dzięki refleksji także w Javie można wywoływać dowolne metody.



Wśród niestandardowych rozszerzeń dodanych przez firmę Microsoft do pochodzącego od Javy języka J++ (i jego następcy C#) znajduje się jeszcze inny typ wskaźników do metod o nazwie **delegacja** (ang. *delegate*). Nie jest to to samo co klasa Method opisywana w tym podrozdziale. Bardziej przydatne od delegacji są jednak klasy wewnętrzne (które wprowadzamy w kolejnym rozdziale).

Przypomnijmy sobie, że za pomocą metody `get 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 osłony. Założymy 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);
```

Skoro znamy już zasady używania obiektów klasy `Method`, wykorzystajmy je w praktyce. Listing 5.17 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 5.17 przedstawia pełny kod generycznego tabulatora i kilka przykładowych wywołań.

Listing 5.17. 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 Exception
    {
        // 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;
}

/**
* Drukuję 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 zwrotnym typu double
*/
public static void printTable(double from, double to, int n, Method f)
{
    //Drukowanie metody jako nagłówka tabeli.
    System.out.println(f);

    double dx = (to - from) / (n - 1);

    for (double x = from; x <= to; x += dx)
    {
        try
        {
            double y = (Double) f.invoke(null, x);
            System.out.printf("%10.4f | %10.4f%n", x, y);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
}

```

Powyzszy program pokazuje, ze z obiektami klasy Method možna zrobic wszystko to co ze wskañnikami do funkcji w jzyku 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 wskañnik 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 użyć interfejsu i klas wewnętrznych (które są tematem kolejnego rozdziału). Zgadzamy się zwłaszcza ze stanowiskiem projektantów jzyka Java i nie polecamy używania obiektów typu Method dla funkcji zwrotnych. Dzięki użyciu interfejsów dla metod zwrotnych (zobacz nastepny 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 niejawny powinien mieć wartość `null`. Wartości typów podstawowych należy przekazywać w obiektach osłonowych. Wartości zwrotne typów podstawowych muszą być odpakowywane.

5.8. 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 podklaś 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 podklaś `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 początek 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 nieświąteczne:

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Z tego powodu dziedziczenie nie jest właściwą techniką w tym przypadku.

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 nic 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 sprawdeń 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 klasy wewnętrzne.

6

Interfejsy i klasy wewnętrzne

W tym rozdziale:

- Interfejsy
- Klonowanie obiektów
- Interfejsy i metody zwrotne
- Klasy wewnętrzne
- Klasy pośredniczące

Znamy już wszystkie podstawowe narzędzia programowania obiektowego. W tym rozdziale poznamy kilka bardziej zaawansowanych, a powszechnie stosowanych technik. Wiedza ta, mimo iż mniej intuicyjna, stanowi uzupełnienie poznanego dotychczas zestawu narzędzi programistycznych.

Na początku zajmiemy się **interfejsami** (ang. *interface*), które opisują, **co** klasy powinny robić, ale nie określają **w jaki sposób**. Klasa może **implementować** jeden lub więcej interfejsów. Obiektów klas implementujących interfejsy można używać zawsze wtedy, kiedy wymagana jest zgodność z określonym interfejsem. Następnie przejdziemy do klonowania obiektów (czasami nazywanego kopowaniem głębokim). Klon obiektu to nowy obiekt z takim samym stanem jak pierwowzór. Modyfikacje klonu nie mają wpływu na oryginalny obiekt.

Kolejne zagadnienie to **klasy wewnętrzne** (ang. *inner class*). Z technicznego punktu widzenia są one nieco skomplikowane, ponieważ ich definicje znajdują się wewnętrz innych klas, a ich metody mają dostęp do pól klas je zawierających. Klasy wewnętrzne znajdują zastosowanie przy projektowaniu zbiorów współpracujących ze sobą klas. W szczególności umożliwiają pisanie zwięzłego i profesjonalnego kodu obsługującego zdarzenia GUI.

Rozdział zamyka opis obiektów **pośrednich**, implementujących dowolne interfejsy. Obiekty te to bardzo wyspecjalizowane narzędzia, których używają programiści narzędzi systemowych. Przy pierwszej lekturze tej książki można ten podrozdział pominąć.

6.1. Interfejsy

W Javie interfejs nie jest klasą, ale zestawem **wymagań**, które muszą być spełnione, aby klasa została uznana za zgodną z danym interfejsem.

Typowy dostawca usług mówi coś takiego: „Jeśli twoja klasa spełnia wymagania określonego interfejsu, ja wykonam moją usługę”. Przeanalizujmy konkretny przykład. Metoda `sort` z klasy `Array` sortuje tablice obiektów, ale pod jednym warunkiem: obiekty te muszą należeć do klas, które implementują interfejs `Comparable`.

Interfejs `Comparable` wygląda następująco:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

Oznacza to, że każda klasa implementująca powyższy interfejs musi mieć metodę `compareTo`, która przyjmuje parametr typu `Object` i zwraca liczbę całkowitą.



Od Java SE 5.0 interfejs `Comparable` jest typem sparametryzowanym:

```
public interface Comparable<T>
{
    int compareTo(T other); // Parametr jest typu T.
}
```

Na przykład klasa implementująca interfejs `Comparable<Employee>` musi zawierać metodę:

```
int compareTo(Employee other)
```

Nadal można używać „surowego” typu `Comparable` bez parametru typu, ale wtedy konieczne jest wykonywanie na własną rękę rzutowania parametru metody `compareTo` na odpowiedni typ.

Nie ma potrzeby stosowania słowa kluczowego `public` w deklaracjach metod w interfejsie, ponieważ wszystkie metody są automatycznie publiczne.

Oczywiście jest jeszcze jedno wymaganie, którego nie można opisać w interfejsie. W wywołaniu `x.compareTo(y)` metoda `compareTo` musi **porównać** dwa podane obiekty i zwrócić informację na temat tego, który z nich jest większy. Liczba ujemna oznacza, że większy jest `y`, zero, że obiekty są równe, a liczba dodatnia, że większy jest `x`.

Ten interfejs ma tylko jedną metodę, ale są interfejsy, które mają ich więcej. Później przekonamy się, że interfejsy mogą nawet zawierać definicje stałych. Ważniejsze jest jednak to, czego interfejs **nie może zawierać**. Interfejsy nie mogą zawierać pól obiektowych ani imple-

mentować metod. Dostarczanie pól obiektowych i implementacja metod są czynnościami, które należą do klas implementujących interfejsy. Interfejs można kojarzyć z klasą abstrakcyjną, która nie zawiera żadnych pól obiektowych — nieco dalej opisujemy różnice między tymi dwoma konstrukcjami.

Przejdźmy do metody `sort` klasy `Array`. Chcemy posortować tablicę obiektów klasy `Employee`. W związku z tym klasa `Employee` musi **implementować** interfejs `Comparable`.

Aby klasa implementowała określony interfejs, należy wykonać dwie czynności:

- 1 Zadeklarować, że klasa będzie implementowała dany interfejs.
- 2 Zdefiniować wszystkie metody danego interfejsu.

Implementacja interfejsu jest wyrażana za pomocą słowa kluczowego `implements`:

```
class Employee implements Comparable
```

Następnie należy zdefiniować metodę `compareTo`. Powiedzmy, że chcemy porównywać pracowników pod względem wysokości ich pensji. Poniżej znajduje się odpowiednia implementacja metody `compareTo`:

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    return Double.compare(salary, other.salary);
}
```

W przykładzie tym użyliśmy statycznej metody `Double.compare`, która zwraca wartość ujemną, gdy pierwszy argument jest mniejszy od drugiego, 0 — gdy argumenty są równe, oraz dodatnią wartość w pozostałych przypadkach.



W deklaracji metody `compareTo` w interfejsie nie użyto słowa kluczowego `public`, ponieważ wszystkie metody w **interfejsie** są publiczne. Natomiast w implementacji interfejsu słowo `to` musi się pojawić w deklaracji metody. W przeciwnym przypadku kompilator uzna, że metoda jest widoczna w obrębie pakietu — co jest domyślnym zachowaniem dla klas. Następnie kompilator zgłasza, że nadano mniejsze uprawnienia dostępu.

Od Java SE 5.0 powyższe zadanie można wykonać nieco lepiej. Zaimplementujemy interfejs `Comparable<Employee>`.

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
    ...
}
```

Warto zauważyć, że mało eleganckie rzutowanie parametru `Object` zniknęło.



Metoda `compareTo` interfejsu `Comparable` zwraca liczbę całkowitą. Jeśli obiekty nie są równe, nie ma znaczenia, jaka liczba dodatnia lub ujemna zostanie zwrócona. Ta elastyczność może się okazać przydatna przy porównywaniu pól przechowujących liczby całkowite. Niech na przykład każdy pracownik ma unikatowy identyfikator w postaci liczby całkowitej. Chcemy wykonać sortowanie według identyfikatorów. W takim przypadku wystarczy zwrócić wynik działania `id - inny.id`. Wartość ta będzie ujemna, jeśli pierwszy identyfikator jest mniejszy od drugiego, będzie wynosiła 0, jeśli są takie same, lub będzie dodatnia w przeciwnym przypadku. Istnieje jednak jedna pułapka: porównywane liczby nie mogą być zbyt duże, ponieważ mogą spowodować przekroczenie zakresu. Jeśli wiadomo, że identyfikatory nie mogą mieć wartości ujemnych lub że ich maksymalna wartość bezwzględna nie przekracza wartości `(Integer.MAX_VALUE-1)/2`, nie ma problemu.

Oczywiście sztuczka ta nie nadaje się do stosowania z liczbami zmiennoprzecinkowymi. Różnica `salary - inna.salary` może zostać zaokrąglona do 0, jeśli porównywane liczby mają bardzo zbliżone, ale nie identyczne wartości. Wywołanie `Double.compare(x, y)` zwraca -1, gdy $x < y$, lub 1, gdy $x > 0$.

Wiemy już, co klasa musi zrobić, aby móc skorzystać z usługi sortowania — musi zaimplementować metodę `compareTo`. To nadzwyczaj rozsądne. Musi istnieć jakiś sposób na porównywanie obiektów przez metodę `sort`. Ale czemu klasa `Employee` nie może definiować metody `compareTo` bez implementacji interfejsu `Comparable`?

Powodem wprowadzenia interfejsów w Javie było to, że jest to język ze **ścisłą kontrolą typów**. Podczas tworzenia wywołania metody kompilator musi mieć możliwość sprawdzenia, czy ta metoda w ogóle istnieje. W metodzie `sort` można znaleźć następujące instrukcje:

```
if (a[i].compareTo(a[j]) > 0)
{
    // Zamiana miejscami obiektów a[i] i a[j].
    .
}
```

Kompilator musi wiedzieć, czy `a[i]` rzeczywiście udostępnia metodę `compareTo`. Jeśli `a` jest tablicą obiektów klasy implementującej interfejs `Comparable`, wiadomo, że istnieje metoda `compareTo`, ponieważ każda klasa implementująca interfejs `Comparable` musi ją definiować.



Można się spodziewać, że metoda `sort` klasy `Arrays` przyjmuje tablicę `Comparable[]`, dzięki czemu kompilator może zgłaszać błędy, jeśli metoda ta zostanie wywołana przy użyciu tablicy zawierającej obiekty nieimplementujące interfejsu `Comparable`. Niestety tak nie jest. W zamian metoda `sort` przyjmuje tablicę `Object[]` i stosuje mało eleganckie rzutowanie:

```
// Kod z biblioteki standardowej — niezalecany
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
    // Zamiana miejscami obiektów a[i] i a[j].
    .
}
```

Jeśli obiekt `a[i]` nie należy do klasy implementującej interfejs `Comparable`, maszyna wirtualna zgłasza wyjątek.

Listing 6.1 przedstawia pełny kod programu sortującego tablicę egzemplarzy klasy Employee, która z kolei została zaprezentowana na listingu 6.2.

Listing 6.1. interfaces/EmployeeSortTest.java

```
package interfaces;

import java.util.*;

/**
 * Ten program demonstruje sposób użycia interfejsu Comparable.
 * @version 1.30 2004-02-27
 * @author Cay Horstmann
 */
public class EmployeeSortTest
{
    public static void main(String[] args)
    {
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Henryk Kwiątek", 35000);
        staff[1] = new Employee("Karol Kowalski", 75000);
        staff[2] = new Employee("Tadeusz Nowak", 38000);

        Arrays.sort(staff);

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

Listing 6.2. interfaces/Employee.java

```
package interfaces;

public class Employee implements Comparable<Employee>
{
    private String name;
    private double salary;

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }
}
```

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

/**
 * Porównuje pracowników według wysokości pensji.
 * @param other inny obiekt klasy Employee
 * @return wartość ujemna, jeśli pracownik ma niższą pensję niż inny (other) pracownik,
 * 0, jeśli pensje są równe, w przeciwnym razie liczba dodatnia
 */
public int compareTo(Employee other)
{
    return Double.compare(salary, other.salary);
}
```

java.lang.Comparable<T> 1.0

- `int compareTo(T other)`

Porównuje obiekt z obiektem `other` i zwraca liczbę ujemną, jeśli pierwszy obiekt jest mniejszy od drugiego, zero, jeśli obiekty są równe, lub liczbę dodatnią w przeciwnym przypadku.

java.util.Arrays 1.2

- `static void sort(Object[] a)`

Sortuje zawartość tablicy `a` przy użyciu zoptymalizowanego algorytmu mergesort. Wszystkie elementy tablicy muszą należeć do klas, które implementują interfejs `Comparable`, i muszą dać się porównać.

java.lang.Integer 7

- `static int compare(int x, int y)`

Zwraca ujemną liczbę całkowitą, gdy $x < y$, zero, gdy x i y są równe, oraz dodatnią liczbę całkowitą w pozostałych przypadkach.

java.lang.Double 7

- `static int compare(double x, double y)`

Zwraca ujemną liczbę całkowitą, gdy $x < y$, zero, gdy x i y są równe, oraz dodatnią liczbę całkowitą w pozostałych przypadkach.

6.1.1. Własności interfejsów

Interfejsy nie są klasami. Nie można utworzyć egzemplarza interfejsu za pomocą operatora `new`:

```
x = new Comparable(. . .); // błęd
```



Zgodnie ze standardem języka „implementator musi zapewnić, że dla wszystkich x i y $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ ” (oznacza to, że jeśli wywołanie $x.\text{compareTo}(y)$ spowoduje wyjątek, to $y.\text{compareTo}(x)$ również musi spowodować wyjątek). Słowo sgn oznacza znak liczby, $\text{sgn}(n)$ ma wartość -1 , jeśli n jest wartością ujemną, 0 jeśli n jest równe 0 , lub 1 , jeśli n jest wartością dodatnią. Innymi słowy, jeśli parametry metody `compareTo` zostaną zamienione miejscami, znak (ale niekoniecznie sama wartość) wyniku również musi zmienić się na przeciwny.

Podobnie jak w przypadku metody `equals`, mogą wystąpić problemy z dziedziczeniem.

Ponieważ klasa `Manager` rozszerza klasę `Employee`, implementuje interfejs `Comparable` \rightarrow `<Employee>`, a nie `Comparable<Manager>`. Jeśli w klasie `Manager` zostanie przesłonięta metoda `compareTo`, należy liczyć się z porównywaniem zwykłych pracowników z kierownikami. Nie można zwyczajnie rzutować typu zwykłego pracownika na kierownika:

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other; //nie
        ...
    }
    ...
}
```

To łamie zasadę „antysymetrii”. Jeśli x jest typu `Employee`, a y typu `Manager`, wywołanie $x.\text{compareTo}(y)$ nie spowoduje wyjątku — obiekty zostaną porównane jako zwykli pracownicy. Jednak odwrotne wywołanie $y.\text{compareTo}(x)$ spowoduje wyjątek `ClassCastException`.

Jest to taka sama sytuacja jak w przypadku metody `equals`, którą opisywaliśmy w rozdziale 5. Rozwiązanie jest również takie samo. Istnieją dwa osobne scenariusze.

Jeśli porównywanie w podklasach opiera się na innych zasadach, należy zatrzymać porównywanie obiektów, które należą do innych klas. Każda metoda `compareTo` powinna się zaczynać od następującego testu:

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

Jeśli algorytmy porównywania obu klas są takie same, należy utworzyć tylko jedną metodę `compareTo` w nadklasie i zadeklarować ją jako finalną.

Przymijmy na przykład, że chcemy, aby kierownicy byli lepsi od zwykłych pracowników, bez względu na pensję. Co z pozostałymi klasami, jak `Executive` i `Secretary`? Aby ustalić „porządek dziobania”, należy w klasie `Employee` zdefiniować klasę o nazwie np. `rank`. Niech każda podklasa przesłania metodę `rank` i implementuje metodę `compareTo`, która bierze pod uwagę wartości `rank`.

Mimo że nie można tworzyć obiektów interfejsów, dopuszczalne jest deklarowanie ich zmiennych.

Comparable x; //OK

Zmienna interfejsowa musi się odwoływać do obiektu klasy implementującej określony interfejs:

x = new Employee(. . .); // W porządku, pod warunkiem że klasa Employee implementuje // interfejs comparable.

Podobnie jak można sprawdzić za pomocą operatora `instanceof`, czy obiekt należy do danej klasy, można przy użyciu niniejszego operatora sprawdzić, czy dany obiekt implementuje określony interfejs:

```
if (anObject instanceof Comparable) { . . . }
```

Podobnie jak w przypadku klas, można tworzyć hierarchie interfejsów. W ten sposób mogą powstawać łańcuchy interfejsów przechodzące od najwyższego stopnia uogólnienia do najwyższego stopnia specjalizacji. Wyobraźmy sobie, że mamy interfejs o nazwie `Moveable`.

```
public interface Moveable
{
    void move(double x, double y);
}
```

Wtedy nietrudno sobie wyobrazić sens istnienia interfejsu `Powered`, który go rozszerzał:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

Podczas gdy w interfejsie nie może być pól obiektowych ani metod statycznych, mogą być stałe. Na przykład:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
    double SPEED_LIMIT = 95; // Statyczna finalna stała publiczna.
}
```

Metody w interfejsie są zawsze publiczne, ale stałe są statyczne, finalne i publiczne.



Metody w interfejsach można oznaczać słowem kluczowym `public`, a stałe słowami `public static final`. Niektórzy programiści robią to z przyzwyczajenia, a inni z kolei z chęci poprawienia czytelności kodu. Jednak specyfikacja języka Java nie zaleca stosowania niepotrzebnych słów kluczowych. My stosujemy się do niniejszego zalecenia.

Niektóre interfejsy zawierają wyłącznie definicje stałych. Na przykład interfejs `SwingConstants` z biblioteki standardowej definiuje stałe `NORTH`, `SOUTH`, `HORIZONTAL` itd. Każda klasa implementująca ten interfejs dziedziczy wszystkie te stałe. W metodach takiej klasy można pisać odwołania typu `NORTH` zamiast `SwingConstants.NORTH`. Trzeba jednak przyznać, że taki sposób używania interfejsów wydaje się nieco „zwyrodniały”, więc nie zalecamy go.

Podczas gdy klasa może mieć tylko jedną nadklasę, może implementować **kilka** interfejsów. Dzięki temu programista zyskuje pełną dowolność, jeśli chodzi o definiowanie zachowań klas. Na przykład w języku Java istnieje bardzo ważny interfejs o nazwie `Cloneable` (szczegółowy opis tego interfejsu znajduje się w kolejnym podrozdziale). Jeśli określona klasa implementuje ten interfejs, będzie można za pomocą metody `clone` klasy `Object` wykonać dokładną kopię obiektu tej klasy. Przypuśćmy zatem, że chcemy, aby tworzona przez nas klasa miała zarówno funkcjonalność porównywania, jak i klonowania. W takim przypadku wystarczy zaimplementować dwa opisane do tej pory interfejsy:

```
class Employee implements Cloneable, Comparable
```

Poszczególne interfejsy należy oddzielić przecinkami.

6.1.2. Interfejsy a klasy abstrakcyjne

Każdy, kto przeczytał sekcję o klasach abstrakcyjnych w rozdziale 5., może się zastanawiać, czemu projektanci Javy zadawali sobie trud wprowadzania interfejsów. Czemu interfejs `Comparable` nie może być zwykłą klasą abstrakcyjną?

```
abstract class Comparable      // czemu nie?
{
    public abstract int compareTo(Object other);
}
```

W takiej sytuacji klasa `Employee` rozszerzałaby klasę abstrakcyjną i definiowała metodę `compareTo`:

```
class Employee extends Comparable      // czemu nie?
{
    public int compareTo(Object other) { . . . }
}
```

Zasignalizowany problem ma niestety związek z zastosowaniem klas abstrakcyjnych do wyrażania ogólnych własności. Każda klasa może rozszerzać tylko jedną klasę. Przypuśćmy, że klasa `Employee` rozszerza już inną klasę, np. `Person`. W takiej sytuacji nie może już dziedziczyć po innej klasie.

```
class Employee extends Person, Comparable      // błąd
```

Jednak każda klasa może implementować dowolną liczbę interfejsów:

```
class Employee extends Person implements Comparable // OK
```

Inne języki, np. C++, zezwalają na dziedziczenie po więcej niż jednej klasie. Nazywa się to **dziedziczeniem wielokrotnym** (ang. *multiple inheritance*). Projektanci Javy postanowili zrezygnować z tej funkcji, ponieważ komplikuje ona język (jak w C++) lub odbija się na wydajności (jak w języku Eiffel).

Interfejsy mają większość zalet dziedziczenia wielokrotnego, a są pozbawione wad związanych z komplikacją i efektywnością języka.



Jezyk C++ obsługuje dziedziczenie wielokrotne z wszystkimi jego komplikacjami, jak bazowe klasy wirtualne, zasady dominacji i poprzeczne rzutowanie wskaźników. Niektóre programistów języka C++ korzysta z dziedziczenia wielokrotnego, a niektórzy twierdzą nawet, że nie powinno się go używać w ogóle. Inni programiści zalecają wykorzystywać dziedziczenie wielokrotne tylko w stylu „mix-in”. W takim przypadku główna klasa bazowa opisuje obiekt macierzysty, a dodatkowe klasy bazowe (tzw. mix-ins) mogą dostarczać dodatkowe cechy. Ten styl przypomina znaną z Javy metodę jednej klasy bazowej i dodatkowych interfejsów. Jednak klasy dodatkowe w C++ mogą dodawać domyślne zachowania, podczas gdy interfejsy w Javie nie.

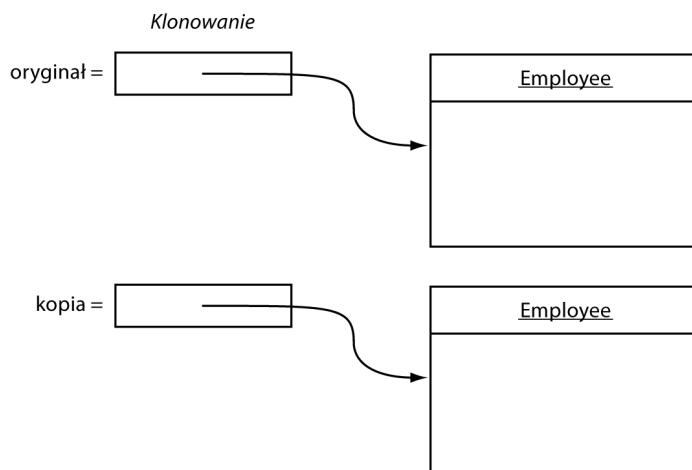
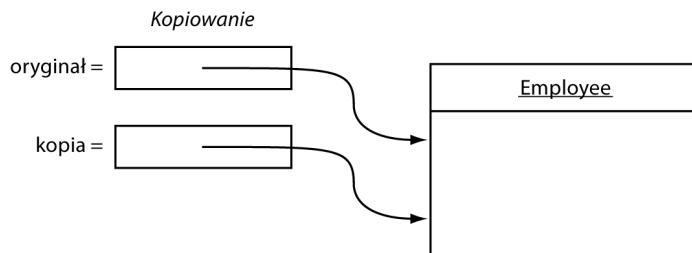
6.2. Klonowanie obiektów

Jeśli skopiujemy zmienną, oryginał i kopia są referencjami do tego samego obiektu (zobacz rysunek 6.1). Oznacza to, że zmiana jednej z nich pociąga za sobą zmianę w drugiej.

```
Employee original = new Employee("Jan Kowalski", 50000);
Employee copy = original;
copy.raiseSalary(10); // To wywołanie także dotyczy oryginału.
```

Rysunek 6.1.

Kopiowanie
i klonowanie



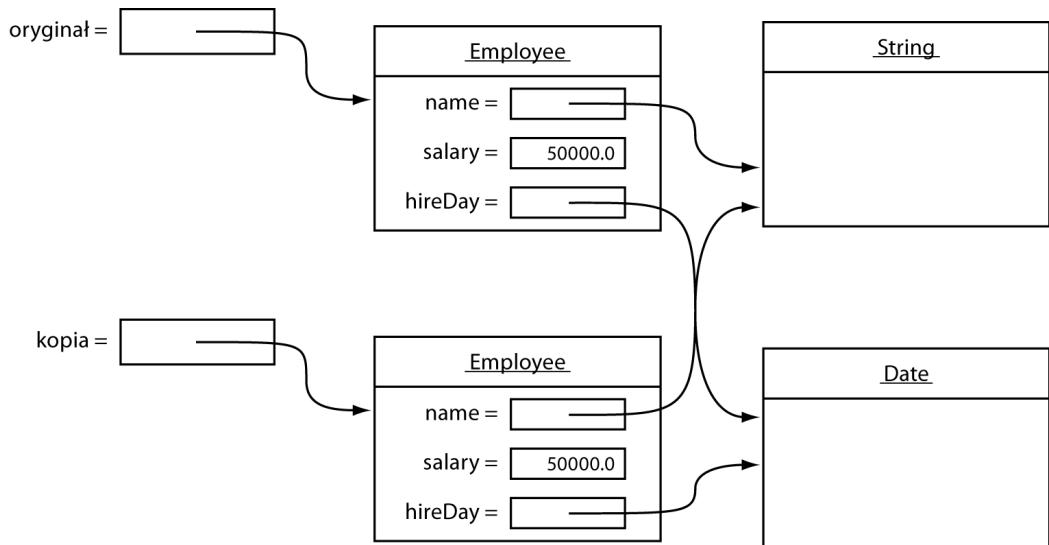
Aby stworzona kopia była całkiem nowym obiektem, którego stan początkowy jest taki sam jak pierwotzoru, ale z czasem mogły pojawić się różnice, należy użyć metody `clone`.

```
Employee copy = original.clone();
copy.raiseSalary(10); // Oryginalny obiekt pozostaje bez zmian.
```

Operacja ta nie jest jednak wcale prosta. Metoda `clone` jest metodą chronioną klasy `Object`, a to oznacza, że we własnym kodzie nie można do niej uzyskać w prosty sposób dostępu. Tylko klasa `Employee` może klonować obiekty typu `Employee`. Ograniczenie to nie zostało wprowadzone bez powodu. Przeanalizujmy, jak może wyglądać implementacja metody `clone` w klasie `Object`. Nie ma ona żadnych informacji na temat obiektów do sklonowania, a więc jedynie wyjście to robienie kopii pole po polu. Jeśli jednak klonowany obiekt zawiera refe-

rencje do podobiektów, skopiowanie takiego pola spowoduje powstanie nowej referencji do tego samego podobiektu. W związku z tym obiekt oryginalny i jego kopia nadal dysponują pewnymi wspólnymi danymi.

Do zobrazowania tego zjawiska wykorzystamy klasę `Employee`, która została wprowadzona w rozdziale 4. Rysunek 6.2 przedstawia, co się dzieje, kiedy obiekt tej klasy zostanie sklonowany przez metodę `clone` klasy `Object`. Widać wyraźnie, że standardowa operacja klonowania jest „pływka” — nie kopiuje obiektów, do których odwołują się zmienne znajdujące się w innych obiektach.



Rysunek 6.2. Kopiowanie płytke

Czy takie płytke kopiowanie w czymś przeszkadza? To zależy. Jeśli podobiekt wspólnie dzielony przez inny obiekt i jego klon jest **niezmienialny**, współdzielenie nie ma żadnych negatywnych skutków. Jest tak na pewno wtedy, gdy podobiekt należy do niezmienialnej klasy, np. `String`. Podobiekt może też pozostać nietknietły przez cały okres życia obiektu, jeśli żadna metoda ustawiająca nie zostanie wywołana na jego rzecz ani żadna metoda nie utworzy do niego referencji.

Często jednak zdarza się, że podobiekt jest zmienialny. W takim przypadku konieczne jest przedefiniowanie metody `clone` w taki sposób, aby wykonywała **kopiowanie głębokie** (ang. *deep copying*), polegające na sklonowaniu także podobiektów. W naszym przykładzie pole `hireDay` jest obiektem klasy `Date`, która jest zmienialna.

W przypadku każdej klasy należy podjąć następujące decyzje:

1. Czy standardowa metoda `clone` wystarczy.
2. Czy można standardową metodę `clone` uzupełnić wywołaniami `clone` na rzecz zmienialnych podobiektów.
3. Czy metoda `clone` nie powinna być wywoływana.

Trzecia z wymienionych opcji jest domyślna. Aby móc wybrać którąś z pozostałych, klasa musi:

1. Implementować interfejs `Cloneable`.
2. Przedefiniować metodę `clone`, dodając do niej modyfikator `public`.



Metoda `clone` w klasie `Object` jest chroniona, przez co nie można w kodzie po prostu umieścić wywołania typu `jakisObiekt.clone()`. Ale czy metody chronione nie są dostępne w podklasach i czy każda klasa nie jest podklassą klasy `Object`? Na szczęście zasady dotyczące dostępu chronionego są bardziej wyrafinowane (zobacz rozdział 5.). Podklasa może wywołać chronioną metodę `clone` tylko w celu sklonowania **swojego własnego** obiektu. Aby obiekty mogły być klonowane przez dowolną metodę, metoda `clone` musi być przedefiniowana jako publiczna.

W takim przypadku użycie interfejsu `Cloneable` nie ma nic wspólnego z normalnym zastosowaniem interfejsów. W szczególności **nie** określa on metody `clone` — metoda ta jest dziedziczona po klasie `Object`. Interfejs jest tu jedynie znacznikiem informującym, że projektant klasy wie, na czym polega klonowanie. Obiekty są tak wrażliwe na punkcie klonowania, że generują kontrolowany wyjątek, jeśli tylko obiekt żądający sklonowania nie implementuje interfejsu `Cloneable`.



Interfejs `Cloneable` jest jednym z kilku **interfejsów znacznikowych** (ang. *tagging interface, marker interface*) Javy. Przypomnijmy, że typowym zastosowaniem interfejsów, jak `Comparable`, jest zapewnienie, że określona klasa zawiera implementację określonej metody lub metod. Interfejs znacznikowy nie ma żadnych metod, a jego jedynym celem jest zezwolenie na użycie operatora `instanceof` do sprawdzenia typu:

```
if (obj instanceof Cloneable) . . .
```

Nie zalecamy stosowania interfejsów znacznikowych.

Nawet jeśli standardowa implementacja metody `clone` (kopiowanie płytke) jest wystarczająca, nadal konieczne jest zaimplementowanie interfejsu `Cloneable`, przedefiniowanie metody `clone` na publiczną i wywołanie metody `super.clone()`. Spójrzmy na przykład:

```
class Employee implements Cloneable
{
    // Zwiększenie widoczności na public i zmiana typu zwrotnego.
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```



Przed Java SE 5.0 metoda `clone` zawsze zwracała typ `Object`. Obecnie kowariantne typy zwrotne umożliwiają określenie odpowiedniego typu dla metod `clone`.

Metoda `clone`, którą oglądaliśmy przed chwilą, nie rozszerza w żaden sposób funkcjonalności metody `Object.clone`. Jedyne jej zadanie to upublicznenie metody. Aby zrobić głębską kopię, należy bardziej się postarać, aby skopiować zmienialne pola obiektowe.

Poniższa metoda `clone` tworzy głęboką kopię obiektu:

```
class Employee implements Cloneable
{
    ...
    public Employee clone() throws CloneNotSupportedException
    {
        // Wywołanie metody Object.clone().
        Employee cloned = (Employee) super.clone();

        // Klonowanie pól zmienialnych.
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

Metoda `clone` klasy `Object` może spowodować wyjątek `CloneNotSupportedException` — robi to, kiedy jest wywoływana na rzecz obiektu, którego klasa nie implementuje interfejsu `Cloneable`. Oczywiście klasy `Employee` i `Date` implementują ten interfejs, a więc wyjątku nie będzie. Kompilator o tym jednak nie wie. Dlatego zadeklarowaliśmy wyjątek:

```
public Employee clone() throws CloneNotSupportedException
```

Czy nie lepiej byłoby przechwycić ten wyjątek?

```
public Employee clone()
{
    try
    {
        return (Employee) super.clone();
    }
    catch (CloneNotSupportedException e) { return null; }
    // To się nie stanie, ponieważ interfejs Cloneable jest zaimplementowany.
}
```

Takie podejście jest odpowiednie dla klas finalnych. W pozostałych sytuacjach lepiej zostawić specyfikator `throws` tam, gdzie jego miejsce.

Należy uważać na klonowanie podklaś. Jeśli na przykład metoda `clone` została zdefiniowana dla klasy `Employee`, można jej użyć do klonowania obiektów klasy `Manager`. Czy metoda `clone` klasy `Employee` poradzi sobie w takiej sytuacji? To zależy od tego, jakie pola zawiera klasa `Manager`. W tym przypadku nie ma problemu, ponieważ pole `bonus` jest typu podstawowego. Jednak klasa `Manager` mogłaby mieć pola wymagające głębokiego kopирования lub takie, które nie są klonowalne. Nie ma żadnej gwarancji, że programista podklaśy wniósł odpowiednie poprawki do metody `clone`, aby odpowiednio wykonywała swoje zadanie. Z tego powodu metoda `clone` w klasie `Object` jest chroniona. Nie możemy jednak skorzystać z tej wygody, jeśli chcemy, aby użytkownicy naszej klasy wywoływali metodę `clone`.

Czy należy implementować metodę `clone` we własnych klasach? Jeśli użytkownicy muszą tworzyć głębokie kopie ich obiektów, to tak. Niektórzy programiści uważają, że należy w ogóle unikać metody `clone` i zamiast niej implementować inną, przeznaczoną do tego samego celu. Zgadzamy się, że metoda `clone` nie jest idealna, ale nie pozbędziemy się problemów z nią.

związań, przerzucając się na inną metodę. W każdym razie klonowanie jest znacznie rzadziej wykonywaną operacją, niż się wydaje. Implementację metody `clone` zawiera mniej niż 5 procent wszystkich klas w bibliotece standardowej.

Program na listingach 6.3 i 6.4 klonuje obiekt klasy `Employee`, a następnie wywołuje dwie metody zmieniające. Metoda `raiseSalary` zmienia wartość pola `salary`, podczas gdy `setHireDay` zmienia stan pola `hireDay`. Żadna ze zmian nie ma wpływu na oryginalny obiekt, ponieważ metoda `clone` wykonuje kopiowanie głębokie.



Wszystkie typy tablicowe mają publiczną (niechronioną) metodę `clone`. Można za jej pomocą tworzyć nowe tablice zawierające kopie wszystkich elementów. Na przykład:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // Nie zmienia elementu luckyNumbers[5].
```

Listing 6.3. `clone/CloneTest.java`

```
package clone;

/**
 * Ten program demonstruje klonowanie.
 * @version 1.10 2002-07-01
 * @author Cay Horstmann
 */
public class CloneTest
{
    public static void main(String[] args)
    {
        try
        {
            Employee original = new Employee("Jan W. Kowalski", 50000);
            original.setHireDay(2000, 1, 1);
            Employee copy = original.clone();
            copy.raiseSalary(10);
            copy.setHireDay(2002, 12, 31);
            System.out.println("original=" + original);
            System.out.println("copy=" + copy);
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}
```

Listing 6.4. `clone/Employee.java`

```
package clone;

import java.util.Date;
import java.util.GregorianCalendar;

public class Employee implements Cloneable
```

```

{
    private String name;
    private double salary;
    private Date hireDay;

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
        hireDay = new Date();
    }

    public Employee clone() throws CloneNotSupportedException
    {
        // Wywołanie metody Object.clone().
        Employee cloned = (Employee) super.clone();

        // Klonowanie pól zmienialnych.
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }

    /**
     * Ustawia datę zatrudnienia na podany dzień.
     * @param year rok zatrudnienia
     * @param month miesiąc zatrudnienia
     * @param day dzień zatrudnienia
     */
    public void setHireDay(int year, int month, int day)
    {
        Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();

        // Przykład zmiany pola obiektowego.
        hireDay.setTime(newHireDay.getTime());
    }

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

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

```



Rozdział 1. drugiego tomu przedstawia alternatywny sposób klonowania obiektów, polegający na zastosowaniu serializacji. Niniejsza technika jest łatwa w implementacji i bezpieczna, ale niestety mało wydajna.

6.3. Interfejsy a sprzężenie zwrotne

W programowaniu często wykorzystywane jest zjawisko **sprzężenia zwrotnego** (ang. *callback*). Ten styl programowania polega na określeniu działań, które mają zostać wykonane w odpowiedzi na określone zdarzenia. Może to być na przykład wykonanie pewnych czynności w odpowiedzi na kliknięcie przycisku lub wybór elementu w menu. Ponieważ nie potrafimy jeszcze implementować interfejsów użytkownika, przeanalizujemy prostszą, ale podobną sytuację.

Pakiet `javax.swing` zawiera klasę `Timer`, którą można wykorzystać do odmierzania czasu. Jeśli na przykład program zawiera zegar, niniejsza klasa może wysyłać co sekundę sygnał, którego programista użyje do aktualizacji tarczy zegara.

Konstruując zegar, można ustawić przedział czasu oraz zdefiniować dowolne działania, które mają zostać wykonane po upłynięciu każdego kolejnego przedziału.

Jak poinformować zegar, co ma robić? W wielu językach programowania w tym celu podaje się nazwę funkcji, która ma być wywoływaną. Jednak klasy w bibliotece standardowej Javy są obiektowe. Trzeba przekazać obiekt jednej z tych klas, a zegar wywoła jedną z metod takiego obiektu. Przekazywany obiekt ma tę przewagę nad funkcją, że może zawierać dodatkowe dane.

Oczywiście zegar musi wiedzieć, którą metodę ma wywołać. Dodatkowo obiekt przekazywany do zegara musi należeć do klasy implementującej interfejs `ActionListener` z pakietu `java.awt.event`. Oto ten interfejs:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

Zegar wywołuje metodę `actionPerformed` po upłynięciu wyznaczonego czasu.



Jak dowiedzieliśmy się w rozdziale 5., w Javie istnieje odpowiednik wskaźników do funkcji w postaci obiektów `Method`. Są one jednak trudne w użyciu, wolniejsze i nie można ich sprawdzać pod kątem bezpieczeństwa w czasie komilacji. W sytuacji, w której w C++ należy użyć wskaźnika do funkcji, w Javie powinno się rozważyć użycie interfejsu.

Wyobraźmy sobie, że chcemy, aby co dziesięć sekund drukowany był napis Kiedy usłyszysz dźwięk, będzie godzina.... Aby wykonać to zadanie, można zdefiniować klasę implementującą interfejs `ActionListener` i w metodzie o nazwie `actionPerformed` umieścić wszystkie instrukcje, które mają zostać wykonane.

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
```

```

        Toolkit.getDefaultToolkit().beep();
    }
}

```

Zatrzymajmy się na chwilę przy parametrze ActionEvent metody actionPerformed. Dostarcza on informacji dotyczących zdarzenia, jak np. obiekt źródłowy, który je wygenerował — więcej na ten temat zostało napisane w rozdziale 8. W tym przypadku jednak dane zdarzenia nie mają znaczenia, a więc można bez obaw zignorować wspomniany parametr.

Następnie utworzymy obiekt naszej klasy i użyjemy go w konstruktorze klasy Timer.

```

ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);

```

Pierwszy parametr konstruktora Timer określa liczbę milisekund, które mają dzielić kolejne powiadomienia — chcemy być powiadamiani co dziesięć sekund. Drugi parametr to obiekt nasłuchujący (ang. *listener object*).

Na koniec uruchamiamy zegar.

```
t.start();
```

Co dziesięć sekund będzie wyświetlany komunikat typu:

Kiedy usłyszysz dźwięk, będzie godzina Fri Dec 14 10:26:40 CET 2007
po którym nastąpi odtworzenie dźwięku.

Listing 6.5 przedstawia opisywany program w całości. Po uruchomieniu zegara wyświetlane jest okno dialogowe z przyciskiem *OK* zamkającym program. W czasie oczekiwania na reakcję użytkownika co dziesięć sekund wyświetlane są aktualne data i godzina.

Listing 6.5. timer/TimerTest.java

```

package timer;

/**
 * @version 1.00 2000-04-13
 * @author Cay Horstmann
 */

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;
// Powyższy import ma na celu zapobiec konfliktowi z klasą java.util.Timer.

public class TimerTest
{
    public static void main(String[] args)
    {
        ActionListener listener = new TimePrinter();

        // Konstrukcja zegara wywołującego obiekt nasłuchujący
        // co dziesięć sekund.
        Timer t = new Timer(10000, listener);
    }
}

```

```

        t.start();

        JOptionPane.showMessageDialog(null, "Zamknąć program?");
        System.exit(0);
    }
}

class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
        Toolkit.getDefaultToolkit().beep();
    }
}

```

Po uruchomieniu tego programu trzeba cierpliwie odczekać dziesięć sekund. Wprawdzie okno dialogowe *Zamknąć program?* pojawia się od razu, ale dźwięk jest odtwarzany po raz pierwszy po upływie dziesięciu sekund.

Warto zauważyć, że klasa `javax.swing.Timer` została w tym programie zimportowana bezpośrednio. Ma to na celu zapobiec konfliktowi klas `javax.swing.Timer` i `java.util.Timer`, które służą do planowania zadań wykonywanych w tle.

javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`

Wyświetla okno dialogowe z komunikatem i przyciskiem *OK*. Okno jest wyświetlane na środku komponentu parent. Jeśli parent ma wartość `null`, okno wyświetla się na środku ekranu.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`

Tworzy zegar powiadamiający obiekt `listener` o upłynięciu liczby milisekund określonej przez parametr `interval`.

- `void start()`

Uruchamia zegar, który wywołuje metodę `actionPerformed` na rzecz obiektów nasłuchujących.

- `void stop()`

Zatrzymuje zegar. Po zatrzymaniu zegar nie wywołuje metody `actionPerformed`.

javax.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`

Zwraca standardowy zestaw narzędzi, na który składają się dane dotyczące środowiska GUI.

- void beep()
- Odtwarza krótki alarm dźwiękowy.

6.4. Klasa wewnętrzna

Klasa wewnętrzna (ang. *inner class*) charakteryzuje się tym, że jest zdefiniowana wewnątrz innej klasy. Są trzy powody, dla których warto definiować tego typu klasy:

- Metody klas wewnętrznych mają dostęp do danych w obszarze, w którym są zdefiniowane — wliczając dane, które w innym przypadku byłyby prywatne.
- Klasę wewnętrzna można ukryć przed innymi klasami tego samego pakietu.
- **Anonimowe klasy wewnętrzne** są przydatne przy definiowaniu metod zwrotnych, ponieważ umożliwiają uniknięcie pisania dużych ilości kodu.

Ponieważ temat ten jest nieco skomplikowany, rozbijemy go na kilka kolejno omówionych części:

- prezentację przykładu prostej klasy wewnętrznej uzyskującej dostęp do pola obiektowego swojej klasy zewnętrznej;
- reguły składniowe klas wewnętrznych;
- omówienie klas wewnętrznych i przekonanie się, w jaki sposób odbywa się ich konwersja na zwykłe klasy; zbyt wrażliwi czytelnicy mogą pominąć tę sekcję;
- opis **lokalnych klas wewnętrznych** (ang. *local inner class*), które mają dostęp do zmiennych lokalnych otaczających je klas;
- wprowadzenie do **anonimowych klas wewnętrznych** (ang. *anonymous inner class*) oraz pokazanie, jak się ich używa do implementacji metod zwrotnych;
- przedstawienie sposobu tworzenia wewnętrznych klas pomocniczych przy użyciu **statycznych klas wewnętrznych**.

6.4.1. Dostęp do stanu obiektu w klasie wewnętrznej

Składnia klas wewnętrznych jest nieco skomplikowana. Dlatego zdecydowaliśmy się na zaprezentowanie prostego, ale raczej sztucznego przykładu przedstawiającego sposób ich użycia. Refaktoryzowaliśmy klasę TimerTest i wydzieliliśmy klasę TalkingClock. Obiekt tej klasy jest tworzony przy użyciu dwóch parametrów: odstępu czasu pomiędzy komunikatami i znacznika włączającego lub wyłączającego alarm dźwiękowy.

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
```



W C++ istnieją **klasy zagnieżdżone**. Klasa zagnieżdżona znajduje się w obrębie klasy zewnętrznej. Oto typowy przykład: klasa listy dowiązań definiuje klasę przechowującą dowiązań i klasę definiującą pozycję iteradora.

```
class LinkedList
{
public:
    class Iterator // klasa zagnieżdżona
    {
public:
    void insert(int x);
    int erase();
    . . .
};

private:
    class Link // klasa zagnieżdżona
    {
public:
    Link* next;
    int data;
    . . .
};
```

Zagnieżdżanie wyraża relacje między **klasami**, a nie **obiektami**. Obiekty klasy `LinkedList` **nie** mają podobiektów typu `Iterator` lub `Link`.

Ma to dwie zalety dotyczące **kontroli nazw i kontroli dostępu**. Ponieważ klasa `Iterator` jest zagnieżdżona w klasie `LinkedList`, na zewnątrz nazywa się `LinkedList::Iterator`, dzięki czemu nie ma możliwości wystąpienia konfliktu z inną klasą o nazwie `Iterator`. W Javie ta zaleta nie ma większego znaczenia, ponieważ podobnie działają pakiety. Zauważmy, że klasa `Link` nie jest **prywatnym** składnikiem klasy `LinkedList`. Jest w pełni ukryta przed pozostałym kodem. Dzięki temu można bezpiecznie używać w niej pól publicznych. Dostęp do nich mogą uzyskać metody klasy `LinkedList` (która musi mieć do nich dostęp), a dla pozostałej części programu są one niewidoczne. Przed wprowadzeniem klas wewnętrznych w Javie tego rodzaju kontrola nie była możliwa.

Jednak klasy wewnętrzne w Javie mają dodatkową funkcjonalność, której brak klasom zagnieżdżonym w C++. Obiekt klasy wewnętrznej zawiera niejawną referencję do obiektu klasy zewnętrznej, który go utworzył. Za pomocą tego wskaźnika obiekt ma dostęp do stanu obiektu zewnętrznego. Szczegóły tego mechanizmu są opisane w dalszej części tego rozdziału.

W Javie statyczne klasy wewnętrzne nie mają tego wskaźnika. Są odpowiednikiem klas zagnieżdżonych w C++.

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

public class TimePrinter implements ActionListener
// klasa wewnętrzna
{
    . . .
}
```

Należy zauważyć, że klasa TimePrinter znajduje się wewnętrznie klasy TalkingClock. **Nie** oznacza to jednak, że każdy obiekt klasy TalkingClock ma pole TimePrinter. Jak się niebawem przekonamy, obiekty klasy TimePrinter są tworzone przez metody klasy TalkingClock.

Poniżej znajduje się kod klasy TimePrinter. Zauważmy, że metoda actionPerformed przed odtworzeniem alarmu sprawdza stan znacznika beep.

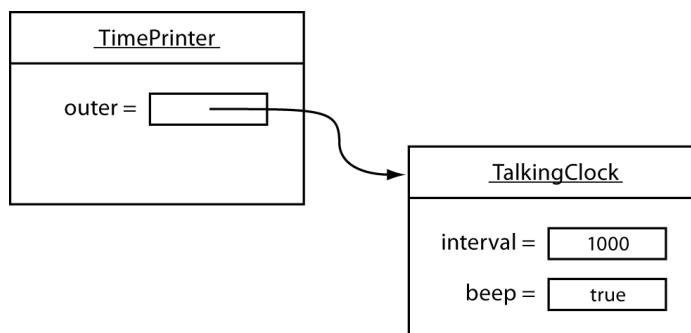
```
private class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

Dzieje się coś zaskakującego. Klasa TimePrinter nie ma pola, czyli zmiennej o nazwie beep. W zamian beep odnosi się do pola obiektu TalkingClock, który utworzył obiekt TimePrinter. Jest to innowacyjne podejście. Normalnie metoda odwoływałaby się do pól obiektu, który ją wywołał. Metody klas wewnętrznych mają dostęp zarówno do własnych pól, **jak i** pól obiektu zewnętrznego.

Aby to działało, obiekt klasy wewnętrznej zawsze ma niejawną referencję do obiektu, który go utworzył (zobacz rysunek 6.3).

Rysunek 6.3.

Obiekt klasy wewnętrznej zawiera referencję do obiektu klasy zewnętrznej



Ta referencja jest niewidoczna w definicji klasy wewnętrznej. Dla jasności nazwiemy referencję do obiektu **zewnętrznego** outer. W takiej sytuacji metoda actionPerformed jest równoważna z poniższą:

```
public void actionPerformed(ActionEvent event)
{
    Date now = new Date();
    System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

Referencja klasy zewnętrznej jest ustawiana w konstruktorze. Kompilator modyfikuje konstruktory wszystkich klas wewnętrznych, dodając parametr referencji do obiektu klasy zewnętrznej. Ponieważ klasa TimePrinter nie definiuje żadnego konstruktora, kompilator syntetyzuje konstruktor domyślny, w wyniku czego powstaje poniższy kod:

```
public TimePrinter(TalkingClock clock) // Kod wygenerowany automatycznie.
{
    outer = clock;
}
```

Przypominamy jeszcze raz, że outer nie jest słowem kluczowym Javy, a służy nam tylko do ilustracji mechanizmów rządzących klasami wewnętrznymi.

Kiedy metoda start tworzy obiekt TimePrinter, kompilator przekazuje referencję this do aktualnego obiektu TalkingClock konstruktorowi:

```
ActionListener listener = new TimePrinter(this); // Automatycznie dodany parametr.
```

Listing 6.6 przedstawia kompletny program testujący naszą klasę wewnętrzna. Jeszcze raz wróćmy do kontroli dostępu. Gdyby klasa TimePrinter była zwykłą klasą, musiałaby uzyskać dostęp do znacznika beep za pomocą metody publicznej klasy TalkingClock. Użycie klasy wewnętrznej zmienia sytuację na lepsze. Nie ma konieczności tworzenia akcesorów, których potrzebuje tylko jedna klasa.



Klasę TimePrinter można było zadeklarować jako prywatną. W takim przypadku obiekty tej klasy mogłyby być tworzone wyłącznie przez metody klasy TalkingClock. Tylko klasy wewnętrzne mogą być prywatne. Zwykłe klasy zawsze mają zasięg pakietowy lub publiczny.

Listing 6.6. innerClass/InnerClassTest.java

```
package innerClass;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * Ten program demonstruje sposób użycia klas wewnętrznych.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class InnerClassTest
{
    public static void main(String[] args)
    {
        TalkingClock clock = new TalkingClock(1000, true);
        clock.start();

        // Niech program działa, dopóki użytkownik nie wciśnie przycisku OK.
        JOptionPane.showMessageDialog(null, "Zamknąć program?");
        System.exit(0);
    }
}

/**
 * Zegar drukujący informacje o czasie w równych odstępach czasu.
```

```

/*
class TalkingClock
{
    private int interval;
    private boolean beep;

    /**
     * Tworzy obiekt TalkingClock.
     * @param interval odstęp czasu pomiędzy kolejnymi komunikatami (w milisekundach)
     * @param beep wartość true oznacza, że dźwięk ma być odtwarzany
     */
    public TalkingClock(int interval, boolean beep)
    {
        this.interval = interval;
        this.beep = beep;
    }

    /**
     * Włączanie zegara.
     */
    public void start()
    {
        ActionListener listener = new TimePrinter();
        Timer t = new Timer(interval, listener);
        t.start();
    }

    public class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}

```

6.4.2. Specjalne reguły składniowe dotyczące klas wewnętrznych

W poprzednim podrozdziale referencję klasy wewnętrznej do obiektu klasy zewnętrznej nazwaliśmy outer. Właściwa składnia jest jednak nieco bardziej skomplikowana. Wyrażenie:

OuterClass.this

określa referencję do klasy zewnętrznej. Na przykład metoda actionPerformed klasy wewnętrznej TimePrinter może wyglądać następująco:

```

public void actionPerformed(ActionEvent event)
{
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}

```

Z drugiej strony konstruktor klasy wewnętrznej można napisać przy użyciu następującej składni:

```
outerObject.new InnerClass(parametry konstrukcyjne)
```

Na przykład:

```
ActionListener listener = this.new TimePrinter();
```

W tym przypadku referencja do klasy zewnętrznej nowo utworzonego obiektu klasy `TimePrinter` zostaje ustawiona na referencję `this` metody, która tworzy obiekt klasy wewnętrznej. Jest to najczęściej spotykany przypadek. Kwalifikator `.this` jak zwykle nie jest konieczny. Można też jednak ustawić referencję do klasy zewnętrznej na inny obiekt, jeśli poda się bezpośrednio jego nazwę. Ponieważ klasa `TimePrinter` jest wewnętrzną klasą publiczną, można na przykład utworzyć obiekt `TimePrinter` dla dowolnego obiektu `TalkingClock`:

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Należy zauważyć, że odwołanie do klasy wewnętrznej wygląda następująco:

```
OuterClass.InnerClass
```

jeśli znajduje się poza klasą zewnętrzną.

6.4.3. Czy klasy wewnętrzne są potrzebne i bezpieczne?

Wielu programistów dodanie klas wewnętrznych do Java 1.1 uznało za poważne odejście od filozofii tworzenia języka prostszego niż C++. Nie da się ukryć, że składnia klas wewnętrznych jest skomplikowana (będzie jeszcze bardziej skomplikowana, gdy zaczniemy zajmować się anonimowymi klasami wewnętrznymi). Interakcje klas wewnętrznych z innymi funkcjami języka, takimi jak kontrola dostępu i zabezpieczenia, nie są jasne.

Czy dodanie funkcjonalności, która jest bardziej elegancka i interesująca niż potrzebna, jest dla Javy początkiem drogi donikąd, którą przeszło tak wiele języków?

Mimo że nie podejmiemy próby udzielenia jednoznacznej odpowiedzi na to pytanie, warto odnotować, że klasy wewnętrzne są zjawiskiem znany **kompilatorowi**, a nie maszynie wirtualnej. Są one konwertowane na zwykłe pliki klas, których nazwy składają się z nazw klasy zewnętrznej i wewnętrznej rozdzielonych symbolem dolara. Maszyna wirtualna nie ma żadnych specjalnych danych na ich temat.

Na przykład klasa `TimePrinter` zdefiniowana w klasie `TalkingClock` jest zamieniana na plik klasy o nazwie `TalkingClock$TimePrinter.class`. Aby to sprawdzić, można przeprowadzić następujący eksperyment: uruchom program `ReflectionTest` z rozdziału 5. i podaj do zbadania klasę `TalkingClock$TimePrinter` albo użyj narzędzia `javap`:

```
javap -private NazwaKlasy
```

Zostaną zwrócone następujące dane:



Przy podawaniu nazwy klasy w wierszu polecień w systemie UNIX należy pamiętać, aby symbol \$ zastąpić symbolem zastępczym. To znaczy program `ReflectionTest` należy uruchomić w następujący sposób:

```
java reflection.ReflectionTest klasaWewnętrzna.TalkingClock\$TimePrinter
```

a narzędzie `javap` następująco:

```
javap -private klasaWewnętrzna.TalkingClock\$TimePrinter
```

```
public class TalkingClock$TimePrinter
{
    public TalkingClock$TimePrinter(TalkingClock);
    public void actionPerformed(java.awt.event.ActionEvent);
    final TalkingClock this$0;
}
```

Jak widać, kompilator wygenerował dodatkowe pole obiektowe `this$0` dla referencji do klasy zewnętrznej (nazwa `this$0` jest utworzona przez kompilator, a więc nie można używać jej w kodzie programu). Jest też parametr konstruktora `TalkingClock`.

Skoro kompilator może wykonać taką transformację, czy nie można by było zrobić tego własnoręcznie? Spróbujmy. Klasę `TimePrinter` zdefiniowalibyśmy jako zwykłą klasę, na zewnątrz klasy `TalkingClock`. Podeczas konstrukcji obiektu `TimePrinter` przekazujemy mu referencję `this` obiektu, który go tworzy.

```
class TalkingClock
{
    . . .

    public void start()
    {
        ActionListener listener = new TimePrinter(this);
        Timer t = new Timer(interval, listener);
        t.start();
    }
}

class TimePrinter implements ActionListener
{
    private TalkingClock outer;
    . . .

    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
}
```

Przejdźmy teraz do metody `actionPerformed`. Musi ona mieć dostęp do `outer.beep`.

```
if (outer.beep) . . . // błęd
```

W tym miejscu pojawia się problem. Klasa wewnętrzna ma dostęp do danych prywatnych klasy zewnętrznej, ale nasza zewnętrzna klasa `TimePrinter` nie.

Dowodzi to, że klasy wewnętrzne rzeczywiście mają większe możliwości niż zwykłe klasy, ponieważ mają większe prawa dostępu.

Niektórych może ciekawić, w jaki sposób klasy wewnętrzne uzyskują swoje zwiększone prawa dostępu, skoro są konwertowane na zwykłe klasy o dziwnych nazwach — maszyna wirtualna nic o nich nie wie. Aby rozwiązać tę zagadkę, zbadajmy jeszcze raz klasę TalkingClock za pomocą programu ReflectionTest:

```
class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int, boolean);

    static boolean access$0(TalkingClock);
    public void start();
}
```

Zwróciły uwagę na statyczną metodę access\$0, którą kompilator dodał do klasy zewnętrznej. Zwraca ona pole beep obiektu, który jest przekazany jako parametr. (Nazwa metody może być trochę inna, np. access\$000 — wszystko zależy od kompilatora).

Metodę tę wywołują metody klasy wewnętrznej. Instrukcja:

```
if (beep)
```

w metodzie actionPerformed klasy TimePrinter wykonuje następujące wywołanie:

```
if (access$0(outer));
```

Czy nie stanowi to zagrożenia bezpieczeństwa? Niestety tak. Wywołanie metody access\$0 w celu odczytania wartości prywatnego pola beep nie jest trudne. Oczywiście nazwa access\$0 nie jest dozwoloną nazwą dla metody w Javie, ale haker znający dobrze strukturę plików klas może z łatwością utworzyć taki plik zawierający instrukcje maszyny wirtualnej wywołujące tę metodę. Można do tego celu użyć na przykład edytora heksadecymalnego. Ponieważ ukryte metody dostępu są widoczne w obrębie pakietu, kod hakera musiałby zostać umieszczony w tym samym pakiecie co atakowana klasa.

Podsumowując, jeśli klasa wewnętrzna ma dostęp do prywatnego pola danych, to także inne klasy dodane do pakietu klasy zewnętrznej będą miały do niego dostęp. Zrobienie tego wymaga jednak determinacji i wiedzy. Programista nie może przypadkowo uzyskać dostępu. Musi w tym celu utworzyć lub zmodyfikować specjalny plik klasy.

6.4.4. Lokalne klasy wewnętrzne

Analizując uważnie kod klasy TalkingClock, można spostrzec, że nazwa typu TimePrinter jest potrzebna tylko jeden raz — przy tworzeniu obiektu tego typu w metodzie start.

W takiej sytuacji można zdefiniować klasę **lokalnie w obrębie jednej metody**.



Syntetyzowane konstruktory i metody bywają nieco zawiłe. Wyobraźmy sobie, że klasa TimePrinter jest prywatną klasą wewnętrzną. W maszynie wirtualnej klasy prywatne nie istnieją, a więc kompilator — radząc sobie najlepiej, jak może — tworzy klasę o zasięgu pakietowym z prywatnym konstruktorem:

```
private TalkingClock$TimePrinter(TalkingClock);
```

Oczywiście tego konstruktora nie da się wywołać, dlatego istnieje jeszcze jeden o zasięgu pakietowym:

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1)
```

Niniejszy konstruktor wywołuje ten pierwszy z wymienionych.

Wywołanie tego konstruktora w metodzie start klasy TalkingClock jest konwertowane przez kompilator na poniższy kod:

```
new TalkingClock$TimePrinter(this, null)
```

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
}
```

W definicjach klas lokalnych nie stosuje się specyfikatorów dostępu (public lub private). Nie ma do nich dostępu spoza bloku, w którym zostały zdefiniowane.

Wielką zaletą klas lokalnych jest to, że pozostają ukryte przed światem zewnętrznym. Nie ma do nich dostępu nawet kod otaczającej klasy. W omawianym przypadku tylko metoda start wie o istnieniu klasy TimePrinter.

6.4.5. Dostęp do zmiennych finalnych z metod zewnętrznych

Klasy lokalne mają jeszcze jedną cechę, która wyróżnia je na tle pozostałych klas wewnętrznych. Mają one dostęp nie tylko do pól klas je otaczających, ale również do zmiennych lokalnych. Zmienne te muszą być jednak finalne. Oto typowy przykład. Przeniesiemy parametry interval i beep z konstruktora klasy TalkingClock do metody start.

```
public void start(int interval, final boolean beep)
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
```

```
    {
        Date now = new Date();
        System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}

ActionListener listener = new TimePrinter();
Timer t = new Timer(interval, listener);
t.start();
}
```

Należy zauważyć, że klasa TalkingClock nie musi już zawierać pola beep. Odwołuje się do zmiennej parametrycznej beep metody start.

Może nie jest to zaskakujące. Wiersz:

```
if (beep) . . .
```

znajduje się w całości w metodzie start, więc dlaczego nie powinien mieć dostępu do wartości zmiennej beep?

Aby to sprawdzić, przeanalizujemy uważnie przepływ sterowania.

- 1 Następuje wywołanie metody start.
- 2 W wyniku wywołania konstruktora klasy wewnętrznej TimePrinter następuje inicjacja zmiennej obiektowej listener.
3. Referencja listener zostaje przekazana do konstruktora Timer, następuje uruchomienie zegara i zakończenie metody start. W tym momencie parametr beep metody start przestaje istnieć.
4. Chwilę później metoda actionPerformed wykonuje wiersz if(beep)....

Aby metoda actionPerformed działała, klasa TimePrinter musiała skopiować wartość pola beep jako zmienną lokalną metody start przed zniknięciem wartości parametru beep. Właśnie to się stało. W powyższym przykładzie kompilator tworzy nazwę TalkingClock\$1TimePrinter dla lokalnej klasy wewnętrznej. Analiza klasy TalkingClock\$1TimePrinter za pomocą programu ReflectionTest da następujący wynik:

```
class TalkingClock$1TimePrinter
{
    TalkingClock$1TimePrinter(TalkingClock, boolean);

    public void actionPerformed(java.awt.event.ActionEvent);

    final boolean val$beep;
    final TalkingClock this$0;
}
```

Zwróciły uwagę na parametr boolean w konstruktorze i zmienną obiektową val\$beep. W trakcie tworzenia obiektu wartość beep jest przekazywana do konstruktora i zapisywana w polu o nazwie val\$beep. Kompilator wykrywa obecność zmiennych lokalnych, tworzy odpowiadające im pola obiektowe oraz kopiuje te zmienne do konstruktora, po czym wykorzystuje je do inicjacji utworzonych pól.

Z punktu widzenia programisty taki dostęp do zmiennych lokalnych jest bardzo pożądanym rozwiązaniem. Umożliwia ono uproszczenie klas wewnętrznych dzięki redukcji liczby pól, które trzeba zadeklarować jawnie.

Wiemy już, że metody klas lokalnych mogą się odwoływać wyłącznie do finalnych zmiennych lokalnych. Dlatego parametr beep w omawianym przykładzie został zadeklarowany jako `final`. Zmienna lokalna ze słowem kluczowym `final` w deklaracji nie może być modyfikowana po tym, jak zostanie raz zainicjowana. Dzięki temu mamy gwarancję, że zmienna lokalna i jej kopia w klasie lokalnej mają zawsze tę samą wartość.



Wiemy już, że zmienne `finalne` mogą być używane w charakterze stałych:

```
public static final double SPEED_LIMIT = 90;
```

Słowo kluczowe `final` można stosować do zmiennych lokalnych, obiektowych i statycznych. Jego znaczenie jest zawsze takie samo: do danej zmiennej można przypisać wartość tylko **jeden raz**. Nie można jej później zmienić.

Nie ma jednak konieczności inicjowania zmiennej `finalnej` po jej zdefiniowaniu. Na przykład `finalna` zmienna parametryczna `beep` jest inicjowana po wywołaniu metody `start` (jeśli metoda jest wywoływana kilka razy, każde wywołanie tworzy własny parametr `beep`). Zmienna obiektowa `val$beep` dostępna w klasie wewnętrznej `TalkingClock$1TimePrinter` jest ustawiana jeden raz w konstruktorze klasy wewnętrznej. Zmienna `finalna`, która nie została zainicjowana, nazywa się **pustą zmienną finalną** (ang. *blank final variable*).

Ograniczenia wprowadzane przez słowo `final` bywają niewygodne. Wyobraźmy sobie, że chcemy aktualizować licznik w otaczającym go zakresie. W poniższym kodzie chcemy sprawdzić liczbę wywołań metody `compareTo` w trakcie sortowania.

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
{
    public int compareTo(Date other)
    {
        counter++; // błąd
        return super.compareTo(other);
    }
};
Arrays.sort(dates);
System.out.println(counter + " porównań.");
```

Zmienna `counter` nie może być zadeklarowana jako `finalna`, ponieważ musi być aktualizowana. Nie można jej zastąpić typem `Integer`, ponieważ obiekty tego typu są niezmienialne. Rozwiązanie polega na użyciu tablicy o rozmiarze 1:

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
{
    public int compareTo(Date other)
    {
        counter[0]++;
    }
};
```

```
        return super.compareTo(other);
    }
};
```

Zmienna tablicowa jest zadeklarowana jako finalna, ale to jedynie oznacza, że nie może się odwoływać do innej tablicy. Elementy tablicy można bez przeszkód zmieniać.

Kiedy klasy wewnętrzne zostały wynalezione, prototyp kompilatora automatycznie wykonywał transformację wszystkich zmiennych lokalnych, które były modyfikowane w klasie wewnętrznej. Jednak niektórzy programiści nie byli zadowoleni z tego, że kompilator za ich plecami tworzył obiekty na stercie. Dlatego zdecydowano się na wprowadzenie ograniczenia ze słowem kluczowym `final`. Niewykluczone, że w przyszłych wersjach Javy decyzja ta zostanie zmieniona.

6.4.6. Anonimowe klasy wewnętrzne

Używając lokalnych klas wewnętrznych, można pójść o krok dalej. Jeśli chcemy utworzyć tylko jeden obiekt jakiejś klasy, to nie trzeba nawet nadawać jej nazwy. Taka klasa nazywa się **anonimową klasą wewnętrzną** (ang. *anonymous inner class*).

```
public void start(int interval, final boolean beep)
{
    ActionListener listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    Timer t = new Timer(interval, listener);
    t.start();
}
```

Składnia zastosowana powyżej jest bez wątpienia bardzo enigmatyczna. Jej znaczenie jest następujące: „utworzenie nowego obiektu klasy implementującej interfejs `ActionListener`, którego wymagana metoda `actionPerformed` znajduje się w klamrach `{ }`”.

Ogólna składnia jest następująca:

```
new NadTyp(parametry konstrukcyjne)
{
    Metody i dane klasy wewnętrznej.
}
```

`NadTyp` w tym przypadku może być interfejsem, np. `ActionListener`. Klasa wewnętrzna implementuje wtedy ten interfejs. Jeśli natomiast `NadTyp` jest klasą, klasa wewnętrzna ją rozszerza.

Anonimowa klasa wewnętrzna nie może mieć konstruktora, ponieważ nazwa konstruktora musi być taka sama jak nazwa klasy, a tak przecież nie jest. W związku z tym parametry

konstrukcyjne podaje się konstruktorowi **nadklasy**. Zwłaszcza klasa wewnętrzna nie może mieć parametrów konstrukcyjnych, jeśli implementuje interfejs. Niemniej konieczne jest wpisanie nawiasów, jak poniżej:

```
new TypInterfejsowy()
{
    metody i dane
}
```

Aby zauważyc różnicę pomiędzy tworzeniem obiektu klasy a tworzeniem obiektu anonimowej klasy wewnętrznej rozszerzającej tę klasę, trzeba uważnie się przyjrzeć.

```
Person queen = new Person("Maria");
// Obiekt klasy Person.
Person count = new Person("Dracula") { . . . };
// Obiekt klasy wewnętrznej rozszerzającej klasę Person.
```

Jeśli po zamknięciu nawiasu zawierającego parametry konstrukcyjne znajduje się klamra otwierająca, oznacza to, że definiowana jest anonimowa klasa wewnętrzna.

Czy anonimowe klasy wewnętrzne są doskonałym pomysłem, czy idealnym sposobem na zaciemnienie kodu? Odpowiedź prawdopodobnie leży gdzieś pośrodku. Jeśli kod klasy wewnętrznej jest krótki i prosty, to może pozwolić programiście zaoszczędzić nieco pisania. Z drugiej strony właśnie takie oszczędzające czas cechy prowadzą do zwycięstwa w konkursie na najbardziej niejasny kod napisany w Javie.

Listing 6.7 przedstawia pełny kod źródłowy programu z zegarem, w którym użyto anonimowej klasy wewnętrznej. W porównaniu z programem z listingu 6.6 ta wersja jest znacznie krótsza, a przy odrobinie praktyki równie łatwa do zrozumienia.

Listing 6.7. anonymousInnerClass/AnonymousInnerClassTest.java

```
package anonymousInnerClass;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * Ten program demonstruje zastosowanie anonimowych klas wewnętrznych.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class AnonymousInnerClassTest
{
    public static void main(String[] args)
    {
        TalkingClock clock = new TalkingClock();
        clock.start(1000, true);

        // Niech program działa, dopóki użytkownik nie wciśnie przycisku OK.
        JOptionPane.showMessageDialog(null, "Zamknąć program?");
        System.exit(0);
    }
}
```

```

    }

}

/**
 * Zegar drukujący informacje o czasie w równych odstępach czasu.
 */
class TalkingClock
{
    /**
     * Tworzy obiekt TalkingClock.
     * @param interval odstęp czasu pomiędzy kolejnymi komunikatami (w milisekundach)
     * @param beep wartość true oznacza, że dźwięk ma być odtwarzany
     */
    public void start(int interval, final boolean beep)
    {
        ActionListener listener = new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                Date now = new Date();
                System.out.println("Kiedy usłyszysz dźwięk, będzie godzina " + now);
                if (beep) Toolkit.getDefaultToolkit().beep();
            }
        };
        Timer t = new Timer(interval, listener);
        t.start();
    }
}

```



Poniżej przedstawiona jest sztuczka o nazwie **inicjacja z podwójną klamrą** (ang. *double brace initialization*), w której wykorzystuje się składnię klas wewnętrznych. Przypuśćmy, że chcemy utworzyć listę tablicową i przekazać ją do metody:

```

ArrayList<String> friends = new ArrayList<>();
favorites.add("Henryk");
favorites.add("Tomasz");
invite(friends);

```

Jeśli lista nie będzie już więcej potrzebna, dobrze by było zdefiniować ją jako anonimową. Jak wówczas jednak dodać do niej elementy? Ano tak:

```
invite(new ArrayList<String>() {{ add("Henryk"); add("Tomasz"); }})
```

Zwrót uwagę na podwójne klamry. Zewnętrzne służą do utworzenia anonimowej podklasy klasy `ArrayList`, a wewnętrzne to blok konstrukcyjny obiektu (patrz rozdział 4.).



Często wygodnie jest utworzyć anonimową podkласę, która jest niemal identyczna jak jej nadklasa. Trzeba jednak wówczas uważać na metodę `equals`. W rozdziale 5. zalecamy, aby w metodzie tej używać następującego testu:

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

Anonimowa podklasa go nie przejdzie.



W danych dziennika albo debugera często zamieszcza się nazwę bieżącej klasy, np.:

```
System.out.println("Coś się stało w " + getClass());
```

Jednak w przypadku metody statycznej to się nie uda. Wywołanie getClass() jest tożsame z this.getClass(), a statyczne metody nie mają this. Dlatego w zamian należy użyć poniższego wyrażenia:

```
new Object(){}.getClass().getEnclosingClass() //pobiera klasę metody statycznej
```

Instrukcja new Object() {} tworzy anonimowy obiekt anonimowej podklasy klasy Object, a metoda getEnclosingClass pobiera jego klasę, czyli klasę zawierającą statyczną metodę.

6.4.7. Statyczne klasy wewnętrzne

Od czasu do czasu chcemy ukryć jedną klasę wewnętrz innej klasie, przy czym nie jest nam potrzebna referencja w klasie wewnętrznej do obiektów klasy zewnętrznej. Aby wyłączyć tworzenie tej referencji, klasę wewnętrzną należy zadeklarować jako statyczną.

Przeanalizujmy typową sytuację, w której wspomniana funkcjonalność może być potrzebna. Mamy znaleźć najmniejszą i największą wartość w tablicy. Oczywiście dla każdej z nich piszemy po jednej metodzie, w wyniku czego tablica jest przemierzana dwa razy. Kod byłby bardziej efektywny, gdyby obie wartości sprawdzały równocześnie, przechodząc przez tablice tylko jeden raz.

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
    if (min > v) min = v;
    if (max < v) max = v;
}
```

Jednak metoda musi zwrócić dwie liczby. Problem ten można rozwiązać, definiując klasę o nazwie Pair przechowującą dwie wartości:

```
class Pair
{
    private double first;
    private double second;

    public Pair(double f, double s)
    {
        first = f;
        second = s;
    }
    public double getFirst() { return first; }
    public double getSecond() { return second; }
}
```

Dzięki temu funkcja minmax może zwrócić obiekt typu Pair.

```
class ArrayAlg
{
    public static Pair minmax(double[] values)
    {
        .
        .
        return new Pair(min, max);
    }
}
```

Wywołującą tę funkcję sprawdza odpowiedzi za pomocą metod `getFirst` i `getSecond`:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Oczywiście słowo `Pair` jest bardzo często używane, przez co w dużym projekcie może się zdarzyć, że jakiś inny programista również wpadnie na doskonały pomysł utworzenia klasy o nazwie `Pair`, tylko że przechowującej na przykład dwa łańcuchy. Aby uniknąć potencjalnego konfliktu nazw, `Pair` można uczynić publiczną klasą wewnętrzną w klasie `ArrayAlg`. Wtedy klasa `Pair` na zewnątrz będzie znana pod nazwą `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

W tym jednak przypadku, w przeciwieństwie do poprzednich sytuacji, nie chcemy, aby obiekt `Pair` zawierał referencje do jakichkolwiek innych obiektów. Tworzenie referencji można wyłączyć, stosując słowo kluczowe `static` w deklaracji klasy wewnętrznej:

```
class ArrayAlg
{
    public static class Pair
    {
        .
        .
    }
}
```

Oczywiście tylko klasy wewnętrzne mogą być statyczne. Statyczna klasa wewnętrzna różni się od zwykłej klasy wewnętrznej tylko tym, że obiekt takiej klasy nie zawiera referencji do obiektu klasy zewnętrznej, który go wygenerował. W omawianym przykładzie użycie statycznej metody wewnętrznej było konieczne, ponieważ obiekt tej klasy jest tworzony wewnątrz metody statycznej:

```
public static Pair minmax(double[] d)
{
    .
    .
    return new Pair(min, max);
}
```

Gdyby klasa `Pair` nie była statyczna, kompilator zgłosiłby błąd polegający na braku niejawnego obiektu typu `ArrayAlg` do inicjacji obiektu klasy wewnętrznej.



Wewnętrznych klas statycznych należy używać zawsze wtedy, kiedy klasa wewnętrzna nie wymaga dostępu do obiektu klasy zewnętrznej. Niektórzy programiści statyczne klasy wewnętrzne nazywają **klasami zagnieżdzonymi**.



Klasy wewnętrzne deklarowane w interfejsach są automatycznie statyczne i publiczne.

Listing 6.8 przedstawia kompletny kod źródłowy klasy `ArrayAlg` i klasy zagnieżdżonej `Pair`.

Listing 6.8. staticInnerClass/StaticInnerClassTest.java

```
package staticInnerClass;

/**
 * Ten program demonstruje zastosowanie statycznych klas wewnętrznych.
 * @version 1.01 2004-02-27
 * @author Cay Horstmann
 */
public class StaticInnerClassTest
{
    public static void main(String[] args)
    {
        double[] d = new double[20];
        for (int i = 0; i < d.length; i++)
            d[i] = 100 * Math.random();
        ArrayAlg.Pair p = ArrayAlg.minmax(d);
        System.out.println("min = " + p.getFirst());
        System.out.println("max = " + p.getSecond());
    }
}

class ArrayAlg
{
    /**
     * Para liczb zmiennoprzecinkowych.
     */
    public static class Pair
    {
        private double first;
        private double second;

        /**
         * Tworzy parę dwóch liczb zmiennoprzecinkowych.
         * @param f pierwsza liczba
         * @param s druga liczba
         */
        public Pair(double f, double s)
        {
            first = f;
            second = s;
        }

        /**
         * Zwraca pierwszą liczbę z pary.
         * @return pierwsza liczba
         */
        public double getFirst()
        {
            return first;
        }
    }
}
```

```
 /**
 * Zwraca drugą liczbę z pary.
 * @return druga liczba
 */
public double getSecond()
{
    return second;
}

/**
 * Znajduje największą i najmniejszą wartość w tablicy.
 * @param values tablica liczb zmiennoprzecinkowych
 * @return para liczb, w której pierwsza liczba określa wartość najmniejszą, a druga
 * największą
 */
public static Pair minmax(double[] values)
{
    double min = Double.MAX_VALUE;
    double max = Double.MIN_VALUE;
    for (double v : values)
    {
        if (min > v) min = v;
        if (max < v) max = v;
    }
    return new Pair(min, max);
}
```

6.5. Klasy proxy

Ostatnią część tego rozdziału poświęciliśmy **klasom proxy**, które zostały wprowadzone w Java SE 1.3. Ich zastosowanie sprawdza się do tworzenia w trakcie działania programu nowych klas implementujących określone interfejsy. Klasy proxy są potrzebne tylko wtedy, gdy w czasie komplikacji nie wiadomo jeszcze, które interfejsy dana klasa ma implementować. Sytuacje takie zdarzają się rzadko, a więc osoby niezainteresowane zaawansowanymi technikami mogą tę część książki pominąć. Jednak w pewnego typu aplikacjach systemowych swoboda oferowana przez klasy pośredniczące może się okazać niezwykle pomocna.

Wyobraźmy sobie, że chcemy utworzyć obiekt klasy implementującej jeden lub więcej interfejsów, których dokładnej charakterystyki w czasie komplikacji jeszcze nie znamy. Jest to trudny do rozwiązania problem. Do utworzenia samej klasy można użyć metody `newInstance` lub wykorzystać refleksję w celu znalezienia konstruktora. Nie można jednak utworzyć obiektu interfejsu. Trzeba utworzyć nową klasę w działającym programie.

W niektórych programach rozwiązanie tego problemu polega na wygenerowaniu kodu, zapisaniu go w pliku, wywołaniu kompilatora i załadowaniu powstałego pliku klasy. Metoda ta jest niestety powolna oraz wymaga współpracy programu z kompilatorem. Lepszym rozwiązaniem jest technika **klas proxy**. Klasa proxy może tworzyć nowe klasy w czasie działania programu.

Implementuje określone przez programistę interfejsy. Klasa proxy zawiera następujące metody:

- wszystkie metody wymagane przez określone interfejsy;
- wszystkie metody zdefiniowane w klasie Object (toString, equals itd.).

Ale przecież nie można w czasie działania programu napisać kodu dla nowych klas. W zamian trzeba dostarczyć **obiekt obsługujący wywołanie**, czyli tzw. **invocation handler**. Jest to obiekt dowolnej klasy, która implementuje interfejs InvocationHandler. Zawiera on tylko jedną metodę:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Kiedy wywoływana jest jakaś metoda na rzecz obiektu klasy proxy, następuje wywołanie metody invoke obiektu obsługującego wywołanie przy użyciu obiektu klasy Method i parametrów oryginalnego wywołania. Obiekt obsługujący wywołanie musi się dowiedzieć, jak obsłużyć wywołanie.

Aby utworzyć obiekt klasy proxy, należy użyć metody newProxyInstance klasy Proxy. Niniejsza metoda pobiera trzy parametry:

- **Mechanizm ładowania klas** (ang. *class loader*). Zgodnie z modelem zabezpieczeń Javy dozwolone jest używanie różnych mechanizmów ładowania klas systemowych, klas pobranych z internetu itd. Więcej na temat tzw. loaderów klas piszemy w rozdziale 9. w drugim tomie. Obecnie w miejsce tego parametru wstawiamy null, aby użyć domyślnego loadera.
- Tablica obiektów klasy Class — po jednym dla każdego interfejsu, który ma być zaimplementowany.
- Obiekt obsługujący wywołanie.

Dwa pytania pozostają bez odpowiedzi. Jak zdefiniować obiekt obsługujący wywołanie? Co można zrobić z powstałym obiektem proxy? Odpowiedź na powyższe pytania zależy oczywiście od problemu, który chcemy rozwiązać za pomocą mechanizmu klas proxy. Klasę te można wykorzystać do wielu celów, np.:

- przesyłanie wywołań metod do zdalnych serwerów;
- łączenie zdarzeń interfejsu użytkownika z akcjami w działającym programie;
- śledzenie wywołań metod w celu ułatwienia lokalizacji błędów.

W przykładowym programie użyjemy klas proxy i obiektów obsługi wywołań do śledzenia wywołań metod. Zdefiniujemy klasę osłonową TraceHandler przechowującą opakowany obiekt. Jej metoda invoke drukuje nazwę i parametry metody, która ma być wywołana, a następnie wywołuje tę metodę przy użyciu opakowanego obiektu jako parametru niejawnego.

```
class TraceHandler implements InvocationHandler
{
    private Object target;

    public TraceHandler(Object t)
    {
```

```

        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        // Wydruk nazwy i parametrów metody.

        // Wywołanie metody.
        return m.invoke(target, args);
    }
}

```

Poniższy kod przedstawia sposób tworzenia obiektu proxy, który jest odpowiedzialny za śledzenie metod:

```

Object value = . . .;
// Tworzenie osłony.
InvocationHandler handler = new TraceHandler(value);
// Tworzenie obiektu proxy dla jednego lub większej liczby interfejsów.
Class[] interfaces = new Class[] { Comparable.class };
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);

```

Dzięki temu, jeśli metoda któregoś z interfejsów zostanie wywołana na rzecz obiektu proxy, zostanie wydrukowana jej nazwa i parametry oraz nastąpi wywołanie tej metody na rzecz obiektu value.

W programie na listingu 6.9 obiekty proxy są wykorzystywane do śledzenia wyszukiwania binarnego. Zapełniamy tablicę obiektami proxy liczb całkowitych od 1 do 1000. Następnie za pomocą metody binarySearch klasy Arrays w tablicy tej wyszukujemy jedną losową liczbę. Na koniec wyświetlamy pasujący element.

```

Object[] elements = new Object[1000];
// Zapełnienie tablicy obiektami proxy liczb całkowitych od 1 do 1000.
for (int i = 0; i < elements.length; i++)
{
    Integer value = i + 1;
    elements[i] = Proxy.newInstance(. . .); // obiekt proxy wartości
}
// Tworzenie losowej liczby całkowitej.
Integer key = new Random().nextInt(elements.length) + 1;
// Wyszukiwanie losowej liczby całkowitej.
int result = Arrays.binarySearch(elements, key);
// Wydruk pasującej wartości, jeśli istnieje.
if (result >= 0) System.out.println(elements[result]);

```

Klasa Integer implementuje interfejs Comparable. Obiekty proxy należą do klasy, która jest definiowana w czasie działania programu (ma nazwę typu \$Proxy0). Ta klasa również implementuje interfejs Comparable, jednak jej metoda compareTo wywołuje metodę invoke handlera obiektu proxy.



Jak widzieliśmy wcześniej w tym rozdziale, klasa Integer implementuje interfejs Comparable<Integer>. Jednak w czasie działania programu wszystkie typy parametryzowane są czyszczone i tworzony jest obiekt proxy przy użyciu obiektu class srodkowej klasy Comparable.

Metoda `binarySearch` wykonuje wywołania podobne do poniższego:

```
if (elements[i].compareTo(key) < 0) . . .
```

Ponieważ tablica została zapelniona obiektami proxy, metoda `compareTo` wywołuje metodę `invoke` klasy `TraceHandler`. Ta z kolei metoda drukuje nazwę i parametry metody, a następnie wywołuje metodę `compareTo` na rzecz opakowanego obiektu `Integer`.

Na końcu programu znajduje się następujący wiersz kodu:

```
System.out.println(elements[result]);
```

Listing 6.9. proxy/ProxyTest.java

```
package proxy;

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

/**
 * Ten program demonstruje użycie klas proxy.
 * @version 1.00 2000-04-13
 * @author Cay Horstmann
 */
public class ProxyTest
{
    public static void main(String[] args)
    {
        Object[] elements = new Object[1000];

        // Wstawienie do tablicy obiektów proxy liczb całkowitych z przedziału 1 – 1000.
        for (int i = 0; i < elements.length; i++)
        {
            Integer value = i + 1;
            InvocationHandler handler = new TraceHandler(value);
            Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class }, 
                handler);
            elements[i] = proxy;
        }

        // Tworzenie losowej liczby całkowitej.
        Integer key = new Random().nextInt(elements.length) + 1;

        // Szukanie liczby.
        int result = Arrays.binarySearch(elements, key);

        // Drukowanie dopasowanej wartości, jeśli zostanie znaleziona.
        if (result >= 0) System.out.println(elements[result]);
    }
}

/**
 * Obiekt obsługujący wywołanie, który drukuje nazwę metody i parametry, a następnie
 * wywołuje oryginalną metodę.
 */
class TraceHandler implements InvocationHandler
{
    private Object target;
```

```
/***
 * Tworzy obiekt TraceHandler.
 * @param t parametr niejawnny wywołania metody
 */
public TraceHandler(Object t)
{
    target = t;
}

public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
{
    // Drukowanie argumentu niejawnego.
    System.out.print(target);
    // Drukowanie nazwy metody.
    System.out.print("." + m.getName() + "(");
    // Drukowanie argumentów jawnych.
    if (args != null)
    {
        for (int i = 0; i < args.length; i++)
        {
            System.out.print(args[i]);
            if (i < args.length - 1) System.out.print(", ");
        }
    }
    System.out.println(")");
    // Wywołanie rzeczywistej metody.
    return m.invoke(target, args);
}
}
```

Metoda `println` wywołuje metodę `toString` na rzecz obiektu proxy. Wywołanie to również jest przekierowywane do obiektu obsługującego wywołanie.

Wynik działania programu jest następujący:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

Można zaobserwować, jak algorytm wyszukiwania binarnego namierza liczbę, dzieląc przeszukiwany obszar na dwie połowy na każdym etapie. Zauważmy, że metoda `toString` jest w obiekcie proxy, mimo iż nie należy do interfejsu `Comparable` — w kolejnym podrozdziale dowiemy się, że niektóre metody klasy `Object` są zawsze w obiektach proxy.

6.5.1. Właściwości klas proxy

Skoro wiemy już, jak działają klasy proxy, przeanalizujemy ich właściwości. Nie zapominajmy, że klasy te są tworzone w czasie działania programu, a po utworzeniu stają się zwykłymi klasami, tak jak wszystkie inne klasy działające w maszynie wirtualnej.

Każda klasa proxy rozszerza klasę `Proxy`. Klasa proxy ma tylko jedno pole obiektowe — obiekt obsługujący wywołanie, które jest zdefiniowane w nadklasie `Proxy`. Wszystkie dodatkowe dane potrzebne do przeprowadzenia działań obiektu proxy muszą być zapisane w obiekcie obsługi wywołania. Na przykład gdy obiekty `Comparable` uczyniliśmy obiektami proxy w programie na listingu 6.9, obiekt `TraceHandler` opakowywał rzeczywiste obiekty.

Wszystkie klasy proxy przesyłają metody `toString`, `equals` i `hashCode` klasy `Object`. Tak jak wszystkie metody klas proxy, wywołują one tylko metodę `invoke` obiektu obsługującego wywołanie. Pozostałe metody klasy `Object` (np. `clone` i `getClass`) nie są przedefiniowywane.

Nazwy klas proxy nie są zdefiniowane. Klasa `Proxy` w maszynie wirtualnej firmy Sun generuje nazwy zaczynające się od przedrostka `$Proxy`.

Każdy mechanizm ładujący klasę i uporządkowany zestaw interfejsów ma tylko jedną klasę proxy. Jeśli zatem wywołamy metodę `newProxyInstance` dwa razy przy użyciu tego samego mechanizmu ładowania klas i tablicy interfejsów, otrzymamy dwa obiekty tej samej klasy. Klasę tę można także uzyskać za pomocą metody `getProxyClass`:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

Klasy proxy są zawsze publiczne i finalne. Jeśli wszystkie interfejsy implementowane przez klasę proxy są publiczne, klasa taka nie należy do żadnego pakietu. W przeciwnym przypadku wszystkie niepubliczne interfejsy muszą należeć do tego samego pakietu. Wtedy klasa proxy również przynależy do tego pakietu.

Aby sprawdzić, czy określony obiekt `Class` reprezentuje klasę proxy, należy wywołać metodę `isProxyClass` klasy `Proxy`.

java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`

W definicji tej metody należy umieścić działania, które mają być wykonane przy każdym wywołaniu metody na rzecz obiektu proxy.

java.lang.reflect.Proxy 1.3

- `static Class getProxyClass(ClassLoader loader, Class[] interfaces)`

Zwraca klasę proxy, która implementuje dane interfejsy.

- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

Tworzy nowy egzemplarz klasy proxy, która implementuje dane interfejsy. Wszystkie metody wywołują metodę `invoke` danego obiektu obsługi.

- static boolean isProxyClass(Class c)

Zwraca wartość true, jeśli c jest klasą proxy.

Na tym kończy się ostatni rozdział dotyczący podstaw języka Java. Z interfejsami i klasami wewnętrznymi będziemy się spotykać bardzo często. Natomiast klasy proxy są zaawansowaną techniką, która leży w sferze zainteresowań przede wszystkim twórców narzędzi, a nie programistów aplikacji. W rozdziale 7. zaczynamy naukę programowania grafiki i interfejsów użytkownika.

7

Grafika

W tym rozdziale:

- Wprowadzenie do biblioteki Swing
- Tworzenie ramek
- Pozycjonowanie ramek
- Wyświetlanie informacji w komponencie
- Figury 2D
- Kolory
- Kroje czcionek
- Wyświetlanie obrazów

Do tej pory tworzyliśmy tylko programy, które pobierały dane z klawiatury, przetwarzają je i wyświetlały wyniki w konsoli. Większość użytkowników oczekuje jednak nieco więcej. Ani nowoczesne programy, ani strony internetowe nie działają w ten sposób. W tym rozdziale zaczynamy naukę pisania programów z graficznym interfejsem użytkownika (GUI). Nauczymy się przede wszystkim ustawiać rozmiar i położenie okien na ekranie, wyświetlać tekst pisany różnymi krojami czcionki, wyświetlać obrazy itd. Cały zdobyty tu warsztat przyda nam się w kolejnych rozdziałach, w których będziemy tworzyć ciekawe projekty.

Dwa następne rozdziały opisują przetwarzanie zdarzeń, takie jak wciśnięcie klawisza na klawiaturze lub kliknięcie przyciskiem myszy, oraz dodawanie do aplikacji takich elementów interfejsu jak menu i przyciski. Po zapoznaniu się z tymi trzema rozdziałami będziesz dysponować wiedzą, która jest niezbędna do pisania aplikacji graficznych. Bardziej zaawansowane techniki związane z programowaniem grafiki zostały opisane w drugim tomie.

Osoby zainteresowane wyłącznie programowaniem po stronie serwera, które nie mają w planach pisania GUI, mogą bezpiecznie pominąć te rozdziały.

7.1. Wprowadzenie do pakietu Swing

W Java 1.0 udostępniono bibliotekę klas o nazwie Abstract Window Toolkit (AWT), która dostarczała podstawowe narzędzia do programowania GUI. Podstawowa biblioteka AWT przerzuca zadania dotyczące tworzenia elementów interfejsu oraz obsługi ich zachowań na natywne narzędzia GUI danej platformy (Windows, Solaris, Mac OS X itd.). Jeśli na przykład przy użyciu oryginalnej biblioteki AWT umieszczono w oknie pole tekstowe, dane wprowadzane do niego były w rzeczywistości obsługiwane przez odpowiednik tego pola w systemie. Dzięki temu program napisany w Javie mógł teoretycznie działać na dowolnej platformie, a jego styl był taki sam jak innych programów w danym systemie. Stąd wziął się slogan firmy Sun: „Napisz raz, uruchamiaj wszędzie”.

Metoda programowania oparta na odpowiednikach sprawdzała się w przypadku prostych aplikacji. Natomiast szybko wyszło na jaw, że napisanie wysokiej jakości przenośnej biblioteki graficznej opartej na natywnych elementach interfejsu użytkownika jest niezwykle trudnym zadaniem. Elementy interfejsu, jak menu, paski przewijania i pola tekstowe, mogą się nieco różnić na różnych platformach. Przez to trudno było stworzyć program działający identycznie w różnych systemach. Ponadto niektóre środowiska graficzne (jak np. X11/Motif) nie dysponują tak szeroką gamą elementów interfejsu jak systemy Windows czy Mac OS X. Ten fakt ograniczał zasobność przenośnej biblioteki do tych elementów, które można znaleźć na wszystkich platformach. W wyniku tego aplikacje GUI budowane na bazie AWT ustępowały wyglądem i funkcjonalnością natywnym aplikacjom takich systemów jak Windows czy Mac OS X. Na domiar złego na różnych platformach biblioteka AWT zawierała różne błędy. Programiści narzekali, że muszą testować swoje aplikacje na wszystkich platformach, co złośliwie nazywano „napisz raz, testuj wszędzie”.

W 1996 roku firma Netscape stworzyła bibliotekę GUI o nazwie IFC (ang. *Internet Foundation Classes*), w której zastosowano zupełnie inne podejście. Elementy interfejsu użytkownika, jak przyciski, menu itd., **były rysowane** w pustym oknie. Rola systemu polegała tylko na wyświetlaniu okien i rysowaniu w nich. Dzięki temu widgety firmy Netscape wyglądały i działały zawsze tak samo, bez względu na platformę. Firma Sun podjęła współpracę z Netscape w celu udoskonalenia tej technologii, czego owocem była biblioteka o nazwie kodoowej Swing. Biblioteka ta została udostępniona w postaci dodatku w Java 1.1, a do biblioteki standardowej wcielono ją w Java SE 1.2.

Zgodnie z myślą Duke'a Ellingtona, że „Nic nie ma znaczenia, jeśli jest pozbawione swingu”, Swing stał się oficjalną nazwą zestawu narzędzi do tworzenia GUI, niewykorzystującego systemowych odpowiedników elementów. Swing wchodzi w skład Java Foundation Classes (JFC). Biblioteka JFC jest bardzo duża i zawiera wiele innych narzędzi poza Swingiem. Zaliczają się do nich interfejsy API dostępności, grafiki dwuwymiarowej oraz obsługi operacji przeciągania i upuszczania.

Oczywiście elementy interfejsu oparte na Swingu pojawiają się z pewnym opóźnieniem w stosunku do komponentów używanych przez AWT. Z naszego doświadczenia wynika, że różnica ta nie powinna stanowić problemu na żadnym w miarę niezbyt starym sprzęcie. Z drugiej strony argumenty przemawiające za użyciem Swinga są przytaczające:

- Swing udostępnia bogaty i wygodny w użyciu zestaw elementów interfejsu użytkownika.



Biblioteka Swing nie zastąpiła AWT, tylko została zbudowana w oparciu o architekturę swojej poprzedniczki. Komponenty Swing oferują znacznie większe możliwości. Używając biblioteki Swing, zawsze korzysta się z podstawowej biblioteki AWT, zwłaszcza przy obsłudze zdarzeń. Od tej pory pisząc „Swing”, mamy na myśli klasy rysujące interfejs użytkownika, a pisząc „AWT”, myślimy o podstawowych mechanizmach okien, takich jak obsługa zdarzeń.

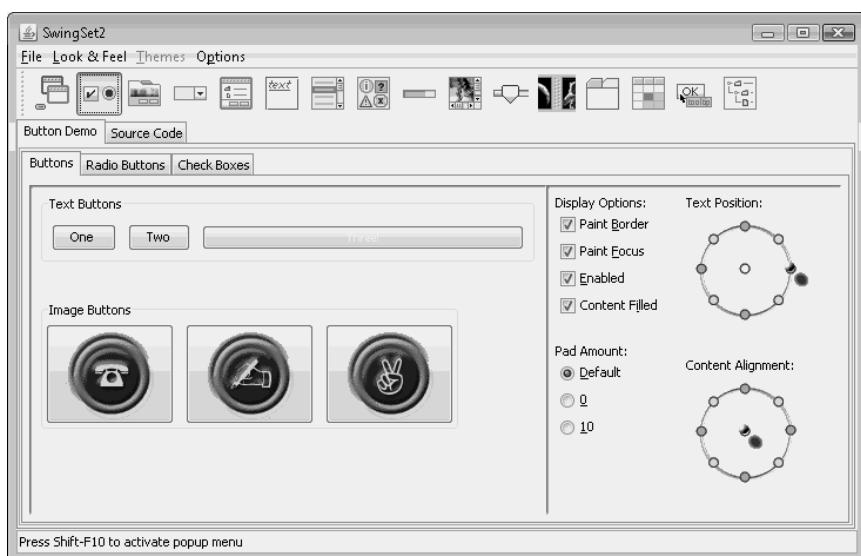
- Swing w niewielkim stopniu zależy od platformy, dzięki czemu jest mniej podatny na błędy związane z danym systemem.
- Sposób działania i wygląd elementów Swinga jest taki sam na różnych platformach.

Niemniej trzecia z wymienionych zalet może być uważana za wadę — jeśli elementy interfejsu użytkownika wyglądają tak samo na wszystkich platformach, to znaczy, że wyglądają **inaczej** niż elementy natywne, co z kolei oznacza, że będą słabiej znane użytkownikom.

W pakiecie Swing problem ten został rozwiązyany w bardzo elegancki sposób. Programista piszący program przy użyciu klas Swing może nadać mu specyficzny charakter. Rysunki 7.1 i 7.2 przedstawiają ten sam program w stylu systemu Windows i GTK.

Rysunek 7.1.

Styl systemu Windows

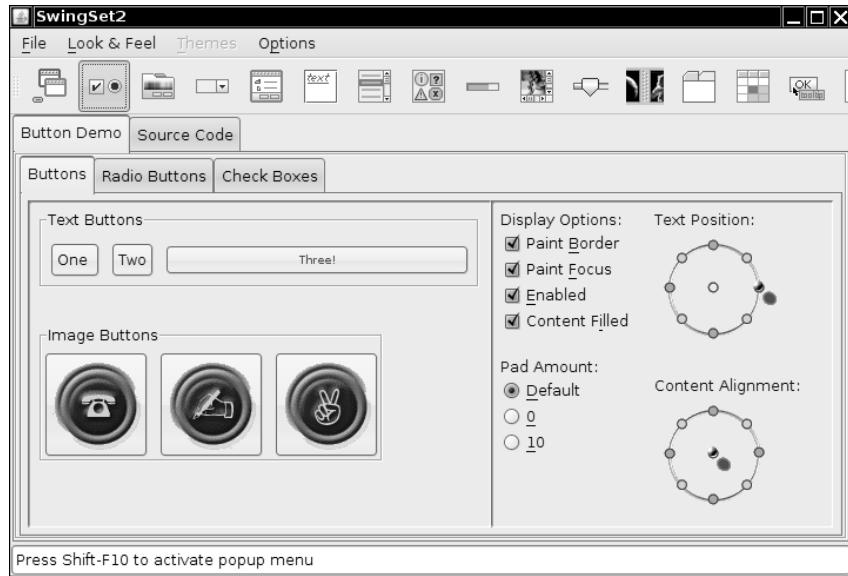


Dodatkowo firma Sun opracowała niezależny od platformy styl o nazwie **Metal**, który później przez specjalistów od marketingu przechrzczono na **Java look and feel**. Jednak większość programistów nadal używa określenia „Metal” i my również trzymamy się tej konwencji w tej książce.

Ze względu na krytyczne głosy kierowane pod adresem stylu Metal, który według niektórych wydawał się zbyt ciężki, w Java SE 5.0 nieco go odświeżono (zobacz rysunek 7.3). Obecnie styl Metal obsługuje wiele motywów — różnych wersji kolorystycznych i zestawów czcionek. Motyw domyślny nazywa się Ocean.

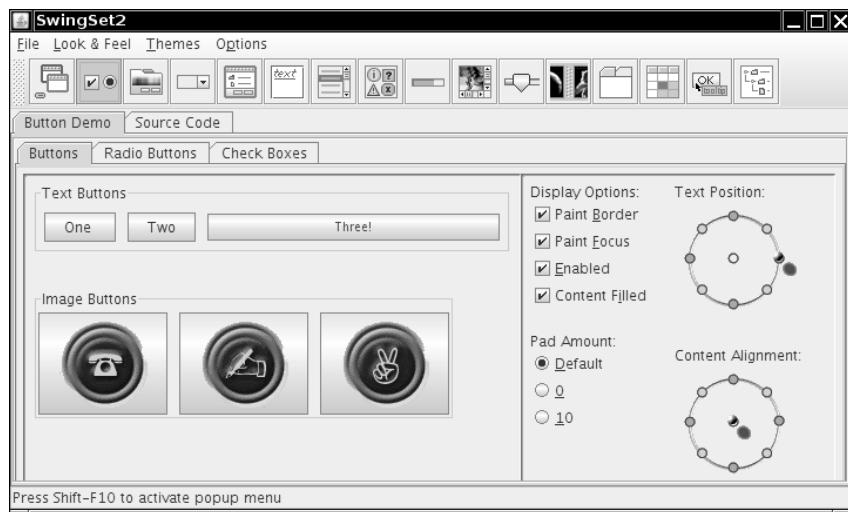
Rysunek 7.2.

Styl GTK



Rysunek 7.3.

Motyw Ocean stylu Metal



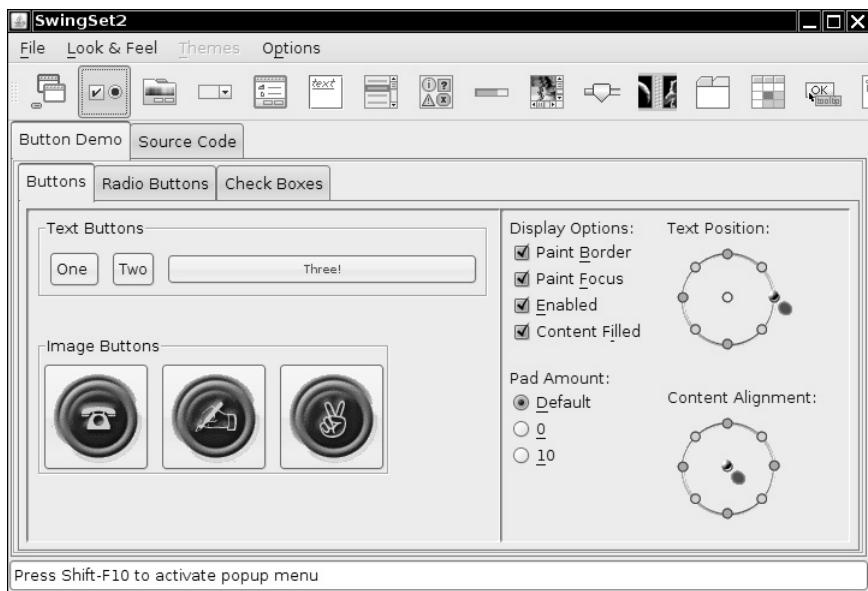
W Java SE 6 poprawiono obsługę natywnego stylu systemu Windows i GTK. Aktualnie aplikacje Swing obsługują schematy kolorów i pulsujące przyciski oraz paski przewijania, które stały się ostatnio modne.

W Java 7 dodano nowy styl o nazwie Nimbus (rysunek 7.4), ale nie jest on domyślnie dostępny. W stylu tym wykorzystywana jest grafika wektorowa zamiast bitmapowej, dzięki czemu interfejsy tworzone przy jego użyciu są niezależne od rozmiaru ekranu.

Niektórzy użytkownicy wolą, aby aplikacje w Javie wyglądały tak jak inne programy na danej platformie, inni wolą styl Metal, a jeszcze inni preferują styl całkiem innego producenta. Jak przekonamy się w rozdziale 8., umożliwienie użytkownikom wyboru dowolnego stylu jest bardzo łatwe.

Rysunek 7.4.

Styl Nimbus



W Javie możliwe jest rozszerzenie istniejącego stylu, a nawet utworzenie całkiem nowego. W tej książce nie mamy wystarczająco dużo miejsca, aby opisać ten żmudny proces polegający na określeniu sposobu rysowania każdego komponentu. Jednak niektórzy programiści pokusili się o to, zwłaszcza ci, którzy przenosili programy w Javie na nietypowe platformy, jak terminale sklepowe czy urządzenia kieszonkowe. Zbiór ciekawych stylów można znaleźć pod adresem <http://www.javootoo.com/>.

W Java SE 5.0 wprowadzono styl o nazwie Synth, który upraszcza cały ten proces. W Synth styl można definiować poprzez dostarczenie obrazów i deskryptorów XML. Nie trzeba nic programować.

Styl Napkin (<http://napkinlaf.sourceforge.net>) nadaje elementom interfejsu użytkownika wygląd przypominający odręczne rysowanie. Wysyłając do klienta utworzony w nim prototyp, dajemy wyraźny sygnał, że nie jest to jeszcze ukończony produkt.

Programowanie interfejsu użytkownika w Javie opiera się obecnie w większości na pakiecie Swing. Jest tylko jeden godny uwagi wyjątek. Środowisko zintegrowane Eclipse używa zestawu narzędzi graficznych o nazwie SWT, który podobnie jak AWT odzwierciedla natywne komponenty na różnych platformach. Artykuły na temat SWT można znaleźć na stronie <http://www.eclipse.org/articles/>.

Firma Oracle pracuje nad alternatywną technologią o nazwie JavaFX, która może w przyszłości zastąpić Swing. Jeśli chcesz dowiedzieć się o niej więcej, zajrzyj na stronę <http://www.oracle.com/technetwork/java/javafx/overview>.

Każdy, kto pisał programy dla systemu Microsoft Windows w językach Visual Basic lub C#, wie, jak dużym ułatwieniem są graficzne narzędzia do projektowania układu i edytory zasobów. Narzędzia te umożliwiają zaprojektowanie całej wizualnej strony aplikacji oraz automatycznie generują większość (często całość) kodu GUI. Dla Javy również dostępne są narzędzia

wspomagające budowanie GUI, ale naszym zdaniem, aby się nimi sprawnie posługiwać, trzeba najpierw nauczyć się robić to samodzielnie. Pozostała część tego rozdziału została poświęcona opisowi technik wyświetlania okien i rysowania w nich różnych elementów.

7.2. Tworzenie ramki

Okno najwyższego poziomu, tzn. takie, które nie jest zawarte w żadnym innym oknie, nazywa się w Javie **ramką** (ang. *frame*). W bibliotece AWT dla tego typu okien utworzono klasę o nazwie `Frame`. Wersja Swing tej klasy nosi nazwę `JFrame` i ją rozszerza. Ramka `JFrame` jest jednym z niewielu komponentów Swinga, które nie są rysowane w obszarze roboczym. W związku z tym elementy dodatkowe (przyciski, pasek tytułu, ikony itd.) są rysowane przez system operacyjny, a nie klasy Swing.



Nazwy większości klas komponentów Swing zaczynają się od litery `J`, np. `JButton`, `JFrame` itd. Istnieją także klasy `Button` i `Frame`, ale są to komponenty AWT. Jeśli litera `J` zostanie przypadkowo pominięta, program może przejść komplikację bez problemu, ale mieszania komponentów AWT i Swing może spowodować nietypowe zachowania lub wygląd aplikacji.

W tym podrozdziale przejrzymy najpopularniejsze metody pracy z komponentem Swing o nazwie `JFrame`. Listing 7.1 przedstawia prosty program wyświetlający na ekranie pustą ramkę widoczną na rysunku 7.5.

Listing 7.1. simpleFrame/SimpleFrameTest.java

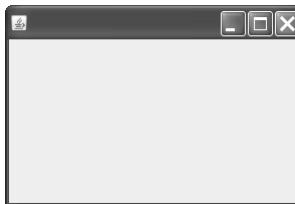
```
package simpleFrame;

import java.awt.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class SimpleFrameTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                SimpleFrame frame = new SimpleFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

Rysunek 7.5.

Najprostsza
widoczna ramka



```
class SimpleFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public SimpleFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```

Przeanalizujemy powyższy program wiersz po wierszu.

Klasy Swing znajdują się w pakiecie `javax.swing`. Nazwa pakietu `javax` oznacza, że jest to pakiet rozszerzający Javy, a nie podstawowy. Swing jest uznawany za rozszerzenie ze względów historycznych. Jest jednak dostępny w każdej wersji Java SE od 1.2.

Domyślny rozmiar ramki 0×0 jest raczej mało atrakcyjny. Zdefiniowaliśmy podklasę o nazwie `SimpleFrame`, której konstruktor ustawia rozmiar na 300×200 pikseli. Jest to jedyna różnica pomiędzy klasami `SimpleFrame` i `JFrame`.

W metodzie `main` klasy `SimpleFrameTest` tworzony jest obiekt klasy `SimpleFrame`, który następnie został uwidoczniony.

W każdym programie opartym na Swingu trzeba poradzić sobie z dwoma zagadnieniami technicznymi.

Po pierwsze, konfiguracja każdego komponentu Swing musi się odbywać w **wątku dystrybucji zdarzeń** (ang. *event dispatch thread*), który steruje wysyłaniem zdarzeń, jak kliknięcia przyciskiem myszy lub wcisnięcie klawisza na klawiaturze, do elementów interfejsu użytkownika. Poniższy fragment programu wykonuje instrukcje w wątku dystrybucji zdarzeń:

```
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        instrukcje
    }
});
```

Szczegółowy opis tego zagadnienia znajduje się w rozdziale 14. Na razie potraktujmy to jako magiczny fragment kodu potrzebny do uruchomienia programu Swing.



Wiele programów nie inicjuje interfejsu użytkownika w wątku dystrybucji zdarzeń. Nie ma żadnych przeszkód, aby operację te przeprowadzać w wątku głównym. Niestety ze względu na to, że komponenty Swing stawały się coraz bardziej złożone, programiści w firmie Sun nie mogli zagwarantować bezpieczeństwa tej metody. Prawdopodobieństwo wystąpienia błędu jest niezwykle niskie, ale nikt nie chciałby być tym pechowcem, któremu się to przydarzy. Lepiej pisać właściwy kod, nawet jeśli wygląda nieco tajemniczo.

Po drugie, określamy, co ma się stać, kiedy użytkownik zamknie ramkę aplikacji. W tym przypadku chcemy, aby program został zamknięty. Odpowiedzialna jest za to poniższa instrukcja:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

W programach składających się z wielu ramek program nie powinien kończyć działania w wyniku zamknięcia jednej z nich. Przy standardowych ustawieniach, jeśli użytkownik zamknie ramkę, zostanie ona ukryta, a program nie zakończy działania (dobrze by było, gdyby program był wyłączany w chwili zamknięcia jego **ostatniej** ramki, ale Swing działa inaczej).

Samo utworzenie ramki nie oznacza, że zostanie ona wyświetlona. Ramki na początku swojego istnienia są niewidoczne. Dzięki temu programista może dodać do nich wszystkie komponenty, zanim ukażą się po raz pierwszy. Aby wyświetlić ramkę, metoda `main` wywołuje na jej rzecz metodę `setVisible`.



Przed Java SE 5.0 można było używać metody `show` dziedziczonej przez klasę `JFrame` po nadklasie `Window`. Nadklassą klasy `Window` jest `Component`, która także zawiera metodę `show`. Stosowanie metody `Component.show` zaczęto odradzać w Java SE 1.2. W zamian, aby wyświetlić komponent, należy użyć metody `setVisible(true)`. Natomiast metoda `Window.show` **nie była** odradzana aż do Java SE 1.4. Możliwość uwidocznienia okna i przeniesienia go na przód była nawet przydatna. Niestety metoda `show` dla okien również jest odradzana od Java SE 5.0.

Po rozplanowaniu instrukcji inicjujących metoda `main` kończy działanie. Zauważmy, że zakończenie metody `main` nie oznacza zamknięcia programu, a jedynie głównego wątku. Wątek dystrybucji zdarzeń podtrzymuje działanie programu aż do jego zakończenia poprzez zamknięcie ramki lub wywołanie metody `System.exit`.

Uruchomiony program przedstawia rysunek 7.5 — jest to zwykłe szare okno najwyższego poziomu. Jak widać, pasek tytułu i pozostałe dodatki, jak zaokrąglone rogi służące do zmiany rozmiaru okna, zostały narysowane przez system operacyjny, a nie klasy Swing. Elementy te będą wyglądały inaczej, jeśli uruchomimy ten program w systemach Windows, GTK czy Mac OS X. Biblioteka Swing rysuje wszystko, co znajduje się wewnątrz ramki. W tym przypadku jej zadanie sprowadza się jedynie do wypełnienia domyślnym kolorem tła.



Od Java SE 1.4 istnieje możliwość wyłączenia wszelkich dodatków za pomocą wywołania metody `frame.setUndecorated(true)`.

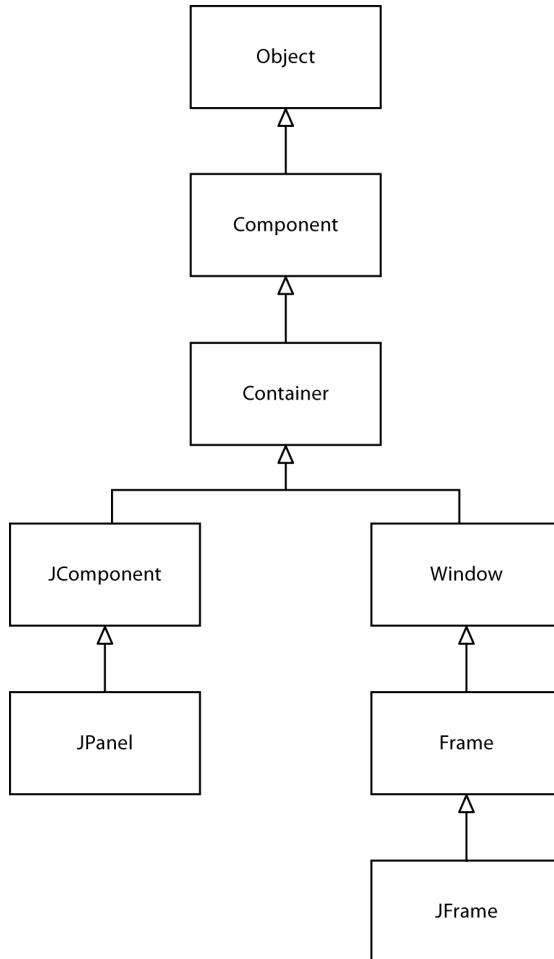
7.3. Pozycjonowanie ramki

Klasa `JFrame` udostępnia tylko kilka metod zmieniających wygląd ramek. Oczywiście większość metod służących do zmiany wymiarów i położenia ramki jest w klasie `JFrame` dziedziczona po różnych nadklasach. Poniżej znajduje się lista najważniejszych z tych metod:

- `setLocation` i `setBounds` — ustawiają położenie ramki.
- `setIconImage` — określa ikonę wyświetlaną w pasku tytułu, w zasobniku systemowym itd.
- `setTitle` — ustawia tekst w pasku tytułu.
- `setResizable` — pobiera wartość logiczną określającą, czy użytkownik może zmieniać rozmiar ramki.

Rysunek 7.6 przedstawia hierarchię dziedziczenia klasy `JFrame`.

Rysunek 7.6.
Hierarchia dziedziczenia klas ramek i komponentów w pakietach AWT i Swing





W wyciągach z API do tego podrozdziału przedstawiamy te metody, które naszym zdaniem mają największe znaczenie przy nadawaniu ramkom odpowiedniego stylu. Niektóre z nich są udostępnione w klasie JFrame. Inne z kolei pochodzą od różnych nadklas klasy JFrame. Czasami konieczne może być przeszukanie dokumentacji API w celu znalezienia metod przeznaczonych do określonego celu. Niestety jest to dość żmudna praca, jeśli chodzi o metody dziedziczone. Na przykład metoda toFront ma zastosowanie do obiektów typu JFrame, ale ponieważ jest dziedziczona po klasie Window, w dokumentacji JFrame nie ma jej opisu. Jeśli uważasz, że powinna istnieć metoda wykonująca określone działanie, ale nie ma jej w dokumentacji danej klasy, przeszukaj dokumentację metod nadklas tej klasy. Na górze każdej strony w dokumentacji API znajdują się odnośniki do nadklas, a lista metod dziedziczonych znajduje się pod zestawieniem nowych i przesłoniętych metod.

Według dokumentacji API metody służące do zmieniania rozmiaru i kształtu ramek znajdują się w klasach Component (będącej przodkiem wszystkich obiektów GUI) i Window (będącej nadkąską klasy Frame). Na przykład do zmiany położenia komponentu można użyć metody setLocation z klasy Component. Wywołanie:

```
setLocation(x, y)
```

umieści lewy górny róg komponentu w odległości x pikseli od lewej krawędzi ekranu i y pikseli od góry. Wartości (0, 0) oznaczają lewy górny róg ekranu. Podobnie metoda setBounds w klasie Component umożliwia zmianę rozmiaru i położenia komponentu (zwłaszcza ramki JFrame) w jednym wywołaniu:

```
setBounds(x, y, width, height)
```

Istnieje też możliwość pozostawienia decyzji o położeniu okna systemowi. Wywołanie:

```
setLocationByPlatform(true);
```

przed wyświetleniem okna spowoduje, że system we własnym zakresie określi jego położenie (ale nie rozmiar). Zazwyczaj nowe okno jest wyświetlane z nieznacznym przesunięciem względem poprzedniego.



W przypadku ramki współrzędne metod setLocation i setBounds są względne do całego ekranu. W rozdziale 9. dowiemy się, że współrzędne komponentów znajdujących się w kontenerze odnoszą się do tego kontenera.

7.3.1. Własności ramek

Wiele metod klas komponentów występuje w parach get-set, jak poniższe metody klasy Frame:

```
public String getTitle()  
public void setTitle(String title)
```

Taka para metod typu get-set nazywa się **własnością** (ang. *property*). Własność ma nazwę i typ. Nazwa jest taka sama jak słowo uzyskane w wyniku opuszczenia członu get i zamianienia pierwszej litery powstałego słowa na małą. Na przykład klasa Frame ma właściwość o nazwie title i typie String.

Z założenia `title` jest własnością ramki. Ustawienie (`set`) niniejszej własności powoduje zmianę tytułu na ekranie użytkownika. Pobranie jej (`get`) zwraca wartość, która została wcześniej ustawiona.

Nie wiemy (i nie obchodzi nas to), jak klasa `Frame` implementuje tę własność. Prawdopodobnie wykorzystuje swój odpowiednik ramki do przechowywania tytułu. Możliwe, że ma następujące pole:

```
private String title; //nie jest wymagane dla własności
```

Jeśli klasa zawiera pasujące pole, nie wiemy (lub nie obchodzi nas to), jak metody dostępne i ustawiające są zaimplementowane. Prawdopodobnie po prostu odczytują i ustawiają dane pole. Możliwe, że robią jeszcze coś, np. powiadamiają system o każdej zmianie tytułu.

Jest tylko jeden wyjątek od konwencji `get-set`: metody własności typu logicznego zaczynają się od przedrostka `is`. Na przykład dwie przedstawione poniżej metody definiują własność `locationByPlatform`:

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

Znacznie więcej na temat własności piszemy w rozdziale 8. drugiego tomu.



Wiele języków programowania, zwłaszcza Visual Basic i C#, standardowo obsługuje własności. Niewykluczone, że w przyszłości w Javie również pojawi się podobna konstrukcja językowa.

7.3.2. Określanie rozmiaru ramki

Pamiętajmy, że jeśli nie ustawimy własnego rozmiaru ramek, wszystkie będą miały wymiary 0×0 pikseli. Aby nie komplikować naszych przykładowych programów, ustawiamy ramki na rozmiar do przyjęcia na większości ekranów. Jednak w profesjonalnej aplikacji należy sprawdzić rozdzielczość ekranu użytkownika i napisać podprogram zmieniający rozmiar ramek w zależności od potrzeby. Na przykład okno, które wygląda znakomicie na ekranie laptopa, na ekranie o wysokiej rozdzielczości skurczy się do rozmiarów znaczka pocztowego.

Aby sprawdzić rozmiar ekranu, należy wykonać następujące działania: wywołaj statyczną metodę `getDefaultToolkit` klasy `Toolkit` w celu utworzenia obiektu typu `Toolkit` (klasa `Toolkit` jest zbiornikiem rozmaitych metod, które współpracują z systemem). Następnie wywołaj metodę `getScreenSize`, która zwraca rozmiar ekranu w postaci obiektu `Dimension`. Obiekt tego typu przechowuje wysokość i szerokość w zmiennych publicznych (!) o nazwach `width` i `height`. Poniżej przedstawiamy opisywany fragment programu:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

Rozmiar ramki ustawiamy na połowę ekranu i pozwalamy systemowi na ustalenie położenia ramki:

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

Dodatkowo dostarczymy ikonę. Do wygodnego ładowania obrazów można wykorzystać klasę `ImageIcon`. Poniżej przedstawiony jest sposób jej użycia:

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

Ikona może się pojawić w różnych miejscach, w zależności od systemu operacyjnego. W systemie Windows pojawi się na przykład w lewym górnym rogu okna oraz będzie widoczna wśród aktywnych zadań pojawiających się w wyniku naciśnięcia kombinacji klawiszy *Alt+Tab*.

Listing 7.2 przedstawia pełny kod omawianego programu. Po jego uruchomieniu zwróć uwagę na ikonę *Core Java*.

Listing 7.2. sizedFrame/SizedFrameTest.java

```
package sizedFrame;

import java.awt.*;
import javax.swing.*;

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class SizedFrameTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new SizedFrame();
                frame.setTitle("SizedFrame");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

class SizedFrame extends JFrame
{
    public SizedFrame()
    {
        // Sprawdzenie wymiarów ekranu.

        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth = screenSize.width;

        // Ustawienie szerokości i wysokości ramki oraz polecenie systemowi, aby ustalił jej położenie.
```

```

setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);

// Ustawienie ikony i tytułu.

Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
}
}

```

Jeszcze kilka dodatkowych wskazówek dotyczących obsługi ramek:

- Jeśli ramka zawiera tylko standardowe komponenty, jak przyciski i pola tekstowe, jej rozmiar można ustawić za pomocą metody pack. Rozmiar zostanie ustawiony na najmniejszy, w którym zmieszcza się wszystkie komponenty. Często rozmiar głównej ramki jest ustawiany na największą wartość. Od Java SE 1.4 ramkę można zmaksymalizować za pomocą następującego wywołania:

```
frame.setExtendedState(Frame.MAXIMIZED_BOTH);
```

- Dobrym pomysłem jest zapisanie położenia i rozmiaru ramki ustawionych przez użytkownika i zastosowanie tych ustawień przy ponownym uruchomieniu aplikacji. Jak to zrobić, dowiemy się w rozdziale 10., przy okazji omawiania API preferencji.
- Jeśli aplikacja wykorzystuje kilka ekranów, ich rozmiary można sprawdzić za pomocą metod GraphicsEnvironment i GraphicsDevice.
- Ponadto klasa GraphicsDevice umożliwia uruchomienie programu w trybie pełnoekranowym.

java.awt.Component 1.0

- boolean isVisible()
- void setVisible(boolean b)

Sprawdza lub ustawia właściwość widoczności. Komponenty są początkowo widoczne z wyjątkiem komponentów najwyższego poziomu, jak JFrame.

- void setSize(int width, int height) 1.1

Ustawia szerokość i wysokość komponentu.

- void setLocation(int x, int y) 1.1

Przenosi komponent w inne miejsce. Współrzędne x i y są względne do kontenera lub ekranu, jeśli komponent jest komponentem najwyższego poziomu (np. JFrame).

- void setBounds(int x, int y, int width, int height) 1.1

Przesuwa komponent i zmienia jego rozmiar.

- Dimension getSize() 1.1

- void setSize(Dimension d) 1.1

Pobiera lub ustawia właściwość size komponentu.

java.awt.Window 1.0■ **void toFront()**

Przenosi okno przed wszystkie pozostałe okna.

■ **void toBack()**

Przenosi okno na sam dół stosu okien i odpowiednio przestawia pozostałe widoczne okna.

■ **boolean isLocationByPlatform() 5.0**■ **void setLocationByPlatform(boolean b) 5.0**

Pobiera lub ustawia właściwość `locationByPlatform`. Jeśli zostanie ona ustawiona przed wyświetleniem okna, platforma wybierze odpowiednią lokalizację.

java.awt.Frame 1.0■ **boolean isResizable()**■ **void setResizable(boolean b)**

Pobiera lub ustawia właściwość `resizable`. Jeśli jest ona ustawiona, użytkownik może zmieniać rozmiar ramki.

■ **String getTitle()**■ **void setTitle(String s)**

Pobiera lub ustawia właściwość `title` określającą tekst na pasku tytułu.

■ **Image getIconImage()**■ **void setIconImage(Image image)**

Pobiera lub ustawia właściwość `iconImage`, która określa ikonę ramki. System może wyświetlić ikonę jako dodatek w ramce lub w innym miejscu.

■ **boolean isUndecorated() 1.4**■ **void setUndecorated(boolean b) 1.4**

Pobiera lub ustawia właściwość `undecorated`. Jeśli ta właściwość jest ustawiona, ramka nie zawiera żadnych dodatków, jak pasek tytułu czy przycisk zamkający. Ta metoda musi być wywołana przed wyświetleniem ramki.

■ **int getExtendedState() 1.4**■ **void setExtendedState(int state) 1.4**

Pobiera lub ustawia stan rozszerzonego okna. Możliwe stany to:

Frame.NORMAL

Frame.ICONIFIED

Frame.MAXIMIZED_HORIZ

Frame.MAXIMIZED_VERT

Frame.MAXIMIZED_BOTH

java.awt.Toolkit 1.0

- static Toolkit getDefaultToolkit()

Zwraca standardowy zestaw narzędzi.

- Dimension getScreenSize()

Pobiera rozmiar ekranu.

javax.swing.ImageIcon 1.2

- ImageIcon(String filename)

Tworzy ikonę, której obraz jest przechowywany w pliku.

- Image getImage()

Pobiera obraz ikony.

7.4. Wyświetlanie informacji w komponencie

W tym podrozdziale nauczymy się wyświetlać informacje w ramce. Zamiast przykładowego programu wyświetlającego tekst w konsoli, który widzieliśmy w rozdziale 3., zaprezentujemy program wyświetlający tekst w ramce, jak na rysunku 7.7.

Rysunek 7.7.

Ramka wyświetlająca tekst



Łańcuch tekstowy można wydrukować bezpośrednio na ramce, ale jest to uznawane za zły zwyczaj programistyczny. Ramki w Javie są z założenia kontenerami do przechowywania komponentów, takich jak pasek menu i inne elementy interfejsu użytkownika. Z reguły rysowanie odbywa się na innym dodanym do ramki komponencie.

Struktura ramki JFrame jest zadziwiająco skomplikowana. Rysunek 7.8 przedstawia schemat budowy takiej ramki. Jak widać, ramka JFrame składa się z czterech warstw. Trzy z nich — podstawowa (ang. *root pane*), warstwowa (ang. *layered pane*) i przezroczysta (ang. *glass pane*) — nie są dla nas interesujące. Ich przeznaczeniem jest organizacja paska menu i warstwy z treścią oraz implementacja stylu. Część interesująca programistów Swing to **warstwa treści** (ang. *content pane*). Podczas projektowania ramki komponenty dodaje się do warstwy treści, stosując kod podobny do poniższego:

```
Container contentPane = frame.getContentPane();
Component c = . . . ;
contentPane.add(c);
```

Do Java SE 1.4 metoda `add` klasy `JFrame` powodowała wyjątek wyświetlający komunikat „Do not use `JFrame.add()`. Use `JFrame.getContentPane().add()` instead”. W Java SE 5.0 zrezygnowano z takiej edukacji programistów, czego wyrazem jest zezwolenie na wywoływanie metody `JFrame.add` na rzecz warstwy z treścią.

Dzięki temu od Java SE 5.0 można stosować krótki zapis:

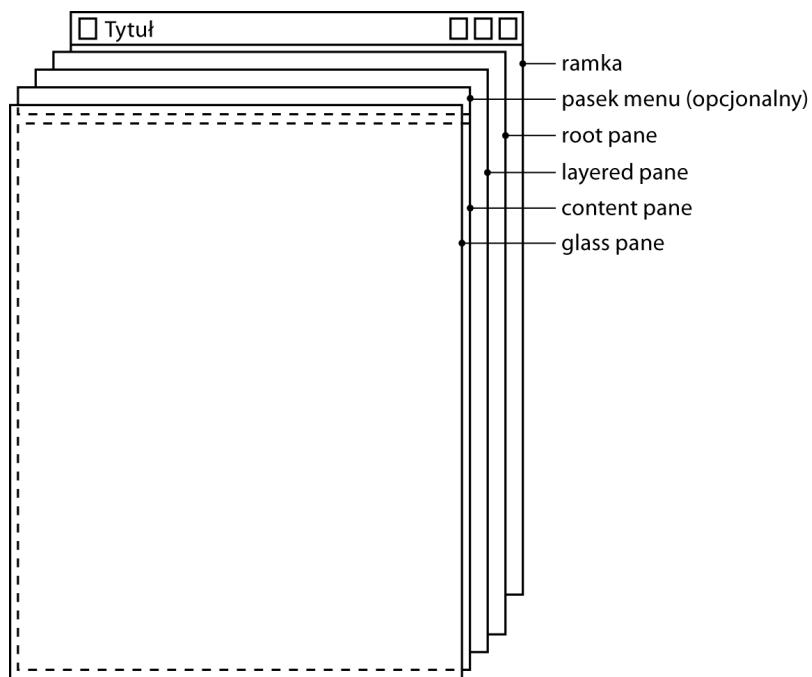
```
frame.add(c);
```

Tym razem chcemy dodać do ramki jeden komponent, na którym narysujemy nasz komunikat. Aby móc rysować na komponencie, należy zdefiniować klasę rozszerzającą klasę `JComponent` i przesłonić w niej metodę `paintComponent` klasy nadzędnej.

Metoda `paintComponent` przyjmuje jeden parametr typu `Graphics`. Obiekt typu `Graphics` zawiera ustawienia dotyczące rysowania obrazów i tekstu, jak czcionka czy aktualny kolor. Rysowanie w Javie zawsze odbywa się za pośrednictwem obiektu `Graphics`. Udostępnia on metody rysujące wzory, obrazy i tekst.

Rysunek 7.8.

Wewnętrzna struktura ramki `JFrame`



Parametr `Graphics` jest podobny do kontekstu urządzeń (ang. *device context*) w systemie Windows i kontekstu graficznego (ang. *graphics context*) w programowaniu dla systemu X11.

Poniższy fragment programu tworzy komponent, na którym można rysować:

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
```

```
{
    kod rysujący
}
```

Za każdym razem, kiedy okno musi być ponownie narysowane, metoda obsługi zdarzeń informuje o tym komponent. Powoduje to uruchomienie metod `paintComponent` wszystkich komponentów.

Nigdy nie należy wywoływać metody `paintComponent` samodzielnie. Jest ona wywoływaną automatycznie, gdy trzeba ponownie narysować jakąś część aplikacji, i nie należy zaburzać tego automatycznego procesu.

Jakiego rodzaju czynności uruchamiają tę automatyczną reakcję? Rysowanie jest konieczne, na przykład kiedy użytkownik zwiększy rozmiar okna albo je zminimalizuje, a następnie zmaksymalizuje. Jeśli zostanie otwarte nowe okno, które częściowo przykryje stare, to po zamknięciu tego nowego okna przykryta część starego zostanie zniszczona i trzeba ją będzie ponownie narysować (system graficzny nie zapisuje pikseli, które znajdują się pod spodem). Oczywiście przy pierwszym uruchomieniu okna musi ono przetworzyć kod określający sposób i miejsce rysowania początkowych elementów.



Aby wymusić ponowne rysowanie ekranu, należy użyć metody `repaint`. Wywoła ona metody `paintComponent` wszystkich komponentów, które mają prawidłowo skonfigurowany obiekt `Graphics`.

Z przedstawionego powyżej fragmentu kodu wnioskujemy, że metoda `paintComponent` przyjmuje tylko jeden parametr typu `Graphics`. Jednostką miary obiektów `Graphics` wyświetlanymi na ekranie są piksele. Para współrzędnych $(0, 0)$ określa lewy górny róg komponentu, na którego powierzchni odbywa się rysowanie.

Wyświetlanie tekstu jest specjalnym rodzajem rysowania. Klasa `Graphics` udostępnia metodę `drawString` o następującej składni:

```
g.drawString(text, x, y)
```

Chcemy narysować łańcuch: *To nie jest program „Witaj, świecie”* w oryginalnym oknie w odległości około jednej czwartej szerokości od lewej krawędzi i połowy wysokości od krawędzi górnej. Mimo że nie potrafimy jeszcze mierzyć długości łańcuchów, zaczniemy rysowanie w punkcie o współrzędnych $(75, 100)$. Oznacza to, że pierwsza litera łańcucha jest przesunięta w prawo o 75 pikseli i w dół o 100 pikseli (w rzeczywistości o 100 pikseli w dół przesunięta jest podstawowa linia pisma — więcej na ten temat w dalszej części rozdziału). Kod opisanej metody `paintComponent` znajduje się poniżej:

```
class NotHelloWorldComponent extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    public void paintComponent(Graphics g)
    {
        g.drawString("To nie jest program „Witaj, świecie”", MESSAGE_X, MESSAGE_Y);
```

```

    }
    . .
}
```

W końcu komponent powinien informować użytkownika o tym, jaki ma być jego rozmiar. Przesłonimy metodę `getPreferredSize`, aby zwracała obiekt klasy `Dimension` z preferowaną szerokością i wysokością:

```

class NotHelloWorldComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    .
    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
        DEFAULT_HEIGHT); }
}
```

Gdy umieścimy w ramce jakieś komponenty i będziemy chcieli użyć ich preferowanych rozmiarów, to zamiast `setSize` wywołamy metodę `pack`:

```

class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
        add(new NotHelloWorldComponent());
        pack();
    }
}
```

Listing 7.3 zawiera pełny kod programu.



Niektórzy programiści wolą rozszerzać klasę `JPanel` zamiast `JComponent`. Obiekt `JPanel` jest z założenia **kontenerem** na inne komponenty, ale można także na nim rysować. Jest jednak między nimi jedna różnica — panel jest **nieprzezroczysty**, czyli rysuje wszystkie piksele w swoim obrębie. Najprostszym sposobem na zrobienie tego jest narysowanie panelu z kolorowym tłem za pomocą wywołania metody `super.paintComponent` w metodzie `paintComponent` każdej podklasy panelu:

```

class NotHelloWorldPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        procedury rysujące
    }
}
```

Listing 7.3. notHelloWorld/NotHelloWorld.java

```

package notHelloWorld;

import javax.swing.*;
import java.awt.*;

/**
 * @version 1.32 2007-06-12

```

```

* @author Cay Horstmann
*/
public class NotHelloWorld
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new NotHelloWorldFrame();
                frame.setTitle("NotHelloWorld");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca panel z komunikatem.
*/
class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
        add(new NotHelloWorldComponent());
        pack();
    }
}

/**
 * Panel wyświetlający komunikat.
*/
class NotHelloWorldPanel extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public void paintComponent(Graphics g)
    {
        g.drawString("To nie jest program „Witaj, świecie.”", MESSAGE_X, MESSAGE_Y);
    }
    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
        DEFAULT_HEIGHT); }
}

```

javax.swing.JFrame 1.2

■ Container getContentPane()

Zwraca obiekt ContentPane dla ramki JFrame.

- Component add(Component c)

Dodaje i zwraca dany komponent do warstwy treści ramki (przed Java SE 5.0 ta metoda powodowała wyjątek).

java.awt.Component **1.0**

- void repaint()

Powoduje ponowne jak najszybsze narysowanie komponentu.

- Dimension getPreferredSize()

Metoda, którą należy prześłonić, aby zwracała preferowany rozmiar komponentu.

javax.swing.JComponent **1.2**

- void paintComponent(Graphics g)

Metoda, którą należy prześłonić w celu zdefiniowania sposobu rysowania określonego komponentu.

java.awt.Window **1.0**

- void pack()

Zmienia rozmiar okna, biorąc pod uwagę preferowane rozmiary znajdujących się w nim komponentów.

7.5. Figury 2D

Od Java 1.0 klasa Graphics udostępnia metody rysujące linie, prostokąty, elipsy itd. Ich możliwości są jednak bardzo ograniczone. Nie ma na przykład możliwości ustawienia grubości linii ani obracania figur.

W Java 1.2 wprowadzono bibliotekę Java2D udostępniającą szeroki wachlarz metod graficznych. W tym rozdziale opisujemy tylko podstawy tej biblioteki — więcej bardziej zaawansowanych informacji na ten temat znajduje się w rozdziale 7. w drugim tomie.

Aby narysować figurę biblioteki Java2D, trzeba utworzyć obiekt klasy Graphics2D. Klasa ta jest podklassą klasy Graphics. Od Java SE 2 metody takie jak paintComponent automatycznie odbierają obiekty klasy Graphics2D. Wystarczy zastosować rzutowanie:

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```

Figury geometryczne w bibliotece Java2D są obiektami. Istnieją klasy reprezentujące linie, prostokąty i elipsy:

Line2D
Rectangle2D
Ellipse2D

Wszystkie te klasy implementują interfejs Shape.



Biblioteka Java2D obsługuje także bardziej skomplikowane figury, jak łuki, krzywe drugiego i trzeciego stopnia oraz trajektorie. Więcej informacji na ten temat znajduje się w rozdziale 7. drugiego tomu.

Aby narysować figurę, należy najpierw utworzyć obiekt klasy implementującej interfejs Shape, a następnie wywołać metodę draw klasy Graphics2D. Na przykład:

```
Rectangle2D rect = . . .;
g2.draw(rect);
```



Przed pojawieniem się biblioteki Java2D do rysowania figur używano metod klasy Graphics, np. drawRectangle. Na pierwszy rzut oka te stare wywołania wydają się prostsze, ale używając biblioteki Java2D, pozostawiamy sobie różne możliwości do wyboru — można później ulepszyć rysunki za pomocą rozmaitych narzędzi dostępnych w bibliotece Java2D.

Biblioteka Java2D nieco komplikuje programowanie. W przeciwieństwie do metod rysujących z wersji 1.0, w których współrzędne były liczbami całkowitymi, figury Java2D używają współrzędnych zmiennoprzecinkowych. W wielu przypadkach stanowi to duże ułatwienie dla programisty, ponieważ może on określić figury przy użyciu lepiej znanych mu jednostek (jak milimetry lub cale), które później są konwertowane na piksele. Biblioteka Java2D wykonuje obliczenia o pojedynczej precyzyji na liczbach typu float w większości wykonywanych wewnętrznie działań. Pojedyncza precyza w zupełności wystarcza — celem obliczeń geometrycznych jest przecież ustanie pikseli na ekranie lub w drukarce. Dopóki błędy zaokrąglania mieszczą się w zakresie jednego piksela, rezultat wizualny nie cierpi. Ponadto obliczenia na liczbach typu float są na niektórych platformach szybsze, a dodatkowo wartości tego typu zajmują o połowę mniej miejsca niż wartości typu double.

Czasami jednak obliczenia na liczbach typu float bywają niewygodne, ponieważ Java nie-wzruszenie wymaga rzutowania, jeśli niezbędna jest konwersja wartości typu double na typ float. Przeanalizujmy na przykład poniższą instrukcję:

```
float f = 1.2; // blqdf
```

Ta instrukcja spowoduje błąd komilacji, ponieważ stała 1.2 jest typu double i komilator obawia się utraty danych. Rozwiążaniem jest dodanie przyrostka F do stałej zmiennoprzecinkowej:

```
float f = 1.2F; // OK
```

Teraz przyjrzyjmy się poniższej instrukcji:

```
Rectangle2D r = . . .
float f = r.getWidth();           // błąd
```

Instrukcja ta spowoduje błąd komplikacji z tego samego powodu co poprzednia. Metoda `getWidth` zwraca wartość typu `double`. W tym przypadku rozwiązaniem jest rzutowanie:

```
float f = (float) r.getWidth(); // OK
```

Ponieważ stosowanie przyrostków i rzutowania nie jest wygodne, projektanci biblioteki Java2D postanowili utworzyć **dwie wersje** każdej klasy reprezentującej figurę: jedną ze współrzędnymi typu `float` dla oszczędnych programistów i jedną ze współrzędnymi typu `double` dla leniwych (w tej książce zaliczamy się do tych drugich, czyli stosujemy współrzędne typu `double`, gdzie tylko możemy).

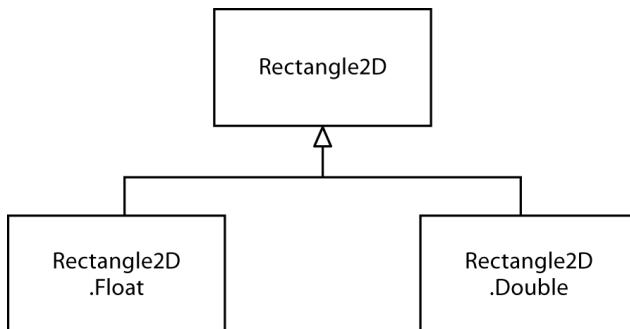
Projektanci biblioteki zastosowali ciekawą i początkowo wprowadzającą w błąd metodę pakowania obu wersji klas. Przyjrzyjmy się klasie `Rectangle2D`. Jest to abstrakcyjna klasa mająca dwie konkretne podklasy, które są dodatkowo statycznymi klasami wewnętrznymi:

```
Rectangle2D.Float
Rectangle2D.Double
```

Rysunek 7.9 przedstawia diagram dziedziczenia.

Rysunek 7.9.

Klasy prostokątów 2D



Najlepiej ignorować fakt, że obie te konkretne klasy są statyczne i wewnętrzne — to tylko taki chwyt, który pozwala uniknąć nazw `FloatRectangle2D` i `DoubleRectangle2D` (więcej informacji na temat statycznych klas wewnętrznych znajduje się w rozdziale 6.).

Tworząc obiekt klasy `Rectangle2D.Float`, wartości określające współrzędne należy podawać jako typu `float`. W przypadku klasy `Rectangle2D.Double` współrzędne muszą być typu `double`.

```
Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Ponieważ zarówno klasa `Rectangle2D.Float`, jak i `Rectangle2D.Double` rozszerzają wspólną klasę `Rectangle2D`, a metody w tych podklasach przesłaniają metody nadklasy, zapamiętywanie typu figury nie przynosi właściwie żadnych korzyści. Referencje do prostokątów można przechowywać w zmiennych typu `Rectangle2D`.

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Oznacza to, że użycie klas wewnętrznych jest konieczne tylko przy tworzeniu obiektów figur.

Parametry konstruktora określają lewy górny róg, szerokość i wysokość prostokąta.



Klasa `Rectangle2D.Float` zawiera jedną metodę, której nie dziedziczy po klasie `Rectangle2D`. Jest to metoda `setRect(float x, float y, float h, float w)`. Metody tej nie można użyć, jeśli referencja do obiektu typu `Rectangle2D.Float` jest przechowywana w zmiennej typu `Rectangle2D`. Nie jest to jednak duża strata — klasa `Rectangle2D` zawiera metodę `setRect` z parametrami typu `double`.

Metody klasy `Rectangle2D` przyjmują parametry i zwracają wartości typu `double`. Na przykład metoda `getWidth` zwraca wartość typu `double`, nawet jeśli szerokość jest zapisana w postaci liczby typu `float` w obiekcie typu `Rectangle2D.Float`.



Aby całkowicie pozbyć się wartości typu `float`, należy używać klas typu `Double`. Jednak w programach tworzących wiele tysięcy figur warto rozważyć użycie klas `Float` ze względu na oszczędność pamięci.

Wszystko, co napisaliśmy do tej pory na temat klas `Rectangle2D`, dotyczy również pozostałych klas reprezentujących figury. Dodatkowo istnieje klasa o nazwie `Point2D`, której podklasy to `Point2D.Float` i `Point2D.Double`. Poniższy fragment programu tworzy obiekt takiej klasy:

```
Point2D p = new Point2D.Double(10, 20);
```

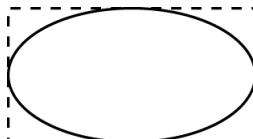


Klasa `Point2D` jest niezwykle przydatna — zastosowanie obiektów `Point2D` jest znacznie bliższe idei programowania obiektowego niż używanie oddzielnych wartości `x` i `y`. Wiele konstruktów przyjmuje parametry typu `Point2D`. Zalecamy stosowanie obiektów tej klasy, gdzie się da — dzięki nim obliczenia geometryczne są często dużo prostsze.

Klasy `Rectangle2D` i `Ellipse2D` dziedziczą po wspólnej nadklasie `RectangularShape`. Wprawdzie elipsa nie jest prostokątna, ale można na niej opisać prostokąt (zobacz rysunek 7.10).

Rysunek 7.10.

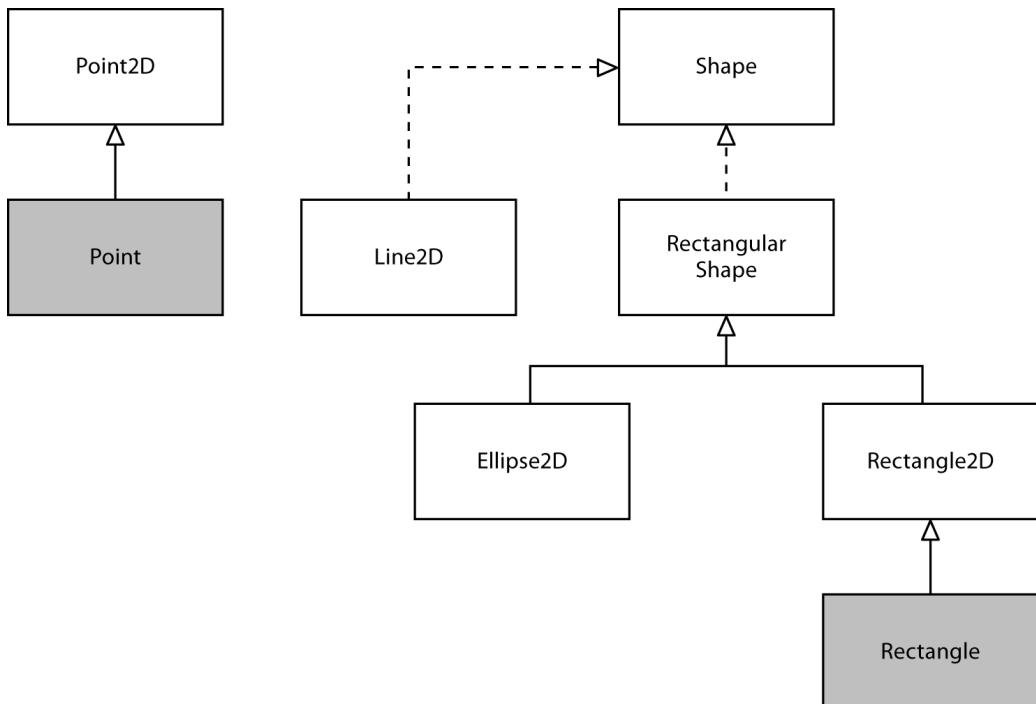
Prostokąt opisany na elipsie



Klasa `RectangularShape` definiuje ponad 20 metod wspólnych dla tych figur. Zaliczają się do nich metody `getWidth`, `getHeight`, `getCenterX` i `getCenterY` (niestety w czasie pisania tej książki nie było metody `getCenter` zwracającej obiekt typu `Point2D`).

Dodatkowo do hierarchii klas reprezentujących figury dodano kilka starszych klas z Java 1.0. Klasy `Rectangle` i `Point`, które przechowują prostokąt i punkt przy użyciu współrzędnych całkowitych, rozszerzają klasę `Rectangle2D` i `Point2D`.

Rysunek 7.11 przedstawia relacje pomiędzy klasami figur. Klasy Double i Float zostały pominięte, a klasy spadkowe wyróżniono szarym tłem.



Rysunek 7.11. Relacje między klasami figur

Tworzenie obiektów typu `Rectangle2D` i `Ellipse2D` jest prostym zadaniem. Należy podać:

- współrzędne `x` i `y` lewego górnego rogu,
- wysokość i szerokość.

W przypadku elipsy te wartości dotyczą opisanego na niej prostokąta. Na przykład instrukcja:

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

utworzy elipsę wpisaną w prostokąt, którego lewy górny róg znajduje się w punkcie o współrzędnych (150, 200) o szerokości 100 i wysokości 50.

Czasami jednak współrzędne lewego górnego rogu nie są od razu dostępne. Często zdarza się, że dostępne są dwa punkty leżące naprzeciw siebie, ale nie są to rogi górny lewy i prawy dolny. Nie można utworzyć prostokąta w poniższy sposób:

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py); // błąd
```

Jeśli `p` nie jest lewym górnym rogiem, jedna lub obie współrzędne będą miały wartości ujemne i prostokąt się nie pojawi. W takim przypadku należy najpierw utworzyć pusty prostokąt i użyć metody `setFrameFromDiagonal`:

```
Rectangle2D rect = new Rectangle2D.Double();
rect setFrameFromDiagonal(px, py, qx, qy);
```

Jeszcze lepiej, jeśli p i q są punktami rogów reprezentowanymi przez obiekty typu `Point2D`:

```
rect.setFrameFromDiagonal(p, q);
```

Przy tworzeniu elipsy zazwyczaj znane są środek, szerokość i wysokość opisanego na niej prostokąta, a nie jego rogi (które nawet nie leżą na elipsie). Metoda `setFrameFromCenter` przyjmuje punkt środkowy, ale wymaga także jednego z czterech rogów. W związku z tym elipsę zazwyczaj tworzy się następująco:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2, centerY - height / 2,
width, height);
```

Aby utworzyć linię, należy podać jej punkt początkowy i końcowy w postaci obiektów `Point2D` lub par liczb:

```
Line2D line = new Line2D.Double(start, end);
```

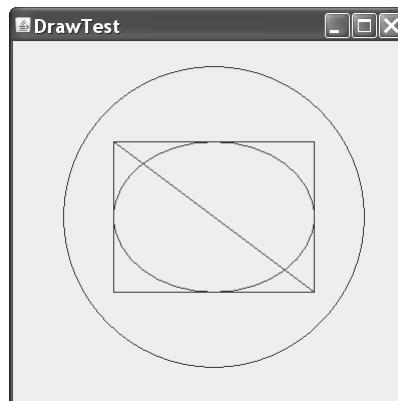
lub

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

Program przedstawiony na listingu 7.4 rysuje prostokąt, elipsę znajdująca się wewnątrz tego prostokąta, przekątną prostokąta oraz koło o takim samym środku jak prostokąt. Rysunek 7.12 przedstawia wynik działania tego programu.

Rysunek 7.12.

Rysowanie figur geometrycznych



Listing 7.4. draw/DrawTest.java

```
package draw;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class DrawTest
{
    public static void main(String[] args)
    {
```

```
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        JFrame frame = new DrawFrame();
        frame.setTitle("DrawTest");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
});

/**
 * Ramka zawierająca panel z rysunkami.
 */
class DrawFrame extends JFrame
{
    public DrawFrame()
    {
        add(new DrawComponent());
        pack();
    }
}

/**
 * Komponent wyświetlający prostokąty i elipsy.
 */
class DrawComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        // Rysowanie prostokąta.

        double leftX = 100;
        double topY = 100;
        double width = 200;
        double height = 150;

        Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
        g2.draw(rect);

        // Rysowanie elipsy.

        Ellipse2D ellipse = new Ellipse2D.Double();
        ellipse setFrame(rect);
        g2.draw(ellipse);

        // Rysowanie przekątnej.

        g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
    }
}
```

```

// Rysowanie koła z takim samym środkiem.

double centerX = rect.getCenterX();
double centerY = rect.getCenterY();
double radius = 150;

Ellipse2D circle = new Ellipse2D.Double();
circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
g2.draw(circle);
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}

```

java.awt.geom.RectangularShape 1.2

- double getCenterX()
- double getCenterY()
- double getMinX()
- double getMinY()
- double getMaxX()
- double getMaxY()

Zwraca współrzędną x lub y punktu środkowego, punktu o najmniejszych lub największych współrzędnych prostokąta.

- double getWidth()
- double getHeight()

Zwraca szerokość lub wysokość prostokąta.

- double getX()
- double getY()

Zwraca współrzędną x lub y lewego górnego rogu prostokąta.

java.awt.geom.Rectangle2D.Double 1.2

- Rectangle2D.Double(double x, double y, double w, double h)

Tworzy prostokąt z lewym górnym rogiem w podanym miejscu i o podanej szerokości i długości.

java.awt.geom.Rectangle2D.Float 1.2

- Rectangle2D.Float(float x, float y, float w, float h)

Tworzy prostokąt z lewym górnym rogiem w podanym miejscu i o podanej szerokości i długości.

java.awt.geom.Ellipse2D.Double **1.2**

- `Ellipse2D.Double(double x, double y, double w, double h)`

Rysuje elipsę wpisaną w prostokąt, którego lewy górnny róg znajduje się w podanym miejscu i który ma określone wysokość oraz szerokość.

java.awt.geom.Point2D.Double **1.2**

- `Point2D.Double(double x, double y)`

Rysuje punkt o podanych współrzędnych.

java.awt.geom.Line2D.Double **1.2**

- `Line2D.Double(Point2D start, Point2D end)`

- `Line2D.Double(double startX, double startY, double endX, double endY)`

Rysuje linię między dwoma podanymi punktami.

7.6. Kolory

Metoda `setPaint` z klasy `Graphics2D` ustawia kolor, który jest stosowany we wszystkich kolejnych rysunkach graficznych. Na przykład:

```
g2.setPaint(Color.RED);
g2.drawString("Uwaga!", 100, 100);
```

Figury zamknięte (np. prostokąt czy elipsa) można w takiej sytuacji wypełnić za pomocą metody `fill` (zamiast `draw`):

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // Wypełnienie prostokąta rect kolorem czerwonym.
```

Aby zastosować kilka kolorów, należy wybrać kolor, zastosować metodę `draw` lub `fill`, a następnie wybrać inny kolor i ponownie zastosować metodę `draw` lub `fill`.



Metoda `fill` rysuje o jeden piksel mniej po prawej i na dole. Jeśli na przykład narysujemy prostokąt `new Rectangle2D.Double(0, 0, 10, 20)`, to rysunek będzie obejmował piksele o współrzędnych $x = 10$ i $y = 20$. Jeśli wypełnimy ten prostokąt kolorem, piksele te nie zostaną pokolorowane.

Do definiowania kolorów służy klasa `java.awt.Color`. W klasie tej dostępnych jest 13 następujących predefiniowanych stałych reprezentujących kolory:

BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED,
WHITE, YELLOW



Przed Java SE 1.4 stałe określające kolory były pisane małymi literami, np. `Color.red`. Było to sprzeczne z przyjętą konwencją pisania stałych wielkimi literami. Obecnie nazwy tych stałych można pisać wielkimi lub, ze względu na zgodność wstępczą, małymi literami.

Niestandardowy kolor można zdefiniować, tworząc obiekt klasy `Color` i podając wartości trzech składowych: czerwonego, zielonego i niebieskiego. Wartość każdego ze składników (zajmujących po jednym bajcie) musi należeć do zbioru 0 – 255. Poniższy kod przedstawia sposób wywołania konstruktora klasy `Color` z parametrami określającymi stopień czerwieni, niebieskiego i zieleni:

```
Color(int redness, int greenness, int blueness)
```

Poniżej znajduje się przykładowa procedura tworząca niestandardowy kolor:

```
g2.setPaint(new Color(0, 128, 128)); //niebieskozielony
g2.drawString("Witaj!", 75, 125);
```



Poza jednolitymi kolorami można także stosować bardziej skomplikowane ustawienia, jak różne odcienie czy obrazy. Więcej informacji na ten temat znajduje się w drugim tomie w rozdziale o zaawansowanych technikach AWT. Jeśli zamiast obiektu typu `Graphics2D` zostanie użyty obiekt `Graphics`, kolory należy ustawiać za pomocą metody `setColor`.

Do ustawiania **koloru tła** służy metoda `setBackground` z klasy `Component`, będącej nadklassą klasy `JComponent`.

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

Istnieje też metoda `setForeground`, która określa kolor elementów rysowanych na komponencie.



Metody `brighter()` i `darker()` — jak sama nazwa wskazuje — sprawiają, że aktualnie używany kolor staje się jaśniejszy bądź ciemniejszy. Ponadto metoda `brighter` jest dobrym sposobem na wyróżnienie wybranego elementu. W rzeczywistości metoda ta nieznacznie rozjaśnia kolor. Aby kolor był dużo jaśniejszy, można tę metodę zastosować trzy razy: `c.brighter().brighter().brighter()`.

Znacznie więcej predefiniowanych nazw kolorów znajduje się w klasie `SystemColor`. Stałe tej klasy określają kolory stosowane do rozmaitych elementów systemu użytkownika. Na przykład instrukcja:

```
p.setBackground(SystemColor.window)
```

ustawia kolor tła komponentu na domyślny dla wszystkich okien w systemie użytkownika (tło jest wstawiane przy każdym rysowaniu okna). Kolory zdefiniowane w klasie `SystemColor` są szczególnie przydatne, kiedy chcemy narysować elementy interfejsu użytkownika nieodbiegające kolorystyką od standardowych elementów w systemie. Tabela 7.1 zawiera nazwy kolorów systemowych oraz ich opisy.

Tabela 7.1. System kolorów

Nazwa	Zastosowanie
desktop	Kolor tła pulpitu
activeCaption	Kolor belki tytułu aktywnego okna
activeCaptionText	Kolor tekstu na belce tytułu
activeCaptionBorder	Kolor obramowania aktywnej belki
inactiveCaption	Kolor nieaktywnej belki
inactiveCaptionText	Kolor tekstu nieaktywnej belki
inactiveCaptionBorder	Kolor obramowania nieaktywnej belki
window	Tło okna
windowBorder	Kolor obramowania okna
windowText	Kolor tekstu w oknie
menu	Tło menu
menuText	Kolor tekstu w menu
text	Kolor tła tekstu
textText	Kolor tekstu
textInactiveText	Kolor tekstu nieaktywnych elementów sterujących
textHighlight	Kolor tła wyróżnionego tekstu
textHighlightText	Kolor wyróżnionego tekstu
control	Kolor tła elementów sterujących
controlText	Kolor tekstu w elementach sterujących
controlLtHighlight	Słabe wyróżnienie elementów sterujących
controlHighlight	Silne wyróżnienie elementów sterujących
controlShadow	Kolor cienia elementów sterujących
controlDkShadow	Ciemniejszy kolor cienia elementów sterujących
scrollbar	Kolor tła dla suwaków
info	Kolor tła dla tekstu pomocy
infoText	Kolor tekstu pomocy

java.awt.Color 1.0

■ `Color(int r, int g, int b)`

Tworzy obiekt reprezentujący kolor.

Parametry: r Wartość barwy czerwonej

g Wartość barwy zielonej

b Wartość barwy niebieskiej

java.awt.Graphics **1.0**

- `Color getColor()`
- `void setColor(Color c)`

Pobiera lub ustawia kolor. Wszystkie następne rysunki będą miały ten kolor.

Parametry: `c` Nowy kolor

java.awt.Graphics2D **1.2**

- `Paint getPaint()`
- `void setPaint(Paint p)`

Pobiera lub ustawia właściwość `paint` danego kontekstu graficznego. Klasa `Color` implementuje interfejs `Paint`. W związku z tym za pomocą tej metody można ustawić atrybut `paint` na jednolity kolor.

- `void fill(shape s)`

Wypełnia figurę aktualnym kolorem.

java.awt.Component **1.0**

- `Color getBackground()`
- `void setBackground(Color c)`

Pobiera lub ustawia kolor tła.

Parametry: `c` Nowy kolor tła

- `Color getForeground()`
- `void setForeground(Color c)`

Pobiera lub ustawia kolor frontu.

Parametry: `c` Nowy kolor frontu

7.7. Czcionki

Program przedstawiony na początku tego rozdziału wyświetlał łańcuch tekstu pisany domyślną czcionką. Często jednak zdarza się, że tekst musi być napisany inną czcionką. Identyfikatorem czcionki jest jej **nazwa**. Nazwa czcionki składa się z **nazwy rodziny czcionek**, np. Helvetica, i opcjonalnego przyrostka, np. Bold. Na przykład nazwy Helvetica i Helvetica Bold należą do rodziny czcionek o nazwie Helvetica.

Aby sprawdzić, jakie czcionki są dostępne w danym komputerze, należy wywołać metodę `getAvailableFontFamilyNames` z klasy `GraphicsEnvironment`. Ta metoda zwraca tablicę nazw wszystkich dostępnych czcionek w postaci łańcuchów. Egzemplarz klasy `GraphicsEnvironment` reprezentujący środowisko graficzne systemu użytkownika można utworzyć za

pomocą statycznej metody `getLocalGraphicsEnvironment`. Poniższy program drukuje nazwy wszystkich czcionek znajdujących się w systemie:

```
import java.awt.*;  
  
public class ListFonts  
{  
    public static void main(String[] args)  
    {  
        String[] fontNames = GraphicsEnvironment  
            .getLocalGraphicsEnvironment()  
            .getAvailableFontFamilyNames();  
        for (String fontName : fontNames)  
            System.out.println(fontName);  
    }  
}
```

W jednym z systemów początek tej listy wygląda następująco:

```
Abadi MT Condensed Light  
Arial  
Arial Black  
Arial Narrow  
Arioso  
Baskerville  
Binner Gothic  
...
```

Lista ta zawiera ponad 70 pozycji.

Nazwy czcionek mogą być znakami towarowymi, a ich projekty mogą w niektórych jurysdykcjach podlegać prawom autorskim. W związku z tym dystrybucja czcionek często wiąże się z uiszczaniem opłat licencyjnych ich właścicielowi. Oczywiście, podobnie jak są tanie podróbki drogich perfum, istnieją też podróbki czcionek imitujące oryginały. Na przykład imitacja czcionki Helvetica w systemie Windows nosi nazwę Arial.

Jako wspólny punkt odniesienia w bibliotece AWT zdefiniowano pięć **logicznych** nazw czcionek:

```
SansSerif  
Serif  
Monospaced  
Dialog  
DialogInput
```

Czcionki te są zawsze zamieniane na czcionki, które znajdują się w danym urządzeniu. Na przykład w systemie Windows czcionka SansSerif jest zastępowana czcionką Arial.

Dodatkowo pakiet SDK firmy Sun zawsze zawiera trzy rodziny czcionek: Lucida Sans, Lucida Bright i Lucida Sans Typewriter.

Aby narysować znak daną czcionką, najpierw trzeba utworzyć obiekt klasy `Font`. Konieczne jest podanie nazwy i stylu czcionki oraz rozmiaru w punktach drukarskich. Poniższa instrukcja tworzy obiekt klasy `Font`:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

Trzeci argument określa rozmiar w punktach. Jednostka ta jest powszechnie stosowana w typografii do określania rozmiaru czcionek. Jeden punkt jest równy 1/72 cala, czyli około 0,35 mm.

W konstruktorze klasy `Font` można użyć logicznej nazwy czcionki zamiast nazwy fizycznej. Styl (zwykły, **pogrubiony**, kursywa lub **pogrubiona kursywa**) określa drugi argument konstruktora `Font`, który może mieć jedną z poniższych wartości:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```



Sposób odwzorowania logicznych nazw czcionek na fizyczne jest określony w pliku `fontconfig.properties` w katalogu `jre/lib` znajdującym się w folderze instalacji Javy. Informacje na temat tego pliku znajdują się pod adresem <http://docs.oracle.com/javase/7/docs/technotes/guides/intl/fontconfig.html>.

Pliki czcionek można wczytywać w formatach TrueType lub PostScript type 1. Potrzebny jest do tego strumień wejściowy dla danej czcionki — zazwyczaj z pliku lub adresu URL (więcej informacji na temat strumieni znajduje się w rozdziale 1. drugiego tomu). Następnie należy wywołać statyczną metodę `Font.createFont`:

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

Zastosowana została zwykła czcionka o rozmiarze 1 punktu. Do określenia żądanego rozmiaru czcionki należy użyć metody `deriveFont`:

```
Font f = f1.deriveFont(14.0F);
```



Istnieją dwie przeciążone wersje metody `deriveFont`. Jedna z nich (przyjmująca parametr typu `float`) ustawia rozmiar czcionki, a druga (przyjmująca parametr typu `int`) ustawia styl czcionki. W związku z tym instrukcja `f1.deriveFont(14)` ustawia styl, a nie rozmiar czcionki! Styl czcionki będzie w tym przypadku kursywą, ponieważ binarna reprezentacja liczby 14 ustawia bit `ITALIC`.

Fonty Javy zawierają symbole i znaki ASCII. Na przykład znak `\u2297` fontu Dialog to znak `\otimes`. Dostępne są tylko te symbole, które zdefiniowano w zestawie znaków Unicode.

Poniższy fragment programu wyświetla napis *Witaj, świecie!* standardową czcionką bezszyfrową systemu z zastosowaniem pogrubienia i o rozmiarze 14 punktów:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Witaj, świecie!";
g2.drawString(message, 75, 100);
```

Teraz **wypośrodkujemy** nasz napis w zawierającym go komponencie. Do tego celu potrzebne są informacje o szerokości i wysokości łańcucha w pikselach. O wymiarach tych decydują trzy czynniki:

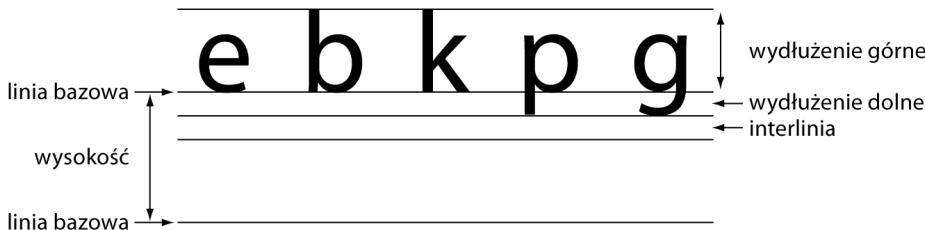
- czcionka (w tym przypadku jest to pogrubiona czcionka bezszeryfowa o rozmiarze 14 punktów),
- łańcuch (w tym przypadku *Witaj, świecie!*),
- urządzenie, na którym łańcuch będzie wyświetlany (w tym przypadku ekran monitora).

Obiekt reprezentujący własności czcionki urządzenia z ekranem tworzymy za pomocą metody `getFontRenderContext` z klasy `Graphics2D`. Zwraca ona obiekt klasy `FontRenderContext`. Obiekt ten należy przekazać metodzie `getStringBounds` z klasy `Font`:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

Metoda `getStringBounds` zwraca prostokąt, w którym mieści się łańcuch.

Do interpretacji wymiarów tego prostokąta potrzebna jest znajomość podstawowych pojęć z zakresu składu tekstów (zobacz rysunek 7.13). **Linia bazowa** (ang. *baseline*) to teoretyczna linia, na której opiera się dolna część liter, np. *e*. **Wydłużenie górne** (ang. *ascent*) to odstęp dzielący linię bazową i **linię górną pisma** (ang. *ascender*), która określa górną granicę liter, takich jak *b* lub *k* czy też wielkich liter. **Wydłużenie dolne** (ang. *descent*) to odległość pomiędzy linią bazową a **linią dolną pisma** (ang. *descender*), która stanowi granicę dolnej części takich liter jak *p* lub *g*.



Rysunek 7.13. Pojęcia z zakresu składu tekstów

Interlinia (ang. *leading*) to odstęp pomiędzy wydłużeniem dolnym jednej linii a wydłużeniem górnym następnej linii (termin pochodzi od pasków ołówku używanych przez zecerów do oddzielania linii). **Wysokość** (ang. *height*) czcionki to odległość pomiędzy następującymi po sobie liniami bazowymi i jest równa sumie wydłużenia dolnego, leadingu i wydłużenia górnego.

Szerokość prostokąta zwracanego przez metodę `getStringBounds` określa szerokość tekstu. Wysokość natomiast jest równa sumie wydłużenia dolnego, leadingu i wydłużenia górnego. Prostokąt ma swój początek na linii bazowej łańcucha. Góra współrzędna `y` prostokąta ma wartość ujemną. W związku z tym szerokość, wysokość i wydłużenie górne łańcucha można sprawdzić następująco:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Aby sprawdzić wydłużenie dolne lub leading, należy użyć metody `getLineMetrics` klasy `Font`. Zwraca ona obiekt klasy `LineMetrics`, dysponujący metodami do sprawdzania wydłużenia dolnego i leadingu:

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

W poniższym fragmencie programu wykorzystano wszystkie opisane powyżej informacje do umieszczenia łańcucha na środku zawierającego go komponentu:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x, y) = lewy górnny róg tekstu
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// Dodanie wydłużenia górnego do y w celu sięgnięcia do linii bazowej.
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

Aby ułatwić sobie zrozumienie techniki wyśrodkowywania tekstu, warto sobie uzmysolić, że metoda `getWidth()` zwraca szerokość komponentu. Pewna część tej przestrzeni, `bounds.getWidth()`, jest zajmowana przez tekst. Reszta powinna być podzielona na dwie równe części, rozmieszczone po obu stronach tekstu. Ten sam sposób rozumowania dotyczy wysokości.



Kiedy konieczne jest obliczenie wymiarów układu bez użycia metody `paintComponent`, nie można uzyskać obiektu obrazowania czcionki typu `Graphics2D`. W zamian należy wywołać metodę `getFontMetrics` klasy `JComponent`, a następnie metodę `getFontRenderContext`.

```
FontRenderContext context = getFontMetrics(f).getFontRenderContext();
```

Przykładowy program przedstawiony poniżej nie tylko drukuje napis, ale także linię bazową i prostokąt otaczający napis. Rysunek 7.14 przedstawia wynik działania tego programu. Listing 7.5 zawiera jego kod.

Rysunek 7.14.
Linia bazowa
i prostokąt
otaczający
łańcuch



Listing 7.5. font/FontTest.java

```
package font;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.33 2007-04-14

```

```
* @author Cay Horstmann
*/
public class FontTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new FontFrame();
                frame.setTitle("FontTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka z komponentem zawierającym tekst.
 */
class FontFrame extends JFrame
{
    public FontFrame()
    {
        add(new FontComponent());
        pack();
    }
}

/**
 * Komponent z tekstem w ramce na środku.
 */
class FontComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        String message = "Witaj, świecie!";

        Font f = new Font("Serif", Font.BOLD, 36);
        g2.setFont(f);

        // Sprawdzenie rozmiaru tekstu.

        FontRenderContext context = g2.getFontRenderContext();
        Rectangle2D bounds = f.getStringBounds(message, context);

        // set (x, y) = lewy górnny róg tekstu

        double x = (getWidth() - bounds.getWidth()) / 2;
        double y = (getHeight() - bounds.getHeight()) / 2;
```

```

// Dodanie wydłużenia górnego do y w celu sięgnięcia do linii bazowej.

double ascent = -bounds.getY();
double baseY = y + ascent;

// Rysowanie komunikatu.

g2.drawString(message, (int) x, (int) baseY);

g2.setPaint(Color.LIGHT_GRAY);

// Rysowanie linii bazowej.

g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));

// Rysowanie otaczającego tekst prostokąta.

Rectangle2D rect = new Rectangle2D.Double(x, y, bounds.getWidth(),
    bounds.getHeight());
g2.draw(rect);
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
    DEFAULT_HEIGHT); }
}

```

java.awt.Font 1.0

- `Font(String name, int style, int size)`

Tworzy obiekt reprezentujący czcionkę.

Parametry:	<code>name</code>	Nazwa czcionki — może być nazwa typu Helvetica Bold lub logiczna nazwa typu Serif lub SansSerif
	<code>style</code>	Styl: <code>Font.PLAIN</code> , <code>Font.BOLD</code> , <code>Font.ITALIC</code> lub <code>Font.BOLD + Font.ITALIC</code>
	<code>size</code>	Rozmiar w punktach (na przykład 12)

- `String getFontName()`

Pobiera nazwę czcionki (typu Helvetica Bold).

- `String getFamily()`

Pobiera nazwę rodziny czcionek (np. Helvetica).

- `String getName()`

Pobiera nazwę logiczną (np. SansSerif), jeśli czcionka została utworzona z nazwy logicznej. W przeciwnym przypadku zwraca nazwę czcionki.

- `Rectangle 2D getStringBounds(String s, FontRenderContext context) 1.2`

Zwraca prostokąt otaczający łańcuch. Prostokąt ma swój początek na linii bazowej łańcucha. Górna współrzędna y prostokąta ma wartość równą odwrotności wydłużenia górnego. Wysokość prostokąta jest równa sumie wydłużenia górnego, dolnego i leadingu. Szerokość jest równa szerokości tekstu.

- `LineMetrics getLineMetrics(String s, FontRenderContext context)` **1.2**
Zwraca ona obiekt klasy `LineMetrics` dysponujący metodami do sprawdzania wydłużenia dolnego i leadingu.
- `Font deriveFont(int style)` **1.2**
- `Font deriveFont(float size)` **1.2**
- `Font deriveFont(int style, float size)` **1.2**
Zwraca nową czcionkę różniącą się od aktualnej tylko rozmiarem podanym jako argument.

java.awt.font.LineMetrics **1.2**

- `float getAscent()`
Pobiera wydłużenie górne czcionki — odległość linii bazowej od wierzchołków wielkich liter.
- `float getDescent()`
Pobiera wydłużenie dolne czcionki — odległość linii bazowej od podstaw liter sięgających dolnej linii pisma.
- `float getLeading()`
Pobiera leading czcionki — odstęp pomiędzy spodem jednej linii tekstu a wierzchołkiem następnej.
- `float getHeight()`
Pobiera całkowitą wysokość czcionki — odległość pomiędzy dwiema liniami bazowymi tekstu (wydłużenie dolne + leading + wydłużenie górne).

java.awt.Graphics **1.0**

- `Font getFont()`
- `void setFont(Font font)`
Pobiera lub ustawia czcionkę. Czcionka ta będzie stosowana w kolejnych operacjach rysowania tekstu.
- Parametry:** `font` Czcionka
- `void drawString(String str, int x, int y)`
Rysuje łańcuch przy użyciu aktualnej czcionki i koloru.
- Parametry:** `str` Łańcuch
- `x` Współrzędna x początku łańcucha
- `y` Współrzędna y linii bazowej łańcucha

java.awt.Graphics2D 1.2

- `FontRenderContext getFontRenderContext()`

Pobiera kontekst wizualizacji czcionki, który określa cechy czcionki w kontekście graficznym.

- `void drawString(String str, float x, float y)`

Rysuje łańcuch przy zastosowaniu aktualnej czcionki i koloru.

Parametry: str Łańcuch

x Współrzędna x początku łańcucha

y Współrzędna y linii bazowej łańcucha

javax.swing.JComponent 1.2

- `FontMetrics getFontMetrics(Font f) 5.0`

Pobiera cechy czcionki. Klasa `FontMetrics` jest prekursorem klasy `LineMetrics`.

java.awt.FontMetrics 1.0

- `FontRenderContext getFontRenderContext() 1.2`

Pobiera kontekst wizualizacji czcionki.

7.8. Wyświetlanie obrazów

Poznaliśmy techniki tworzenia prostych rysunków składających się z linii i figur geometrycznych. Bardziej złożone obrazy, jak zdjęcia, mają zazwyczaj inne pochodzenie, np. przenosi się je do komputera za pomocą skanera lub wytwarza w wyspecjalizowanym do tego celu oprogramowaniu. W drugim tomie nauczymy się tworzyć obrazy złożone z pojedynczych pikseli zapisanych w tablicy — technika ta jest często używana na przykład podczas tworzenia obrazów fraktalnych.

Obrazy zapisane w postaci plików na dysku lub w internecie można wczytać do aplikacji w Javie i wyświetlić na obiektach `Graphics`. Obrazy można wczytywać na wiele sposobów. W poniższym przykładzie użyta jest znana nam już klasa `ImageIcon`:

```
Image image = new ImageIcon(filename).getImage();
```

Zmienna `image` zawiera referencję do obiektu opakowującego obraz. Można go wyświetlić za pomocą metody `drawImage` z klasy `Graphics`:

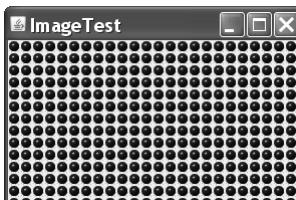
```
public void paintComponent(Graphics g)
{
    ...
    g.drawImage(image, x, y, null);
}
```

Program z listingu 7.6 robi nawet więcej, ponieważ wypełnia całe okno wieloma obrazami. Rezultat tego widać na rysunku 7.15. Za to kaskadowe wypełnienie odpowiedzialna jest metoda `paintComponent`. Najpierw rysujemy jeden obraz w lewym górnym rogu, a następnie zapełniamy całe okno za pomocą metody `copyArea`:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
    for (int j = 0; j * imageHeight <= getHeight(); j++)
        if (i + j > 0)
            g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
```

Rysunek 7.15.

Okno wypełnione kopiami jednego obrazu



Listing 7.6 przedstawia pełny kod źródłowy opisywanego programu.

Listing 7.6. image/ImageTest.java

```
package image;

import java.awt.*;
import javax.swing.*;

/**
 * @version 1.33 2007-04-14
 * @author Cay Horstmann
 */
public class ImageTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ImageFrame();
                frame.setTitle("ImageTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka zawierająca komponent obrazu.
     */
    class ImageFrame extends JFrame
    {
        public ImageFrame()
        {
            add(new ImageComponent());
        }
    }
}
```

```

        pack();
    }

}

/*
 * Komponent wyświetlający powielony obraz.
 */
class ImageComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private Image image;

    public ImageComponent()
    {
        image = new ImageIcon("blue-ball.gif").getImage();
    }

    public void paintComponent(Graphics g)
    {
        if (image == null) return;

        int imageWidth = image.getWidth(this);
        int imageHeight = image.getHeight(this);

        // Rysowanie obrazu w lewym górnym rogu.
        g.drawImage(image, 0, 0, null);
        // Powielenie obrazu w obrębie komponentu.

        for (int i = 0; i * imageWidth <= getWidth(); i++)
            for (int j = 0; j * imageHeight <= getHeight(); j++)
                if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j
                                           * imageHeight);
    }

    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
                                                               DEFAULT_HEIGHT); }
}


```

java.awt.Graphics **1.0**

■ `boolean drawImage(Image img, int x, int y, ImageObserver observer)`

Rysuje obraz w naturalnym rozmiarze. Uwaga: to wywołanie może zwrócić wartość przed narysowaniem obrazu.

Parametry:	<code>img</code>	Obraz do narysowania
	<code>x</code>	Współrzędna x lewego górnego rogu
	<code>y</code>	Współrzędna y lewego górnego rogu
	<code>observer</code>	Obiekt powiadający o postępie procesu wizualizacji (może być wartość null)

- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`

Rysuje obraz o zmienionych wymiarach. System dopasowuje rozmiar obrazu do obszaru o podanej szerokości i wysokości. Uwaga: to wywołanie może zwrócić wartość przed narysowaniem obrazu.

Parametry:	<code>img</code>	Obraz do narysowania
	<code>x</code>	Współrzędna x lewego górnego rogu
	<code>y</code>	Współrzędna y lewego górnego rogu
	<code>width</code>	Szerokość obrazu
	<code>height</code>	Wysokość obrazu
	<code>observer</code>	Obiekt powiadamiający o postępie procesu wizualizacji (może być wartość <code>null</code>)

- `void copyArea(int x, int y, int width, int height, int dx, int dy)`

Kopiuje obszar ekranu.

Parametry:	<code>x</code>	Współrzędna x lewego górnego rogu obszaru źródłowego
	<code>y</code>	Współrzędna y lewego górnego rogu obszaru źródłowego
	<code>width</code>	Szerokość obszaru źródłowego
	<code>height</code>	Wysokość obszaru źródłowego
	<code>dx</code>	Odległość w poziomie od obszaru źródłowego do obszaru docelowego
	<code>dy</code>	Odległość w pionie od obszaru źródłowego do obszaru docelowego

Na tym zakończymy wprowadzenie do grafiki w Javie. Bardziej zaawansowane techniki, takie jak grafika 2D i obróbka obrazów, zostały opisane w drugim tomie. W kolejnym rozdziale dowiemy się, jak programy reagują na dane wprowadzane przez użytkownika.

8

Obsługa zdarzeń

W tym rozdziale:

- Podstawy obsługi zdarzeń
- Akcje
- Zdarzenia generowane przez mysz
- Hierarchia zdarzeń AWT

Obsługa zdarzeń ma fundamentalne znaczenie w programach z graficznym interfejsem użytkownika. Każdy, kto chce tworzyć interfejsy graficzne w Javie, musi opanować obsługę zdarzeń. Ten rozdział opisuje model obsługi zdarzeń biblioteki AWT. Do opisywanych zagadnień należą przechwytywanie zdarzeń w komponentach interfejsu użytkownika i urządzeniach wejściowych, a także **akcje** (ang. *actions*), czyli bardziej strukturalna metoda przetwarzania zdarzeń.

8.1. Podstawy obsługi zdarzeń

Każdy system operacyjny posiadający graficzny interfejs użytkownika stale monitoruje zachodzące w nim zdarzenia, jak naciśkanie klawiszy na klawiaturze czy kliknięcia przyciskiem myszy. Informacje o tych zdarzeniach są przesyłane do uruchomionych programów. Następnie każdy program podejmuje samodzielna decyzję, w jaki sposób, jeśli w ogóle, zareagować na te zdarzenia. W takich językach jak Visual Basic relacje pomiędzy zdarzeniami a kodem są oczywiste. Programista pisze kod obsługi każdego interesującego go zdarzenia i umieszcza go w tzw. **procedurze obsługi zdarzeń** (ang. *event procedure*). Na przykład z przyciskiem o nazwie `HelpButton` w języku Visual Basic może być skojarzona procedura obsługi zdarzeń o nazwie `HelpButton_Click`. Kod tej procedury jest wykonywany w odpowiedzi na każde kliknięcie tego przycisku. Każdy komponent GUI w języku Visual Basic reaguje na ustalony zestaw zdarzeń — nie można zmienić zdarzeń, na które reaguje dany komponent.

Natomiast programiści czystego języka C zajmujący się zdarzeniami muszą pisać procedury nieprzerwanie monitorujące kolejkę zdarzeń w celu sprawdzenia, jakie powiadomienia przesyła system operacyjny (z reguły do tego celu stosuje się pętlę zawierającą bardzo rozbudowaną instrukcję switch!). Technika ta jest oczywiście bardzo mało elegancka i sprawia wiele problemów podczas pisania kodu. Jej zaletą jest natomiast to, że nie ma żadnych ograniczeń dotyczących zdarzeń, na które można reagować, w przeciwieństwie do innych języków, np. Visual Basica, które wkładają bardzo dużo wysiłku w ukrywanie kolejki zdarzeń przed programistą.

W środowisku programistycznym Javy przyjęto podejście pośrednie pomiędzy językami Visual Basic a C, jeśli chodzi o oferowane możliwości, a co za tym idzie — także złożoność. Poruszając się w zakresie zdarzeń, które obsługuje biblioteka AWT, programista ma pełną kontrolę nad sposobem przesyłania zdarzeń ze **źródeł zdarzeń** (ang. *event sources*), np. przycisków lub pasków przewijania, do **słuchaczy zdarzeń** (ang. *event listener*). Na słuchacza zdarzeń można desygnować **każdy** obiekt — w praktyce wybiera się ten obiekt, który z łatwością może wykonać odpowiednie działania w odpowiedzi na zdarzenie. **Ten delegacyjny model zdarzeń** daje znacznie większe możliwości niż język Visual Basic, w którym słuchacz jest ustalony z góry.

Źródła zdarzeń dysponują metodami, w których można rejestrować słuchaczy zdarzeń. Kiedy ma miejsce określone zdarzenie, źródło wysyła powiadomienie o nim do wszystkich obiektów nasłuchujących, które zostały dla niego zarejestrowane.

Jak można się spodziewać, informacje o zdarzeniu w języku obiektowym, takim jak Java, są pakowane w **obiekcie zdarzeń** (ang. *event object*). W Javie wszystkie obiekty zdarzeń należą do klasy `java.util.EventObject`. Oczywiście istnieją też podklasy reprezentujące każdy typ zdarzenia, takie jak `ActionEvent` czy `WindowEvent`.

Różne źródła zdarzeń mogą dostarczać różnego rodzaju zdarzeń. Na przykład przycisk może wysyłać obiekty `ActionEvent`, podczas gdy okno wysyła obiekty `WindowEvent`.

Podsumujmy, co już wiemy na temat obsługi zdarzeń w bibliotece AWT:

- Obiekt nasłuchujący jest egzemplarzem klasy implementującej specjalny **interfejs nasłuchu** (ang. *listener interface*).
- Źródło zdarzeń to obiekt, który może rejestrować obiekty nasłuchujące i wysyłać do nich obiekty zdarzeń.
- Źródło zdarzeń wysyła obiekty zdarzeń do wszystkich zarejestrowanych słuchaczy w chwili wystąpienia zdarzenia.
- Informacje zawarte w obiekcie zdarzeń są wykorzystywane przez obiekty nasłuchujące przy podejmowaniu decyzji dotyczącej reakcji na zdarzenie.

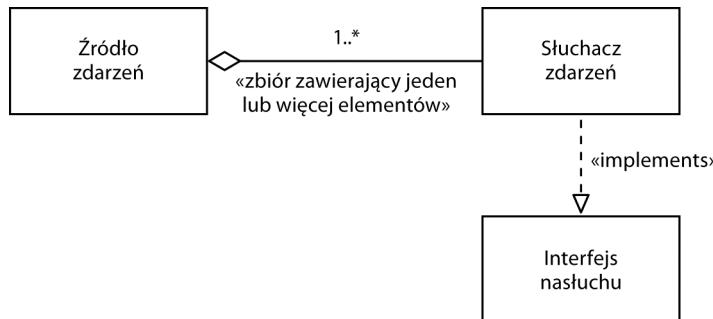
Rysunek 8.1 przedstawia relacje pomiędzy klasami obsługi zdarzeń a interfejsami.

Poniżej znajduje się przykładowa definicja słuchacza:

```
ActionListener listener = . . .;  
 JButton button = new JButton("OK");  
 button.addActionListener(listener);
```

Rysunek 8.1.

Relacje pomiędzy źródłami zdarzeń a słuchaczami



Od tej pory obiekt `listener` będzie powiadamiany o każdym zdarzeniu akcji w przycisku. Jak się można domyślić, zdarzenie akcji w przypadku przycisku to jego kliknięcie.

Klasa implementująca interfejs `ActionListener` musi definiować metodę o nazwie `actionPerformed`, która jako parametr przyjmuje obiekt typu `ActionEvent`:

```

class MyListener implements ActionListener
{
    ...
    public void actionPerformed(ActionEvent event)
    {
        // Instrukcje wykonywane w odpowiedzi na kliknięcie przycisku.
        ...
    }
}
  
```

Kiedy użytkownik kliknie przycisk, obiekt typu `JButton` tworzy obiekt typu `ActionEvent` i wywołuje metodę `listener.actionPerformed(event)`, przekazując do niej ten obiekt zdarzenia. Źródło zdarzeń, takie jak przycisk, może mieć kilka słuchaczy. W takim przypadku kliknięcie przycisku przez użytkownika powoduje wywołanie metod `actionPerformed` wszystkich słuchaczy.

Rysunek 8.2 przedstawia relacje pomiędzy źródłem zdarzeń, słuchaczem zdarzeń i obiektem zdarzeń.

8.1.1. Przykład — obsługa kliknięcia przycisku

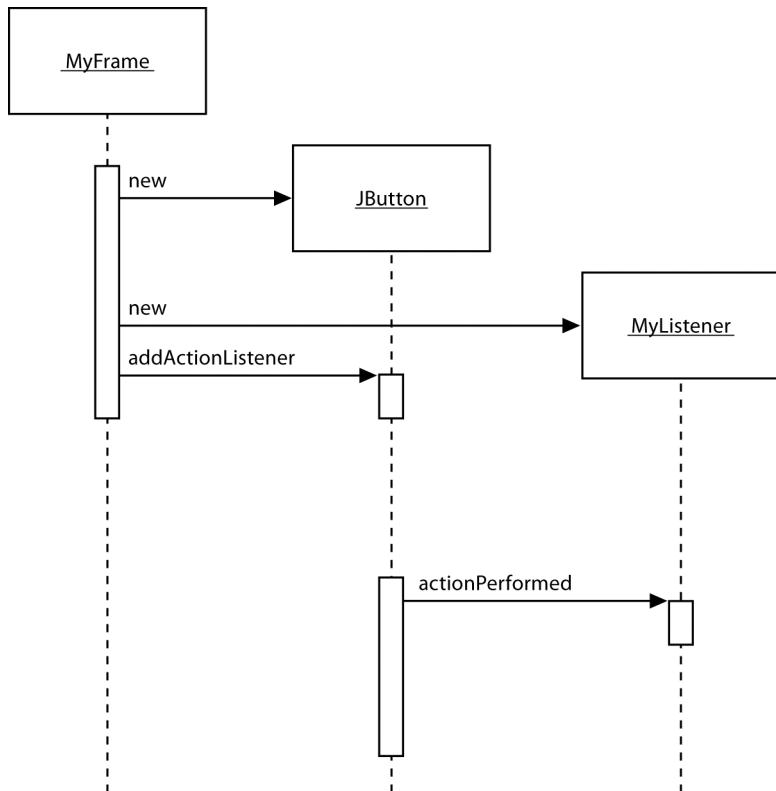
Aby nabrać biegłości w posługiwaniu się modelem delegacji zdarzeń, przeanalizujemy szczegółowo prosty program reagujący na kliknięcie przycisku. Utworzymy panel zawierający trzy przyciski, których zdarzeń będą nasłuchiwać trzy obiekty nasłuchujące.

W tym przypadku za każdym razem, gdy użytkownik kliknie jeden z przycisków na panelu, skojarzony z tym przyciskiem obiekt odbierze obiekt typu `ActionEvent` oznaczający kliknięcie przycisku. W odpowiedzi obiekt nasłuchujący zmieni kolor tła panelu.

Przed przejściem do programu, który nasłuchuje kliknięć przycisków, musimy najpierw zapoznać się z techniką tworzenia i dodawania przycisków do panelu (więcej informacji na temat elementów GUI znajduje się w rozdziale 9.).

Rysunek 8.2.

Powiadamianie o zdarzeniach



Tworzenie przycisku polega na podaniu jego konstruktorowi łańcucha określającego etykietę przycisku, ikony lub jednego i drugiego. Poniżej znajdują się przykłady tworzenia dwóch przycisków:

```

 JButton yellowButton = new JButton("Żółty");
 JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
  
```

Przyciski do panelu dodaje się za pomocą metody add:

```

 JButton yellowButton = new JButton("Żółty");
 JButton blueButton = new JButton("Niebieski");
 JButton redButton = new JButton("Czerwony");

 buttonPanel.add(yellowButton);
 buttonPanel.add(blueButton);
 buttonPanel.add(redButton);
  
```

Wynik powyższych działań przedstawia rysunek 8.3.

Następnie konieczne jest dodanie procedur nasłuchujących tych przycisków. Do tego potrzebne są klasy implementujące interfejs ActionListener, który — jak już wspominaliśmy — zawiera tylko jedną metodę: actionPerformed. Sygnatura tej metody jest następująca:

```

 public void actionPerformed(ActionEvent event)
  
```

Rysunek 8.3.

Panel
z przyciskami



Interfejs ActionListener nie ogranicza się tylko do kliknięć przycisków. Znajduje on zastosowanie w wielu innych sytuacjach, takich jak:

- wybór elementu z pola listy za pomocą dwukrotnego kliknięcia,
- wybór elementu menu,
- kliknięcie klawisza *Enter* w polu tekstowym,
- upływ określonej ilości czasu dla komponentu Timer.

Więcej szczegółów na ten temat znajduje się w tym i kolejnym rozdziale.

Sposób użycia interfejsu ActionListener jest taki sam we wszystkich sytuacjach: metoda `actionPerformed` (jedyna w interfejsie ActionListener) przyjmuje obiekt typu `ActionEvent` jako parametr. Ten obiekt zdarzenia dostarcza informacji o zdarzeniu, które miało miejsce.

Reakcją na kliknięcie przycisku ma być zmiana koloru tła panelu. Zadany kolor będziemy przechowywać w klasie nasłuchującej:

```
class ColorAction implements ActionListener
{
    private Color backgroundColor;

    public ColorAction(Color c)
    {
        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        // Ustawienie koloru tła panelu.
        . . .
    }
}
```

Następnie dla każdego koloru tworzymy osobny obiekt i każdy z nich rejestrujemy jako słuchacza przycisku.

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Jeśli użytkownik kliknie na przykład przycisk z napisem Żółty, zostanie wywołana metoda `actionPerformed` obiektu `yellowAction`. Pole `backgroundColor` tego obiektu ma wartość `color.YELLOW`.

Został jeszcze tylko jeden problem do rozwiązania. Obiekt typu `ColorAction` nie ma dostępu do zmiennej `buttonPanel`. Można to rozwiązać na jeden z dwóch sposobów. Można zapisać panel w obiekcie `ColorAction` i skonstruować go w konstruktorze `ColorAction`. Wygodniej jednak byłoby, gdyby `ColorAction` była klasą wewnętrzną klasy `ButtonFrame`. Dzięki temu jej metody miałyby automatycznie dostęp do zewnętrznego panelu (więcej informacji na temat klas wewnętrznych znajduje się w rozdziale 6.).

Zastosujemy drugą z opisanych metod. Poniżej przedstawiamy klasę `ColorAction` wewnętrzna klasy `ButtonFrame`:

```
class ButtonPanel extends JFrame
{
    private JPanel buttonPanel;
    ...
    private class ColorAction implements ActionListener
    {
        private Color backgroundColor;

        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    }
}
```

Przypatrzmy się uważniej metodzie `actionPerformed`. Klasa `ColorAction` nie posiada pola `buttonPanel`. Ma go natomiast zewnętrzna klasa `ButtonFrame`.

Jest to bardzo często spotykana sytuacja. Obiekty nasłuchu zdarzeń często muszą wykonywać działania, które mają wpływ na inne obiekty. Klasę nasłuchującą często można umieścić w strategicznym miejscu wewnętrz klasy, której obiekt ma mieć zmieniony stan.

Listing 8.1 przedstawia kompletny program. Kliknięcie jednego z przycisków powoduje zmianę koloru tła panelu przez odpowiedniego słuchacza akcji.

Listing 8.1. button/ButtonFrame.java

```
package button;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z panelem zawierającym przyciski
 */
public class ButtonFrame extends JFrame
{
    private JPanel buttonPanel;
    private static final int DEFAULT_WIDTH = 300;
```

```

private static final int DEFAULT_HEIGHT = 200;

public ButtonFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    // Tworzenie przycisków
    JButton yellowButton = new JButton("Żółty");
    JButton blueButton = new JButton("Niebieski");
    JButton redButton = new JButton("Czerwony");

    buttonPanel = new JPanel();

    // Dodanie przycisków do panelu
    buttonPanel.add(yellowButton);
    buttonPanel.add(blueButton);
    buttonPanel.add(redButton);

    // Dodanie panelu do ramki
    add(buttonPanel);

    // Utworzenie akcji przycisków
    ColorAction yellowAction = new ColorAction(Color.YELLOW);
    ColorAction blueAction = new ColorAction(Color.BLUE);
    ColorAction redAction = new ColorAction(Color.RED);

    // Powiązanie akcji z przyciskami
    yellowButton.addActionListener(yellowAction);
    blueButton.addActionListener(blueAction);
    redButton.addActionListener(redAction);
}

/**
* Słuchacz akcji ustawiający kolor dla panelu.
*/
private class ColorAction implements ActionListener
{
    private Color backgroundColor;

    public ColorAction(Color c)
    {
        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        buttonPanel.setBackground(backgroundColor);
    }
}

```

javax.swing.JButton 1.2

- JButton(String label)
- JButton(Icon icon)

- JButton(String label, Icon icon)

Tworzy przycisk. Łańcuch etykiety może zawierać sam tekst lub (od Java SE 1.3) kod HTML, np. "<html>0k</html>".

java.awt.Container **1.0**

- Component add(Component c)

Dodaje komponent c do kontenera.

8.1.2. Nabywanie biegłości w posługiwaniu się klasami wewnętrznymi

Niektórzy programiści nie przepadają za klasami wewnętrznymi, ponieważ uważają, że klasy i obiekty o dużych rozmiarach spowalniają działanie programu. Przyjrzyjmy się temu twierdzeniu. Nie potrzebujemy nowej klasy dla każdego elementu interfejsu użytkownika. W naszym programie wszystkie trzy przyciski współdzielą jedną klasę nasłuchującą. Oczywiście każdy z nich posiada osobny obiekt nasłuchujący. Ale obiekty te nie są duże. Każdy z nich zawiera wartość określającą kolor i referencję do panelu. A tradycyjne rozwiązywanie z zastosowaniem instrukcji `if-else` również odwołuje się do tych samych obiektów kolorów przechowywanych przez słuchaczy akcji, tylko że jako zmienne lokalne, a nie pola obiektów.

Poniżej przedstawiamy dobry przykład tego, jak anonimowe klasy wewnętrzne mogą uprościć kod programu. W programie na listingu 8.1 z każdym przyciskiem związane są takie same działania:

- 1 Utworzenie przycisku z etykietą.
- 2 Dodanie przycisku do panelu.
- 3 Utworzenie słuchacza akcji z odpowiednim kolorem.
- 4 Dodanie słuchacza akcji.

Napiszemy metodę pomocniczą, która będzie upraszczała te czynności:

```
public void makeButton(String name, Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    ColorAction action = new ColorAction(backgroundColor);
    button.addActionListener(action);
}
```

Teraz wystarczą tylko następujące wywołania:

```
makeButton("Złoty", Color.YELLOW);
makeButton("Niebieski", Color.BLUE);
makeButton("Czerwony", Color.RED);
```

Możliwe są dalsze uproszczenia. Zauważmy, że klasa `ColorAction` jest potrzebna tylko **jeden raz** — w metodzie `makeButton`. A zatem można ją przerobić na klasę anonimową:

```

public void makeButton(String name, final Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    });
}

```

Kod słuchacza akcji stał się znacznie prostszy. Metoda actionPerformed odwołuje się do zmiennej parametrycznej backgroundColor (podobnie jak w przypadku wszystkich zmiennych lokalnych wykorzystywanych w klasie wewnętrznej, parametr ten musi być finalny).

Nie jest potrzebny żaden jawnego konstruktora. Jak widzieliśmy w rozdziale 6., mechanizm klas wewnętrznych automatycznie generuje konstruktor zapisujący wszystkie finalne zmienne lokalne, które są używane w jednej z metod klasy wewnętrznej.



Anonimowe klasy wewnętrzne potrafią zmylić niejednego programistę. Można jednak przyzwyczać się do ich rozszyfrowywania, wyrabiając sobie umiejętność pomijania wzrokiem kodu procedury:

```

button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        buttonPanel.setBackground(backgroundColor);
    }
});

```

Akcja przycisku ustawia kolor tła. Dopóki procedura obsługi zdarzeń składa się z tylko kilku instrukcji, wydaje się, że z odczytem nie powinno być problemów, zwłaszcza jeśli w sferze naszych zainteresowań nie leżą mechanizmy klas wewnętrznych.

java.util.EventObject 1.1

- Object getSource()

Zwraca referencję do obiektu, w którym wystąpiło zdarzenie.

java.awt.event.ActionEvent 1.1

- String getActionCommand()

Zwraca łańcuch polecenia skojarzonego z danym zdarzeniem akcji. Jeśli zdarzenie pochodzi od przycisku, łańcuch polecenia jest taki sam jak etykieta przycisku, chyba że został zmieniony za pomocą metody setActionCommand.



Słuchaczem przycisku może być obiekt **każdej** klasy, która implementuje interfejs ActionListener. My wolimy używać obiektów nowej klasy, która została utworzona specjalnie z myślą o wykonywaniu akcji przycisku. Jednak niektórzy programiści nie czują się pewnie w stosowaniu klas wewnętrznych i wybierają inne podejście. Implementują interfejs ActionListener w kontenerze źródła zdarzeń. Następnie kontener ten ustawia **sam siebie** jako słuchacza w następujący sposób:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

W tej sytuacji żaden z trzech przycisków nie ma osobnego słuchacza. Dysponują one wspólnym obiektem, którym jest ramka przycisku. W związku z tym metoda actionPerformed musi sprawdzić, który przycisk został kliknięty.

```
class ButtonFrame extends JFrame implements ActionListener
{
    ...
    public void actionPerformed(ActionEvent event)
    {
        Object source = event.getSource();
        if (source == yellowButton) ...
        else if (source == blueButton) ...
        else if (source == redButton) ...
        else ...
    }
}
```

Jak widać, metoda ta jest nieco zagmatwana, przez co nie zalecamy jej stosowania.

8.1.3. Tworzenie słuchaczy zawierających jedno wywołanie metody

Prostych słuchaczy zdarzeń można tworzyć bez użycia klas wewnętrznych. Wyobraźmy sobie na przykład, że mamy przycisk z etykietą *Load*, którego procedura obsługi zdarzeń zawiera tylko jedną metodę:

```
frame.loadData();
```

Oczywiście można użyć anonimowej klasy wewnętrznej:

```
loadButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData();
    }
});
```

Jednak klasa EventHandler może utworzyć takiego słuchacza automatycznie za pomocą następującego wywołania:

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Oczywiście nadal konieczne jest zainstalowanie procedury obsługi:

```
loadButton.addActionListener(
    EventHandler.create(ActionListener.class, frame, "loadData"));
```

Jeśli słuchacz wywołuje metodę z jednym parametrem, który można uzyskać z parametru zdarzenia, można użyć innego rodzaju metody `create`. Na przykład wywołanie:

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

jest równoznaczne z:

```
new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData(((JTextField) event.getSource()).getText());
    }
}
```

Nazwy właściwości `source` i `text` zamieniają się w wywołaniu metod `getSource` i `getText`.



Drugi argument w wywołaniu `EventHandler.create` musi należeć do klasy **publicznej**. W przeciwnym razie mechanizm refleksji nie będzie w stanie znaleźć i wywołać docelowej metody.

java.beans.EventHandler 1.4

- `static Object create(Class listenerInterface, Object target, String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

Tworzy obiekt klasy pośredniczącej implementującej dany interfejs. Albo podana metoda, albo wszystkie metody interfejsu wykonują dane akcje na rzecz obiektu docelowego.

Akcją może być metoda lub właściwość obiektu docelowego. Jeśli jest to właściwość, wykonywana jest jej metoda ustawiająca. Na przykład akcja `text` jest zamieniana na wywołanie metody `setText`.

Właściwość zdarzenia składa się z jednej lub większej liczby nazw właściwości oddzielonych kropkami. Pierwsza właściwość jest wczytywana z parametru metody nasłuchującej. Druga właściwość pochodzi od powstałego obiektu itd. Wynik końcowy staje się parametrem akcji. Na przykład właściwość `source.text` jest zamieniana na wywołanie metod `getSource` i `getText`.

8.1.4. Przykład — zmiana stylu

Domyślnym stylem programów pisanych przy użyciu Swinga jest Metal. Istnieją dwa sposoby na zmianę stylu. Pierwszy z nich polega na utworzeniu pliku *swing.properties* w katalogu *jre/lib* w miejscu instalacji Javy. W pliku tym należy ustawić własność *swing.defaultlaf* na nazwę klasy stylu, który chcemy zastosować. Na przykład:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Zauważmy, że styl Metal jest zlokalizowany w pakiecie *javax.swing*. Pozostałe style znajdują się w pakiecie *com.sun.java* i nie muszą być obecne w każdej implementacji Javy. Obecnie ze względu na prawa autorskie pakiety stylów systemów Windows i Mac OS X są dostępne wyłącznie z wersjami środowiska uruchomieniowego Javy przeznaczonymi dla tych systemów.



Ponieważ w plikach własności linie zaczynające się od znaku `#` są ignorowane, można w takim pliku umieścić kilka stylów i wybierać je wedle upodobania, odpowiednio zmieniając położenie znaku `#`:

```
#swing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Aby zmienić styl w ten sposób, konieczne jest ponowne uruchomienie programu. Programy Swing wczytują plik *swing.properties* tylko jeden raz — przy uruchamianiu.

Drugi sposób polega na dynamicznej zmianie stylu. Należy wywołać statyczną metodę *UIManager.setLookAndFeel* oraz przekazać do niej nazwę klasy wybranego stylu. Następnie wywołujemy statyczną metodę *SwingUtilities.updateComponentTreeUI* w celu odświeżenia całego zbioru komponentów. Metodzie tej wystarczy przekazać tylko jeden komponent, a pozostałe znajdzie ona samodzielnie. Metoda *UIManager.setLookAndFeel* może spowodować kilka wyjątków, jeśli nie znajdzie żądanego stylu lub jeśli wystąpi błąd podczas ładowania stylu. Jak zwykle nie zgłębiamy kodu obsługującego wyjątki, ponieważ szczegółowo zajmiemy się tym w rozdziale 11.

Poniższy przykładowy fragment programu przedstawia sposób przełączenia na styl Motif:

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
    UIManager.setLookAndFeel(plaf);
    SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

Aby odnaleźć wszystkie zainstalowane style, należy użyć wywołania:

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

W takiej sytuacji nazwę każdego stylu i jego klasy można uzyskać następująco:

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

Listing 8.2 przedstawia pełny kod programu demonstrującego przełączanie stylów (zobacz rysunek 8.4). Program ten jest podobny do programu z listingu 8.1. Idąc za radą z poprzedniej sekcji, akcję przycisku, polegającą na zmianie stylu, określiliśmy za pomocą metody pomocniczej `makeButton` i anonimowej klasy wewnętrznej.

Listing 8.2. plaf/PlafFrame.java

```
package plaf;

import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z panelem zawierającym przyciski zmieniające styl.
 */
public class PlafFrame extends JFrame
{
    private JPanel buttonPanel;
    public PlafFrame()
    {
        buttonPanel = new JPanel();
        UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
        for (UIManager.LookAndFeelInfo info : infos)
            makeButton(info.getName(), info.getClassName());

        add(buttonPanel);
        pack();
    }

    /**
     * Tworzy przycisk zmieniający styl.
     * @param name nazwa przycisku
     * @param plafName nazwa klasy stylu
     */
    void makeButton(String name, final String plafName)
    {
        // Dodanie przycisku do panelu.

        JButton button = new JButton(name);
        buttonPanel.add(button);

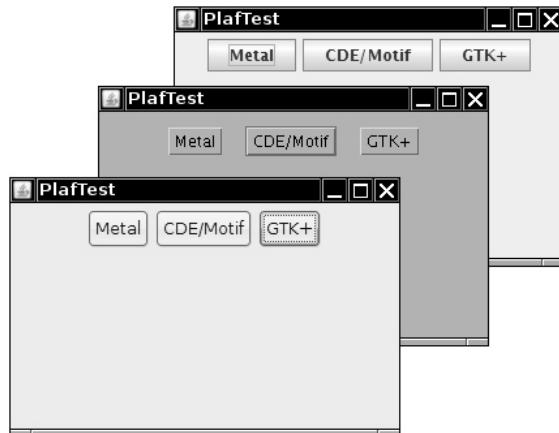
        // Ustawienie akcji przycisku.

        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                // Akcja przycisku — przełączenie na nowy styl.
                try
                {
                    UIManager.setLookAndFeel(plafName);
                    SwingUtilities.updateComponentTreeUI(PlafFrame.this);
                    pack();
                }
                catch (Exception e)
                {

```

Rysunek 8.4.

Zmienianie stylu



```
        e.printStackTrace();
    }
}
});  
}
```

Program ten ma jeden interesujący fragment. Metoda actionPerformed wewnętrznej klasy nasłuchowej musi przekazać referencję this zewnętrznej klasy PlafFrame do metody update →ComponentTreeUI. Przypomnijmy z rozdziału 6., że wskaźnik this zewnętrznego obiektu należy oznać przedrostkiem w postaci nazwy klasy zewnętrznej:

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```

javax.swing.UIManager 1.2

- static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()

Tworzy tablicę obiektów reprezentujących zainstalowane style.

- ## ■ static setLookAndFeel(String className)

Ustawia aktualny styl, wykorzystując do tego podaną nazwę klasy (np. `javax.swing.plaf.metal.MetalLookAndFeel`).

javax.swing.UIManager.LookAndFeelInfo **1.2**

- ## ■ String getName()

Zwraca nazwę stylu.

- ## ■ String getClassName()

Zwraca nazwę klasy implementującej dany styl.

8.1.5. Klasy adaptacyjne

Nie wszystkie zdarzenia są tak łatwe w obsłudze jak kliknięcie przycisku. W profesjonalnym programie należy stale sprawdzać, czy użytkownik nie zamknie głównej ramki, aby zapobiec ewentualnej utracie jego danych. Gdy użytkownik zamknie ramkę, powinno wyświetlać się okno dialogowe monitujące o potwierdzenie tego zamiaru.

Kiedy użytkownik zamknie okno, obiekt klasy JFrame jest źródłem zdarzenia WindowEvent. Aby przechwycić to zdarzenie, konieczny jest odpowiedni obiekt nasłuchujący, który należy dodać do listy słuchaczy okna ramki.

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

Obiekt nasłuchujący okna musi należeć do klasy implementującej interfejs WindowListener. Interfejs ten zawiera siedem metod. Ramka wywołuje jedną z nich w odpowiedzi na jedno z siedmiu zdarzeń, które mogą mieć miejsce w przypadku okna. Nazwy tych metod mówią same za siebie. Należy tylko wyjaśnić, że iconified w systemie Windows oznacza to samo co minimized. Poniżej widać cały interfejs WindowListener:

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```



Aby sprawdzić, czy okno zostało zmaksymalizowane, należy zainstalować obiekt WindowStateListener — zobacz wyciąg z API na końcu tej sekcji.

W Javie klasa, która implementuje dany interfejs, musi definiować wszystkie jego metody. W tym przypadku oznacza to implementację **siedmiu** metod. Przypomnijmy jednak, że interesuje nas tylko jedna z nich, o nazwie windowClosing.

Oczywiście nic nie stoi na przeszkodzie, aby zaimplementować ten interfejs, wstawić wywołanie System.exit(0) do metody windowClosing i napisać sześć nicnierobiących funkcji dla pozostałych metod:

```
class Terminator implements WindowListener
{
    public void windowClosing(WindowEvent e)
    {
        if (użytkownik potwierdza)
            System.exit(0);
    }
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
```

```
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
}
```

Pisanie sześciu metod, które nic nie robią, jest tym rodzajem pracy, której nikt nie lubi. Zadanie to ułatwiają **klasy adaptacyjne** (ang. *adapter class*) dostępne z każdym interfejsem nasłuchującym w bibliotece AWT, który ma więcej niż jedną metodę. Klasy te implementują wszystkie metody interfejsów, którym odpowiadają, ale metody te nic nie robią. Na przykład klasa `WindowAdapter` zawiera definicje siedmiu nicnieroziących metod. Oznacza to, że klasa adaptacyjna automatycznie spełnia wymagania techniczne stawiane przez Java, a dotyczące implementacji odpowiadającego jej interfejsu nasłuchującego. Klasę adaptacyjną można rozszerzyć, definiując w podklasie metody odpowiadające niektórym, ale nie wszystkim typom zdarzeń interfejsu (interfejsy, które mają tylko jedną metodę, np. `ActionListener`, nie potrzebują metod adaptacyjnych).

Rozszerzymy klasę `WindowAdapter`. Odziedziczymy po niej sześć nicnieroziących metod, a metodę `windowClosing` przesłoniemy:

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (użytkownik potwierdza)
            System.exit(0);
    }
}
```

Teraz możemy zarejestrować obiekt typu `Terminator` jako słuchacza zdarzeń:

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Każde zdarzenie okna wygenerowane przez ramkę jest przekazywane do obiektu `listener` za pomocą wywołania jednej z jego siedmiu metod (zobacz rysunek 8.5). Sześć z nich nie robi nic, a metoda `windowClosing` wywołuje metodę `System.exit(0)`, zamkując tym samym aplikację.



Jeśli w nazwie metody rozszerzanej klasy adaptacyjnej znajdzie się błąd, komputer go nie wykryje. Jeśli na przykład w klasie rozszerzającej `WindowAdapter` zostanie zdefiniowana metoda `windowIsClosing`, nowa klasa będzie zawierała osiem metod, a metoda `windowClosing` nie będzie nic robiła.

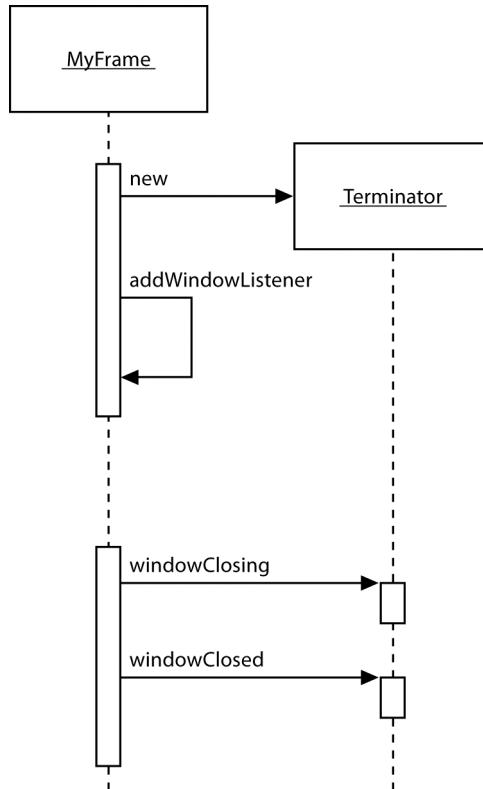
Utworzenie klasy rozszerzającej klasę adaptacyjną `WindowAdapter` jest krokiem naprzód, ale można posunąć się jeszcze dalej. Nie ma potrzeby nadawać obiektowi `listener` nazwy. Wystarczy napisać:

```
frame.addWindowListener(new Terminator());
```

Ale czemu poprzestawać na tym? Klasa nasłuchująca może być anonimową klasą wewnętrzną ramki.

Rysunek 8.5.

Obiekt
nasłuchujący
zdarzeń
dotyczących okna



```

frame.addWindowListener(new
    WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        if (uzytkownik potwierdza)
            System.exit(0);
    }
});

```

Powyzszy fragment programu ma następujące działanie:

- Definiuje klasę bez nazwy, rozszerzającą klasę WindowAdapter.
- Do utworzonej anonimowej klasy dodaje metodę windowClosing (podobnie jak wcześniej, metoda ta zamknie program).
- Dziedziczy sześć pozostałych nicnierobiących metod po klasie WindowAdapter.
- Tworzy obiekt tej nowej klasy — obiekt również nie ma nazwy.
- Przekazuje ten obiekt do metody addWindowListener.

Powtarzamy jeszcze raz, że do składni wewnętrznych klas anonimowych trzeba się przyzwyczaić. Dzięki nim można pisać tak zwięzły kod, jak to tylko możliwe.

java.awt.event.WindowListener 1.1

- `void windowOpened(WindowEvent e)`
Jest wywoływana po otwarciu okna.
- `void windowClosing(WindowEvent e)`
Jest wywoływana, kiedy użytkownik wyda polecenie menedżera okien, aby zamknąć okno. Okno zostanie zamknięte tylko wtedy, gdy zostanie wywołana jego metoda `hide` lub `dispose`.
- `void windowClosed(WindowEvent e)`
Jest wywoływana po zamknięciu okna.
- `void windowIconified(WindowEvent e)`
Jest wywoływana po zminimalizowaniu okna.
- `void windowDeiconified(WindowEvent e)`
Jest wywoływana po przywróceniu okna.
- `void windowActivated(WindowEvent e)`
Jest wywoływana po uaktywnieniu okna. Aktywna może być tylko ramka lub okno dialogowe. Z reguły menedżer okien zaznacza w jakiś sposób aktywne okno, np. podświetlając pasek tytułu.
- `void windowDeactivated(WindowEvent e)`
Jest wywoływana po dezaktywowaniu okna.

java.awt.event.WindowStateListener 1.4

- `void windowStateChanged(WindowEvent event)`
Jest wywoływana po zmaksymalizowaniu, zminimalizowaniu lub przywróceniu okna do normalnego rozmiaru.

java.awt.event.WindowEvent 1.1

- `int getNewState() 1.4`
- `int getOldState() 1.4`

Zwraca nowy i stary stan okna w zdarzeniu zmiany stanu okna. Zwracana liczba całkowita jest jedną z następujących wartości:

`Frame.NORMAL`
`Frame.ICONIFIED`
`Frame.MAXIMIZED_HORIZ`
`Frame.MAXIMIZED_VERT`
`Frame.MAXIMIZED_BOTH`

8.2. Akcje

Często jedną opcję można wybrać na kilka różnych sposobów. Użytkownik może wybrać odpowiednią funkcję w menu, nacisnąć określony klawisz lub przycisk na pasku narzędzi. Zaprogramowanie takiej funkcjonalności w modelu zdarzeń AWT jest proste — należy wszystkie zdarzenia związać z tym samym obiektem nasłuchującym. Wyobraźmy sobie, że `blueAction` jest obiektem nasłuchującym akcji, którego metoda `actionPerformed` zmienia kolor tła na niebieski. Jeden obiekt można związać jako słuchacza z kilkoma źródłami zdarzeń:

- przyciskiem paska narzędzi z etykietą *Niebieski*;
- elementem menu z etykietą *Niebieski*;
- skrótem klawiszowym *Ctrl+N*.

Dzięki temu zmiana koloru będzie wykonywana zawsze w taki sam sposób, bez znaczenia, czy wywoła ją kliknięcie przycisku, wybór elementu menu, czy naciśnięcie klawisza.

W pakiecie Swing dostępna jest niezwykle przydatna struktura opakowująca polecenia i wiążąca je z różnymi źródłami zdarzeń — interfejs `Action`. **Akcja** to obiekt, który opakowuje:

- opis polecenia (łańcuch tekstowy i opcjonalna ikona),
- parametry niezbędne do wykonania polecenia (w naszym przypadku wymagany kolor).

Interfejs `Action` zawiera następujące metody:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Pierwsza z tych metod jest już nam znana z interfejsu `ActionListener`. Należy dodać, że interfejs `Action` rozszerza interfejs `ActionListener`. W związku z tym wszędzie, gdzie powinien się znaleźć obiekt `ActionListener`, można użyć obiektu `Action`.

Dwie kolejne metody włączają i wyłączają akcję oraz sprawdzają, czy akcja jest aktualnie włączona. Kiedy akcja jest związana z menu lub paskiem narzędzi i jest wyłączona, odpowiadająca jej opcja ma kolor szary.

Metody `putValue` i `getValue` zapisują i pobierają pary nazwa – wartość z obiektu akcji. Nazwy akcji i ikony są zapisywane w obiektach akcji za pomocą dwóch predefiniowanych łańcuchów: `Action.NAME` i `Action.SMALL_ICON`:

```
action.putValue(Action.NAME, "Niebieski");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

Tabela 8.1 przedstawia zestawienie wszystkich predefiniowanych nazw tablicowych akcji.

Tabela 8.1. Predefiniowane stałe interfejsu Action

Nazwa	Wartość
NAME	Nazwa akcji — wyświetlna na przyciskach i elementach menu
SMALL_ICON	Mała ikona — może być wyświetlana na przyciskach, pasku narzędzi lub elementach menu
SHORT_DESCRIPTION	Krótki opis ikony — wyświetlany w etykiecie narzędzia
LONG_DESCRIPTION	Długi opis ikony — do użytku w pomocy internetowej; żaden komponent Swinga nie używa tej wartości
MNEMONIC_KEY	Skrót akcji — wyświetlany na elementach menu (zobacz rozdział 9.)
ACCELERATOR_KEY	Skrót klawiaturowy; żaden komponent Swinga nie używa tej wartości
ACTION_COMMAND_KEY	Używana w przestarzałej już metodzie registerKeyboardAction
DEFAULT	Własność pasująca do wszystkiego; żaden komponent Swinga nie używa tej wartości

Jeśli obiekt akcji jest dodawany do menu lub paska narzędzi, jego nazwa i ikona są automatycznie pobierane i wyświetlane w menu lub na pasku narzędzi. Wartość właściwości SHORT_DESCRIPTION zamienia się w dymek opisujący narzędzie.

Pozostałe dwie metody interfejsu Action umożliwiają powiadamianie innych obiektów, zwłaszcza menu i pasków narzędzi, które są źródłem akcji, o zmianach właściwości obiektu akcji. Jeśli na przykład menu jest dodawane jako obiekt nasłuchujący zmian właściwości obiektu akcji i obiekt ten zostanie następnie wyłączony, menu zostanie wywołane, a nazwa akcji będzie szara. Obiekty nasłuchu zmian właściwości są ogólną konstrukcją stanowiącą część modelu komponentów JavaBean. Więcej informacji na temat Beanów i ich właściwości znajduje się w drugim tomie.

Nie należy zapominać, że Action to **interfejs**, a nie klasa. Każda klasa implementująca go musi definiować wszystkie siedem metod, które opisaliśmy. Na szczęście jakiś dobry człowiek napisał klasę o nazwie `AbstractAction`, która implementuje wszystkie te metody z wyjątkiem `actionPerformed`. Klasa ta zajmuje się zapisywaniem par nazwa – wartość i zarządzaniem obiektami nasłuchującymi zmian właściwości. Wystarczy rozszerzyć klasę `AbstractAction` i zdefiniować metodę `actionPerformed`.

Utworzmy obiekt wykonujący polecenia zmiany koloru. Zapiszemy nazwę polecenia, ikonę i żądanego koloru. Kolor zapiszemy w tablicy par nazwa – wartość dostarczanej przez klasę `AbstractAction`. Poniżej znajduje się kod źródłowy klasy `ColorAction`. Konstruktor ustawia pary nazwa – wartość, a metoda `actionPerformed` wykonuje akcję zmiany koloru.

```
public class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon, Color c)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("color", c);
        putValue(Action.SHORT_DESCRIPTION, "Ustaw kolor panelu na " + name.toLowerCase());
    }
}
```

```

public void actionPerformed(ActionEvent event)
{
    Color c = (Color) getValue("color");
    buttonPanel.setBackground(c);
}
}

```

Nasz przykładowy program tworzy trzy obiekty tej klasy, np.:

```

Action blueAction = new ColorAction("Niebieski", new ImageIcon("blue-ball.gif"),
→Color.BLUE);

```

Teraz konieczne jest związywanie akcji z przyciskiem. Jest to łatwe, ponieważ możemy użyć konstruktora JButton, który przyjmuje obiekt typu Action.

```

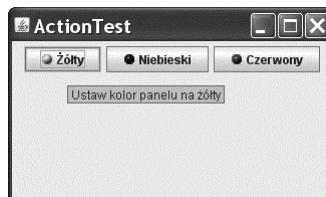
JButton blueButton = new JButton(blueAction);

```

Konstruktor odczytuje nazwę i ikonę z akcji, ustawia krótki opis jako etykietę oraz ustawia akcję jako słuchacza. Ikony i etykietę przedstawia rysunek 8.6.

Rysunek 8.6.

Przyciski zawierają ikony z obiektów akcji



W kolejnym rozdziale wykażemy, że również łatwe jest dodawanie tej samej akcji do menu.

Na koniec przypiszemy obiekty akcji do klawiszy, dzięki czemu akcje te będą wykonywane, kiedy użytkownik wpisze polecenia z klawiatury. Kojarzenie akcji z klawiszami należy zacząć od wygenerowania obiektu klasy KeyStroke. Klasa ta opakowuje opis klawisza. Do utworzenia obiektu typu KeyStroke nie używa się konstruktora, ale statycznej metody getKeyStroke klasy KeyStroke.

```

KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl N");

```

Do zrozumienia następnego etapu potrzebna jest znajomość pojęcia **aktywności komponentu** (ang. *keyboard focus*). Interfejs użytkownika może się składać z wielu przycisków, menu, pasków przewijania i innych komponentów. Kiedy zostanie naciśnięty klawisz, zdarzenie to zostaje wysłane do aktywnego komponentu. Komponent ten jest z reguły (choć nie zawsze) w jakiś sposób wizualnie wyróżniony. Na przykład w stylu Javy tekst na aktywnym przycisku ma cienką obwódkę. Fokus (aktywność komponentu) można przenosić na różne komponenty za pomocą klawisza *Tab*. Naciśnięcie klawisza spacji powoduje kliknięcie aktywnego przycisku. Inne klawisze wywołują inne działania. Na przykład klawisze strzałek mogą sterować paskiem przewijania.

Jednak my nie chcemy wysyłać zdarzenia naciśnięcia klawisza do aktywnego komponentu. W przeciwnym razie każdy przycisk musiałby znać procedurę obsługi kombinacji klawiszy *Ctrl+Y*, *Ctrl+B* i *Ctrl+R*.

Jest to bardzo powszechny problem. Jednak projektanci biblioteki Swing znaleźli dla niego proste rozwiązanie. Każdy JComponent posiada trzy **mapy wejścia** (ang. *input maps*), z których każda odwzorowuje obiekty KeyStroke na związane z nimi akcje. Mapy te odpowiadają trzem różnym sytuacjom (zobacz tabela 8.2).

Tabela 8.2. Mapy klawiaturowe

Znacznik	Wywołuje działanie, gdy
WHEN_FOCUSED	komponent jest aktywny
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	komponent zawiera komponent aktywny
WHEN_IN_FOCUSED_WINDOW	komponent znajduje się w tym samym oknie co komponent aktywny

Powyzsze mapy są sprawdzane w następującej kolejności w wyniku naciśnięcia klawisza:

- 1 Sprawdzenie mapy WHEN_FOCUSED aktywnego komponentu. Jeśli dany skrót klawiaturowy istnieje, następuje wykonanie powiązanego z nim działania. Jeśli działanie zostaje wykonane, następuje zatrzymanie sprawdzania warunków.
- 2 Następuje sprawdzenie map WHEN_ANCESTOR_OF_FOCUSED_COMPONENT aktywnego komponentu, a następnie jego komponentów nadzędnych. Gdy zostanie znaleziona mapa z danym skrótem klawiaturowym, następuje wykonanie działania. Jeśli działanie zostaje wykonane, następuje zatrzymanie sprawdzania warunków.
- 3 Odszukanie wszystkich **widocznych** i **włączonych** komponentów w aktywnym oknie, w których mapie WHEN_IN_FOCUSED_WINDOW znajduje się dany skrót klawiaturowy. Umożliwienie tym komponentom (w kolejności zgodnej z rejestracją zdarzeń naciśnięcia klawisza) wykonania odpowiednich działań. Po wykonaniu pierwszego działania następuje zatrzymanie przetwarzania. Ta część procesu może być źródłem problemów, jeśli dany skrót klawiaturowy pojawia się w więcej niż jednej mapie WHEN_IN_FOCUSED_WINDOW.

Mapę wejścia komponentu tworzy się za pomocą metody getInputMap. Na przykład:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Warunek WHEN_FOCUSED powoduje, że ta mapa będzie sprawdzana, gdy komponent jest aktywny. Nam potrzebna jest inna mapa. Aktywny jest jeden z przycisków, nie panel. Do wstawienia skrótów klawiszy zmieniających kolor nadaje się jedna z pozostałych dwóch map. W naszym przykładowym programie użyjemy mapy WHEN_ANCESTOR_OF_FOCUSED_COMPONENT.

Klasa InputMap nie odwzorowuje bezpośrednio obiektów KeyStroke w postaci obiektów Action. W zamian odwzorowuje w postaci dowolnych obiektów, a druga mapa, zaimplementowana w klasie ActionMap, mapuje obiekty na akcje. Dzięki temu łatwiej jest współdzielić te same akcje przez skróty klawiaturowe pochodzące z różnych map wejścia.

A zatem każdy komponent posiada trzy mapy wejścia i jedną mapę akcji (ang. *action map*). Aby je powiązać, trzeba wymyślić nazwy dla akcji. Klawisz można powiązać z akcją w następujący sposób:

```
imap.put(KeyStroke.getKeyStroke("ctrl Z"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

W przypadku akcji niewykonującej żadnych działań zwyczajowo stosuje się łańcuch `none`. W ten sposób można łatwo dezaktywować klawisz:

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```



Dokumentacja JDK zaleca stosowanie jako klucza akcji jej nazwy. Naszym zdaniem nie jest to dobre rozwiązanie. Nazwa akcji jest wyświetlaną na przyciskach i elementach menu, w związku z czym może się zmieniać w zależności od kaprysów projektanta interfejsu oraz może być przetłumaczona na wiele języków. Takie niestałe łańcuchy nie są dobrym wyborem w przypadku klawiszy wyszukiwania. Zalecamy wymyślenie nazw akcji niezależnych od wyświetlanych nazw.

Poniżej znajduje się zestawienie działań, które trzeba wykonać, aby wywołać to samo działanie w odpowiedzi na zdarzenie naciśnięcia przycisku, wyboru elementu z menu lub naciśnięcia klawisza:

1. Utwórz podklasę klasy `AbstractAction`. Można użyć tej samej klasy dla wielu spokrewnionych akcji.
2. Utwórz obiekt powyższej klasy akcji.
3. Utwórz przycisk lub element menu z obiektu powyższej klasy akcji. Konstruktor odczyta etykietę i ikonę z tego obiektu.
4. W przypadku akcji uruchamianych przez naciśnięcie klawisza konieczne jest wykonanie dodatkowych czynności. Najpierw należy zlokalizować komponent najwyższego poziomu w oknie, np. panel zawierający wszystkie pozostałe elementy.
5. Pobierz mapę `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` komponentu najwyższego poziomu. Utwórz obiekt klasy `KeyStroke` reprezentujący odpowiedni skrót klawiaturowy. Utwórz obiekt będący kluczem działania, np. łańcuch opisujący akcję. Wstaw tę parę danych (klawisz, klucz działania) do mapy wejścia.
6. Pobierz mapę akcji komponentu najwyższego poziomu. Dodaj parę klucz akcji – obiekt akcji do tej mapy.

Listing 8.3 przedstawia kompletny kod programu mapującego przyciski i klawisze na obiekty akcji. Można go wypróbować — kliknięcie jednego z przycisków lub naciśnięcie kombinacji klawiszy `Ctrl+Z`, `Ctrl+N` lub `Ctrl+C` spowoduje zmianę koloru panelu.

Listing 8.3. action/ActionFrame.java

```
package action;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z panelem, który demonstruje akcje zmiany koloru.
 */


```

```

public class ActionFrame extends JFrame
{
    private JPanel buttonPanel;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public ActionFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        buttonPanel = new JPanel();

        // Definicje akcji
        Action yellowAction = new ColorAction("Żółty",
        ↳new ImageIcon("yellow-ball.gif"),
        Color.YELLOW);
        Action blueAction = new ColorAction("Niebieski",
        ↳new ImageIcon("blue-ball.gif"), Color.BLUE);
        Action redAction = new ColorAction("Czerwony",
        ↳new ImageIcon("red-ball.gif"), Color.RED);

        // Dodanie przycisków dla akcji
        buttonPanel.add(new JButton(yellowAction));
        buttonPanel.add(new JButton(blueAction));
        buttonPanel.add(new JButton(redAction));

        // Dodanie panelu do ramki
        add(buttonPanel);

        // Powiązanie klawiszy Z, N i C z nazwami
        InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_
        ↳FOCUSUSED_COMPONENT);
        imap.put(KeyStroke.getKeyStroke("ctrl Z"), "panel.yellow");
        imap.put(KeyStroke.getKeyStroke("ctrl N"), "panel.blue");
        imap.put(KeyStroke.getKeyStroke("ctrl C"), "panel.red");

        // Powiązanie nazw z akcjami
        ActionMap amap = buttonPanel.getActionMap();
        amap.put("panel.yellow", yellowAction);
        amap.put("panel.blue", blueAction);
        amap.put("panel.red", redAction);
    }

    public class ColorAction extends AbstractAction
    {
        /**
         * Tworzy akcję zmiany koloru.
         * @param name nazwa, która pojawi się na przycisku
         * @param icon ikona, która pojawi się na przycisku
         * @param c kolor tła
         */
        public ColorAction(String name, Icon icon, Color c)
        {
            putValue(Action.NAME, name);
            putValue(Action.SMALL_ICON, icon);
            putValue(Action.SHORT_DESCRIPTION, "Ustaw kolor panelu na " +
            ↳name.toLowerCase());
        }
    }
}

```

```

        putValue("color", c);
    }

    public void actionPerformed(ActionEvent event)
    {
        Color c = (Color) getValue("color");
        buttonPanel.setBackground(c);
    }
}
}

```

javax.swing.Action 1.2

- boolean isEnabled()
 - void setEnabled(boolean b)
- Pobiera lub ustawia właściwość enabled akcji.
- void putValue(String key, Object value)
- Wstawia parę nazwa – wartość do obiektu akcji.

Parametry: key Nazwa właściwości, która ma zostać zapisana z obiektem akcji. Może to być dowolny łańcuch, ale jest kilka nazw o z góry zdefiniowanym znaczeniu — zobacz tabelę 8.1.

value Obiekt powiązany z nazwą.

- Object getValue(String key)

Zwraca wartość z zapisanej pary nazwa – wartość.

javax.swing.KeyStroke 1.2

- static KeyStroke getKeyStroke(String description)

Tworzy skrót klawiaturowy z czytelnego dla człowieka opisu (ciągu łańcuchów rozdzielonych spacjami). Opis zaczyna się od zera lub większej liczby modyfikatorów shift control ctrl meta alt altGraph, a kończy się łańcuchem typed i łańcuchem składającym się z jednego znaku (na przykład typed a) lub opcjonalnym specyfikatorem zdarzenia (pressed — domyślny, lub released) i kodem klawisza. Kod klawisza, jeśli ma przedrostek VK_, powinien odpowiadać stałej KeyEvent, na przykład INSERT odpowiada KeyEvent.VK_INSERT.

javax.swing.JComponent 1.2

- ActionMap getActionMap() 1.3

Zwraca mapę wiążącą klucze mapy akcji (które mogą być dowolnymi obiektami) z obiektami klasy Action.

- InputMap getInputMap(int flag) 1.3

Pobiera mapę wejścia, która odwzorowuje klawisze w postaci kluczów mapy akcji.

Parametry:	flag	Warunek określający, kiedy element aktywny ma wywołać akcję — jedna z wartości z tabeli 8.2.
-------------------	------	--

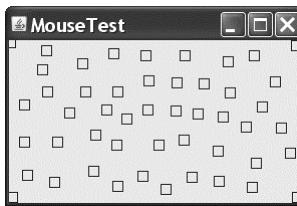
8.3. Zdarzenia generowane przez mysz

Takie zdarzenia jak kliknięcie przycisku lub elementu w menu za pomocą myszy nie wymagają pisania procedur obsługi. Te zdarzenia są obsługiwane automatycznie przez różne elementy interfejsu użytkownika. Aby jednak umożliwić rysowanie za pomocą myszy, konieczne jest przechwycenie zdarzeń ruchu, kliknięcia i przeciągania myszy.

W tym podrozdziale prezentujemy prosty edytor grafiki pozwalający umieszczać, przesuwać i usuwać kwadraty z obszaru roboczego (zobacz rysunek 8.7).

Rysunek 8.7.

Program
obsługujący
zdarzenia myszy



Kiedy użytkownik naciśnie przycisk myszy, wywoływane są trzy metody nasłuchujące: `mousePressed` po naciśnięciu przycisku, `mouseReleased` po zwolnieniu przycisku myszy i `mouseClicked`. Jeśli w sferze zainteresowań leżą wyłącznie pełne kliknięcia, pierwsze dwie z wymienionych metod można pominąć. Wywołując metody `getX` i `getY` na rzecz obiektu klasy `MouseEvent`, można sprawdzić współrzędne `x` i `y` wskaźnika myszy w chwili kliknięcia. Do rozróżnienia pojedynczych, podwójnych i potrójnych (!) kliknięć służy metoda `getClickCount`.

Niektórzy projektanci interfejsów tworzą kombinacje klawiszy połączone z kliknięciami myszką, np. `Ctrl+Shift+kliknięcie`. Naszym zdaniem jest to postępowanie niegodne naśladowania. Osoby, które nie zgadzają się z naszą opinią, może przekonać fakt, że sprawdzenie przycisków myszy i klawiszy specjalnych jest niezwykle zagmatwanym zadaniem — niebawem się o tym przekonamy.

Aby sprawdzić, które modyfikatory zostały ustawione, należy użyć maski bitowej. W oryginalnym API dwie maski przycisków są równoważne z maskami klawiszy specjalnych, mianowicie:

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

Zrobiono tak, aby użytkownicy posiadający myszkę z jednym przyciskiem mogli naśładować pozostałe przyciski za pomocą klawiszy specjalnych (ang. *modifier keys*). Od Java SE 1.4 zaproponowano jednak inną metodę. Od tej pory istnieją następujące maski:

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
```

```
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

Metoda `getModifiersEx` zwraca dokładne informacje o przyciskach myszy i klawiszach specjalnych użytych w zdarzeniu myszy.

Pamiętajmy, że maska `BUTTON3_DOWN_MASK` w systemie Windows sprawdza prawy (nie główny) przycisk myszy. Na przykład poniższy fragment programu sprawdza, czy prawy przycisk myszy jest wciśnięty:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
    . . . // procedury obsługi zdarzenia kliknięcia prawym przyciskiem myszy
```

W przykładowym programie definiujemy zarówno metodę `mousePressed`, jak i `mouseClicked`. Jeśli użytkownik kliknie piksel nieznajdujący się w obrębie żadnego z narysowanych kwadratów, zostanie dodany nowy kwadrat. Działanie to zostało zaimplementowane w metodzie `mousePressed`, a więc kwadrat pojawia się natychmiast po kliknięciu, przed zwolnieniem przycisku myszy. Dwukrotne kliknięcie przyciskiem myszy w obrębie narysowanego kwadratu powoduje jego usunięcie. Implementacja tej funkcji została umieszczona w metodzie `mouseClicked`, ponieważ konieczne jest sprawdzenie liczby kliknięć:

```
public void mousePressed(MouseEvent event)
{
    current = find(event.getPoint());
    if (current == null) // nie w obrębie kwadratu
        add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
    current = find(event.getPoint());
    if (current != null && event.getClickCount() >= 2)
        remove(current);
}
```

Kiedy kurSOR myszy przesuwa się nad oknem, odbiera ono stały strumień zdarzeń ruchu myszy. Zauważmy, że są osobne interfejsy `MouseListener` i `MouseMotionListener`. Wyróżniono je z częścią zwiększenia efektywności. Kiedy użytkownik przesuwa mysz, powstaje cała masa zdarzeń dotyczących tej czynności. Obiekt nasłuchujący, który oczekuje na **kliknięcia**, nie jest zajmowany przez nieinteresujące go **zdarzenia ruchu**.

Nasz testowy program przechwytuje zdarzenia ruchu i w odpowiedzi na nie zmienia wygląd kurSORa (na krzyżek). Odpowiedzialna jest za to metoda `getPredefinedCursor` z klasy `Cursor`. Tabela 8.3 przedstawia stałe podawane jako argument wspomnianej funkcji oraz reprezentowane przez nie kurSory w systemie Windows.

Poniżej znajduje się kod źródłowy metody `mouseMoved` z klasy `MouseMotionListener` zdefiniowanej w naszym przykładowym programie:

Tabela 8.3. Przykładowe kursory

Ikona	Stała	Ikona	Stała
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

```
public void mouseMoved(MouseEvent event)
{
    if (find(event.getPoint()) == null)
        setCursor(Cursor.getDefaultCursor());
    else
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```



Można zdefiniować własne typy kurSORów. Służy do tego metoda `createCustomCursor` z klasy `Toolkit`:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img, new Point(10, 10), "dynamite
→stick");
```

Pierwszy argument tej metody określa plik graficzny przedstawiający kurSOR. Drugi wyznacza przesunięcie punktu aktywnego kurSORa. Trzeci jest łańcuchem opisującym kurSOR. łańcuch ten może służyć zwiększeniu dostępności. Na przykład program czytający z ekranu używany przez osobę niedowidzącą może przeczytać opis takiego kurSORa.

Jeśli w czasie przesuwania myszy użytkownik kliknie jej przycisk, generowane są wywołania metody `mouseDragged` zamiast `mouseMoved`. Nasz przykładowy program zezwala na przeciąganie kwadratów pod kurSorem. Efekt ten uzyskaliśmy, aktualizując położenie przeciąganego kwadratu, tak aby jego środek znajdował się w tym samym miejscu co punkt centralny myszki. Następnie ponownie rysujemy obszar roboczy, aby ukazać nowe położenie kurSORa myszki.

```
public void mouseDragged(MouseEvent event)
{
    if (current != null)
    {
        int x = event.getX();
        int y = event.getY();
```

```

        current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
        repaint();
    }
}

```



Metoda `mouseMoved` jest wywoływana tylko wtedy, gdy kursor znajduje się w obrębie komponentu. Natomiast metoda `mouseDragged` jest wywoływana nawet wtedy, gdy kursor opuści komponent.

Istnieją jeszcze dwie inne metody obsługujące zdarzenia myszy: `mouseEntered` i `mouseExited`. Są one wywoływane, gdy kursor myszy wchodzi do komponentu lub go opuszcza.

Na zakończenie wyjaśnimy sposób nasłuchiwanego zdarzeń generowanych przez mysz. Kliknięcia przyciskiem myszy są raportowane przez metodę `mouseClicked` należącą do interfejsu `MouseListener`. Ponieważ wiele aplikacji korzysta wyłącznie z kliknięć myszką i występują one bardzo często, zdarzenia ruchu myszy i przeciągania zostały zdefiniowane w osobnym interfejsie o nazwie `MouseMotionListener`.

W naszym programie interesują nas oba rodzaje zdarzeń generowanych przez mysz. Zdefiniowaliśmy dwie klasy wewnętrzne o nazwach `MouseHandler` i `MouseMotionHandler`. Pierwsza z nich jest podklassą klasy `MouseAdapter`, ponieważ definiuje tylko dwie z pięciu metod interfejsu `MouseListener`. Klasa `MouseMotionHandler` implementuje interfejs `MouseMotionListener`, co znaczy, że zawiera definicje obu jego metod. Listingi 8.4 i 8.5 przedstawiają kod źródłowy omawianego programu.

Listing 8.4. mouse/MouseFrame.java

```

package mouse;

import javax.swing.*;

/**
 * Ramka zawierająca okienko do testowania myszy
 */
public class MouseFrame extends JFrame
{
    public MouseFrame()
    {
        add(new MouseComponent());
        pack();
    }
}

```

Listing 8.5. mouse/MouseComponent.java

```

package mouse;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

```

```
/*
 * Komponent z działaniami myszy, do którego można dodawać (lub z którego można usuwać) kwadraty.
 */
public class MouseComponent extends JComponent
{
    private static final int SIDELENGTH = 10;
    private ArrayList<Rectangle2D> squares;
    private Rectangle2D current;

    public MouseComponent()
    {
        squares = new ArrayList<>();
        current = null;

        addMouseListener(new MouseHandler());
        addMouseMotionListener(new MouseMotionHandler());
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        //Rysowanie wszystkich kwadratów
        for (Rectangle2D r : squares)
            g2.draw(r);
    }

    /**
     * Znajduje pierwszy kwadrat zawierający punkt.
     * @param p punkt
     * @return pierwszy kwadrat zawierający punkt p
     */
    public Rectangle2D find(Point2D p)
    {
        for (Rectangle2D r : squares)
        {
            if (r.contains(p)) return r;
        }
        return null;
    }

    /**
     * Dodaje kwadrat do zbioru.
     * @param p środek kwadratu
     */
    public void add(Point2D p)
    {
        double x = p.getX();
        double y = p.getY();

        current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH /
            2, SIDELENGTH,
            SIDELENGTH);
        squares.add(current);
        repaint();
    }
}
```

```

    /**
 * Usuwa kwadrat ze zbioru.
 * @param s kwadrat, który ma być usunięty
 */
public void remove(Rectangle2D s)
{
    if (s == null) return;
    if (s == current) current = null;
    squares.remove(s);
    repaint();
}
// Kwadrat zawierający kurSOR
private class MouseHandler extends MouseAdapter
{
    public void mousePressed(MouseEvent event)
    {
        // Dodanie nowego kwadratu, jeśli kurSOR nie jest wewnątrz innego kwadratu
        current = find(event.getPoint());
        if (current == null) add(event.getPoint());
    }

    public void mouseClicked(MouseEvent event)
    {
        // Usunięcie kwadratu w wyniku jego dwukrotnego kliknięcia
        current = find(event.getPoint());
        if (current != null && event.getClickCount() >= 2) remove(current);
    }
}

private class MouseMotionHandler implements MouseMotionListener
{
    public void mouseMoved(MouseEvent event)
    {
        // Ustawienie kursora na krzyżk, jeśli znajduje się wewnątrz
        // kwadratu

        if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
        else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
    }

    public void mouseDragged(MouseEvent event)
    {
        if (current != null)
        {
            int x = event.getX();
            int y = event.getY();

            // Przeciągnięcie aktualnego kwadratu w celu wyśrodkowania go w punkcie (x, y)
            current setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
                SIDELENGTH);
            repaint();
        }
    }
}

```

java.awt.event.MouseEvent **1.1**

- int getX()
- int getY()
- Point getPoint()

Zwraca współrzędne x (pozioma) i y (pionowa) lub punkt, w którym miało miejsce zdarzenie, mierząc od lewego górnego rogu komponentu będącego źródłem zdarzenia.

- int getClickCount()

Zwraca liczbę kolejnych kliknięć przyciskiem myszy związanych z danym zdarzeniem (odstęp czasu oddzielający zdarzenia określone jako kolejne zależy od systemu).

java.awt.event.InputEvent **1.1**

- int getModifiersEx() **1.4**

Zwraca rozszerzone modyfikatory zdarzenia. Do sprawdzania zwróconych wartości służą następujące maski:

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

- static String getModifiersExText(int modifiers) **1.4**

Zwraca łańcuch typu Shift+Button1, opisujący rozszerzone modyfikatory w danym zbiorze znaczników.

java.awt.Toolkit **1.0**

- public Cursor createCustomCursor(Image image, Point hotSpot, String name) **1.2**

Tworzy nowy obiekt niestandardowego kurSORA.

Parametry:	image	Obraz reprezentujący kurSOR
	hotSpot	Punkt centralny kurSORA (na przykład końcówka strzałki lub środek krzyżyka)
	name	Opis kurSORA wspomagający dostępność w specjalnych środowiskach

java.awt.Component **1.0**

- public void setCursor(Cursor cursor) **1.1**

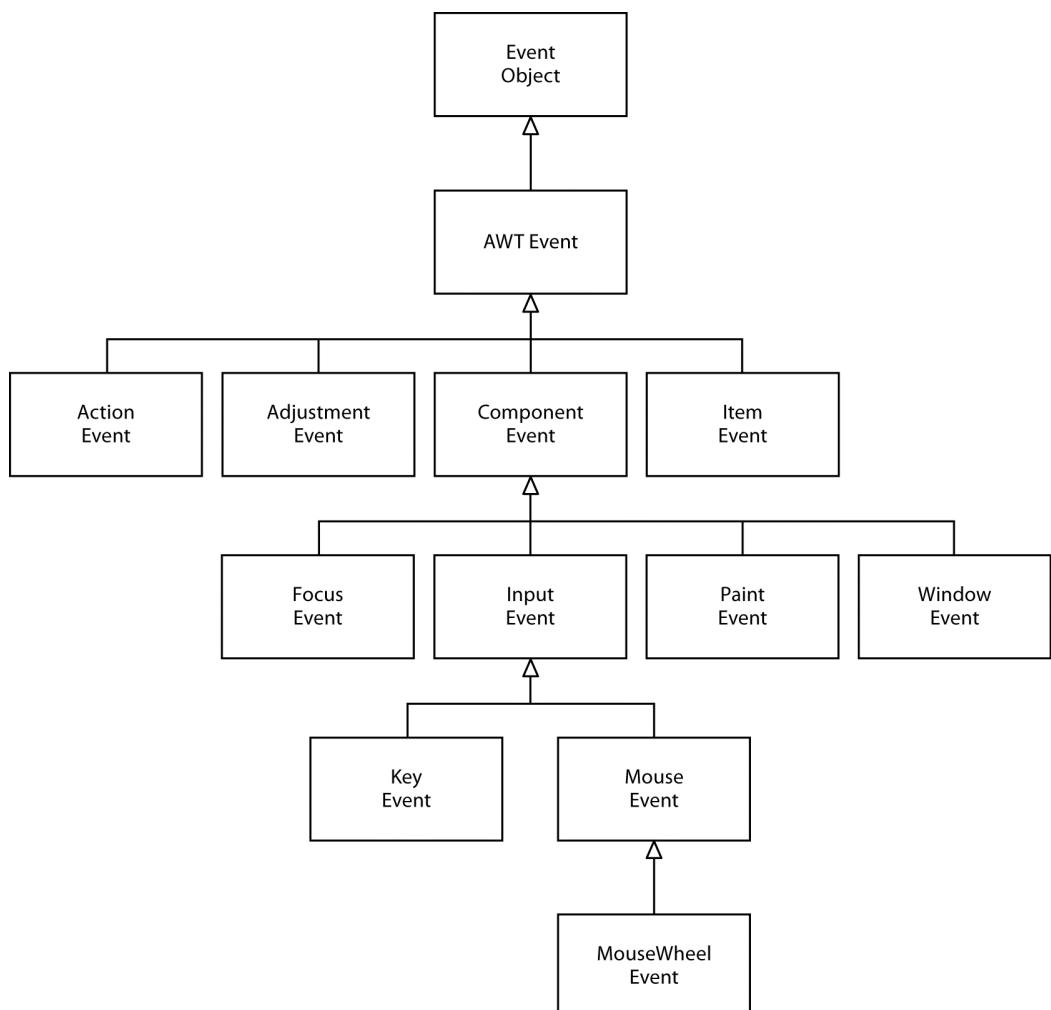
Ustawia obraz kurSORA na określony kurSOR.

8.4. Hierarchia zdarzeń w bibliotece AWT

Mając już pewne rozeznanie w temacie obsługi zdarzeń, na zakończenie tego rozdziału zrobimy krótki przegląd architektury obsługi zdarzeń biblioteki AWT.

Jak wspominaliśmy wcześniej, zdarzenia w Javie są obsługiwane w metodologii obiektowej, a wszystkie zdarzenia pochodzą od klasy `EventObject` z pakietu `java.util` (nazwą wspólnej nadklasy nie jest `Event`, ponieważ taką nazwę nosi klasa zdarzeń w starym modelu zdarzeń — mimo że model ten jest obecnie odradzany, jego klasy nadal wchodzą w skład biblioteki Javy).

Klasa `EventObject` posiada podkласę `AWTEvent` będącą nadklasą wszystkich klas zdarzeniowych AWT. Rysunek 8.8 przedstawia diagram dziedziczenia zdarzeń AWT.



Rysunek 8.8. Diagram dziedziczenia klas zdarzeniowych AWT

Niektóre komponenty Swing generują obiekty zdarzeniowe jeszcze innych typów zdarzeń. Rozszerzają one bezpośrednio klasę EventObject, a nie AWTEvent.

Obiekty zdarzeniowe zawierają informacje o zdarzeniach przesyłanych przez źródło zdarzeń do swoich słuchaczy. W razie potrzeby można przeanalizować obiekty zdarzeniowe, które zostały przekazane do obiektów nasłuchujących, co zrobiliśmy w przykładzie z przykładem za pomocą metod getSource i getActionCommand.

Niektóre klasy zdarzeniowe AWT są dla programisty Javy bezużyteczne. Na przykład biblioteka AWT wstawia do kolejki zdarzeń obiekty PaintEvent, ale obiekty te nie są dostarczane do słuchaczy. Programiści Javy nie nasłuchują zdarzeń rysowania. Przesłaniają metodę paint →Component, aby móc kontrolować ponowne rysowanie. Ponadto AWT generuje pewne zdarzenia, które są potrzebne tylko programistom systemowym. Nie opisujemy tych specjalnych typów zdarzeń.

8.4.1. Zdarzenia semantyczne i niskiego poziomu

Biblioteka AWT rozróżnia zdarzenia **niskiego poziomu** i zdarzenia **semantyczne**. Zdarzenie semantyczne jest dziełem użytkownika (jest to np. kliknięcie przycisku). Dlatego zdarzenie ActionEvent jest zdarzeniem semantycznym. Zdarzenia niskiego poziomu to takie zdarzenia, które umożliwiają zaistnienie zdarzeń semantycznych. W przypadku kliknięcia przycisku jest to jego naciśnięcie, szereg ruchów myszą i zwolnienie (ale tylko jeśli zwolnienie nastąpi w obrębie przycisku). Może to być naciśnięcie klawisza mające miejsce po wybraniu przycisku przez użytkownika za pomocą klawisza *Tab* i naciśnięcie go za pomocą spacji. Podobnie semantycznym zdarzeniem jest przesunięcie paska przewijania, a ruch myszą jest zdarzeniem niskiego poziomu.

Poniżej znajduje się lista najczęściej używanych klas zdarzeń semantycznych pakietu java.awt.event:

- ActionEvent — kliknięcie przycisku, wybór elementu z menu, wybór elementu listy, naciśnięcie klawisza *Enter* w polu tekstowym.
- AdjustmentEvent — przesunięcie paska przewijania.
- ItemEvent — wybór jednego z pól do wyboru lub elementów listy.

Do najczęściej używanych klas zdarzeń niskiego poziomu zaliczają się:

- KeyEvent — naciśnięcie lub zwolnienie klawisza.
- MouseEvent — naciśnięcie lub zwolnienie przycisku myszy, poruszenie lub przeciagnięcie myszą.
- MouseWheelEvent — pokręcenie kółkiem myszy.
- FocusEvent — uaktywnienie lub dezaktywacja elementu.
- WindowEvent — zmiana stanu okna.

Tych zdarzeń nasłuchują następujące interfejsy:

```

ActionListener
AdjustmentListener
FocusListener
ItemListener
KeyListener
MouseListener
MouseMotionListener
MouseWheelListener
WindowListener
WindowFocusListener
WindowStateListener

```

Niektóre interfejsy nasłuchujące AWT, te zawierające więcej niż jedną metodę, posiadają odpowiadające im klasy adaptacyjne, które implementują wszystkie ich metody (pozostałe interfejsy mają tylko jedną metodę, a więc utworzenie dla nich klas adaptacyjnych nie dałoby żadnych korzyści). Poniższe klasy adaptacyjne są często używane:

```

FocusAdapter
KeyAdapter
MouseAdapter
MouseMotionAdapter
WindowAdapter

```

Tabela 8.4 przedstawia najważniejsze interfejsy nasłuchowe, zdarzenia i źródła zdarzeń biblioteki AWT.

Tabela 8.4. Obsługa zdarzeń

Interfejs	Metody	Parametry/ metody dostępu	Zdarzenia generowane przez
ActionListener	actionPerformed	ActionEvent ■ getActionCommand ■ getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent ■ getAdjustable ■ getAdjustmentType ■ getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent ■ getItem ■ getItemSelectable ■ getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent ■ isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent ■ getKeyChar ■ getKeyCode ■ getKeyModifiersText ■ getKeyText ■ isActionText	Component

Tabela 8.4. Obsługa zdarzeń — ciąg dalszy

Interfejs	Metody	Parametry/ metody dostępu	Zdarzenia generowane przez
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent ■ getClickCount ■ getX ■ getY ■ getPoint ■ translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	MouseWheelMoved	MouseWheelEvent ■ getWheelRotation ■ getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent ■ getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent ■ getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent ■ getOldState ■ getNewState	Window

Pakiet `javax.swing.event` zawiera dodatkowe zdarzenia specyficzne dla komponentów Swinga. Niektóre z nich opisujemy w następnym rozdziale.

Na tym zakończymy opis technik obsługi zdarzeń AWT. W następnym rozdziale nauczymy się wykorzystywać najpopularniejsze komponenty Swinga oraz szczegółowo przeanalizujemy generowane przez nie zdarzenia.

9

Komponenty Swing interfejsu użytkownika

W tym rozdziale:

- Swing a wzorzec projektowy Model-View-Controller
- Wprowadzenie do zarządzania rozkładem
- Wprowadzanie tekstu
- Komponenty wyboru
- Menu
- Zaawansowane techniki zarządzania rozkładem
- Okna dialogowe

Głównym celem poprzedniego rozdziału było przedstawienie technik wykorzystania modelu zdarzeń w Javie. W międzyczasie postawiliśmy pierwsze kroki w tworzeniu graficznego interfejsu użytkownika. Ten rozdział opisuje najważniejsze narzędzia potrzebne do budowy w pełni funkcjonalnego GUI.

Zacznijemy od przeglądu architektury, na której opiera się Swing. Znajomość podstawowych mechanizmów działania ułatwia naukę efektywnego wykorzystania niektórych bardziej zaawansowanych komponentów. W następnej kolejności omówimy najczęściej używane komponenty Swing, czyli pola tekstowe, przełączniki (ang. *radio button*) i menu. Następnie przejdziemy do rozmieszczania tych komponentów w oknie niezależnie od wybranego stylu interfejsu za pomocą narzędzi zarządcy układu (ang. *layout manager*). Na zakończenie rozdziału nauczymy się tworzyć okna dialogowe Swing.

Ten rozdział opisuje podstawowe komponenty, takie jak komponenty tekstowe, przyciski i suwaki. Są to najczęściej używane komponenty, niezbędne w większości interfejsów. Bardziej zaawansowane komponenty zostały opisane w drugim tomie.

9.1. Swing a wzorzec projektowy

Model-View-Controller

Zgodnie z zapowiedzią zaczniemy od opisu architektury komponentów biblioteki Swing. Najpierw zapoznamy się z ogólnym pojęciem **wzorca projektowego** (ang. *design pattern*), a później przejdziemy do wzorca model-widok-kontroler (ang. *Model-View-Controller* — *MVC*), który miał niemały wpływ na projekt biblioteki Swing.

9.1.1. Wzorce projektowe

Przy rozwiązywaniu problemu z reguły nie dochodzi się do rozwiązania, zaczynając od zera. Zazwyczaj korzysta się z doświadczenia innych programistów, np. zasiegając ich rady w interesujących nas kwestiach. Wzorce projektowe umożliwiają przedstawienie tej wiedzy w poukładowy sposób.

Niedawno inżynierowie oprogramowania zaczęli gromadzić katalogi takich wzorców. Inspirowaną prekursorów w tej dziedzinie były wzorce projektowe architekta Christophera Alexandra. W swojej książce pod tytułem *The Timeless Way of Building* (Oxford University Press, 1979) zaważył on zbiór wzorców projektowych do wykorzystania w pomieszczeniach publicznych i mieszkalnych. Oto przykładowy wzorzec z tej książki:

Umiejscowienie okna

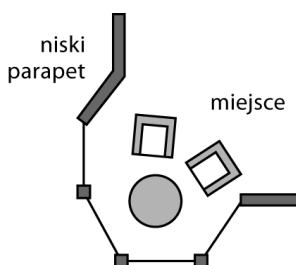
Każdy lubi ławeczki w oknach, okna wykuszowe i duże okna z niskimi parapetami oraz przystawionymi do nich wygodnymi krzesłami... W pokoju pozbawionym takiego miejsca rzadko potrafimy się zrelaksować...

Jeśli w pokoju nie ma takiego miejsca, osoba w nim przebywająca jest rozdzierana przez dwie siły — z jednej strony chce wygodnie usiąść, a z drugiej ciągnie ją do światła.

Oczywiście, jeśli wygodne miejsca — te, w których planujemy spędzać najwięcej czasu — znajdują się z dala od okien, nie ma sposobu na przewyciężenie tego konfliktu...

Wniosek: w każdym pokoju, w którym spędzasz choć trochę czasu w ciągu dnia, wygospodaruj przynajmniej jedno wygodne miejsce przy oknie (rysunek 9.1).

Rysunek 9.1.
Miejsce
przy oknie



Każdy wzorzec w katalogu Alexandra, podobnie jak wzorce projektowe oprogramowania, jest zbudowany według określonego schematu. Najpierw opisywany jest kontekst, czyli sytuacja, która powoduje powstanie problemu. Później następuje opis problemu — z reguły ma on postać zbioru kilku przeciwnych sobie sił. Ostateczne rozwiązanie jest złotym środkiem pomiędzy tymi siłami.

We wzorcu miejsca przy oknie kontekstem jest pokój, w którym spędzamy jakąś część dnia. Przeciwnie siły to chęć usadowienia się w wygodnym miejscu i przyciąganie do światła. Rozwiążanie polega na wygospodarowaniu miejsca przy oknie.

We wzorcu MVC, opisany w kolejnym podrozdziale, kontekstem jest system interfejsu użytkownika, który przedstawia informacje i odbiera dane od użytkownika. Jest kilka sił. Może być wiele różnych reprezentacji tych samych danych, które muszą być aktualizowane wspólnie. Reprezentacja wizualna może się zmieniać, np. w związku z różnymi stylami. Mechanizmy interakcji mogą się zmieniać, na przykład w związku z obsługą poleceń głosowych. Rozwiążanie polega na rozdzielaniu obowiązków na trzy osobne komponenty: model, widok i kontroler.

Wzorzec MVC nie jest jedynym wzorcem, którego użyto przy projektowaniu bibliotek AWT i Swing. Oto kilka innych przykładów:

- Kontenery i komponenty są przykładami wzorca Composite.
- Panel przewijany (ScrollPane) to przykład wzorca Decorator.
- Zarządcy układu reprezentują wzorzec Strategy.

Ważną cechą wzorców projektowych jest to, że przenikają one do kultury. Programiści na całym świecie wiedzą, o co nam chodzi, kiedy mówimy o wzorcu MVC lub Decorator. Dzięki temu wzorce stały się doskonałym narzędziem do opisu problemów związanych z projektowaniem.

Formalny opis wielu wzorców programistycznych znajduje się w nowatorskiej książce poświęconej tej tematyce pod tytułem *Wzorce projektowe* (WNT, Warszawa 2005), której autorem jest Erich Gamma i współpracownicy (tytuł oryginału *Design patterns — Elements of Reusable Object-Oriented Software*). Gorąco polecamy także lekturę doskonałej książki pod tytułem *A System of Patterns* (John Wiley & Sons, 1996), której autorem jest Frank Buschmann i współpracownicy. Naszym zdaniem ta pozycja jest mniej nowatorska od poprzedniej i bardziej przystępna.

9.1.2. Wzorzec Model-View-Controller

Zatrzymajmy się na chwilę nad składnikami każdego komponentu interfejsu użytkownika, takimi jak przyciski, pola wyboru, pola tekstowe czy skomplikowany widok drzewa. Każdy komponent ma trzy cechy:

- **Treść** — np. stan przycisku (wciśnięty lub nie) lub tekst w polu tekstowym.
- **Wygląd** — kolor, rozmiar itd.
- **Zachowanie** — reakcje na zdarzenia.

Nawet na pierwszy rzut oka taki prosty komponent jak przycisk wykazuje w miarę złożone interakcje pomiędzy tymi cechami. Oczywiście wygląd przycisku zależy od stylu. Przycisk w stylu Metal wygląda inaczej niż Windows lub Motif. Dodatkowo na jego wygląd ma wpływ jego stan. Wciśnięcie przycisku oznacza konieczność ponownego narysowania go ze zmienionym wyglądem. Stan zależy od zdarzeń odbieranych przez przycisk. Kiedy użytkownik naciśnie przycisk myszy po uprzednim umieszczeniu kurSORA w obrębie przycisku na ekranie, przycisk ten zostanie wciśnięty.

Oczywiście używając przycisku w programie, nikt nie rozważa dogłębnie jego wewnętrznych mechanizmów i cech. To jest przecież zadanie programisty, który ten przycisk implementował. Natomiast programiści implementujący przyciski muszą bardziej się nad nimi skupić. Ich zadanie polega przecież na takim zaimplementowaniu przycisków i innych komponentów, aby działały bez zarzutów w każdym stylu.

W związku z tym projektanci biblioteki Swing postanowili skorzystać z dobrze znanego wzorca o nazwie **Model-View-Controller** (MVC). Wzorzec ten, podobnie jak wiele innych wzorców projektowych, odwołuje się do jednej z zasad projektowania zorientowanego obiektowo, którą opisywaliśmy w rozdziale 5., a która brzmi: nie obciążaj jednego obiektu zbyt dużą liczbą działań. Nie twórz jednej klasy, która robi wszystko. Styl jednego komponentu zwiąż z jednym obiektem, a treść przechowuj w **innym** obiekcie. Wzorzec projektowy MVC podpowiada, jak to zrobić. Należy napisać trzy osobne klasy:

- **Model** — przechowuje treść.
- **Widok** (ang. *view*) — wyświetla treść.
- **Kontroler** (ang. *controller*) — obsługuje dane wejściowe od użytkownika.

Wzorzec precyzyjnie określa interakcje pomiędzy tymi trzema obiektami. Model przechowuje treść i **nie posiada interfejsu użytkownika**. W przypadku przycisku nie ma tej treści dużo. Stanowi ją tylko niewielki zestaw znaczników określających, czy przycisk jest wciśnięty, czy nie, czy jest aktywny, czy nie itd. Bardziej interesująca jest treść w przypadku pola tekstowego. Jest to obiekt łańcuchowy przechowujący aktualny tekst. **Nie jest to jednak to samo** co widok treści — jeśli treści jest więcej, niż może pomieścić pole tekstowe, użytkownik zobaczy tylko część tekstu (rysunek 9.2).

Rysunek 9.2.

Model i widok pola tekstowego

model "Koń i żółw grali w kości z piękną ćmą u źródła"

widok **grali | w kości**

Model musi zawierać metody zmieniające i sprawdzające treść. Na przykład model tekstowy posiada metody dodające lub usuwające znaki z aktualnego tekstu i zwracające ten tekst w postaci łańcucha. Nie zapomnijmy, że model nie ma charakteru wizualnego. Rysowanie danych przechowywanych w modelu należy do obowiązków widoku.

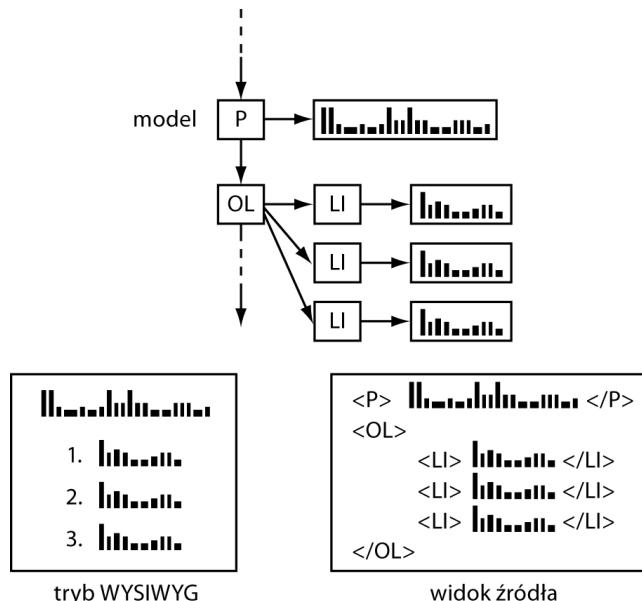


Termin „model” nie jest najlepszym określeniem, ponieważ zazwyczaj oznacza on coś abstrakcyjnego. Na przykład projektanci samochodów i samolotów budują modele, będące imitacjami prawdziwych maszyn. Analogia ta w przypadku wzorca model-widok-kontroler prowadzi jednak na manowce. W tym wzorcu model przechowuje całą treść, a widok dostarcza (pełną lub niepełną) wizualną reprezentację tej treści. Lepszą analogią byłby model pozujący malarzowi. Zadaniem artysty jest przyjrzenie się temu modelowi i stworzenie jego widoku. W zależności od artysty widok ten może być zwykłym portretem, malowidłem impresjonistycznym albo rysunkiem kubistycznym przedstawiającym kończyny w „powykręcanych” proporcjach.

Jedną z zalet wzorca MVC jest to, że model można przedstawiać na różne sposoby, za każdym razem pokazując inną część całości. Na przykład edytor HTML może oferować dwa **równoczesne** widoki treści: widok strony, jakby była wyświetlona w przeglądarce (tryb WYSIWYG), oraz widok źródła (rysunek 9.3). Kiedy model jest aktualizowany za pośrednictwem kontrolera jednego z widoków, oba widoki są informowane o tej zmianie. W momencie odebrania powiadomienia widoki aktualizują się automatycznie. Oczywiście dla takich prostych komponentów jak przycisk nie tworzy się wielu widoków tego samego modelu.

Rysunek 9.3.

Dwa oddzielne widoki tego samego modelu

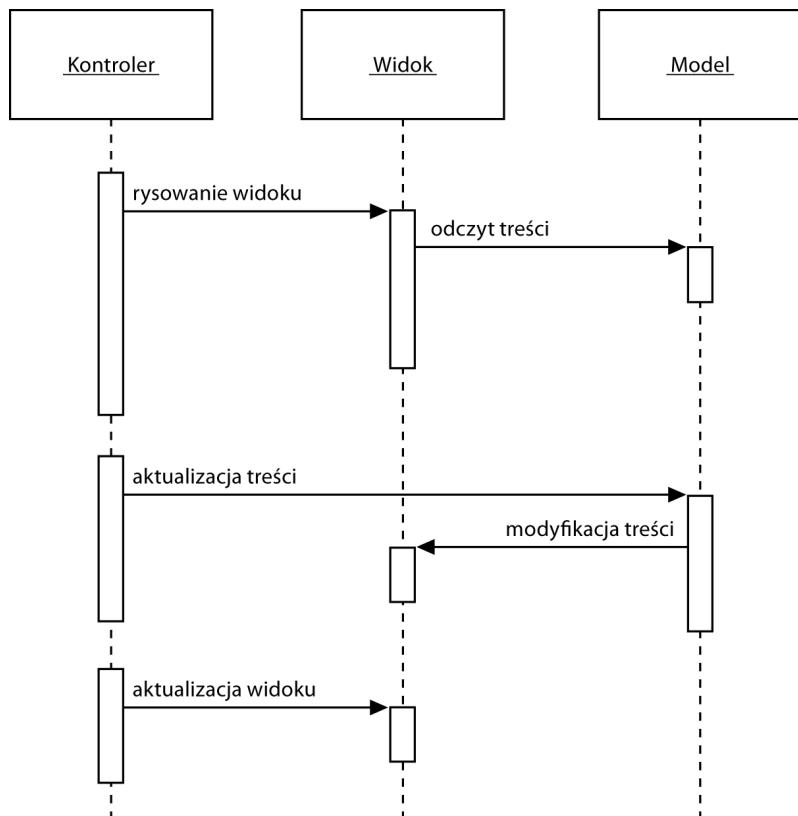


Kontroler obsługuje zdarzenia związane z wprowadzaniem danych przez użytkownika, jak kliknięcie przyciskiem myszy czy naciśnięcie klawisza na klawiaturze. Jeśli na przykład użytkownik naciśnie klawisz litery w polu tekstowym, kontroler wywołuje polecenie modelu dotyczące wstawiania znaków. Następnie model rozkazuje widokowi, aby się zaktualizował. Widok nie otrzymuje żadnych informacji, dlaczego tekst się zmienił. Jeśli natomiast użytkownik naciśnie jeden z klawiszy strzałek, kontroler może wydać widokowi polecenie, aby się przewinął. Przewijanie widoku nie wywiera żadnego wpływu na tekst, a więc model nie wie, że to zdarzenie miało w ogóle miejsce.

Rysunek 9.4 przedstawia relacje pomiędzy modelem, widokiem i kontrolerem.

Rysunek 9.4.

Relacje pomiędzy modelem, widokiem i kontrolerem



Programista Swing nie musi z reguły pamiętać o architekturze model-widok-kontroler. Każdy komponent interfejsu użytkownika posiada klasę osłonową (np. JButton czy JTextField), która przechowuje model i widok. Kiedy programista wysyła pytanie dotyczące treści (np. tekstu w polu tekstowym), klasa osłonowa odpytuje model i zwraca odpowiedź programiście. Żądanie zmiany widoku (np. przeniesienia karetki w polu tekstowym) jest przesyłane przez klasę osłonową do widoku. Czasami jednak klasa osłonowa nie wywiązuje się w pełni ze swojego zadania polegającego na przesyłaniu poleceń. W takim przypadku konieczne jest odszukanie za jej pomocą modelu i praca bezpośrednio na nim (nie trzeba pracować bezpośrednio nad widokiem — to zadanie należy do procedur odpowiedzialnych za styl).

Poza byciem właściwym narzędziem do wykonania danego zadania wzorzec MVC był atrakcyjny dla projektantów biblioteki Swing z jeszcze jednego powodu — pozwalał na implementację obieralnego wyglądu, czyli stylu (ang. *pluggable look and feel*). Model przycisku czy pola tekstowego jest niezależny od stylu, ale oczywiście reprezentacja wizualna jest całkowicie zależna od projektu interfejsu użytkownika w konkretnym stylu. Kontroler może zachowywać się różnie. Na przykład w urządzeniu sterowanym głosem musi obsługiwać całkiem inne zdarzenia niż na standardowym komputerze z klawiaturą i myszą. Dzięki oddzieleniu podstawowego modelu od interfejsu użytkownika projektanci biblioteki Swing mogą wielokrotnie wykorzystywać kod w modelach, a nawet przełączać styl w trakcie działania programu.

Oczywiście wzorce to tylko zestawy wskazówek, a nie ścisły zbiór zasad. Żadnego wzorca nie można zastosować we wszystkich sytuacjach. Na przykład wzorzec dotyczący miejsca przy

oknie może być trudny do zastosowania w niektórych pomieszczeniach. Podobnie projektanci biblioteki Swing zetknęli się z brutalną rzeczywistością, dochodząc do wniosku, że implementacja obieralnego wyglądu nie zawsze pozwala na dobrą realizację wzorca model-widok-kontroler. Modele łatwo można oddzielać, a każdy komponent interfejsu użytkownika posiada klasę modelową. Natomiast zakres działań widoku i kontrolera nie zawsze dają się rozdzielić i są one rozproszone w kilku różnych klasach. Oczywiście użytkownika tych klas to zagadnienie nie dotyczy. W rzeczywistości, jak pisaliśmy już wcześniej, programista nie musi też pamiętać o modelach — może zwyczajnie używać klas opakowujących komponenty.

9.1.3. Analiza MVC przycisków Swing

W poprzednim rozdziale nauczyliśmy się używać przycisków, nic nie wiedząc o ich modelu, widoku i kontrolerze. Ponieważ jednak przyciski są jednym z najprostszych elementów interfejsu użytkownika, stanowią dobry punkt zaczepienia przy nabywaniu biegłości w posługiwaniu się wzorcem model-widok-kontroler. Podobne klasy i interfejsy można spotkać także w bardziej zaawansowanych komponentach Swing.

Klasy modelowe większości komponentów implementują interfejs, którego nazwa kończy się słowem `Model`, np. `ButtonModel`. Klasy implementujące ten interfejs mogą definiować stan różnego rodzaju przycisków. Przyciski nie są zbyt skomplikowane i biblioteka Swing zawiera tylko jedną klasę o nazwie `DefaultButtonModel`, która implementuje wspomniany interfejs.

Pogląd na temat tego, jakiego rodzaju dane są przechowywane przez model przycisku, daje przedstawiona poniżej tabela 9.1, zawierająca wykaz i opis właściwości interfejsu `ButtonModel`.

Tabela 9.1. Właściwości interfejsu `ButtonModel`

Nazwa właściwości	Wartość
<code>actionCommand</code>	Łańcuch polecenia działania związanego z przyciskiem
<code>mnemonic</code>	Skrót klawiaturowy dla przycisku
<code>armed</code>	<code>true</code> , jeśli przycisk został naciśnięty i kursor znajduje się nad nim
<code>enabled</code>	<code>true</code> , jeśli przycisk może być używany
<code>pressed</code>	<code>true</code> , jeśli przycisk został naciśnięty, a przycisk myszy nie został jeszcze zwolniony
<code>rollover</code>	<code>true</code> , jeśli kursor znajduje się nad przyciskiem.
<code>selected</code>	<code>true</code> , jeśli przycisk został włączony (używana w przypadku pól wyboru i przełączników)

Każdy obiekt typu `JButton` przechowuje obiekt modelu przycisku, który można z niego wydobyć.

```
 JButton button = new JButton("Niebieski");
 ButtonModel model = button.getModel();
```

W praktyce programistę to niewiele obchodzi — szczegóły dotyczące stanu przycisku mają znaczenie tylko dla widoku, który go rysuje. Ważne informacje, jak to, czy przycisk jest włączony, są dostępne w klasie `JButton` (oczywiście klasa `JButton` pobiera te informacje z modelu).

Przyjrzyjmy się jeszcze raz interfejsowi `ButtonModel`, aby sprawdzić, czego w nim **nie ma**. Model ten **nie** przechowuje etykiety ani ikony przycisku. Nie ma możliwości sprawdzenia, co znajduje się na fronce przycisku, patrząc tylko na jego model (w podrozdziale 9.4.2 o przełącznikach przekonamy się, że czystość projektu jest źródłem problemów dla programisty).

Warto dodać, że **ten sam** model (czyli `DefaultButtonModel`) jest używany dla przycisków, przełączników, pól tekstowych, a nawet elementów menu. Oczywiście każdy z tych typów przycisków posiada inny widok i kontroler. W stylu Metal przycisk `JButton` używa klasy o nazwie `BasicButtonUI` do reprezentacji widoku i klasy `ButtonUIListener` jako kontrolera. Ogólnie z każdym komponentem Swing związany jest obiekt widoku, którego nazwa kończy się skrótem `UI`. Jednak nie każdy komponent Swing posiada dedykowany obiekt kontrolera.

Po przeczytaniu tego wprowadzenia do mechanizmów działania przycisków `JButton` może nasunąć się jedno pytanie: czym w rzeczywistości jest `JButton`? Jest to po prostu klasa osłonowa dziedzicząca po klasie `JComponent`, która przechowuje obiekt `DefaultButtonModel`, dane widoku (takie jak etykieta i ikona przycisku) oraz obiekt `BasicButtonUI` odpowiedzialny za widok przycisku.

9.2. Wprowadzenie do zarządzania rozkładem

Zanim przejdziemy do opisu poszczególnych komponentów Swing, takich jak pola tekstowe i przełączniki, krótko opiszymy techniki rozmieszczania ich w obrębie ramki. JDK w przeciwieństwie do Visual Basica nie posiada projektanta formy. Pozycjonowanie komponentów interfejsu użytkownika odbywa się za pomocą odpowiednio napisanych procedur.

Oczywiście wiele środowisk programistycznych obsługujących Java udostępnia narzędzia służące do automatyzacji wymienionych zadań. Niemniej bardzo ważna jest dokładna znajomość mechanizmów wewnętrznych, ponieważ nawet najlepsze narzędzia zazwyczaj wymagają ręcznego dostrojenia.

Zaczniemy od programu z rozdziału 8., który zmieniał kolor tła w odpowiedzi na naciśnięcie przycisku (rysunek 9.5).

Rysunek 9.5.

Panel z trzema przyciskami

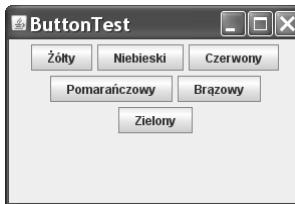


Przyciski te znajdują się na panelu `JPanel` i podlegają **zarządcy rozkładu ciągłego** (ang. *flow layout manager*), czyli domyльнemu zarządcy rozkładu panelu. Rysunek 9.6 pokazuje, co się dzieje, kiedy do panelu dodamy więcej przycisków. Jak widać, kiedy nie ma już miejsca, następuje przejście do nowego wiersza.

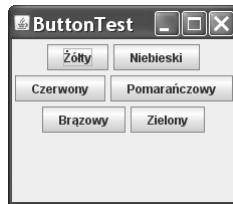
Ponadto przyciski zostają na środku, nawet jeśli rozmiar okna się zmienia (rysunek 9.7).

Rysunek 9.6.

Panel z sześcioma przyciskami zarządzanymi przez zarządcę rozkładu

**Rysunek 9.7.**

Zmiana rozmiaru panelu powoduje automatyczne przegrupowanie przycisków



Ogólnie **komponenty** znajdują się w **kontenerze**, a **zarządcą rozkładu** określa położenie i rozmiar komponentów w kontenerze.

Przyciski, pola tekstowe i inne elementy interfejsu użytkownika rozszerzają klasę Component. Komponenty mogą się znajdować w takich kontenerach jak panele. Ponieważ panele same mogą być umieszczane w innych kontenerach, klasa Container dziedziczy po klasie Component. Rysunek 9.8 przedstawia hierarchię dziedziczenia klasy Component.



Niestety powyższa hierarchia dziedziczenia jest w dwóch miejscach niejasna. Po pierwsze, okna najwyższego rzędu, jak JFrame, są podklasami klasy Container, a więc także klasy Component, ale nie mogą być umieszczane wewnętrz innych kontenerów. Po drugie, klasa JComponent jest podkąsą klasy Container, a nie Component, przez co do JButton można dodawać inne komponenty (choć nie zostałyby one wyświetlane).

Każdy kontener posiada domyślnego zarządcę rozkładu, ale można utworzyć też własny. Na przykład poniższa instrukcja rozmieszcza komponenty w panelu za pomocą klasy GridLayout:

```
panel.setLayout(new GridLayout(4, 4));
```

Programista dodaje komponenty do kontenera. Metoda add tego kontenera przekazuje komponent i dane dotyczące jego umiejscowienia do zarządcy rozkładu.

java.awt.Container 1.0

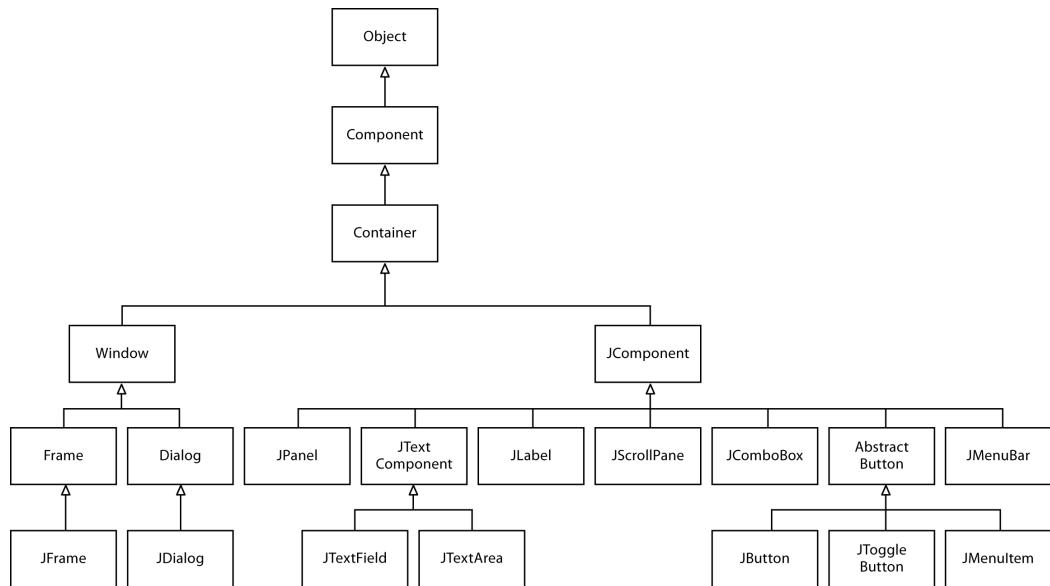
■ void setLayout(LayoutManager m)

Ustawia zarządcę rozkładu dla kontenera.

■ Component add(Component c)

■ Component add(Component c, Object constraints) **1.1**

Dodaje komponent do kontenera i zwraca referencję do tego komponentu.



Rysunek 9.8. Hierarchia dziedziczenia klasy `Component`

Parametry: `c` Komponent, który ma zostać dodany.
`constraints` Identyfikator zrozumiałej dla zarządcy rozkładu.

`java.awt.FlowLayout 1.0`

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`

Parametry: `align` Jedna z trzech wartości: `LEFT`, `CENTER`, `RIGHT`.
`hgap` Przerwa w poziomie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).
`vgap` Przerwa w pionie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).

9.2.1. Rozkład brzegowy

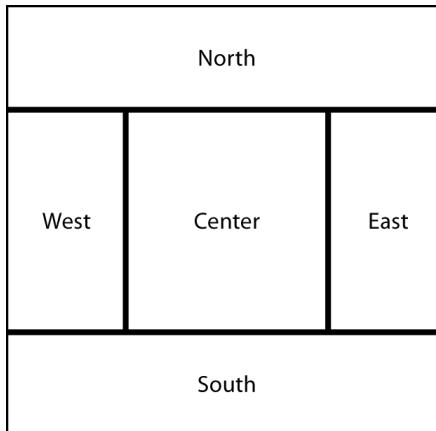
Zarządcą rozkładu brzegowego (ang. *border layout manager*) jest domyślny dla panelu z treścią komponentów `JFrame`. W przeciwieństwie do zarządcy rozkładu ciągłego, który w pełni kontroluje położenie każdego komponentu, zarządcą rozkładu brzegowego pozwala wybrać miejsce dla każdego komponentu. Do określania położenia wykorzystywane są kierunki świata: północ, południe, wschód i zachód, oraz środek (rysunek 9.9).

Na przykład:

```
frame.add(component, BorderLayout.SOUTH);
```

Rysunek 9.9.

Rozkład brzegowy



Najpierw ustawiane są komponenty położone przy brzegach, a resztę powierzchni zajmuje środek. Kiedy zmienia się rozmiar komponentu, zmienia się tylko rozmiar środka, zaś rozmiar elementów brzegowych pozostaje niezmieniony. Dodawanie komponentów polega na zastosowaniu stałych CENTER, NORTH, SOUTH, EAST i WEST klasy BorderLayout. Nie wszystkie miejsca muszą być zajęte. W przypadku braku wartości przyjmowana jest wartość CENTER.



Stałe klasy BorderLayout są zdefiniowane jako łańcuchy. Na przykład stała BorderLayout.SOUTH jest zdefiniowana jako South. Wielu programistów woli używać łańcuchów, ponieważ są krótsze, np. frame.add(component, "South"). Jeśli jednak w łańcuchu znajdzie się błąd, kompilator go nie przechwyci.

W przeciwnieństwie do rozkładu ciągłego, rozkład brzegowy rozciąga komponenty na całą dostępną przestrzeń (rozkład ciągły pozostawia preferowany rozmiar komponentu). W przypadku przycisków może to być problemem:

```
frame.add(yellowButton, BorderLayout.SOUTH); //nie
```

Rysunek 9.10 przedstawia wynik zastosowania powyższego fragmentu programu. Przycisk został rozciągnięty na całą szerokość obszaru południowego ramki. Gdyby został dodany kolejny przycisk, zastąpiłby on swojego poprzednika.

Rysunek 9.10.

Jeden przycisk
w rozkładzie
brzegowym



Ten problem można rozwiązać za pomocą dodatkowych paneli. Spójrzmy na przykład na rysunek 9.11. Trzy przyciski umieszczone na samym dole ekranu znajdują się na panelu. Panel został ustawiony w południowej części panelu z treścią.

Rysunek 9.11.

Panel umieszczony w południowej części ramki



Aby osiągnąć taką konfigurację, najpierw należy utworzyć obiekt `JPanel`, a następnie dodać do niego wszystkie przyciski. Domyślnym zarządcą rozkładu w panelu jest `FlowLayout`, który w tym przypadku stanowi dobry wybór. Poszczególne przyciski należy dodawać do panelu za pomocą omawianej już metody `add`. Położenie i rozmiar przycisków są kontrolowane przez zarządcę rozkładu `FlowLayout`. Dzięki temu przyciski będą się znajdować na środku panelu i nie będą się rozciągać na cały dostępny obszar. Na końcu należy dodać panel do panelu z treścią ramki.

```
 JPanel panel = new JPanel();
 panel.add(yellowButton);
 panel.add(blueButton);
 panel.add(redButton);
 frame.add(panel, BorderLayout.SOUTH);
```

Rozkład brzegowy rozciąga panel na cały obszar południowy.

`java.awt.BorderLayout` **1.0**

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`

Tworzy nowy `BorderLayout`.

Parametry:	<code>hgap</code>	Przerwa w poziomie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).
	<code>vgap</code>	Przerwa w pionie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).

9.2.2. Rozkład siatkowy

Rozkład siatkowy polega na rozmieszczeniu komponentów w wierszach i kolumnach tworzących siatkę. Wszystkie komponenty mają ten sam rozmiar. Przyciski kalkulatora przedstawionego na rysunku 9.12 zostały rozmieszczone za pomocą rozkładu siatkowego. W miarę zwiększania i zmniejszania okna przyciski rosną lub kurczą się, ale wszystkie mają takie same rozmiary.

W konstruktorze obiektu rozkładu siatkowego należy określić, ile wierszy i kolumn ma zostać utworzonych.

```
panel.setLayout(new GridLayout(4, 4));
```

Rysunek 9.12.

Kalkulator



Komponenty są dodawane najpierw do pierwszego wiersza, potem do drugiego itd.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

Listing 9.1 przedstawia kod źródłowy tego kalkulatora. Jest to zwykły kalkulator, a nie wersja z odwróconą notacją polską, która — nie wiedzieć czemu — zyskała sobie tak dużą popularność w publikacjach na temat Javy. W tym programie po dodaniu komponentu do ramki następuje wywołanie metody pack. Metoda ta oblicza wysokość i szerokość ramki, wykorzystując preferowane rozmiary wszystkich komponentów.

Listing 9.1. calculator/CalculatorPanel.java

```
package calculator;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Panel z przyciskami kalkulatora i wyświetlaczem wyniku.
 */
public class CalculatorPanel extends JPanel
{
    private JButton display;
    private JPanel panel;
    private double result;
    private String lastCommand;
    private boolean start;

    public CalculatorPanel()
    {
        setLayout(new BorderLayout());
        result = 0;
        lastCommand = "=";
        start = true;
        // Dodanie wyświetlacza
        display = new JButton("0");
        display.setEnabled(false);
        add(display, BorderLayout.NORTH);
        ActionListener insert = new InsertAction();
        ActionListener command = new CommandAction();
```

```
// Wstawienie przycisków na siatkę 4×4

panel = new JPanel();
panel.setLayout(new GridLayout(4, 4));

 addButton("7", insert);
 addButton("8", insert);
 addButton("9", insert);
 addButton("/", command);

 addButton("4", insert);
 addButton("5", insert);
 addButton("6", insert);
 addButton("*", command);

 addButton("1", insert);
 addButton("2", insert);
 addButton("3", insert);
 addButton("-", command);

 addButton("0", insert);
 addButton(".", insert);
 addButton("=", command);
 addButton("+", command);

 add(panel, BorderLayout.CENTER);
}

/*
* Dodaje przycisk do panelu centralnego.
* @param label etykieta przycisku
* @param listener sluchacz przycisków
*/

private void addButton(String label, ActionListener listener)
{
    JButton button = new JButton(label);
    button.addActionListener(listener);
    panel.add(button);
}

/*
* Ta akcja wstawia łańcuch akcji przycisku na końcu tekstu do wyświetlenia.
*/

private class InsertAction implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String input = event.getActionCommand();
        if (start)
        {
            display.setText("");
            start = false;
        }
        display.setText(display.getText() + input);
    }
}
```

```

    /**
 * Ta akcja wykonuje polecenia określone przez akcję przycisku.
 */
private class CommandAction implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String command = event.getActionCommand();

        if (start)
        {
            if (command.equals("-"))
            {
                display.setText(command);
                start = false;
            }
            else lastCommand = command;
        }
        else
        {
            calculate(Double.parseDouble(display.getText()));
            lastCommand = command;
            start = true;
        }
    }
}

/**
 * Wykonuje oczekujące działania.
 * @param x wartość, która ma być połączona z poprzednim wynikiem.
 */
public void calculate(double x)
{
    if (lastCommand.equals("+")) result += x;
    else if (lastCommand.equals("-")) result -= x;
    else if (lastCommand.equals("*")) result *= x;
    else if (lastCommand.equals("/")) result /= x;
    else if (lastCommand.equals("=")) result = x;
    display.setText(" " + result);
}
}

```

Oczywiście niewiele programów ma tak regularny interfejs jak kalkulator. W praktyce wykorzystywane są niewielkie siatki (zazwyczaj składające się z jednego wiersza lub jednej kolumny), za pomocą których ustawia się niektóre obszary okna. Na przykład panel z rozkładem siatkowym można wykorzystać do utworzenia szeregu przycisków o takich samych rozmiarach.

java.awt.GridLayout **1.0**

- GridLayout(int rows, int columns)
- GridLayout(int rows, int columns, int hgap, int vgap)

Tworzy nowy obiekt GridLayout. Parametr rows lub columns (ale nie oba naraz) może mieć wartość zero, co oznacza dowolną liczbę komponentów w wierszu lub kolumnie.

Parametry:	rows	Liczba wierszy siatki.
	columns	Liczba kolumn siatki.
	hgap	Przerwa w poziomie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).
	vgap	Przerwa w pionie w pikselach (wartości ujemne powodują nachodzenie na siebie elementów).

9.3. Wprowadzanie tekstu

Jesteśmy już gotowi na wprowadzenie elementów Swing interfejsu użytkownika. Zaczniemy od komponentów pozwalających na wprowadzanie i edycję tekstu. Dane tekstowe można odbierać za pomocą komponentów JTextField i JTextArea. Pole tekstowe (ang. *text field*) przyjmuje tylko jeden wiersz tekstu, a obszar tekstowy (ang. *text area*) wiele wierszy. Pole hasła JPasswordField przyjmuje jeden wiersz tekstu, nie pokazując jego treści.

Wszystkie trzy wymienione klasy dziedziczą po klasie JTextComponent. Obiektu samej tej klasy nie można utworzyć, ponieważ jest to klasa abstrakcyjna. Z drugiej strony, jak to często bywa w Javie, podczas przeszukiwania API poszukiwane metody mogą się znajdować w nadklasie JTextComponent, a nie w jednej z jej podklas. Na przykład metody pobierające i ustawiające tekst w polu tekstowym i obszarze tekstowym należą do klasy JTextComponent.

javax.swing.text.JTextComponent 1.2

- String getText()
 - void setText(String text)
- Pobiera lub ustawia tekst komponentu.
- boolean isEditable()
 - void setEditable(boolean b)

Pobiera lub ustawia właściwość editable, która określa, czy użytkownik może edytować zawartość komponentu tekstowego.

9.3.1. Pola tekstowe

Najczęściej pola tekstowe są dodawane do okien za pośrednictwem panelu lub innego kontenera — podobnie jak przyciski:

```
 JPanel panel = new JPanel();
 JTextField textField = new JTextField("Łańcuch testowy", 20);
 panel.add(textField);
```

Powyższy fragment programu dodaje pole tekstowe i inicjuje je, wstawiając do niego łańcuch testowy. Drugi parametr tego konstruktora ustawia szerokość pola. W tym przypadku jest to 20 „kolumn”. Niestety kolumna nie należy do precyzyjnych jednostek miary. Jedna kolumna ma szerokość jednego znaku fontu użytego do napisania tekstu. Dzięki temu, jeśli spodziewanych jest n znaków tekstu lub mniej, szerokość kolumny można ustawić na n . Metoda ta nie sprawdza się jednak dobrze w praktyce. Dla pewności należy zawsze dodać 1 lub 2 do maksymalnej długości danych wejściowych. Ponadto należy pamiętać, że liczba kolumn jest tylko wskazówką dla AWT, która określa **preferowany** rozmiar. Jeśli zarządcą rozkładu jest zmuszony zwiększyć lub zmniejszyć pole tekstowe, może odpowiednio dostosować jego rozmiar. Szerokość kolumny ustawiona w konstruktorze JTextField nie stanowi górnego limitu znaków, które może wprowadzić użytkownika. Możliwe jest wpisanie dłuższego łańcucha, który po przekroczeniu szerokości pola będzie można przewijać. Użytkownicy nie lubią przewijanych pól tekstowych, a więc nie należy skąpić dla nich miejsca. Liczbę kolumn można ustawić ponownie w czasie działania programu za pomocą metody `setColumns()`.



Po zmianie rozmiaru pola tekstowego za pomocą metody `setColumns()` należy wywołać metodę `revalidate()` zawierającą ją kontenera.

```
textField.setColumns(10);
panel.revalidate();
```

Metoda `revalidate()` ponownie ustala rozmiar i rozkład wszystkich komponentów znajdujących się w kontenerze. Po użyciu metody `revalidate()` zarządcą rozkładu zmienia rozmiar kontenera, dzięki czemu może być widoczne pole tekstowe o zmienionym rozmiarze.

Metoda `revalidate()` należy do klasy `JComponent`. Nie zmienia ona rozmiaru komponentu, ale zaznacza go jako kandydata do takiej zmiany. Podejście to pozwala uniknąć powtarzania obliczeń, w przypadku gdy konieczna jest zmiana rozmiaru wielu komponentów. Aby jednak obliczyć ponownie rozmiar wszystkich komponentów w ramce `JFrame`, należy wywołać metodę `validate()` — klasa `JFrame` nie dziedziczy po klasie `JComponent`.

Z reguły pole tekstowe ma na celu umożliwienie użytkownikowi wprowadzenia (lub edycji istniejącego) tekstu. Bardzo często na początku pola te są puste. Aby tak było, należy połuć parametr łańcuchowy w konstruktorze `JTextField`:

```
JTextField textField = new JTextField(20);
```

Tekst w polu tekstowym można zmienić w dowolnej chwili za pomocą metody `setText()` z klasy `JTextComponent`, o której mowa wcześniej. Na przykład:

```
textField.setText("Dzień dobry!");
```

Do sprawdzania, co wpisał użytkownik w polu tekstowym, służy metoda `getText()`. Zwraca ona łańcuch tekstu, który został wprowadzony przez użytkownika. Aby usunąć wiodące i końcowe białe znaki z tekstu znajdującego się w polu tekstowym, należy wywołać metodę `trim()` na rzecz danych zwracanych przez metodę `getText()`:

```
String text = textField.getText().trim();
```

Do ustawiania kroju czcionki tekstu wprowadzanego przez użytkownika służy metoda `setFont`.

javax.swing.JTextField 1.2

- `JTextField(int cols)`

Tworzy puste pole `JTextField` o szerokości określonej przez podaną liczbę kolumn.

- `JTextField(String text, int cols)`

Tworzy pole tekstowe zawierające określony łańcuch znaków i mające określona szerokość w kolumnach.

- `int getColumns()`

- `void setColumns(int cols)`

Pobiera lub ustawia liczbę kolumn pola tekstowego.

javax.swing.JComponent 1.2

- `void revalidate()`

Wymusza ponowne ustalenie położenia i rozmiaru komponentu.

- `void setFont(Font f)`

Ustawia kroj czcionki dla komponentu.

java.awt.Component 1.0

- `void validate()`

Ponownie ustala położenie i rozmiar komponentu. Jeśli komponent jest kontenerem, ponownie ustalane są położenie i rozmiar zawartych w nim komponentów.

- `Font getFont()`

Pobiera nazwę kroju czcionki komponentu.

9.3.2. Etykiety komponentów

Etykiety są komponentami przechowującymi tekst. Nie posiadają żadnych ozdobników (na przykład obramowania). Nie reagują na dane wprowadzane przez użytkownika. Etykieta może być identyfikatorem komponentu. Na przykład pola tekstowe, w przeciwieństwie do przycisków, nie posiadają identyfikujących je etykiet. Aby nadać etykietę komponentowi, który standardowo nie ma identyfikatora, należy:

- 1 Utworzyć komponent `JLabel` z odpowiednim tekstem.
- 2 Umieścić ten komponent w odpowiedniej odległości od komponentu, który ma być identyfikowany, dzięki czemu użytkownik odniesie wrażenie, że etykieta dotyczy właściwego komponentu.

Konstruktor klasy `JLabel` pozwala określić początkowy tekst lub ikonę oraz opcjonalnie wyrównanie treści. Do wyrównania należy używać stałych z interfejsu `SwingConstants`.

Interfejs ten definiuje kilka bardzo przydatnych stałych, jak LEFT, RIGHT, CENTER, NORTH, EAST itd. Klasa JLabel jest jedną z wielu klas Swing, które implementują ten interfejs. W związku z tym etykietę z wyrównaniem do prawej można utworzyć na dwa sposoby:

```
JLabel label = new JLabel("Nazwa użytkownika: ", SwingConstants.RIGHT);
```

lub

```
JLabel label = new JLabel("Nazwa użytkownika: ", JLabel.RIGHT);
```

Metody setText i setIcon umożliwiają ustawienie tekstu i ikony etykiety w czasie działania programu.



W przyciskach, etykietach i elementach menu poza zwykłym tekstem można używać języka HTML. Nie zalecamy jednak tego przy przyciskach, ponieważ zakłócony zostaje oryginalny styl. Natomiast w etykietach HTML pozwala uzyskać ciekawe efekty. Jedyne, co trzeba zrobić, to otoczyć łańcuch etykiety znacznikami <html>...</html>:

```
label = new JLabel("<html><b>Wymagany</b> tekst:</html>");
```

Uwaga: wyświetlenie pierwszego komponentu z etykietą HTML zabiera sporo czasu ze względu na konieczność załadowania dość skomplikowanych procedur obsługujących HTML.

Etykiety można pozycjonować w kontenerze tak samo jak inne komponenty. Oznacza to, że aby umieścić etykietę w pożądanym miejscu, można zastosować opisane wcześniej techniki.

javax.swing.JLabel 1.2

- JLabel(String text)
- JLabel(Icon icon)
- JLabel(String text, int align)
- JLabel(String text, Icon icon, int align)

Tworzy etykietę.

Parametry:	text	Tekst etykiety
	icon	Ikona etykiety
	align	Jedna ze stałych interfejsu SwingConstants: LEFT (domyślna), CENTER lub RIGHT

- String getText()
- void setText(String text)

Pobiera lub ustawia tekst etykiety.

- Icon getIcon()
- void setIcon(Icon icon)

Pobiera lub ustawia ikonę etykiety.

9.3.3. Pola haseł

Pole hasła to specjalny rodzaj pola tekstowego. Znaki wpisywane w takie pole są niewidoczne dla wąszych osób stojących w pobliżu. Zamiast wpisywanych znaków pojawiają się tak zwane **znaki echo** — zazwyczaj gwiazdki. W bibliotece Swing dostępna jest klasa `JPasswordField`, która umożliwia tworzenie tego typu pól.

Pole hasła stanowi jeszcze jeden dowód na potwierdzenie bardzo dużych możliwości wzorca MVC. Pole hasła wykorzystuje do przechowywania danych te same mechanizmy co zwykłe pole tekstowe, ale jego widok został zmieniony w taki sposób, że zamiast wpisywanych znaków pojawiają się znaki zastępcze.

`javax.swing.JPasswordField 1.2`

- `JPasswordField(String text, int columns)`

Tworzy pole hasła.

- `void setEchoChar(char echo)`

Ustawia znak echo dla pola hasła. Jest to wartość doradcza — określony styl może wymusić stosowanie własnego znaku. Wartość 0 przywraca domyślny znak.

- `char[] getPassword()`

Zwraca tekst zawarty w polu hasła. Aby zwiększyć bezpieczeństwo, należy nadpisać zawartość zwróconej tablicy, kiedy nie jest już potrzebna (hasło nie jest zwracane jako łańcuch, ponieważ łańcuchy pozostają w maszynie wirtualnej, aż zostaną usunięte przez system zbierania nieużytków).

9.3.4. Obszary tekstowe

Czasami konieczne jest odebranie od użytkownika tekstu o długości przekraczającej jeden wiersz. Do tego celu służy komponent `JTextArea`. W obszarze tekstowym użytkownik może wpisać dowolną liczbę wierszy, do rozdzielenia których służy klawisz *Enter*. Każdy wiersz kończy się symbolem `\n`. Rysunek 9.13 przedstawia obszar tekstowy w działaniu.

W konstruktorze komponentu `JTextArea` określa się liczbę wierszy i kolumn zajmowanych przez tworzony obszar. Na przykład:

```
textArea = new JTextArea(8, 40); // 8 wierszy po 40 kolumn
```

Parametr `columns` ma takie samo przeznaczenie jak poprzednio. Nadal trzeba pamiętać o dodaniu kilku dodatkowych kolumn. Podobnie jak wcześniej, liczba wierszy i kolumn nie ogranicza możliwości użytkownika. Jeśli tekst jest zbyt długi, pojawiają się paski przewijania. Do zmiany liczby wierszy i kolumn służą, podobnie jak wcześniej, metody `setRows` i `setColumns`. Liczby te określają tylko preferowany rozmiar — zarządcą rozkładu może zmniejszyć lub zwiększyć rozmiar pola tekowego.

Rysunek 9.13.

Komponenty tekstowe



Jeśli tekst nie mieści się w obszarze tekstowym, część tekstu zostaje obcięta. Aby tego uniknąć, można włączyć zawijanie wierszy:

```
textArea.setLineWrap(true); // Włączono zawijanie wierszy.
```

Zawijanie wierszy jest wyłącznie efektem wizualnym. Nie powoduje ono wstawiania znaków \n.

9.3.5. Panele przewijane

W Swingu obszar tekstowy nie posiada pasków przewijania. Aby się pojawiły, należy obszar ten umieścić wewnątrz **panelu przewijanego** (ang. *scroll pane*).

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Po zastosowaniu powyższego fragmentu kodu widokiem obszaru tekstowego zarządza panel przewijany. Paski przewijania pojawiają się automatycznie, jeśli tekst nie mieści się w obszarze tekstowym, oraz same znikają, kiedy tekstu zrobi się mniej. Przewijanie jest obsługiwane wewnątrz panelu przewijanego — pisany przez programistę program nie musi przetwarzać zdarzeń przewijania.

Ten ogólny mechanizm działa we wszystkich komponentach, nie tylko obszarach tekstowych. Aby dodać do komponentu paski przewijania, należy umieścić go w panelu przewijanym.

Listing 9.2 przedstawia różne komponenty tekstowe. Program wyświetla pole tekstowe, pole hasła oraz obszar tekstowy z paskami przewijania. Pole tekstowe i hasła mają etykiety. Kliknięcie przycisku *Wstaw* powoduje wstawienie zawartości pól tekstowych do obszaru tekstowego.

Listing 9.2. text/TextComponentFrame.java

```
package text;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z przykładowymi komponentami tekstowymi.
 */


```

```

public class TextComponentFrame extends JFrame
{
    public static final int TEXTAREA_ROWS = 8;
    public static final int TEXTAREA_COLUMNS = 20;

    public TextComponentFrame()
    {
        final JTextField textField = new JTextField();
        final JPasswordField passwordField = new JPasswordField();

        JPanel northPanel = new JPanel();
        northPanel.setLayout(new GridLayout(2, 2));
        northPanel.add(new JLabel("Nazwa użytkownika: ", SwingConstants.RIGHT));
        northPanel.add(textField);
        northPanel.add(new JLabel("Hasło: ", SwingConstants.RIGHT));
        northPanel.add(passwordField);

        add(northPanel, BorderLayout.NORTH);

        final JTextArea textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
        JScrollPane scrollPane = new JScrollPane(textArea);

        add(scrollPane, BorderLayout.CENTER);

        // Dodanie przycisku wstawiającego tekst do obszaru tekstowego

        JPanel southPanel = new JPanel();

        JButton insertButton = new JButton("Wstaw");
        southPanel.add(insertButton);
        insertButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                textArea.append("Nazwa użytkownika: " + textField.getText() + " "
                + "Hasło: "
                + new String(passwordField.getPassword()) + "\n");
            }
        });
    }

    add(southPanel, BorderLayout.SOUTH);
    pack();
}
}

```



Komponent JTextArea wyświetla wyłącznie czysty tekst, bez specjalnych krojów czcionek i formatowania. Aby wyświetlać sformatowany tekst (np. HTML), można użyć klasy JEditorPane, która została opisana w drugim tomie.

javax.swing.JTextArea 1.2

- JTextArea()
- JTextArea(int rows, int cols)

- `JTextArea(String text, int rows, int cols)`
Tworzy obszar tekstowy.
- `void setColumns(int cols)`
Ustawia preferowaną liczbę kolumn szerokości obszaru tekstowego.
- `void setRows(int rows)`
Ustawia preferowaną liczbę wierszy wysokości obszaru tekstowego.
- `void append(String newText)`
Wstawia podany tekst na końcu tekstu znajdującego się w obszarze tekstowym.
- `void setLineWrap(boolean wrap)`
Włącza lub wyłącza zawijanie wierszy.
- `void setWrapStyleWord(boolean word)`
Wartość `true` oznacza zawijanie wierszy z uwzględnieniem całych wyrazów.
Wartość `false` oznacza zawijanie wierszy bez uwzględnienia całych wyrazów.
- `void setTabSize(int c)`
Ustawia tabulację na `c` kolumn. Należy zauważyć, że tabulatory nie są zamieniane na spacje, ale powodują wyrównanie z kolejnym tabulatorem.

javax.swing.JScrollPane **1.2**

- `JScrollPane(Component c)`
Tworzy panel przewijany wyświetlający zawartość określonego komponentu. Paski przewijania pojawiają się, kiedy komponent jest większy niż widok.

9.4. Komponenty umożliwiające wybór opcji

Umiemy już odbierać dane od użytkowników, ale w wielu sytuacjach lepszym rozwiążaniem jest podanie kilku opcji do wyboru niż pozwolenie użytkownikom na samodzielne wpisanie informacji. Opcje do wyboru może reprezentować zestaw przycisków lub lista elementów (metoda ta nie wymaga sprawdzania błędów). W tym podrozdziale omawiamy tworzenie pól wyboru (ang. *checkbox*), przełączników (ang. *radio button*), list opcji do wyboru oraz suwaków (ang. *slider*).

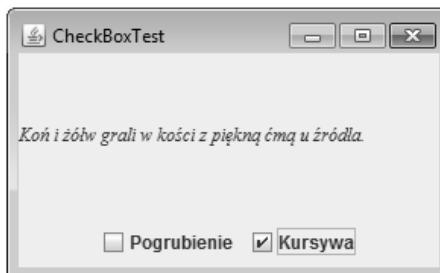
9.4.1. Pola wyboru

Do odbierania danych typu „tak” lub „nie” najlepiej nadają się pola wyboru. Pola te mają automatycznie nadawane etykiety. Zaznaczenie opcji polega na kliknięciu wybranego pola, a usunięcie zaznaczenia na kliknięciu pola, które jest zaznaczone. Do włączania lub wyłączania opcji można też użyć klawisza spacji, kiedy pole wyboru jest aktywne.

Rysunek 9.14 przedstawia prosty program zawierający dwa pola wyboru. Jedno z nich włącza lub wyłącza atrybut kursywy czcionki, a drugie robi to samo z atrybutem pogrubienia. Zauważmy, że pole po prawej stronie jest aktywne, na co wskazuje prostokątna obwódka. Kiedy użytkownik kliknie jedno z pól wyboru, następuje odświeżenie ekranu i zastosowanie nowych atrybutów czcionki.

Rysunek 9.14.

Pola wyboru



Obok pola wyboru musi się znajdować etykieta identyfikacyjna. Jej tekst ustala się w konstruktorze.

```
bold = new JCheckBox("Pogrubienie");
```

Do włączania i wyłączania opcji służy metoda `setSelected`. Na przykład:

```
bold.setSelected(true);
```

Metoda `setSelected` sprawdza aktualny stan każdego pola wyboru. Jest to `false`, jeśli pole wyboru nie jest zaznaczone, a `true`, jeśli jest zaznaczone.

Kliknięcie pola wyboru przez użytkownika uruchamia akcję. Z polem wyboru — jak zawsze — wiążemy słuchacza akcji. W omawianym programie oba pola współdzielą jednego słuchacza.

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

Metoda `actionPerformed` sprawdza stan pól *Kursywa* i *Pogrubienie* oraz ustawia czcionkę panelu na zwykłą, pogrubioną, kursywę lub pogrubioną kursywę.

```
public void actionPerformed(ActionEvent event)
{
    int mode = 0;
    if (bold.isSelected()) mode += Font.BOLD;
    if (italic.isSelected()) mode += Font.ITALIC;
    label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

Listing 9.3 przedstawia pełny kod źródłowy tego programu.

Listing 9.3. checkBox/CheckBoxTest.java

```
package checkBox;

import java.awt.*;
import javax.swing.*;
```

```

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class CheckBoxTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new CheckBoxFrame();
                frame.setTitle("CheckBoxTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

```

javax.swing.JCheckBox 1.2

- `JCheckBox(String label)`
 - `JCheckBox(String label, Icon icon)`
- Tworzy pole wyboru, które nie jest początkowo zaznaczone.
- `JCheckBox(String label, boolean state)`
- Tworzy pole wyboru z podaną etykietą o określonym stanie początkowym.
- `boolean isSelected ()`
 - `void setSelected(boolean state)`

Pobiera lub ustawia stan pola wyboru.

9.4.2. Przelączniki

W poprzednim programie można było zaznaczyć jedno z pól, oba lub nie zaznaczyć żadnego. Istnieje wiele sytuacji, w których chcemy, aby użytkownik mógł wybrać tylko jedną z kilku opcji. Zaznaczenie jednego pola powoduje automatyczne usunięcie zaznaczenia innego. Grupy tego typu pól są też często nazywane **grupami przycisków radiowych**, ponieważ działaniem przypominają przyciski wyboru w starym radiu. Wciśnięcie jednego przycisku powoduje, że wcześniej wciśnięty przycisk wyskakuje. Rysunek 9.15 przedstawia typowy przykład ich zastosowania. Użytkownik ma do wyboru cztery rozmiary czcionki: *Mała*, *Średnia*, *Duża* i *Bardzo duża* — oczywiście możliwy jest wybór tylko jednej opcji naraz.

Implementacja grup przycisków radiowych w Swingu jest łatwa. Dla każdej grupy należy utworzyć obiekt typu `ButtonGroup`. Następnie do tego obiektu dodaje się obiekty typu `JRadio->Button`. Zadaniem obiektu grupy przycisków jest wyłączanie wcześniej włączonego przycisku w odpowiedzi na włączenie innego.

Rysunek 9.15.

Grupa przycisków radiowych (przełączników)



```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton = new JRadioButton("Mała", false);
group.add(smallButton);

JRadioButton mediumButton = new JRadioButton("Średnia", true);
group.add(mediumButton);

. . .
```

Drugi argument konstruktora powinien mieć wartość `true` w przełączniku, który ma być włączony na początku, oraz `false` w pozostałych przełącznikach. Należy pamiętać, że obiekt grupy przycisków kontroluje tylko ich **zachowanie**. Do grupowania przycisków w celu odpowiedniej ich aranżacji należy użyć jakiegoś kontenera, jak np. `JPanel`.

Wracając do rysunków 9.14 i 9.15, można zauważyc, że przyciski radiowe wyglądają inaczej niż pola wyboru. Te drugie są prostokątne i po zaznaczeniu pokazuje się w nich haczyk. Przyciski radiowe są okrągłe, a kiedy się je kliknie, pojawia się w nich kropka.

Mechanizm powiadamiania o zdarzeniach w przełącznikach jest taki sam jak we wszystkich innych przyciskach. Kiedy użytkownik kliką przełącznik, generuje on akcję. Omawiany program zawiera definicję słuchacza akcji, który ustawia odpowiedni rozmiar czcionki:

```
ActionListener listener = new
    ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        // Parametr size odwołuje się do ostatniego parametru metody addRadioButton.
        label.setFont(new Font("Serif", Font.PLAIN, size));
    }
};
```

Porównajmy powyższego słuchacza ze słuchaczem z programu z polami wyboru. Każdy przełącznik posiada osobny obiekt słuchacza. Każdy z tych obiektów ma jasno określone zadanie — ustawienie rozmiaru czcionki na określoną wartość. W przypadku pól wyboru zastosowaliśmy nieco inną metodę. Oba pola mają tego samego słuchacza akcji. Wywołuje on metodę, która sprawdza aktualny stan obu pól.

Czy można by było zastosować tę metodę w przypadku przełączników? Można by było utworzyć jednego słuchacza, który obliczałby rozmiar w następujący sposób:

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```

Woleliśmy jednak zastosować osobne obiekty nasłuchujące akcji, ponieważ ściślej wiążą wartości z przyciskami.



Wiadomo, że w grupie przełączników tylko jeden z nich jest włączony. Przydałaby się możliwość sprawdzenia, który jest włączony, bez potrzeby sprawdzania wszystkich przycisków w grupie. Ponieważ obiekt `ButtonGroup` kontroluje wszystkie przyciski, dobrze by było, gdyby udostępniał referencję do wciśniętego przycisku. Klasa `ButtonGroup` zawiera metodę `getSelection`, ale nie zwraca ona przycisku, który jest wciśnięty. Zwraca natomiast referencję `ButtonModel` do modelu związanego z tym przyciskiem. Niestety żadna z metod klasy `ButtonModel` nie jest zbyt pomocna. Interfejs `ButtonModel` dziedziczy metodę `getSelectedObjects` po interfejsie `ItemSelectable`, która jest bezużyteczna, zwracając wartość `null`. Obiecująca jest metoda `getActionCommand`, ponieważ „polecenie akcji” przełącznika jest jego etykietą tekstową. Jednak polecenie akcji jego modelu ma wartość `null`. Wartości poleceń akcji modeli są ustawiane tylko wtedy, gdy zostaną bezpośrednio ustawione polecenia akcji wszystkich przełączników za pomocą metody `setActionCommand`. Dzięki temu można sprawdzić polecenie akcji aktualnie zaznaczonego przycisku za pomocą wywołania `buttonGroup.getSelection().getActionCommand()`.

Listing 9.4 przedstawia kompletny kod programu ustawiającego rozmiar czcionki za pomocą przełączników.

Listing 9.4. radioButton/RadioButtonFrame.java

```
package radioButton;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z przykładową etykietą tekstową i przełącznikami służącymi do wyboru rozmiaru czcionki
 */
public class RadioButtonFrame extends JFrame
{
    private JPanel buttonPanel;
    private ButtonGroup group;
    private JLabel label;
    private static final int DEFAULT_SIZE = 36;

    public RadioButtonFrame()
    {
        // Dodanie przykładowej etykiety tekstowej

        label = new JLabel("Koń i żółw grali w kości z piękną ćmą u źródła.");
        label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
        add(label, BorderLayout.CENTER);

        // Dodanie przełączników

        buttonPanel = new JPanel();
        group = new ButtonGroup();

        addRadioButton("Mała", 8);
        addRadioButton("Średnia", 12);
        addRadioButton("Duża", 18);
        addRadioButton("Bardzo duża", 36);
    }
}
```

```
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }

    /**
     * Tworzy przełącznik ustawiający rozmiar czcionki przykładowego tekstu.
     * @param name łańcuch identyfikujący przełącznik
     * @param size rozmiar czcionki ustawiany przez ten przełącznik
     */

    public void addRadioButton(String name, final int size)
    {
        boolean selected = size == DEFAULT_SIZE;
        JRadioButton button = new JRadioButton(name, selected);
        group.add(button);
        buttonPanel.add(button);

        // Ten słuchacz ustawia rozmiar czcionki etykiety

        ActionListener listener = new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                // Parametr size odwołuje się do ostatniego parametru metody addRadioButton
                label.setFont(new Font("Serif", Font.PLAIN, size));
            }
        };
        button.addActionListener(listener);
    }
}
```

javax.swing.JRadioButton 1.2

- `JRadioButton(String label, Icon icon)`
Tworzy początkowo niezaznaczony przełącznik.
- `JRadioButton(String label, boolean state)`
Tworzy przełącznik o określonej etykiecie i stanie początkowym.

javax.swing.ButtonGroup 1.2

- `void add(AbstractButton b)`
Dodaje przycisk do grupy.
- `ButtonModel getSelection()`
Zwraca model przycisku.

javax.swing.ButtonModel 1.2

- `String getActionCommand()`
Zwraca polecenie akcji modelu przycisku.

javax.swing.AbstractButton 1.2

- void setActionCommand(String s)

Ustawia polecenie akcji dla przycisku i jego modelu.

9.4.3. Obramowanie

Jeśli jedno okno zawiera kilka grup przełączników, należy w jakiś sposób te grupy oznaczyć. Do tego celu można użyć **obramowania** Swing. Obramowanie można zastosować do każdego komponentu, który rozszerza klasę `JComponent`. Najczęściej obramowanie stosuje się wokół panelu, który zawiera elementy interfejsu użytkownika, jak przełączniki.

Do wyboru jest kilka rodzajów obramowań, ale sposób ich użycia jest taki sam dla wszystkich.

- 1 Wywołaj statyczną metodę klasy `BorderFactory`, która tworzy obramowanie.

Do wyboru są następujące style (zobacz rysunek 9.16):

- `LoweredBevel` (ukos dolny),
- `RaisedBevel` (ukos górnny),
- `Etched` (wgłębienie),
- `Line` (linia),
- `Matte` (linia) — umożliwia określenie grubości poszczególnych krawędzi,
- `Empty` (pusta) — tworzy obramowanie, które w ogóle nie zajmuje miejsca.

- 2 Aby dodać tytuł do obramowania, należy je przekazać do metody `BorderFactory.createTitledBorder`.

- 3 Można pójść na całość i połączyć kilka obramowań: `BorderFactory.createCompoundBorder`.

- 4 Utworzone obramowanie dodaje się do komponentu za pomocą metody `setBorder` z klasy `JComponent`.

Rysunek 9.16.

Rodzaje obramowań



Poniższy fragment programu tworzy obramowanie w stylu `Etched` z tytułem dla panelu:

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched, "Tytuł");
panel.setBorder(titled);
```

Aby sprawdzić, jak wyglądają poszczególne style obramowań, uruchom program z listingu 9.5.

Listing 9.5. border/BorderFrame.java

```
package border;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

/**
 * Ramka z przełącznikami służącymi do wyboru stylu obramowania.
 */
public class BorderFrame extends JFrame
{
    private JPanel demoPanel;
    private JPanel buttonPanel;
    private ButtonGroup group;

    public BorderFrame()
    {
        demoPanel = new JPanel();
        buttonPanel = new JPanel();
        group = new ButtonGroup();

        addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
        addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
        addRadioButton("Etched", BorderFactory.createEtchedBorder());
        addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
        addRadioButton("Matte", BorderFactory.createMatteBorder(10, 10, 10, 10,
            Color.BLUE));
        addRadioButton("Empty", BorderFactory.createEmptyBorder());

        Border etched = BorderFactory.createEtchedBorder();
        Border titled = BorderFactory.createTitledBorder(etched, "Typy obramowania");
        buttonPanel.setBorder(titled);

        setLayout(new GridLayout(2, 1));
        add(buttonPanel);
        add(demoPanel);
        pack();
    }

    public void addRadioButton(String buttonName, final Border b)
    {
        JRadioButton button = new JRadioButton(buttonName);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                demoPanel.setBorder(b);
            }
        });
        group.add(button);
        buttonPanel.add(button);
    }
}
```

Różne obramowania mają różne opcje służące do ustawiania szerokości i koloru. Szczegółowe informacje na ten temat znajdują się w wyciągach z API. Wielbicieli obramowań uciecze fakty, że istnieje klasa `SoftBevelBorder`, służąca do tworzenia obramowań o mniej ostrych rogach, oraz że obramowanie `LineBorder` może mieć także zaokrąglone rogi. Wymienione obramowania można tworzyć wyłącznie za pomocą konstruktorów klas — nie istnieje dla nich metoda `BorderFactory`.

javax.swing.BorderFactory 1.2

- static Border createLineBorder(Color color)
 - static Border createLineBorder(Color color, int thickness)

Tworzy obramowanie w postaci zwykłej linii.

- static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)
 - static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)

Tworzy obramowanie o określonej grubości i wypełnione określonym kolorem lub powtarzającym się obrazem.

- static Border createEmptyBorder()
 - static Border createEmptyBorder(int top, int left, int bottom, int right)

Tworzy puste obramowanie.

- static Border createEtchedBorder()
 - static Border createEtchedBorder(Color highlight, Color shadow)
 - static Border createEtchedBorder(int type)
 - static Border createEtchedBorder(StyleID highlightStyle, Color highlight, StyleID shadowStyle, Color shadow)

Twarzy obramowanie w postaci linii z efektem trójkątowym.

Parametry: highlight, shadow. Kolory dla efektu trójkątnego.

- static Border createBevelBorder(int type)
 - static Border createBevelBorder(int type, Color highlight, Color shadow)
 - static Border createLoweredBevelBorder()
 - static Border createRaisedBevelBorder()

Tworzy obramowanie wyglądające jak wznosząca się lub opadająca skośna powierzchnia

Parametry: highlight, shadow Kolorы для эффекта трехмерного изображения

- static TitledBorder createTitledBorder(String title)
- static TitledBorder createTitledBorder(Border border)
- static TitledBorder createTitledBorder(Border border, String title)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)

Tworzy obramowanie z tytułem o określonych cechach.

Parametry:	title	Tytuł
	border	Obramowanie, do którego ma być dodany tytuł
	justification	Jedna ze stałych TitledBorder: LEFT, CENTER, RIGHT, LEADING, TRAILING, DEFAULT_JUSTIFICATION (do lewej)
	position	Jedna ze stałych TitledBorder: ABOVE_TOP, TOP, BELOW_TOP, ABOVE_BOTTOM, BOTTOM, BELOW_BOTTOM, DEFAULT_POSITION (góra)
	font	Krój czcionki w tytule
	color	Kolor tytułu

- static CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)

Tworzy obramowanie z dwóch rodzajów obramowań.

javax.swing.border.SoftBevelBorder 1.2

- SoftBevelBorder(int type)
- SoftBevelBorder(int type, Color highlight, Color shadow)

Tworzy obramowanie ukośne o mniej ostrych rogach.

Parametry:	highlight, shadow	Kolory dla efektu trójwymiarowego
	type	Wartość EtchedBorder.RAISED lub EtchedBorder.LOWERED

javax.swing.border.LineBorder 1.2

- public LineBorder(Color color, int thickness, boolean roundedCorners)

Tworzy obramowanie o określonym kolorze i grubości. Jeśli parametr roundedCorners ma wartość true, rogi są zaokrąglone.

javax.swing.JComponent 1.2

■ void setBorder(Border border)

Ustawia obramowanie komponentu.

9.4.4. Listy rozwijalne

Jeśli opcji do wyboru jest dużo, przełączniki nie zdają egzaminu, ponieważ zajmują zbyt dużo miejsca. W takim przypadku lepiej użyć listy rozwijalnej (ang. *combo box*). Kliknięcie takiego komponentu powoduje rozwinięcie listy opcji, z których użytkownik może wybrać tylko jedną (zobacz rysunek 9.17).

Rysunek 9.17.

Lista rozwijalna



Jeśli pole listy rozwijalnej jest **edytowalne** (editable), aktualnie wybraną opcję można edytować, tak jakby była polem tekstowym. Dlatego komponent ten jest też nazywany **polem typu kombi** — łączy w sobie elastyczność pól tekstowych z zestawem ustalonych z góry opcji. Do tworzenia tego typu pól służy klasa JComboBox.

Od Java 7 klasa JComboBox jest generyczna. Przykładowo JComboBox<String> przechowuje obiekty typu String, a JComboBox<Integer> — liczby całkowite.

Aby lista rozwijalna była edytowalna, należy użyć metody setEditable. Należy pamiętać, że edytowanie ma wpływ wyłącznie na wybrany element. Nie powoduje to zmiany listy opcji do wyboru.

Bieżący wybór, który mógł zostać zmodyfikowany, jeśli pole jest edytowalne, można pobrać za pomocą metody getSelectedItem. Jednak w edytowalnej liście element ten może być każdego typu, w zależności od tego, jaki edytor odbiera dane edytowane przez użytkownika i zamienia wyniki na obiekty (opis edytorów znajduje się w rozdziale 6. drugiego tomu). Jeśli pole nie jest edytowalne, lepiej zastosować wywołanie:

```
combo.getSelectedItem()
```

Instrukcja ta zwraca wybrany element z poprawnym typem.

W przedstawionym programie użytkownik może wybrać z listy jeden rodzaj czcionki (Serif, SansSerif, Monospaced itd.). Może także wpisać nazwę innego fontu.

Poszczególne opcje wstawia się za pomocą metody addItem. W tym programie metoda ta jest wywoływana tylko w konstruktorze, ale można ją wywoływać w dowolnym miejscu.

```
JComboBox<String> faceCombo = new JComboBox<>();
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
...
```

Ta metoda dodaje łańcuch na końcu listy. Elementy można także dodawać w dowolnym miejscu listy za pomocą metody `InsertItemAt`:

```
faceCombo.insertItemAt("Monospaced", 0); // Dodanie opcji na początku listy.
```

Dodawane elementy mogą być dowolnego typu — lista rozwijalna wywołuje metodę `toString` przed wyświetleniem każdego elementu.

Do usuwania elementów z listy w czasie działania programu służą metody `removeItem` i `removeItemAt`. Pierwszej należy podać treść elementu, który ma być usunięty, a drugiej numer pozycji elementu do usunięcia.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // usunięcie pierwszego elementu
```

Metoda `removeAllItems` usuwa wszystkie elementy naraz.



Metoda `addItem` nie działa wydajnie przy dodawaniu dużej liczby elementów do listy rozwijalnej. Zamiast niej lepiej utworzyć obiekt `DefaultComboBoxModel`, zapełnić go za pomocą metody `addElement`, a następnie wywołać metodę `setModel` z klasy `JComboBox`.

Kiedy użytkownik wybiera element z listy, generowana jest akcja. Aby sprawdzić, który element został wybrany, należy wywołać metodę `getSource` na rzecz parametru zdarzenia w celu uzyskania referencji do listy rozwijalnej, która wysłała to zdarzenie. Następnie należy wywołać metodę `getSelectedItem` sprawdzającą, który element jest aktualnie wybrany. Zwróconą wartość trzeba rzutować na odpowiedni typ, zazwyczaj `String`.

```
public void actionPerformed(ActionEvent event)
{
    label.setFont(new Font(
        faceCombo.getSelectedItem(faceCombo.getSelectedIndex()),
        Font.PLAIN,
        DEFAULT_SIZE));
}
```

Listing 9.6 przedstawia kompletny kod programu.

Listing 9.6. comboBox/ComboBoxFrame.java

```
package comboBox;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z przykładową etykietą tekstową i listą rozwijalną umożliwiającą wybór kroju czcionki.
 */


```

```

public class ComboBoxFrame extends JFrame
{
    private JComboBox<String> faceCombo;
    private JLabel label;
    private static final int DEFAULT_SIZE = 24;

    public ComboBoxFrame()
    {
        // Dodanie tekstu etykiety

        label = new JLabel("Koń i pies grali w kości z piękną ćmą u źródła.");
        label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
        add(label, BorderLayout.CENTER);

        // Tworzenie listy rozwijalnej i dodawanie nazw czcionek

        faceCombo = new JComboBox<>();
        faceCombo.addItem("Serif");
        faceCombo.addItem("SansSerif");
        faceCombo.addItem("Monospaced");
        faceCombo.addItem("Dialog");
        faceCombo.addItem("DialogInput");

        // Słuchacz listy rozwijalnej zmienia król pisma etykiety na czcionkę wybraną przez użytkownika

        faceCombo.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                label.setFont(new
                    Font(faceCombo.getItemAt(faceCombo.getSelectedIndex()), Font.PLAIN,
                    DEFAULT_SIZE));
            }
        });
    }

    // Dodanie listy rozwijalnej do panelu znajdującego się przy południowej krawędzi ramki

    JPanel comboPanel = new JPanel();
    comboPanel.add(faceCombo);
    add(comboPanel, BorderLayout.SOUTH);
    pack();
}
}

```



Do tworzenia list opcji widocznych cały czas służy komponent `JList`. Został on opisany w rozdziale 6. drugiego tomu.

javax.swing.JComboBox **1.2**

- `boolean isEditable()`
- `void setEditable(boolean b)`

Pobiera lub ustawia właściwość `editable` listy rozwijalnej.

- void addItem(Object item)
Dodaje element do listy.
- void insertItemAt(Object item, int index)
Wstawia element do listy na określonej pozycji.
- void removeItem(Object item)
Usuwa element z listy.
- void removeItemAt(int index)
Usuwa element z listy znajdujący się na określonej pozycji.
- void removeAllItems()
Usuwa wszystkie elementy z listy.
- Object getSelectedItem()
Zwraca aktualnie wybrany element.

9.4.5. Suwaki

Listy rozwijalne pozwalają na wybór jednej z kilku opcji. Suwaki natomiast umożliwiają wybór opcji z szerszego spektrum wartości, na przykład jednej ze stu.

Najczęściej stosowana metoda tworzenia suwaków jest następująca:

```
JSlider slider = new JSlider(min, max, initialValue);
```

Jeśli parametry określające wartość minimalną, maksymalną i początkową nie zostaną podane, będą miały wartości odpowiednio 0, 100 i 50.

Aby utworzyć pionowy suwak, należy zastosować następującą metodę:

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

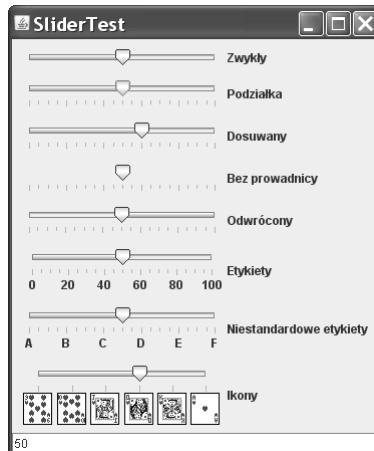
Przedstawione konstruktory tworzą zwykłe suwaki, jak pierwszy na rysunku 9.18. Niebanem nauczymy się ozdabiać suwaki rozmaitymi dodatkami.

W miarę przesuwania przez użytkownika gałki suwaka przyjmuje on kolejne **wartości** od minimalnej do maksymalnej. Kiedy zmienia się wartość, do wszystkich słuchaczy zmian wysyłane jest zdarzenie typu `ChangeEvent`. Aby móc odbierać powiadomienia o zmianach, należy wywołać metodę `addChangeListener` i zainstalować obiekt implementujący interfejs `ChangeListener`. Interfejs ten zawiera jedną metodę o nazwie `stateChanged`. W metodzie tej należy sprawdzić wartość suwaka:

```
public void stateChanged(ChangeEvent event)
{
    JSlider slider = (JSlider) event.getSource();
    int value = slider.getValue();
    ...
}
```

Rysunek 9.18.

Suwaki



Suwak można przyozdobić **podziałką** (ang. *ticks*). W omawianym programie dla drugiego suwaka zastosowano następujące ustawienia:

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

Długa kreska pojawia się co 20 jednostek, a krótsza co pięć. Jednostki odnoszą się do wartości suwaka, nie do pikseli.

Te instrukcje służą tylko do ustawienia liczby jednostek, co ile mają się pojawiać znaczniki w postaci kresek. Aby kreski te zostały uwidocznione, potrzebne jest następujące wywołanie:

```
slider.setPaintTicks(true);
```

Długie i krótkie kreski są wzajemnie niezależne. Można na przykład ustawić długą kreskę co 20 jednostek i krótką co siedem, ale taka skala nie byłaby zbyt klarowna.

Gałkę suwaka można zmusić, aby przyklejała się do **kresek podziałki**. Kiedy użytkownik przeciągnie gałkę suwaka i ją puści, zostanie ona natychmiast dosunięta do najbliższej kreski. Za aktywowanie tego trybu odpowiada poniższa procedura:

```
slider.setSnapToTicks(true);
```



Funkcja dosuwania nie działa tak dobrze, jak można by było tego życzyć. Dopóki gałka suwaka rzeczywiście nie zostanie dosunięta, obiekt nasłuchujący zmian raportuje wartości, które nie odpowiadają kreskom podziałki. Ponadto kliknięcie obok gałki takiego suwaka (czynność ta w innych suwakach powoduje przesunięcie gałki w stronę kliknięcia) nie powoduje jej przesunięcia do kolejnej kreski.

Długie kreski podziałki można oznaczyć etyktetami:

```
slider.setPaintLabels(true);
```

Na przykład w przypadku suwaka o zakresie 0 – 100 i odstępie długich kresek co 20 jednostek etykiety będą następujące: 0, 20, 40, 60, 80, 100.

Istnieje też możliwość zastosowania innych znaczników, takich jak łańcuchy lub ikony (rysunek 9.18). Czynności z tym związane są nieco skomplikowane. Trzeba zapisać tablice mieszającą (ang. *hash table*) kluczami typu Integer i wartościami typu Component. Następnie wywołuje się metodę `setLabelTable`. Komponenty zostaną umieszczone pod kreskami. Zazwyczaj w takim przypadku używane są obiekty typu `JLabel`. Poniższy fragment programu ustawia etykiety A, B, C, D, E, F:

```
Hashtable<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));

labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

Więcej informacji na temat tablic mieszających znajduje się w rozdziale 13.

Program przedstawiony na listingu 9.7 zawiera także suwaki z ikonami jako etykietami kresek podziałki.



Jeśli nie widać etykiet lub kresek podziałki, należy się upewnić, czy zostały wywołane metody `setPaintTicks(true)` i `setPaintLabels(true)`.

Czwarty suwak na rysunku 9.18 jest pozbawiony prowadnicy. Za jej usunięcie odpowiedzialna jest poniższa procedura:

```
slider.setPaintTrack(false);
```

Kierunek piątego suwaka został odwrócony za pomocą poniższej metody:

```
slider.setInverted(true);
```

Poniższy przykładowy program demonstruje wszystkie opisane rodzaje suwaków. Każdy suwak posiada obiekt nasłuchujący typu `ChangeListener`, który wstawia bieżącą wartość suwaka do pola tekstowego umiejscowionego na samym dole ramki.

Listing 9.7. slider/SliderFrame.java

```
package slider;

import java.awt.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca kilka suwaków oraz pole tekstowe pokazujące wartości ustawiane za ich pomocą
 */
public class SliderFrame extends JFrame
{
    private JPanel sliderPanel;
    private JTextField textField;
    private ChangeListener listener;

    public SliderFrame()
    {
```

```
sliderPanel = new JPanel();
sliderPanel.setLayout(new GridBagLayout());

// Wspólny słuchacz wszystkich suwaków
listener = new ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        // Aktualizacja pola tekowego w odpowiedzi na zmianę wartości suwaka
        JSlider source = (JSlider) event.getSource();
        textField.setText(" " + source.getValue());
    }
};

// Zwykły suwak
JSlider slider = new JSlider();
addSlider(slider, "Zwykły");

// Suwak z podziałką
slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Podziałka");

// Suwak z dosuwaniem galki do najbliższej kreski
slider = new JSlider();
slider.setPaintTicks(true);
slider.setSnapToTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Dosuwany");

// Suwak bez prowadnicy
slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setPaintTrack(false);
addSlider(slider, "Bez prowadnicy");

// Suwak o odwróconym działaniu
slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setInverted(true);
addSlider(slider, "Odwrócony");

// Suwak z etykietami liczbowymi
slider = new JSlider();
slider.setPaintTicks(true);
```

```
slider.setPaintLabels(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Etykiety");

// Suwak z etykietami literowymi

slider = new JSlider();
slider.setPaintLabels(true);
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);

Dictionary<Integer, Component> labelTable = new Hashtable<>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
labelTable.put(40, new JLabel("C"));
labelTable.put(60, new JLabel("D"));
labelTable.put(80, new JLabel("E"));
labelTable.put(100, new JLabel("F"));

slider.setLabelTable(labelTable);
addSlider(slider, "Niestandardowe etykiety");

// Suwak z etykietami ikonowymi

slider = new JSlider();
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setSnapToTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(20);

labelTable = new Hashtable<Integer, Component>();

// Dodawanie obrazów kart

labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));

slider.setLabelTable(labelTable);
addSlider(slider, "Ikony");

// Dodawanie pola tekstowego, które wyświetla wartość ustawioną na suwaku

textField = new JTextField();
add(sliderPanel, BorderLayout.CENTER);
add(textField, BorderLayout.SOUTH);
pack();
}

/*
* Dodaje suwak do panelu suwaków i wiąże słuchacza.
* @param s suwak
```

```

* @param description opis suwaka
*/
public void addSlider(JSlider s, String description)
{
    s.addChangeListener(listener);
    JPanel panel = new JPanel();
    panel.add(s);
    panel.add(new JLabel(description));
    panel.setAlignmentX(Component.LEFT_ALIGNMENT);
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.gridy = sliderPanel.getComponentCount();
    gbc.anchor = GridBagConstraints.WEST;
    sliderPanel.add(panel, gbc);
}
}

```

javax.swing.JSlider 1.2

- `JSlider()`
- `JSlider(int direction)`
- `JSlider(int min, int max)`
- `JSlider(int min, int max, int initialValue)`
- `JSlider(int direction, int min, int max, int initialValue)`

Tworzy poziomy suwak o określonym kierunku oraz wartościach minimalnej, maksymalnej i początkowej.

Parametry:	<code>direction</code>	SwingConstants.HORIZONTAL lub SwingConstants.VERTICAL — domyślna jest pierwsza z wymienionych wartości.
	<code>min, max</code>	Najmniejsza i największa wartość suwaka. Domyślne wartości to 0 i 100.
	<code>initialValue</code>	Wartość początkowa suwaka — domyślnie 50.

- `void setPaintTicks(boolean b)`
Jeśli parametr `b` ma wartość `true`, wyświetla podziałkę.
- `void setMajorTickSpacing(int units)`
- `void setMinorTickSpacing(int units)`
Wstawia długie i krótkie kreski podziałki co określoną liczbę jednostek.
- `void setPaintLabels(boolean b)`
Jeśli parametr `b` ma wartość `true`, wyświetla etykiety kresek podziałki.
- `void setLabelTable(Dictionary table)`
Ustawia komponenty stanowiące etykiety kresek. Każda para klucz – wartość w tabeli ma postać `new Integer(wartość)/komponent`.

- void setSnapToTicks(boolean b)

Jeśli parametr b ma wartość true, gałka suwaka dosuwa się do najbliższej kreski podziałki.

- void setPaintTrack(boolean b)

Jeśli parametr b ma wartość true, wyświetlana jest prowadnica, po której przesuwa się gałka suwaka.

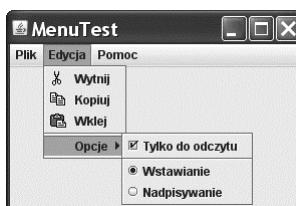
9.5. Menu

Ten rozdział zaczęliśmy od opisu najczęściej używanych komponentów, takich jak przyciski, pola tekstowe i listy rozwijalne. W Swingu można też tworzyć inny rodzaj elementów interfejsu użytkownika — znane z aplikacji posiadających graficzny interfejs menu rozwijalne.

Pasek menu znajdujący się na górze okna zawiera nazwy rozwijalnych menu. Kliknięcie jednej z tych nazw powoduje otwarcie odpowiadającego jej menu, które zawiera różne **elementy menu** oraz **podmenu**. Kiedy użytkownik kliknie element menu, wszystkie menu zostają zamknięte, a do programu wysyłany jest komunikat. Rysunek 9.19 przedstawia typowe menu z podmenu.

Rysunek 9.19.

Menu z podmenu



9.5.1. Tworzenie menu

Proces budowy menu jest bardzo prosty. Najpierw należy utworzyć pasek menu:

```
JMenuBar menuBar = new JMenuBar();
```

Pasek menu jest zwykłym komponentem, który można wstawić w dowolnym miejscu. Zazwyczaj jest on umieszczany na samej górze ramki za pomocą metody `setJMenuBar`:

```
frame.setJMenuBar(menuBar);
```

Dla każdego menu należy utworzyć osobny obiekt:

```
JMenu editMenu = new JMenu("Edycja");
```

Menu najwyższego rzędu dodaje się do paska menu:

```
menuBar.add(editMenu);
```

Elementy menu, separatory i podmenu dodaje się do obiektu menu:

```
JMenuItem pasteItem = new JMenuItem("Wklej");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // podmenu
editMenu.add(optionsMenu);
```

Na rysunku 9.19 separatory znajdują się pod elementami menu *Wklej* oraz *Tylko do odczytu*.

Kiedy użytkownik kliką menu, uruchamia akcję. Każdy element menu musi posiadać obiekt nasłuchujący akcji:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

Metoda `JMenu.add(String s)` dodaje element na końcu menu. Na przykład:

```
editMenu.add("Wklej");
```

Metoda `add` zwraca utworzony element menu, który można przejąć w celu dodania dla niego słuchacza:

```
JMenuItem pasteItem = editMenu.add("Wklej");
pasteItem.addActionListener(listener);
```

Polecenia wykonywane w odpowiedzi na kliknięcie elementu menu często mogą być aktywowane także przez inne elementy interfejsu, jak przyciski na pasku narzędzi. W rozdziale 8. nauczyliśmy się określać polecenia za pośrednictwem obiektów `Action`. Polega to na zdefiniowaniu klasy implementującej interfejs `Action`, zazwyczaj dla wygody rozszerzającej klasę `AbstractAction`. Etykietę elementu menu określa się w konstruktorze obiektu typu `AbstractAction`. Ponadto należy przedefiniować metodę `actionPerformed` na procedurę obsługi akcji. Na przykład:

```
Action exitAction = new AbstractAction("Zakończ") // etykieta elementu menu
{
    public void actionPerformed(ActionEvent event)
    {
        procedury obsługi akcji
        System.exit(0);
    }
};
```

Następnie można dodać akcję do menu:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

Ta procedura dodaje element do menu, wykorzystując do tego celu nazwę akcji. Obiekt akcji staje się jej słuchaczem. Jest to skrócona forma zapisu poniższego fragmentu programu:

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```

javax.swing.JMenu 1.2

■ `JMenu(String label)`

Tworzy menu z określona etykieta.

- `JMenuItem add(JMenuItem item)`
Dodaje element menu (lub menu).
- `JMenuItem add(String label)`
Dodaje element menu z określona etykietą i zwraca ten element.
- `JMenuItem add(Action a)`
Dodaje element menu z określona akcją i zwraca ten element.
- `void addSeparator()`
Dodaje separator do menu.
- `JMenuItem insert(JMenuItem menu, int index)`
Dodaje nowy element menu (lub podmenu) do menu w miejscu określonym przez indeks.
- `JMenuItem insert(Action a, int index)`
Dodaje element menu z określona akcją w miejscu określonym przez indeks.
- `void insertSeparator(int index)`
Dodaje separator do menu.
Parametr: index Miejsce wstawienia separatora
- `void remove(int index)`
- `void remove(JMenuItem item)`
Tworzy element menu dla określonej akcji.

javax.swing.JMenuItem 1.2

- `JMenuItem(String label)`
Tworzy element menu z daną etykietą.
- `JMenuItem(Action a) 1.3`
Tworzy element menu dla danej akcji.

javax.swing.AbstractButton 1.2

- `void setAction(Action a) 1.3`
Ustawia akcję dla przycisku lub elementu menu.

javax.swing.JFrame 1.2

- `void setMenuBar(JMenuBar menubar)`
Ustawia pasek menu w ramce.

9.5.2. Ikony w elementach menu

Elementy menu są bardzo podobne do przycisków. Klasa JMenuItem jest nawet rozszerzeniem klasy AbstractButton. Menu, podobnie jak przyciski, mogą mieć etykietę tekstową, ikonę lub jedno i drugie. Ikonę można określić za pomocą konstruktorów JMenuItem(String, Icon) bądź JMenuItem(Icon) lub metody setIcon, którą klasa JMenuItem dziedziczy po klasie AbstractButton. Na przykład:

```
MenuItem cutItem = new JMenuItem("Wytnij", new ImageIcon("cut.gif"));
```

Na rysunku 9.19 ikony znajdują się obok kilku elementów menu. Przy standardowych ustawieniach tekst jest umieszczany po prawej stronie ikony elementu menu. Aby tekst pojawił się po lewej stronie ikony, należy użyć metody setHorizontalTextPosition, którą klasa JMenuItem dziedziczy po klasie AbstractButton. Na przykład poniższa instrukcja ustawia tekst etykiety elementu menu po lewej stronie ikony:

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

Można także dodać ikonę do akcji:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

Podczas konstruowania elementu menu z akcji wartość Action.NAME staje się etykietą tego elementu, a wartość Action.SMALL_ICON ikoną.

Inna metoda ustawiania ikony polega na użyciu konstruktora AbstractAction:

```
cutAction = new
AbstractAction("Wytnij", new ImageIcon("cut.gif"))
{
    public void actionPerformed(ActionEvent event)
    {
        kod akcji
    }
};
```

javax.swing.JMenuItem 1.2

■ JMenuItem(String label, Icon icon)

Tworzy element menu z określona ikoną i etykieta.

javax.swing.AbstractButton 1.2

■ void setHorizontalTextPosition(int pos)

Określa położenie tekstu w poziomie względem ikony.

Parametr: pos SwingConstants.RIGHT (wyrównanie tekstu do prawej),
SwingConstants.LEFT (wyrównanie tekstu do lewej)

javax.swing.AbstractAction 1.2

■ AbstractAction(string name, Icon smallIcon)

Tworzy akcję abstrakcyjną o określonej nazwie i z określona ikoną.

9.5.3. Pola wyboru i przełączniki jako elementy menu

Pola wyboru i przełączniki jako elementy menu pojawiają się obok nazwy elementu (zobacz rysunek 9.19). Kiedy użytkownik kliknie taki element menu, zostanie on automatycznie zaznaczony lub też zaznaczenie zostanie usunięte.

Elementy te traktuje się tak samo jak wszystkie inne elementy menu. Poniższy fragment kodu tworzy element menu w postaci pola wyboru:

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Tylko do odczytu");
optionsMenu.add(readonlyItem);
```

Przełączniki w elementach menu działają tak samo jak zwykłe przełączniki. Muszą należeć do grupy przycisków, w której zaznaczenie jednego elementu powoduje usunięcie zaznaczenia uprzednio wybranego.

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Wstawianie");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Nadpisywanie");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

W przypadku tych elementów programista nie musi koniecznie wiedzieć, kiedy dokładnie nastąpił wybór elementu. Może natomiast sprawdzić jego stan za pomocą metody `isSelected` (to oczywiście oznacza konieczność przechowywania referencji do tego elementu menu w polu obiektowym). Do ustawiania stanu służy metoda `setSelected`.

javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`

Tworzy element menu w postaci pola wyboru o określonej etykiecie.

- `JCheckBoxMenuItem(String label, boolean state)`

Tworzy element menu w postaci pola wyboru o określonej etykiecie i określonym stanie początkowym (true oznacza zaznaczony).

javax.swing.JRadioButtonMenuItem 1.2

- `JRadioButtonMenuItem(String label)`

Tworzy element menu w postaci przełącznika o określonej etykiecie.

- `JRadioButtonMenuItem(String label, boolean state)`

Tworzy element menu w postaci przełącznika o określonej etykiecie i określonym stanie początkowym (true oznacza zaznaczony).

javax.swing.AbstractButton 1.2

- `boolean isSelected()`

■ void setSelected(boolean state)

Pobiera lub ustawia stan elementu (true oznacza zaznaczony).

9.5.4. Menu podręczne

Menu podręczne (ang. *pop-up menu*) nie jest związane z paskiem menu, tylko pojawia się w różnych miejscach okna (rysunek 9.20).

Rysunek 9.20.

Menu podręczne



Proces tworzenia menu podręcznego wygląda podobnie jak w przypadku zwykłego menu, z tym wyjątkiem, że nie nadaje mu się tytułu.

```
JPopupMenu popup = new JPopupMenu();
```

Elementy do takiego menu dodaje się w typowy sposób:

```
JMenuItem item = new JMenuItem("Wytnij");
item.addActionListener(listener);
popup.add(item);
```

W przeciwieństwie do paska menu, który zawsze znajduje się na samej górze ramki, menu podręczne musi być wyświetlane za pomocą metody `show`. Należy w niej określić komponent nadrzędny menu oraz jego lokalizację za pomocą systemu współrzędnych komponentu nadrzędnego. Na przykład:

```
popup.show(panel, x, y);
```

Zazwyczaj menu podręczne są tak zaprogramowane, aby pojawiały się w odpowiedzi na kliknięcie przez użytkownika określonym przyciskiem myszy (tzw. **pop-up trigger**). W systemach Windows i Linux funkcję tę zazwyczaj pełni prawy przycisk myszy. Za pojawienie się menu kontekstowego w odpowiedzi na kliknięcie przez użytkownika przycisku wyzwalającego menu odpowiada poniższa instrukcja:

```
component.setComponentPopupMenu(popup);
```

Czasami do komponentu posiadającego menu kontekstowe może zostać wstawiony inny komponent, który również posiada takie menu. Komponent podrzędny może odziedziczyć menu kontekstowe elementu nadrzędnego dzięki poniższej instrukcji:

```
child.setInheritsPopupMenu(true);
```

javax.swing.JPopupMenu 1.2

■ void show(Component c, int x, int y)

Wyświetla menu kontekstowe.

Parametry:	c x, y	Komponent, w którym ma się pojawić menu. Współrzędne (w przestrzeni komponentu c) górnego lewego rogu menu kontekstowego.
-------------------	-----------	---

■ **boolean isPopupTrigger(MouseEvent event) 1.3**

Zwraca wartość true, jeśli zdarzenie myszy powoduje pojawienie się menu kontekstowego.

java.awt.event.MouseEvent 1.1

■ **boolean isPopupTrigger()**

Zwraca wartość true, jeśli zdarzenie myszy powoduje pojawienie się menu kontekstowego.

javax.swing.JComponent 1.2

■ **JPopupMenu getComponentPopupMenu() 5.0**

■ **void setComponentPopupMenu(JPopupMenu popup) 5.0**

Pobiera lub ustawia menu kontekstowe dla komponentu.

■ **boolean getInheritsPopupMenu() 5.0**

■ **void setInheritsPopupMenu(boolean b) 5.0**

Pobiera lub ustawia właściwość `inheritsPopupMenu`. Jeśli właściwość ta jest ustawiona, a menu tego komponentu jest null, wykorzystuje menu kontekstowe jego komponentu nadzawanego.

9.5.5. Mnemoniki i akceleratory

Dla zaawansowanego użytkownika programu bardzo ważnym usprawnieniem pracy jest możliwość otwierania menu za pomocą **mnemoników**. Mnemoniki do elementów menu określa się poprzez określenie wybranej litery w konstruktorach tych elementów:

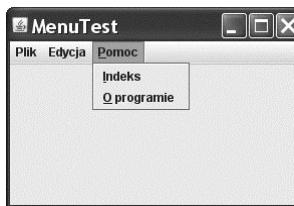
```
JMenuItem aboutItem = new JMenuItem("O programie", 'O');
```

Mnemonik jest wyświetlany automatycznie w menu, a litera mnemoniku jest podkreślona (rysunek 9.21). Na przykład etykieta elementu zdefiniowanego powyżej będzie wyglądała następująco: *O programie* — podkreślona litera *O*. Po rozwinięciu menu wystarczy nacisnąć klawisz *O*, aby wybrać ten element (jeśli litera mnemoniku nie występuje w łańcuchu menu, jej naciśnięcie spowoduje wybór tego elementu, ale mnemonik nie będzie wyświetlany w menu — oczywiście przydatność takich niewidocznych mnemoników stoi pod znakiem zapytania).

Czasami programista nie chce, aby podkreślona została pierwsza litera pasująca do mnemoniku. Jeśli mamy na przykład mnemonik *A* dla elementu menu *Zapisz jako*, możemy sprawić, aby została podkreślona litera *a* w drugim wyrazie (*Zapisz jako*). W Java SE 1.4 wprowadzono możliwość określania, która litera ma zostać podkreślona. Służy do tego metoda `setDisplayedMnemonicIndex`.

Rysunek 9.21.

Mnemoniki



Mając obiekt `Action`, mnemonik można dodać jako wartość klucza `Action.MNEMONIC_KEY`:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('0'));
```

Literę mnemoniku można podać tylko w konstruktorze elementu menu, w konstruktorze samego menu nie. Aby dodać mnemonik dla całego menu, należy użyć metody `setMnemonic`:

```
JMenu helpMenu = new JMenu("Pomoc");
helpMenu.setMnemonic('P');
```

Aby przejść do menu najwyższego poziomu na pasku menu, należy nacisnąć klawisz *Alt* i literę mnemoniku. Aby na przykład przejść do menu *Pomoc*, należy nacisnąć kombinację klawiszy *Alt+P*.

Z pomocą mnemoników można aktywować element aktualnie otwartego menu lub jego podmenu. Natomiast **akceleratory** (ang. *accelerators*) to skróty klawiszowe, które dają dostęp do elementów menu bez jego otwierania. Na przykład w wielu programach akceleratory *Ctrl+O* i *Ctrl+S* odpowiadają elementom *Otwórz* i *Zapisz* w menu *Plik*. Do wiązania elementów menu z klawiszami skrótu (akceleratorami) służy metoda `setAccelerator`. Przyjmuje ona obiekt typu `KeyStroke`. Na przykład poniższa instrukcja wiąże skrót klawiszowy *Ctrl+O* z elementem menu `openItem`:

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl 0"));
```

Naciśnięcie kombinacji klawiszy akceleratora powoduje wybór odpowiedniej opcji z menu i uruchomienie akcji, tak jakby użytkownik wybrał daną opcję wprost z menu.

Akceleratory można wiązać wyłącznie z elementami menu, nie z samymi menu. Akceleratory nie otwierają żadnego menu — bezpośrednio uruchamiają akcję związaną z danym menu.

Teoretycznie tworzenie akceleratora dla elementu menu jest techniką podobną do dodawania akceleratora do komponentu Swing (technikę tę opisaliśmy w rozdziale 8.). Jednak kombinacja klawiszy akceleratora związanego z elementem menu jest automatycznie widoczna w menu (rysunek 9.22).

Rysunek 9.22.

Akceleratory





W systemie Windows kombinacja klawiszy *Ctrl+F4* zamyka okno. Akcelerator ten nie został jednak zaprogramowany w Javie. Jest to skrót systemu operacyjnego. Wspomniana kombinacja klawiszy zawsze uruchamia zdarzenie *WindowClosing* dla aktywnego okna, bez względu na to, czy w menu znajduje się element *Zakończ*.

javax.swing.JMenuItem 1.2

- `JMenuItem(String label, int mnemonic)`

Tworzy element menu z określona etykieta i mnemonikiem.

Parametry: `label` Etykieta

`mnemonic` Znak mnemoniczny, który zostanie podkreślony w etykiecie.

- `void setAccelerator(KeyStroke k)`

Ustawia klawisz `k` jako akcelerator do elementu menu. Klawisz skrótu jest widoczny obok etykiety w menu.

javax.swing.AbstractButton 1.2

- `void setMnemonic(int mnemonic)`

Ustawia mnemonic dla przycisku. Znak ten będzie podkreślony w etykiecie.

- `void setDisplayedMnemonicIndex(int index) 1.4`

Podkreśla znak znajdujący się na pozycji określonej przez parametr `index`. Metody tej należy używać, aby uniknąć podkreślenia pierwszej litery odpowiadającej mnemonicowi.

9.5.6. Aktywowanie i dezaktywowanie elementów menu

W określonych sytuacjach niektóre elementy menu nie powinny być dostępne. Jeśli na przykład jakiś dokument zostanie otwarty w trybie tylko do odczytu, element menu *Zapisz* nie powinien być dostępny. Jedno wyjście polega na usunięciu go za pomocą metody *JMenu.remove*, ale użytkownicy nie przepadają za menu, których zawartość ulega zmianom. Lepiej jest zatem dezaktywować te elementy menu, które w danej sytuacji nie są potrzebne. Element taki ma kolor szary i nie działa (rysunek 9.23).

Rysunek 9.23.

Dezaktywowane elementy menu



Do aktywowania i dezaktywowania elementów menu służy metoda `setEnabled`:

```
saveItem.setEnabled(false);
```

Istnieją dwie strategie aktywowania i dezaktywowania elementów menu. Przy każdej zmianie sytuacji można wywoływać metodę `setEnabled` na rzecz odpowiednich elementów menu lub akcji. Na przykład w odpowiedzi na przejście w tryb tylko do odczytu można zlokalizować elementy menu *Zapisz* i *Zapisz jako* w celu ich dezaktywacji. Inna metoda polega na wyłączeniu elementów menu chwilę przed wyświetleniem tego menu. W takim przypadku konieczna jest rejestracja słuchacza zdarzenia wybrania menu. Pakiet `javax.swing.event` zawiera definicję interfejsu `MenuListener` z trzema metodami:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

Metoda `menuSelected` jest wywoływana **przed** wyświetleniem menu, a zatem można jej używać do aktywacji i dezaktywacji elementów menu. Poniższy fragment programu dezaktywuje polecenia *Zapisz* i *Zapisz jako* w odpowiedzi na zaznaczenie pola wyboru o nazwie *Tylko do odczytu*:

```
public void menuSelected(MenuEvent event)
{
    saveAction.setEnabled(!readonlyItem.isSelected());
    saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```



Dezaktywacja elementów menu bezpośrednio przed wyświetleniem menu jest sprytnym rozwiązaniem, ale nie sprawdza się w przypadku elementów posiadających skróty klawiszowe. Ponieważ wciśnięcie kombinacji klawiszy skrótu nie powoduje otwarcia menu, akcja nie jest dezaktywowana, a więc można ją wyzwolić za pomocą akceleratora.

javax.swing.JMenuItem 1.2

- `void setEnabled(boolean b)`

Aktywuje lub dezaktywuje element menu.

javax.swing.event.MenuListener 1.2

- `void menuSelected(MenuEvent e)`

Wywoływana po wybraniu przez użytkownika menu, ale przed jego otwarciem.

- `void menuDeselected(MenuEvent e)`

Wywoywana po dezaktywacji menu, ale przed jego zamknięciem.

- `void menuCanceled(MenuEvent e)`

Wywoywana po anulowaniu wyboru menu, np. spowodowanym kliknięciem poza jego obrębem.

Listing 9.8 przedstawia program demonstrujący wszystkie omawiane w tym podrozdziale menu: menu zagnieżdżone, dezaktywowane elementy menu, elementy menu w postaci pól wyboru i przełączników, menu kontekstowe oraz mnemoniki i akceleratory.

Listing 9.8. menu/MenuFrame.java

```
package menu;

import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z paskiem menu
 */
public class MenuFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    private Action saveAction;
    private Action saveAsAction;
    private JCheckBoxMenuItem readonlyItem;
    private JPopupMenu popup;

    /**
     * Przykładowa akcja, która drukuje nazwę akcji do wyjścia System.out
     */
    class TestAction extends AbstractAction
    {
        public TestAction(String name)
        {
            super(name);
        }

        public void actionPerformed(ActionEvent event)
        {
            System.out.println("Wybrano " + getValue(Action.NAME));
        }
    }

    public MenuFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JMenu fileMenu = new JMenu("Plik");
        fileMenu.add(new TestAction("Nowy"));

        // Akceleratory

        JMenuItem openItem = fileMenu.add(new TestAction("Otwórz"));
        openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl 0"));

        fileMenu.addSeparator();

        saveAction = new TestAction("Zapisz");
        JMenuItem saveItem = fileMenu.add(saveAction);
        saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));

        saveAsAction = new TestAction("Zapisz jako");
        fileMenu.add(saveAsAction);
        fileMenu.addSeparator();

        fileMenu.add(new AbstractAction("Zakończ"));
```

```

{
    public void actionPerformed(ActionEvent event)
    {
        System.exit(0);
    }
});

// Menu z polem wyboru i przełącznikami

readonlyItem = new JCheckBoxMenuItem("Tylko do odczytu");
readonlyItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        boolean saveOk = !readonlyItem.isSelected();
        saveAction.setEnabled(saveOk);
        saveAsAction.setEnabled(saveOk);
    }
});

ButtonGroup group = new ButtonGroup();

JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Wstawianie");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Nadpisywanie");

group.add(insertItem);
group.add(overtypeItem);

// Ikony

Action cutAction = new TestAction("Wytnij");
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
Action copyAction = new TestAction("Kopiuj");
copyAction.putValue(Action.SMALL_ICON, new ImageIcon("copy.gif"));
Action pasteAction = new TestAction("Wklej");
pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("paste.gif"));

JMenu editMenu = new JMenu("Edycja");
editMenu.add(cutAction);
editMenu.add(copyAction);
editMenu.add(pasteAction);

// Zagnieżdżone menu

JMenu optionMenu = new JMenu("Opcje");

optionMenu.add(readonlyItem);
optionMenu.addSeparator();
optionMenu.add(insertItem);
optionMenu.add(overtypeItem);

editMenu.addSeparator();
editMenu.add(optionMenu);

// Mnemoniki

JMenu helpMenu = new JMenu("Pomoc");

```

```
helpMenu.setMnemonic('P');

JMenuItem indexItem = new JMenuItem("Indeks");
indexItem.setMnemonic('I');
helpMenu.add(indexItem);

// Mnemoniki można także dodawać do akcji
Action aboutAction = new TestAction("O programie");
aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('O'));
helpMenu.add(aboutAction);

// Dodanie wszystkich menu najwyższego rzędu do paska menu

JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

menuBar.add(fileMenu);
menuBar.add(editMenu);
menuBar.add(helpMenu);

// Menu kontekstowe

popup = new JPopupMenu();
popup.add(cutAction);
popup.add(copyAction);
popup.add(pasteAction);

JPanel panel = new JPanel();
panel.setComponentPopupMenu(popup);
add(panel);

// Poniższy wiersz stanowi obejście błędu 4966109
panel.addMouseListener(new MouseAdapter() {});
}

}
```

9.5.7. Paski narzędzi

Pasek narzędzi (ang. *toolbar*) zapewnia szybki dostęp do najczęściej używanych poleceń programu (rysunek 9.24).

Rysunek 9.24.

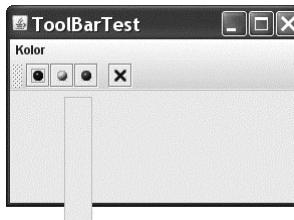
Pasek narzędzi



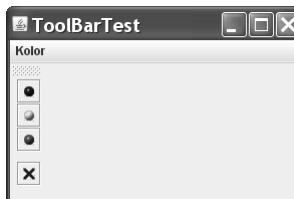
Cechą wyróżniającą paski narzędzi jest ich zdolność do przenoszenia się w różne miejsca. Można za pomocą przeciągania umieszczać je przy jednej z czterech krawędzi ramki (rysunek 9.25). Po zwolnieniu przycisku myszy pasek narzędzi pozostaje w nowej lokalizacji (rysunek 9.26).

Rysunek 9.25.

Przeciąganie paska narzędzi

**Rysunek 9.26.**

Pasek narzędzi w nowej lokalizacji

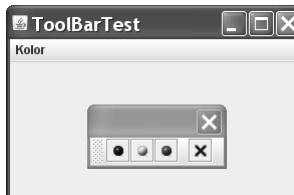


Przeciąganie paska narzędzi jest możliwe w kontenerach z układem krawędziowym lub dowolnym zarządcą rozkładu, który obsługuje ograniczenia North, East, South i West.

Pasek narzędzi można nawet całkiem oddzielić od ramki. Wtedy znajduje się on we własnej ramce (rysunek 9.27). Kiedy ramka zawierająca odłączony pasek narzędzi zostanie zamknięta, pasek ten wraca do swojej pierwotnej ramki.

Rysunek 9.27.

Odłączony pasek narzędzi



Programowanie pasków narzędzi jest łatwym zadaniem. Poniżej do paska dodawany jest element:

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

Klasa JToolBar posiada także metodę służącą do dodawania obiektów Action. Wstawianie obiektów typu Action do paska narzędzi wygląda następująco:

```
bar.add(blueAction);
```

Na pasku pokaże się niewielka ikona akcji.

Do oddzielania grup przycisków służy separator:

```
bar.addSeparator();
```

Na przykład na rysunku 9.24 separator znajduje się pomiędzy trzecim a czwartym przyciskiem.

Następnie pasek narzędzi trzeba wstawić do ramki.

```
add(bar, BorderLayout.NORTH);
```

Można także określić tytuł paska narzędzi, który będzie widoczny po jego odłączeniu:

```
bar = new JToolBar(titleString);
```

Domyślnie paski narzędzi są ułożone poziomo. Aby pasek narzędzi miał pionowe położenie początkowe, należy zastosować jedną z poniższych metod:

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

lub

```
bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

Mimo że na paskach narzędzi najczęściej spotyka się przyciski, mogą się tam znaleźć wszystkie inne komponenty — na przykład lista rozwijalna.

9.5.8. Dymki

Wadą pasków narzędzi jest to, że małe ikony niewiele mówią użytkownikowi o swoim przeznaczeniu. Rozwiązaniem tego problemu są **dymki** (ang. *tooltips*). Dymek pojawia się, kiedy kursor myszy zatrzyma się na chwilę nad przyciskiem. Tekst dymka jest wyświetlany w prostokącie z wypełnieniem w jakimś kolorze. Kiedy kursor myszy zostanie zabrany znaad przycisku, dymek znika (rysunek 9.28).

Rysunek 9.28.

Dymek



W bibliotece Swing dymek można dodać do każdego komponentu JComponent za pomocą metody `setToolTipText`:

```
exitButton.setToolTipText("Zamknij");
```

W przypadku obiektów typu Action dymki wiążą się z kluczami `SHORT_DESCRIPTION`:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Zamknij");
```

Listing 9.9 demonstruje wstawianie tych samych obiektów typu Action do menu i paska narzędzi. Należy zauważyć, że nazwy akcji pokazują się jako nazwy elementów w menu oraz jako krótkie opisy w chmurkach przycisków na pasku narzędzi.

Listing 9.9. toolBar/ToolBarTest.java

```
package toolBar;  
  
import java.awt.*;
```

```

import javax.swing.*;

/**
 * @version 1.13 2007-06-12
 * @author Cay Horstmann
 */
public class ToolBarTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                ToolBarFrame frame = new ToolBarFrame();
                frame.setTitle("ToolBarTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

```

javax.swing.JToolBar 1.2

- JToolBar()
- JToolBar(String titleString)
- JToolBar(int orientation)
- JToolBar(String titleString, int orientation)

Tworzy pasek narzędzi z określonym tytułem i położeniem. Położenie może być poziome SwingConstants.HORIZONTAL (domyślne) lub pionowe SwingConstants.VERTICAL.

- JButton add(Action a)

Tworzy przycisk w pasku narzędzi z nazwą, ikoną, krótkim opisem i wywołaniem zwojnym akcji oraz dodaje ten przycisk na końcu paska.

- void addSeparator()

Wstawia separator na końcu paska narzędzi.

javax.swing.JComponent 1.2

- void setToolTipText(String text)

Ustawia tekst, który będzie wyświetlany w dymku po najechaniu kursorem na dany komponent.

9.6. Zaawansowane techniki zarządzania rozkładem

Do tej pory komponenty interfejsu użytkownika rozmieszczaliśmy, stosując rozkład brzegowy (ang. *border layout*), ciągły (ang. *flow layout*) oraz siatkowy (ang. *grid layout*). Techniki te mogą się okazać niewystarczające w przypadku bardziej zaawansowanych zadań. Ten podrozdział został poświęcony zaawansowanym technikom zarządzania rozkładem komponentów.

Programiści Windowsa mogą się dziwić, że w Javie tak dużo uwagi poświęcono zarządcom rozkładu. W systemie Windows to nic wielkiego — najpierw w edytorze okien dialogowych przeciąga się i upuszcza wybrane komponenty okna, a następnie za pomocą odpowiednich narzędzi ustawia się je zgodnie z wymaganiami. Programiści pracujący w dużych zespołach często w ogóle nie zajmują się układem komponentów, ponieważ robią to za nich wykwalifikowani projektanci interfejsów użytkownika.

Wadą takiego podejścia jest to, że powstały układ trzeba ręcznie modyfikować, jeśli zmieni się rozmiar komponentów. Ale dlaczego komponenty miałyby zmieniać rozmiar? Może to mieć miejsce z dwóch powodów. Po pierwsze, użytkownik może użyć większej czcionki na przyciskach i dla tekstu w oknach dialogowych. Nietrudno się przekonać, że wiele aplikacji w systemie Windows bardzo źle znosi takie modyfikacje. Przyciski nie powiększają się, przez co tekst jest upychany na takiej samej powierzchni jak wcześniej. Po drugie, podobny problem może wystąpić przy lokalizacji programu. Na przykład polecenie „Anuluj” po niemiecku brzmi „Abbrechen”. Jeśli w projekcie przycisku przewidziano tylko tyle miejsca, ile potrzeba dla słowa „Anuluj”, jego niemiecki odpowiednik będzie częściowo obcięty.

Dlaczego przyciski w systemie Windows nie powiększają się, aby pomieścić etykiety? Ponieważ projektant interfejsu użytkownika nie dał żadnych instrukcji, w którym kierunku powinny one rosnąć. Po zakończeniu przeciągania, upuszczania i ustawiania edytor okien dialogowych pamięta tylko położenie i rozmiar każdego komponentu. Nie dysponuje informacjami, **dla czego** te komponenty zostały ustawione w taki sposób.

Zarządcy rozkładu w Javie oferują znacznie lepsze podejście do zagadnienia rozkładu komponentów. Dzięki nim w rozkładzie dostępne są informacje dotyczące powiązań między komponentami. Mialo to szczególne znaczenie w pierwotnej bibliotece AWT, która korzystała z rodzimych elementów interfejsu. Rozmiar przycisku może znacznie różnić się w stylu Motif, w systemach Windows i Mac OS X, a program czy applet nie wie z góry, gdzie będzie uruchamiany. Zróżnicowanie w pewnym stopniu zniknęło wraz z pojawiением się biblioteki Swing. Jeśli aplikacja wymusza określony styl, np. Metal, to wygląda identycznie na wszystkich platformach. Jeśli jednak programista zezwoli użytkownikom na wybór odpowiadającego im stylu, musi przy aranżacji komponentów polegać na elastyczności zarządców rozkładu.

Od Java 1.0 biblioteka AWT udostępnia rozkład *GridBagLayout*, który układa komponenty w wierszach i kolumnach. Rozmiary kolumn i wierszy mogą się zmieniać, a komponenty mogą zajmować po kilka z nich. Ten rozkład jest bardzo elastyczny, ale nie mniej skomplikowany. Sam dźwięk słów *GridBagLayout* jest znany z tego, że przepelnia lękem serca programistów Javy.

Nieudaną próbą uwolnienia programistów od tyranii rozkładu GridBagLayout był projekt w ramach biblioteki Swing rozkładu o nazwie BoxLayout. Cytując za dokumentacją JDK klasy BoxLayout: „Zagnieźdzanie wielu paneli z różnymi kombinacjami poziomych i pionowych paneli (sic!) daje efekt podobny do rozkładu GridBagLayout przy jednoczesnym uniknięciu nadmiernej komplikacji”. Ponieważ jednak każdy blok jest ustawiany osobno, nie można za pomocą rozkładu BoxLayout ustawiać sąsiadujących komponentów zarówno w poziomie, jak i w pionie.

W Java SE 1.4 podjęto jeszcze jedną próbę zastąpienia rozkładu GridBagLayout, której owoceem jest SpringLayout. Komponenty w kontenerze łączą umowne sprężyny (ang. *springs*), które rozciągają się lub kurczą w odpowiedzi na zmiany rozmiaru kontenera, dostosowując położenie komponentów. Brzmi to niezbyt zachęcająco i rzeczywiście rozkład sprężynowy (ang. *spring layout*) szybko poszedł w zapomnienie.

W 2005 roku zespół pracujący nad projektem NetBeans opracował technologię Matisse, która stanowi połoczenie narzędzia do opracowywania rozkładu i zarządcy rozkładu. Projektant interfejsu użytkownika upuszcza komponenty w kontenerze i określa, które z nich powinny się znajdować w jednej linii. Narzędzie konwertuje zamierzenia projektanta na instrukcje dla **zarządcy rozkładu grupowego** (ang. *group layout manager*). Jest to o wiele wygodniejsze podejście niż własnoręczne pisanie całego kodu zarządcy rozkładu. Zarządcą rozkładu grupowego wchodzi obecnie w skład Java SE 6. Zalecamy używanie narzędzi do budowy GUI środowiska NetBeans nawet tym, którzy na co dzień korzystają z innego IDE. Można zaprojektować GUI w NetBeans, a wygenerowany kod przenieść do dowolnego wybranego IDE.

W kolejnym podrozdziale opisujemy rozkład GridBagLayout, ponieważ jest on w powszechnym użyciu i nadal jest najłatwiejszą techniką tworzenia kodu rozkładu w starszych wersjach Javy. Opisujemy strategie, dzięki którym w większości sytuacji rozkład GridBagLayout można w dużym stopniu ujarzmić.

W dalszej kolejności przechodzimy do opisu narzędzia Matisse i zarządcy rozkładu grupowego. Wiedza dotycząca działania zarządcy rozkładu grupowego jest potrzebna do sprawdzenia, czy Matisse wygenerował prawidłowe instrukcje podczas wizualnego pozycjonowania komponentów.

Tematykę zarządców rozkładu kończymy prezentacją sposobu całkowitego pominiecia zarządzania rozkładem i ręcznego ustawiania komponentów oraz pisaniem własnego zarządcy rozkładu.

9.6.1. Rozkład GridBagLayout

Rozkład GridBagLayout jest przodem wszystkich zarządców rozkładu. Można go sobie wyobrazić jako rozkład siatkowy pozbawiony ograniczeń. Wiersze i kolumny mogą mieć różne rozmiary. Sąsiadujące ze sobą komórki można łączyć w celu zrobienia miejsca dla dużych komponentów (zarówno wiele procesorów tekstu, jak i język HTML oferują takie same możliwości edycji tabel — najpierw tworzy się zwykłą siatkę, a następnie scala przylegające komórki w razie potrzeby). Komponent nie musi zajmować całej powierzchni komórki, a jego położenie w komórce można kontrolować.

Przyjrzyjmy się opcjom dotyczącym wyboru własności czcionki na rysunku 9.29. Opierają się one na następujących komponentach:

- dwie listy rozwijalne służące do wyboru kroju i rozmiaru czcionki,
- etykiety list rozwijalnych,
- dwa pola wyboru służące do pogrubienia i pochylenia czcionki,
- obszar tekstowy zawierający przykładowy tekst.

Rysunek 9.29.

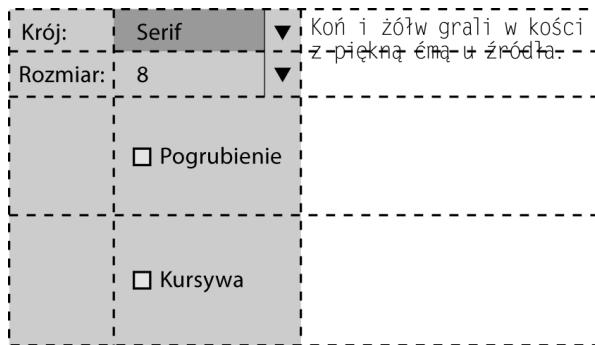
Opcje czcionki



Podzielmy teraz cały kontener na siatkę komórek, jak na rysunku 9.30 (wiersze i kolumny nie muszą mieć takich samych rozmiarów). Każde pole wyboru zajmuje dwie kolumny, a obszar tekstowy zajmuje cztery wiersze.

Rysunek 9.30.

Siatka ujęta do zaprojektowania okna dialogowego



Utworzenie powyższej siatki za pomocą zarządcy GridBagLayout wymaga następujących czynności:

1. Utwórz obiekt typu `GridBagLayout`. Nie trzeba podawać liczby wierszy i kolumn, z których ma się składać siatka. Zarządcą sam spróbuje te informacje zdobyć na podstawie danych dostarczonych później.
2. Ustaw utworzony obiekt typu `GridBagLayout` jako zarządcę rozkładu komponentu.
3. Dla każdego komponentu utwórz obiekt typu `GridBagConstraints`. Określ układ komponentów w siatce poprzez odpowiednie ustawienie wartości pól tego obiektu.
4. Dodaj każdy komponent z jego ograniczeniami (ang. *constraints*) za pomocą poniższego wywołania:

```
add(component, constraints);
```

Poniżej znajduje się przykładowy kod (ograniczenia opisujemy szczegółowo nieco dalej).

```

GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);

```

Sztuka polega na umiejętności ustaleniu stanu obiektu GridBagConstraints. Najważniejsze parametry tego typu obiektów opisane zostały w kolejnych podrozdziałach.

9.6.1.1. Parametry `gridx`, `gridy`, `gridwidth` i `gridheight`

Ograniczenia `gridx`, `gridy`, `gridwidth` i `gridheight` służą do określania lokalizacji komponentu na siatce. Parametry `gridx` i `gridy` określają wiersz i kolumnę, w których ma się znajdować lewy górny róg dodawanego komponentu. Wartości `gridwidth` i `gridheight` określają liczbę kolumn i wierszy zajmowanych przez komponent.

Wartości współrzędnych siatki zaczynają się od 0, tzn. punkt `gridx = 0` i `gridy = 0` jest lewym górnym rogiem. Na przykład współrzędne obszaru tekstowego na rysunku to `gridx = 2` i `gridy = 0`, ponieważ zaczyna się on w kolumnie numer 2 (czyli trzeciej) wiersza o numerze 0. Szerokość tego obszaru wynosi `gridwidth = 1` i `gridheight = 4`, czyli równa się jednej kolumnie i czterem wierszom.

9.6.1.2. Pola `weight`

Każdy komponent w rozkładzie `GridBagLayout` musi mieć ustawione pola `weight` (`weightx` i `weighty`). Wartość 0 powoduje, że komponent nie będzie się rozszerzał ani kurczył w stosunku do rozmiaru początkowego względem określonej osi. W rozkładzie na rysunku 9.29 ustawiliśmy pole `weightx` etykiet na 0. Dzięki temu będą one miały zawsze taki sam rozmiar, bez względu na rozmiar okna. Z drugiej strony, jeśli pola `weight` wszystkich elementów zostaną ustawione na 0, kontener zamiast rozciągać się na całą dostępną przestrzeń, będzie płynął się na jej środku.

Problemy związane z parametrami `weight` polegają na tym, że są one własnością wierszy i kolumn, a nie poszczególnych komórek. Trzeba jednak określić je w kategoriach komórek, ponieważ rozkład `GridBagLayout` nie eksponuje wierszy i kolumn. Wartości parametrów `weight` są obliczane jako maksimum wartości `weight` w każdym wierszu lub kolumnie. Aby zatem wiersz lub kolumna miały stały rozmiar, należy parametr `weight` wszystkich zawartych w nich komponentów ustawić na 0.

Należy pamiętać, że wartości `weight` nie określają względnych rozmiarów kolumn. Określają tylko, jaka część wolnej przestrzeni ma być przydzielona każdemu obszarowi, jeśli kontener przekroczy preferowany rozmiar. Takie działanie trudno opanować intuicyjnie. Zalecamy ustalenie wszystkich parametrów `weight` na 100. Następnie trzeba uruchomić program, aby sprawdzić, jak wygląda. Zmniejszając i zwiększając okno, można sprawdzić, jak dostosowują

się poszczególne wiersze i kolumny. Jeśli wyjdzie, że któryś wiersz lub któraś kolumna nie powinna się powiększać, należy ustawić parametry `weight` wszystkich znajdujących się w niej komponentów na 0. Można wypróbować także inne wartości `weight`, ale zazwyczaj nie przynosi to dobrego rezultatu.

9.6.1.3. Parametry `fill` i `anchor`

Aby komponent nie rozciągał się i nie zapełniał całej dostępnej przestrzeni, należy odpowiednio ustawić ograniczenie `fill`. Parametr ten może przyjmować jedną z czterech wartości, których poprawna postać jest następująca: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL` oraz `GridBagConstraints.BOTH`.

Jeśli komponent nie zajmuje całego dostępnego miejsca, można określić jego położenie w tym obszarze za pomocą pola `anchor`. Dostępne wartości to: `GridBagConstraints.CENTER` (domyślna), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST` oraz `GridBagConstraints.EAST`.

9.6.1.4. Dopełnienie

Komponent można otoczyć dodatkową pustą przestrzenią, odpowiednio ustawiając pole `insets` obiektu `GridBagConstraints`. W tym celu należy odpowiednio ustawić wartości `left`, `top`, `right` i `bottom` obiektu typu `Insets`. Jest to tak zwane **dopełnienie zewnętrzne** (ang. *external padding*).

Wartości `ipadx` i `ipady` określają **dopełnienie wewnętrzne** (ang. *internal padding*). Wartości te są dodawane do minimalnej szerokości i wysokości komponentu. Stanowi to zabezpieczenie przed skurczaniem się komponentu do minimalnych rozmiarów.

9.6.1.5. Inny sposób ustawiania wartości parametrów `gridx`, `gridy`, `gridwidth` i `gridheight`

Dokumentacja biblioteki AWT zaleca, aby zamiast bezwzględnych ustawień wartości `gridx` i `gridy` stosować stałą `GridBagConstraints.RELATIVE`. Komponenty natomiast należy dodawać w określonej kolejności od lewej do prawej w pierwszym wierszu, potem drugim itd.

Liczbę wierszy i kolumn w tym przypadku również określa się za pomocą odpowiednich ustawień parametrów `gridheight` i `gridwidth`. Wyjątek stanowi sytuacja, w której komponent sięga do **ostatniego** wiersza lub kolumny. Wtedy nie należy podawać konkretnej liczby, tylko użyć stałej `GridBagConstraints.REMAINDER`. Stanowi to informację dla zarządcy rozkładu, że dany komponent jest ostatni w wierszu.

Wydaje się, że opisywana metoda daje dobre rezultaty. Niezbyt rozsądne wydaje się jednak ukrywanie rzeczywistych informacji o położeniu przed zarządcą układu w nadziei, że zdoła on później je odzyskać.

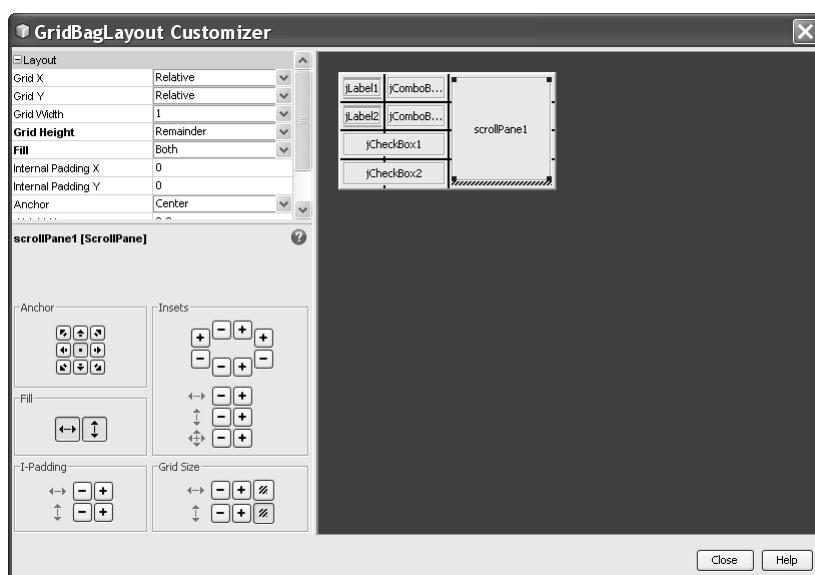
Wszystko to wydaje się nieco skomplikowane, ale poniższa strategia działania znacznie ułatwia opanowanie rozkładu `GridBagLayout`:

1. Narysuj szkic rozkładu komponentów na kartce.
2. Opracuj taką siatkę, w której wszystkie małe komponenty mieszczą się w pojedynczych komórkach, a większe komponenty zajmują po kilka komórek.
3. Oznacz wiersze i kolumny swojej siatki numerami 0, 1, 2, 3 itd. To ułatwia sprawdzenie wartości parametrów `gridx`, `gridy`, `gridwidth` i `gridheight`.
4. Dla każdego komponentu określ, czy ma on wypełniać swoją komórkę w pionie, czy poziomie. Jeśli nie ma wypełniać, zdecyduj o sposobie jego wyrównania. Służą do tego parametry `fill` i `anchor`.
5. Ustaw wszystkie parametry `weight` na 100. Aby dany wiersz lub kolumna zachowały swój domyślny rozmiar na stałe, ustaw wartość parametru `weightx` lub `weighty` wszystkich komponentów należących do tego wiersza lub kolumny na 0.
6. Napisz kod. Dokładnie sprawdź ustawienia `GridBagConstraints`. Jedna niepoprawna wartość może zniszczyć cały układ.
7. Skompiluj program, uruchom go i delektuj się.

Niektóre środowiska do budowy GUI udostępniają nawet wizualne narzędzia służące do określania ograniczeń. Rysunek 9.31 przedstawia okno dialogowe konfiguracji w NetBeans.

Rysunek 9.31.

Określanie ograniczeń rozkładu `GridBagLayout` w środowisku NetBeans



9.6.1.6. Klasa pomocnicza ułatwiająca pracę z ograniczeniami `GridBagLayout`

Najbardziej żmudną czynnością związaną z projektowaniem rozkładu `GridBagLayout` jest pisanie kodu ustawiającego ograniczenia. Większość programistów ułatwia sobie życie, pisząc funkcje lub małe klasy pomocnicze. Taką przykładową klasę prezentujemy pod listingiem omawianego do tej pory programu. Ta klasa ma następujące własności:

- Krótka nazwa — GBC zamiast GridBagConstraints.
- Dziedziczy po klasie GridBagConstraints, dzięki czemu można używać krótszych nazw stałych, np. GBC.EAST.
- Dodawanie komponentu odbywa się przy użyciu obiektu GBC, np.:


```
add(component, new GBC(1, 2));
```
- Posiada dwa konstruktory ustawiające najczęściej używane parametry: gridx i gridy lub gridx, gridy, gridheight i gridwidth.


```
add(component, new GBC(1, 2, 1, 4));
```
- Istnieją wygodne metody typu set dla pól występujących w parach x-y:


```
add(component, new GBC(1, 2).setWeight(100, 100));
```
- Metody set zwracają this, dzięki czemu można je stosować łańcuchowo:


```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));
```
- Metody setInsets tworzą obiekty typu Insets. Poniższa instrukcja tworzy wstawki jednopikselowe:


```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));
```

Listing 9.10 przedstawia kompletny kod programu do zmiany właściwości czcionek. Klasa GBC znajduje się na listingu 9.11. Poniższy kod dodaje komponenty do siatki:

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(bold, new GBC(0, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
```

Dla osób, które opanowały ograniczenia siatki, kod tego typu jest łatwy do odczytania i debogowania.



W kursie na stronie <http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html> znajduje się zalecenie, aby używać tego samego obiektu GridBagConstraints dla wszystkich komponentów. W naszym odczuciu powstały w ten sposób kod jest trudny do odczytania i podatny na błędy. Spójrzmy na przykład na demonstracyjny program dostępny na stronie <http://docs.oracle.com/javase/tutorial/uiswing/events/containerlistener.html>. Czy przyciski z założenia miały się rozciągać, czy może programista zapomniał wyłączyć ograniczenie fill?

Listing 9.10. gridbag/FontFrame.java

```
package gridbag;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;
```

```
/*
 * Ramka zawierająca komponenty ustawiające właściwości czcionki w rozkładzie GridBagLayout
 */
public class FontFrame extends JFrame
{
    public static final int TEXT_ROWS = 10;
    public static final int TEXT_COLUMNS = 20;

    private JComboBox<String> face;
    private JComboBox<Integer> size;
    private JCheckBox bold;
    private JCheckBox italic;
    private JTextArea sample;

    public FontFrame()
    {
        GridBagLayout layout = new GridBagLayout();
        setLayout(layout);

        ActionListener listener = EventHandler.create(ActionListener.class, this,
            "updateSample");

        // Tworzenie komponentów

        JLabel faceLabel = new JLabel("Krój: ");

        face = new JComboBox<>(new String[] { "Serif", "SansSerif", "Monospaced",
            "Dialog",
            "DialogInput" });

        face.addActionListener(listener);

        JLabel sizeLabel = new JLabel("Rozmiar: ");

        size = new JComboBox<>(new Integer[] { 8, 10, 12, 15, 18, 24, 36, 48 });

        size.addActionListener(listener);

        bold = new JCheckBox("Bold");
        bold.addActionListener(listener);

        italic = new JCheckBox("Italic");
        italic.addActionListener(listener);

        sample = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
        sample.setText("Koń i pies grali w kości w piękną ćma u źródła.");
        sample.setEditable(false);
        sample.setLineWrap(true);
        sample.setBorderStyle(BorderFactory.createEtchedBorder());

        // Dodawanie komponentów do siatki przy użyciu klasy pomocniczej GBC

        add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
        add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100,
            0).setInsets(1));
        add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
        add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100,
            0).setInsets(1));
    }
}
```

```
        add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
        add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
        add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
        pack();
        updateSample();
    }

    public void updateSample()
    {
        String fontFace = (String) face.getSelectedItem();
        int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
            + (italic.isSelected() ? Font.ITALIC : 0);
        int fontSize = size.getItemAt(size.getSelectedIndex());
        Font font = new Font(fontFace, fontStyle, fontSize);
        sample.setFont(font);
        sample.repaint();
    }
}
```

Listing 9.11. gridbag/GBC.java

```
package gridbag;

import java.awt.*;

/**
 * Ta klasa upraszcza korzystanie z klasy GridBagConstraints.
 * @version 1.01 2004-05-06
 * @author Cay Horstmann
 */
public class GBC extends GridBagConstraints
{
    /**
     * Tworzy obiekt typu GBC z podanymi wartościami gridx i gridy oraz wszystkimi pozostałymi
     * parametrami ustawnionymi na wartości domyślne.
     * @param gridx współrzędna gridx
     * @param gridy współrzędna gridy
     */
    public GBC(int gridx, int gridy)
    {
        this.gridx = gridx;
        this.gridy = gridy;
    }

    /**
     * Tworzy obiekt typu GBC z podanymi wartościami gridx, gridy, gridwidth i gridheight oraz
     * wszystkimi pozostałymi parametrami ustawnionymi na wartości domyślne.
     * @param gridx współrzędna gridx
     * @param gridy współrzędna gridy
     * @param gridwidth liczba zajmowanych komórek w poziomie
     * @param gridheight liczba zajmowanych komórek w pionie
     */
    public GBC(int gridx, int gridy, int gridwidth, int gridheight)
    {
        this.gridx = gridx;
        this.gridy = gridy;
        this.gridwidth = gridwidth;
        this.gridheight = gridheight;
    }
}
```

```
}

/**
 * Ustawia parametr anchor.
 * @param anchor wartość parametru anchor
 * @return this obiekt do dalszej modyfikacji
 */
public GBC setAnchor(int anchor)
{
    this.anchor = anchor;
    return this;
}

/**
 * Ustawia kierunek zapelniania.
 * @param fill kierunek zapelniania
 * @return this obiekt do dalszej modyfikacji
 */
public GBC setFill(int fill)
{
    this.fill = fill;
    return this;
}

/**
 * Ustawia parametry weight komórek.
 * @param weightx parametr weight w poziomie
 * @param weighty parametr weight w pionie
 * @return this obiekt do dalszej modyfikacji
 */
public GBC setWeight(double weightx, double weighty)
{
    this.weightx = weightx;
    this.weighty = weighty;
    return this;
}

/**
 * Ustawia dodatkową pustą przestrzeń w komórce.
 * @param distance dopełnienie we wszystkich kierunkach
 * @return this obiekt do dalszej modyfikacji
 */
public GBC setInsets(int distance)
{
    this.insets = new Insets(distance, distance, distance, distance);
    return this;
}

/**
 * Ustawia dopełnienia w komórce.
 * @param top odstęp od górnej krawędzi
 * @param left odstęp od lewej krawędzi
 * @param bottom odstęp od dolnej krawędzi
 * @param right odstęp od prawej krawędzi
 * @return obiekt do dalszej modyfikacji
 */
public GBC setInsets(int top, int left, int bottom, int right)
{
```

```

        this.insets = new Insets(top, left, bottom, right);
        return this;
    }

    /**
     * Ustawia dopełnienie wewnętrzne.
     * @param ipadx dopełnienie wewnętrzne poziome
     * @param ipady dopełnienie wewnętrzne pionowe
     * @return obiekt do dalszej modyfikacji
     */
    public GBC setIpad(int ipadx, int ipady)
    {
        this.ipadx = ipadx;
        this.ipady = ipady;
        return this;
    }
}

```

java.awt.GridBagConstraints 1.0

■ int gridx, gridy

Ustawia pierwszą kolumnę i pierwszy wiersz komórki. Domyślana wartość to 0.

■ int gridwidth, gridheight

Określa liczbę kolumn i wierszy zajmowanych przez komórkę. Domyślana wartość to 0.

■ double weightx, weighty

Określa możliwości komórki do powiększania się. Domyślana wartość to 0.

■ int anchor

Określa wyrównanie komponentu wewnątrz komórki. Dostępne są wartości bezwzględne:

NORTHWEST	NORTH	NORTHEAST
-----------	-------	-----------

WEST	CENTER	EAST
------	--------	------

SOUTHWEST	SOUTH	SOUTHEAST
-----------	-------	-----------

oraz ich odpowiednikie niezależne od orientacji:

FIRST_LINE_START	LINE_START	FIRST_LINE_END
------------------	------------	----------------

PAGE_START	CENTER	PAGE_END
------------	--------	----------

LAST_LINE_START	LINE_END	LAST_LINE_END
-----------------	----------	---------------

Tych drugich należy używać, jeśli program ma być lokalizowany w językach, w których kierunek pisma biegnie od lewej do prawej lub od dołu do góry. Wartość domyślana to CENTER.

■ int fill

Określa sposób wypełniania komórki przez komponent. Dostępne wartości to NONE, BOTH, HORIZONTAL i VERTICAL. Wartość domyślana to NONE.

■ int ipadx, ipady

Określa wewnętrzne dopełnienie wokół komponentu. Wartość domyślna to 0.

■ Insets insets

Określa zewnętrzne dopełnienie wzduż krawędzi komórki. Domyślnie brak dopełnienia.

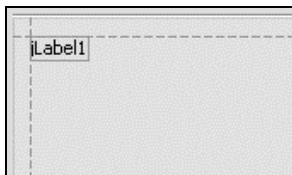
■ GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady) **1.2**

Tworzy obiekt typu GridBagConstraints z wartościami wszystkich pól ustawionymi na wartości podane w argumentach. Firma Sun zaleca, aby konstruktora tego używały wyłącznie automatyczne generatory kodu, ponieważ kod ten jest bardzo nieprzyjazny dla człowieka.

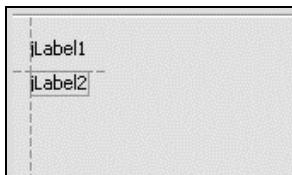
9.6.2. Rozkład grupowy

Przed rozpoczęciem opisu API klasy GroupLayout rzucimy okiem na narzędzie do budowy GUI w NetBeans, które kiedyś nazywało się Matisse. Nie będzie to jednak pełny kurs obsługi tego narzędzia. Więcej informacji o nim można znaleźć na stronie <https://netbeans.org/features/java/swing.html>.

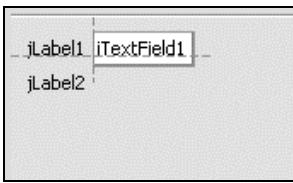
Czynności konieczne do utworzenia układu górnej części okna dialogowego widocznego na rysunku 9.13 są następujące: utwórz nowy projekt i dodaj formę JFrame. Przeciagnij etykietę, aż pojawią się dwie linie pomocnicze oddzielające etykietę od krawędzi kontenera.



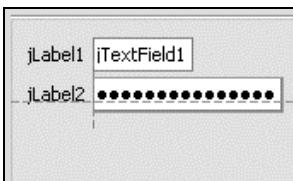
Umieść inną etykietę pod pierwszym wierszem.



Przeciagnij pole tekstowe, aby jego linia bazowa wyrównała się z linią bazową pierwszej etykiety. Ponownie zwróć uwagę na linie pomocnicze.



Na zakończenie ustaw pole hasła w jednej linii z dolną etykietą i kolumną z polem znajdującym się na górze.



Matisse wygeneruje następujący kod:

```

layout.setHorizontalGroup(
    layout.createParallelGroup(GridLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(GridLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel1)
                .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
                .addComponent(jTextField1))
            .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel2)
                .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
                .addComponent(jPasswordField1)))
        .addContainerGap(222, Short.MAX_VALUE)));
layout.setVerticalGroup(
    layout.createParallelGroup(GridLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(GridLayout.Alignment.BASELINE)
            .addComponent(jLabel1)
            .addComponent(jTextField1))
        .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
        .addGroup(layout.createParallelGroup(GridLayout.Alignment.BASELINE)
            .addComponent(jLabel2)
            .addComponent(jPasswordField1))
        .addContainerGap(244, Short.MAX_VALUE)));

```

Wygląda to dość strasznie, ale na szczęście nie trzeba pisać tego kodu własnoręcznie. Znajomość podstaw dotyczących akcji rozkładu jest jednak przydatna, ponieważ umożliwia znajdywanie błędów. Przeanalizujemy podstawową strukturę tego kodu. W wyciągach z API znajdujących się na końcu tego podrozdziału zostało wyjaśnione przeznaczenie wszystkich użytych tu klas i metod.

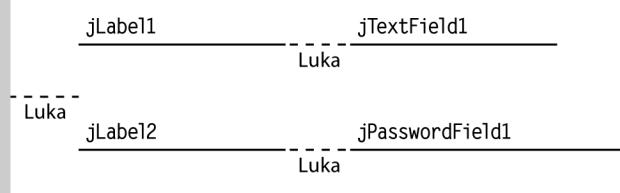
Komponenty są umieszczane w obiektach typu `SequentialGroup` lub `ParallelGroup`. Są to podklasy klasy `GroupLayout.Group`. Grupy mogą zawierać komponenty, luki pustego miejsca i zagnieżdżone grupy. Różne metody `add` klas grup zwracają obiekty grup, dzięki czemu można łączyć je w łańcuchy, np.:

```
group.addComponent(...).addPreferredGap(...).addComponent(...);
```

Jak widać w przykładowym kodzie, rozkład grupowy oddziela obliczenia związanego z ułożeniem w pionie i poziomie.

Ułożenie w poziomie można sobie wyobrazić jako komponenty o wysokości równej 0, jak na poniższym rysunku.

Krawędź kontenera



Są dwie równoległe sekwencje komponentów odpowiadające (nieco uproszczonemu) poniższemu kodowi:

```
.addContainerGap()
.addGroup(layout.createParallelGroup()
    .addGroup(layout.createSequentialGroup()
        .addComponent(jLabel1)
        .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
        .addComponent(jTextField1))
    .addGroup(layout.createSequentialGroup()
        .addComponent(jLabel2)
        .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
        .addComponent(jPasswordField1)))
```

Ale to przecież nie może działać prawidłowo. Skoro etykiety mają różne długości, pole tekstowe i pole hasła nie mogą być wyrównane w jednej linii.

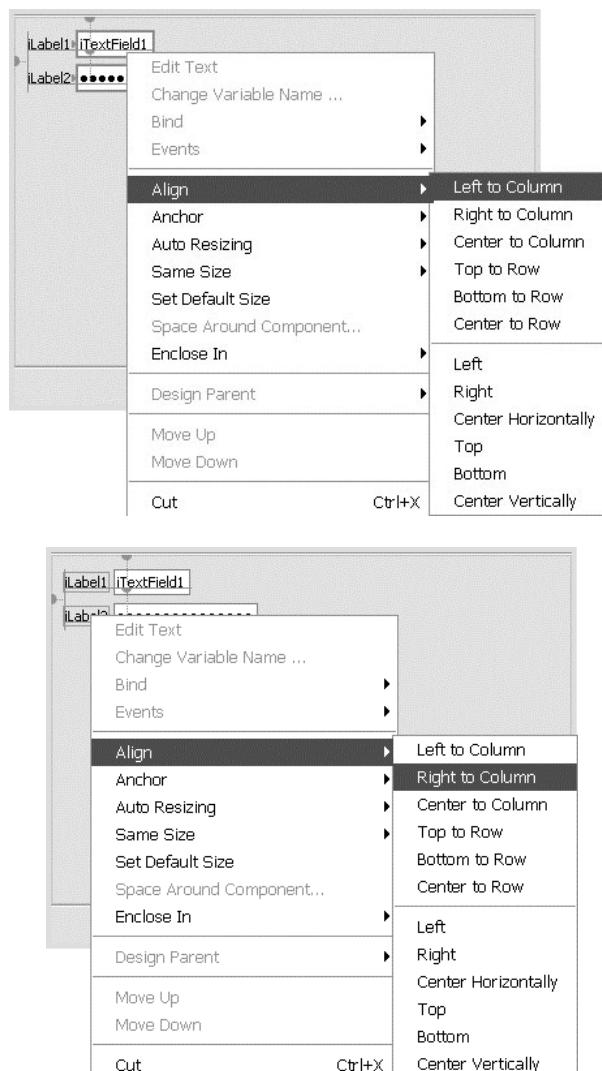
Musimy poinformować program Matisse, że pola mają być wyrównane. Zaznacz oba pola, kliknij prawym przyciskiem myszy i wybierz opcję *Align/Left to Column*. Wyrównaj też etykiety (rysunek 9.32).

Czynności te powodują duże zmiany w kodzie:

```
.addGroup(layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jLabel1, GroupLayout.Alignment.TRAILING)
        .addComponent(jLabel2, GroupLayout.Alignment.TRAILING))
    .addPreferredGap(LayoutConstraint.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jTextField1)
        .addComponent(jPasswordField1)))
```

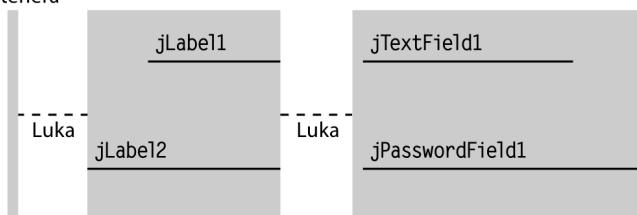
Rysunek 9.32.

Wyrównywanie etykiet i pól tekstowych w Matisse



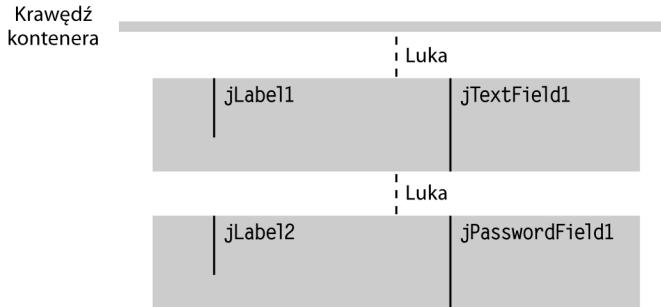
Teraz etykiety i pola znajdują się w równoległych grupach. Pierwsza grupa ma wyrównanie TRAILING (czyli wyrównanie do prawej przy kierunku tekstu w prawo):

Krawędź kontenera



Zdolność Matisse do zamieniania instrukcji projektanta na zagnieżdżone grupy wydaje się magią, ale — jak powiedział Arthur C. Clarke — każdą wystarczająco zaawansowaną technologię można odróżnić od czarów.

Aby wszystko było jasne, przyjrzymy się także pionowym obliczeniom. Tym razem komponenty należy traktować tak, jakby nie miały szerokości. Jest jedna grupa sekwencyjna zawierająca dwie równoległe grupy, rozdzielone pustymi przestrzeniami.



Odpowiadający im kod jest następujący:

```
layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel1)
        .addComponent(jTextField1))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel2)
        .addComponent(jPasswordField1))
```

Jak widać w kodzie, komponenty zostały wyrównane względem linii bazowych (linia bazowa to linia, na której opiera się tekst komponentu).

Można wymusić, aby kilka komponentów miało taki sam rozmiar. Na przykład można sprawić, aby pole tekstowe i pole hasła miały dokładnie takie same szerokości. W tym celu w Matisse należy kliknąć prawym przyciskiem myszy i wybrać opcję *Same Size/Same Width* (rysunek 9.33).

Matisse doda następującą instrukcję do kodu rozkładu:

```
layout.linkSize(SwingConstants.HORIZONTAL, new Component[] {jPasswordField1, jTextField1});
```

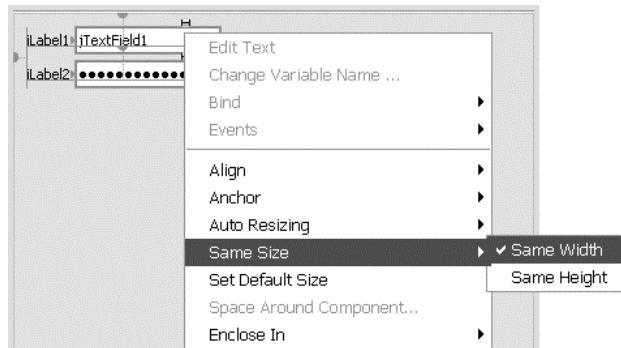
Kod na listingu 9.12 przedstawia rozkład programu z poprzedniego podrozdziału przy użyciu klasy GroupLayout zamiast GridBagLayout. Kod może nie wydawać się ani trochę prostszy niż przedstawiony na listingu 9.10, ale tego nie musielibyśmy pisać. Komponenty rozmieliśmy za pomocą Matisse, a później nieco oczyściliśmy wygenerowany kod.

Listing 9.12. GroupLayout/FontFrame.java

```
package GroupLayout;
import java.awt.*;
```

Rysunek 9.33.

Wymuszanie
tej samej
szerokości
dla dwóch
komponentów



```

import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

/**
 * Ramka, której komponenty zostały ułożone za pomocą zarządcy GroupLayout
 */
public class FontFrame extends JFrame
{
    public static final int TEXT_ROWS = 10;
    public static final int TEXT_COLUMNS = 20;

    private JComboBox<String> face;
    private JComboBox<Integer> size;
    private JCheckBox bold;
    private JCheckBox italic;
    private JScrollPane pane;
    private JTextArea sample;

    public FontFrame()
    {
        ActionListener listener = EventHandler.create(ActionListener.class, this,
            "updateSample");

        // Tworzenie komponentów

        JLabel faceLabel = new JLabel("Król: ");

        face = new JComboBox<>(new String[] { "Serif", "SansSerif", "Monospaced",
            "Dialog", "DialogInput" });

        face.addActionListener(listener);

        JLabel sizeLabel = new JLabel("Rozmiar: ");

        size = new JComboBox<>(new Integer[] { 8, 10, 12, 15, 18, 24, 36, 48 });

        size.addActionListener(listener);

        bold = new JCheckBox("Bold");
        bold.addActionListener(listener);
    }
}

```



```

        GroupLayout.Alignment.
        ↪BASELINE).addComponent(size
        .addComponent(sizeLabel)).
        ↪addPreferredGap(
        LayoutStyle.ComponentPlacement.
        ↪RELATED).addComponent(
        italic, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
        .addPreferredGap(LayoutStyle.
        ↪ComponentPlacement.RELATED)
        .addComponent(bold, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.
        ↪MAX_VALUE)))
        .addContainerGap()));
    pack();
}

public void updateSample()
{
    String fontFace = (String) face.getSelectedItem();
    int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
        + (italic.isSelected() ? Font.ITALIC : 0);
    int fontSize = size.getItemAt(size.getSelectedIndex());
    Font font = new Font(fontFace, fontStyle, fontSize);
    sample.setFont(font);
    sample.repaint();
}
}

```

javax.swing.GroupLayout 6

■ GroupLayout(Container host)

Tworzy obiekt GroupLayout służący do rozkładu komponentów w kontenerze host. Uwaga: konieczne jest wywołanie metody `setLayout` na rzecz obiektu kontenera.

■ void setHorizontalGroup(GroupLayout.Group g)

■ void setVerticalGroup(GroupLayout.Group g)

Ustawia grupę odpowiedzialną za rozkład w poziomie lub pionie.

■ void linkSize(Component... components)

■ void linkSize(int axis, Component... component)

Wymusza taki sam rozmiar komponentów lub taki sam rozmiar względem tylko jednej z osi (SwingConstants.HORIZONTAL lub SwingConstants.VERTICAL).

■ GroupLayout.SequentialGroup createSequentialGroup()

Tworzy grupę, która układa swoich potomków sekwencyjnie.

■ GroupLayout.ParallelGroup createParallelGroup()

■ GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align)

- `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align, boolean resizable)`

Tworzy grupę, która układa swoich potomków równolegle.

Parametry: `align` Wartość `BASELINE`, `LEADING` (domyślna), `TRAILING` lub `CENTER`.

`resizable` Wartość `true`, jeśli grupa może zmieniać rozmiar, `false`, jeśli preferowany rozmiar jest jednocześnie rozmiarem minimalnym i maksymalnym.

- `boolean getHonorsVisibility()`
- `void setHonorsVisibility(boolean b)`

Pobiera lub ustawia właściwość `honorsVisibility`. Wartość `true` oznacza, że komponenty niewidoczne nie będą brane pod uwagę w rozkładzie. Wartość `false` oznacza traktowanie ich jak elementy widoczne. Opcje te pozwalają chwilowo ukryć niektóre komponenty bez zmiany układu.

- `boolean getAutoCreateGaps()`
- `void setAutoCreateGaps(boolean b)`
- `boolean getAutoCreateContainerGaps()`
- `void setAutoCreateContainerGaps(boolean b)`

Pobiera i ustawia właściwości `autoCreateGaps` i `autoCreateContainerGaps`. Wartość `true` oznacza automatyczne dodawanie przerw pomiędzy komponentami lub pomiędzy krawędziami kontenera a komponentami do nich przylegającymi. Wartość domyślna to `false`. Wartość `true` jest przydatna podczas ręcznego tworzenia rozkładu `GroupLayout`.

`javax.swing.GroupLayout.Group`

- `GroupLayout.Group addComponent(Component c)`
- `GroupLayout.Group addComponent(Component c, int minimumSize, int preferredSize, int maximumSize)`

Dodaje komponent do grupy. Wartości parametrów określających rozmiar mogą być nieujemne lub stałymi `GroupLayout.DEFAULT_SIZE` albo `GroupLayout.PREFERRED_SIZE`. Użycie stałej `DEFAULT_SIZE` powoduje wywołanie metody komponentu `getMinimumSize`, `getPreferredSize` lub `getMaximumSize`. Stała `PREFERRED_SIZE` powoduje wywołanie metody `getPreferredSize`.

- `GroupLayout.Group addGap(int size)`
- `GroupLayout.Group addGap(int minimumSize, int preferredSize, int maximumSize)`

Tworzy przerwę o podanym stałym lub elastycznym rozmiarze.

- `GroupLayout.Group addGroup(GroupLayout.Group g)`

Dodaje określoną grupę do grupy.

```
javax.swing.GroupLayout.ParallelGroup
```

- GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align)
- GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align, int minimumSize, int preferredSize, int maximumSize)
- GroupLayout.ParallelGroup addGroup(GroupLayout.Group g, GroupLayout.Alignment align)

Dodaje komponent lub grupę do grupy, stosując określony sposób wyrównania — BASELINE, LEADING,.TRAILING lub CENTER.

```
javax.swing.GroupLayout.SequentialGroup
```

- GroupLayout.SequentialGroup addContainerGap()
 - GroupLayout.SequentialGroup addContainerGap(int preferredSize, int maximumSize)
- Tworzy lukę oddzielającą komponent od krawędzi kontenera.
- GroupLayout.SequentialGroup addPreferredGap(LayoutConstraint.ComponentPlacement type)

Tworzy lukę rozdzielającą komponenty. Parametr type może przyjąć wartość `LayoutConstraint.ComponentPlacement.RELATED` lub `LayoutConstraint.ComponentPlacement.UNRELATED`.

9.6.3. Nieużywanie żadnego zarządcy rozkładu

Zdarzają się sytuacje, w których programista nie chce zaprzątać sobie głowy żadnym zarządcą rozkładu, ponieważ chce jedynie umieścić jakiś komponent w określonym miejscu (czasami nazywa się to **pozycjonowaniem bezwzględnym** — ang. *absolute positioning*). Technika ta nie jest dobrym rozwiązaniem w aplikacjach niezależnych od platformy, ale doskonale nadaje się do szybkiego utworzenia prototypu.

Aby umieścić komponent na stałe w określonym miejscu, należy:

1. Ustawić zarządcę rozkładu na null.
2. Wstawić wybrany komponent do kontenera.
3. Określić położenie i rozmiar:

```
frame.setLayout(null);
JButton ok = new JButton("OK");
frame.add(ok);
ok.setBounds(10, 10, 30, 15);
```

```
java.awt.Component 1.0
```

- void setBounds(int x, int y, int width, int height)
- Ustawia położenie i rozmiar komponentu.

Parametry:	<code>x, y</code>	Nowy lewy górny róg komponentu
	<code>width, height</code>	Nowy rozmiar komponentu

9.6.4. Niestandardowi zarządcy rozkładu

Istnieje możliwość utworzenia własnej klasy zarządzającej rozkładem komponentów w specjalny sposób. W ramach przykładu prezentujemy zarządcę rozmieszczającego komponenty na krawędzi koła (rysunek 9.34).

Rysunek 9.34.

Rozkład kołowy



Niestandardowy zarządcza rozkładu musi implementować interfejs `LayoutManager`. Konieczne jest przesłonięcie następujących metod:

```
void addLayoutComponent(String s, Component c);
void removeLayoutComponent(Component c);
Dimension preferredLayoutSize(Container parent);
Dimension minimumLayoutSize(Container parent);
void layoutContainer(Container parent);
```

Pierwsze dwie są wywoływanie przy dodawaniu i usuwaniu komponentów. Jeśli nie ma wymogu przechowywania żadnych dodatkowych informacji o komponentach, można te metody zdefiniować w taki sposób, aby nic nie robiły. Kolejne dwie metody obliczają przestrzeń wymaganą przez minimalny i preferowany rozkład komponentów. Zazwyczaj wartości te są sobie równe. Piąta metoda wykonuje rzeczywistą pracę i wywołuje metodę `setBounds` na przecz wszystkich komponentów.



Biblioteka AWT udostępnia jeszcze interfejs o nazwie `LayoutManager2` z dziesięcioma metodami. Jego głównym przeznaczeniem jest umożliwienie programistom korzystania z metody `add` z ograniczeniami. Interfejs ten implementują na przykład klasy `BorderLayout` i `GridBagLayout`.

Listing 9.13 przedstawia kod bezużytecznego zarządcy `CircleLayout`, który układa komponenty na krawędzi koła. Klasa ramowa tego programu jest przedstawiona na listingu 9.14.

Listing 9.13. `circleLayout/CircleLayout.java`

```
package circleLayout;
import java.awt.*;
```

```
/*
 * Ramka zawierająca komponenty ułożone w kółko
 */
public class CircleLayout implements LayoutManager
{
    private int minWidth = 0;
    private int minHeight = 0;
    private int preferredWidth = 0;
    private int preferredHeight = 0;
    private boolean sizesSet = false;
    private int maxComponentWidth = 0;
    private int maxComponentHeight = 0;

    public void addLayoutComponent(String name, Component comp)
    {
    }

    public void removeLayoutComponent(Component comp)
    {
    }

    public void setSizes(Container parent)
    {
        if (sizesSet) return;
        int n = parent.getComponentCount();

        preferredWidth = 0;
        preferredHeight = 0;
        minWidth = 0;
        minHeight = 0;
        maxComponentWidth = 0;
        maxComponentHeight = 0;

        // Obliczanie maksymalnych szerokości i wysokości komponentów
        // oraz ustawianie preferowanego rozmiaru na sumę rozmiarów komponentów
        for (int i = 0; i < n; i++)
        {
            Component c = parent.getComponent(i);
            if (c.isVisible())
            {
                Dimension d = c.getPreferredSize();
                maxComponentWidth = Math.max(maxComponentWidth, d.width);
                maxComponentHeight = Math.max(maxComponentHeight, d.height);
                preferredWidth += d.width;
                preferredHeight += d.height;
            }
        }
        minWidth = preferredWidth / 2;
        minHeight = preferredHeight / 2;
        sizesSet = true;
    }

    public Dimension preferredLayoutSize(Container parent)
    {
        setSizes(parent);
        Insets insets = parent.getInsets();
        int width = preferredWidth + insets.left + insets.right;
        int height = preferredHeight + insets.top + insets.bottom;
        return new Dimension(width, height);
    }
}
```

```

        int height = preferredHeight + insets.top + insets.bottom;
        return new Dimension(width, height);
    }

    public Dimension minimumLayoutSize(Container parent)
    {
        setSizes(parent);
        Insets insets = parent.getInsets();
        int width = minWidth + insets.left + insets.right;
        int height = minHeight + insets.top + insets.bottom;
        return new Dimension(width, height);
    }

    public void layoutContainer(Container parent)
    {
        setSizes(parent);

        // Obliczenie środka okręgu

        Insets insets = parent.getInsets();
        int containerWidth = parent.getSize().width - insets.left - insets.right;
        int containerHeight = parent.getSize().height - insets.top - insets.bottom;

        int xcenter = insets.left + containerWidth / 2;
        int ycenter = insets.top + containerHeight / 2;

        // Obliczenie promienia okręgu

        int xradius = (containerWidth - maxComponentWidth) / 2;
        int yradius = (containerHeight - maxComponentHeight) / 2;
        int radius = Math.min(xradius, yradius);

        // Układanie komponentów na okręgu

        int n = parent.getComponentCount();
        for (int i = 0; i < n; i++)
        {
            Component c = parent.getComponent(i);
            if (c.isVisible())
            {
                double angle = 2 * Math.PI * i / n;

                // Środek komponentu
                int x = xcenter + (int) (Math.cos(angle) * radius);
                int y = ycenter + (int) (Math.sin(angle) * radius);

                // Przesunięcie komponentu, aby jego środek znajdował się w punkcie (x, y),
                // a jego rozmiar był rozmiarem preferowanym
                Dimension d = c.getPreferredSize();
                c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
            }
        }
    }
}

```

Listing 9.14. circleLayout/CircleLayoutFrame.java

```
package circleLayout;

import javax.swing.*;

/***
 * Ramka zawierająca przyciski ulożone na obwodzie okręgu
 */
public class CircleLayoutFrame extends JFrame
{
    public CircleLayoutFrame()
    {
        setLayout(new CircleLayout());
        add(new JButton("Żółty"));
        add(new JButton("Niebieski"));
        add(new JButton("Czerwony"));
        add(new JButton("Zielony"));
        add(new JButton("Pomarańczowy"));
        add(new JButton("Fuksjja"));
        add(new JButton("Błękit"));
        pack();
    }
}
```

java.awt.LayoutManager 1.0

- void addLayoutComponent(String name, Component comp)

Dodaje komponent do rozkładu.

Parametry:	name	Identyfikator położenia komponentu
	comp	Komponent, który ma być wstawiony

- void removeLayoutComponent(Component comp)

Usuwa komponent.

- Dimension preferredLayoutSize(Container cont)

Zwraca preferowane wymiary kontenera.

- ## ■ Dimension minimumLayoutSize(Container cont)

Zwraca minimalne wymiary kontenera.

- ## ■ void layoutContainer(Container cont)

Układa komponenty w kontenerze.

9.6.5. Kolejka dostępu

Dodając kilka komponentów do okna, należy przemyśleć **kolejkę dostępu** (ang. *traversal order*) do nich. W chwili pierwszego pojawienia się okna aktywny jest komponent będący na pierwszym miejscu w kolejce dostępu. Naciśnięcie klawisza *Tab* powoduje aktywowanie

kolejnego komponentu (przypomnijmy, że komponentami posiadającymi fokus klawiaturowy można sterować za pomocą klawiatury, np. można kliknąć przycisk za pomocą spacji). Jak wiadomo, wiele osób używa klawisza *Tab* do nawigacji po elementach sterujących interfejsu. Należą do nich między innymi osoby nielubiące używać myszki oraz ci, którzy nie mogą jej używać ze względu na upośledzenia ruchowe bądź też osoby poruszające się po interfejsie za pomocą poleceń głosowych. Są to wystarczające powody do poznania sposobu obsługi kolejki dostępu przez Swinga.

Kolejność dostępu do komponentów jest bardzo prosta i odbywa się od lewej do prawej i od góry do dołu. Na przykład kolejność dostępu do komponentów w programie zmieniającym własności czcionki jest następująca (rysunek 9.35):

- ① Lista rozwijalna *Król*.
- ② Obszar tekstowy z przykładowym tekstem (aby przejść do następnego pola, należy nacisnąć kombinację klawiszy *Ctrl+Tab*; znak tabulatora jest uznawany za tekst).
- ③ Lista rozwijalna *Rozmiar*.
- ④ Pole wyboru *Pogrubienie*.
- ⑤ Pole wyboru *Kursywa*.

Rysunek 9.35.

Kolejność dostępu



Sytuacja komplikuje się, jeśli kontener zawiera inne kontenery. Kiedy aktywowany jest inny kontener, aktywny staje się komponent znajdujący się w jego lewym górnym rogu, a następnie aktywowane są kolejne komponenty w tym kontenerze. W końcu aktywowany jest komponent znajdujący się za wspomnianym kontenerem.

Cechę tę można obrócić na swoją korzyść, grupując powiązane elementy w dodatkowym kontenerze, np. panelu.



Elementy z kolejki dostępu usuwa się za pomocą instrukcji podobnej do poniższej:

```
component.setFocusable(false);
```

Technika ta jest przydatna w przypadku rysowanych komponentów, które nie pobierają danych z klawiatury.

9.7. Okna dialogowe

Wszystkie tworzone do tej pory komponenty były wyświetlane w ramach okna tworzonego w aplikacji. Tego rodzaju sytuacje najczęściej spotyka się w **apletach**, które działają w oknie przeglądarki. Natomiast w samodzielnych aplikacjach często stosowane są oddzielne okna dialogowe służące do podawania informacji użytkownikowi lub pobierania ich od użytkownika.

Podobnie jak większość systemów okienkowych, w bibliotece AWT wyróżnia się **okna dialogowe modalne** (ang. *modal dialog box*) i **okna dialogowe niemodalne** (ang. *modeless dialog box*). Modalne okno dialogowe blokuje dostęp do pozostałych okien aplikacji, dzięki czemu znajduje zastosowanie w sytuacjach, kiedy przed kontynuacją działania program potrzebuje informacji od użytkownika. Na przykład modalne okno dialogowe pojawia się, kiedy użytkownik chce wczytać plik. Przed rozpoczęciem operacji konieczne jest podanie nazwy pliku do wczytania. Aplikacja może kontynuować działanie dopiero po zamknięciu (modalnego) okna dialogowego.

Niemodalne okna dialogowe nie blokują dostępu do reszty aplikacji w trakcie podawania informacji przez użytkownika. Oknem tego typu jest pasek narzędzi. Pasek ten pozostaje widoczny tyle czasu, ile potrzeba, i nie przeszkadza to użytkownikowi w korzystaniu w tym czasie z innych okien aplikacji.

Zacznijemy od najprostszego rodzaju okien dialogowych, czyli okien modalnych wyświetlających jeden komunikat. W bibliotece Swing dostępna jest klasa `JOptionPane`, która umożliwia tworzenie prostych okien dialogowych bez konieczności pisania specjalnego kodu. Następnie zajmiemy się tworzeniem bardziej złożonych okien dialogowych opartych na własnych oknach. Na koniec nauczymy się przenosić dane z aplikacji do okna dialogowego i z powrotem.

Na zakończenie tego podrozdziału prezentujemy dwa standardowe okna dialogowe — wyboru pliku i koloru. Okna dialogowe wyboru pliku są skomplikowane i do ich tworzenia potrzebna jest znajomość klasy Swing `JFileChooser` — napisanie własnej takiej klasy byłoby nie lada wyzwaniem. Okno dialogowe `JColorChooser` służy do wybierania kolorów.

9.7.1. Okna dialogowe opcji

W bibliotece Swing dostępny jest zestaw gotowych do użycia okien dialogowych, które z powodzeniem można wykorzystać do pobrania pojedynczej prostej informacji od użytkownika. Klasa `JOptionPane` udostępnia cztery statyczne metody wyświetlające te proste okna:

- `showMessageDialog` — wyświetla komunikat i czeka, aż użytkownik kliknie przycisk *OK*.
- `showConfirmDialog` — wyświetla komunikat i odbiera potwierdzenie (typu *OK/Cancel*).
- `showOptionDialog` — wyświetla komunikat i odbiera opcję użytkownika z określonego zestawu.

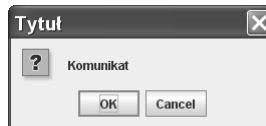
- `showInputDialog` — wyświetla komunikat i odbiera jeden wiersz danych od użytkownika.

Rysunek 9.36 przedstawia typowe okno dialogowe. Składa się ono z następujących komponentów:

- ikona,
- komunikat,
- dwa przyciski opcji.

Rysunek 9.36.

Okno dialogowe opcji



Okno dialogowe przyjmujące dane wejściowe (ang. *input dialog*) zawiera dodatkowy komponent służący do odbierania danych od użytkownika. Może to być pole tekstowe, w którym użytkownik wpisuje dowolny łańcuch tekstowy, albo lista rozwijalna z kilkoma opcjami do wyboru.

Szczegóły wyglądu tych okien dialogowych oraz dobór ikon dla standardowych typów komunikatów zależą od stylu.

Ikona po lewej stronie zależy od **typu komunikatu**, których jest pięć:

ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE

Typ PLAIN_MESSAGE nie ma żadnej ikony. Każdy rodzaj okna dialogowego posiada także metodę, za pomocą której można wstawić własną ikonę.

Każdy typ okna dialogowego pozwala na podanie komunikatu. Może to być łańcuch tekstu, ikona, komponent interfejsu użytkownika lub dowolny inny obiekt. Obiekt komunikatu jest wyświetlany następująco:

String	Rysuje łańcuch.
Icon	Wyświetla ikonę.
Component	Wyświetla komponent.
Object[]	Wyświetla wszystkie obiekty w tablicy jeden nad drugim.
Dowolny inny obiekt	Stosuje metodę <code>toString</code> i wyświetla uzyskany łańcuch.

Opcje te można obejrzeć, uruchamiając program z listingu 9.15.

Oczywiście zdecydowanie najczęściej wykorzystywana jest opcja podawania łańcucha komunikatu. Dostarczenie obiektu Component daje dużą elastyczność, ponieważ można sprawić, aby metoda `paintComponent` narysowała wszystko, co można zechcieć.

Przyciski na dole zależą od typu okna dialogowego i **typu opcji**. Metody `showMessageDialog` i `showInputDialog` dostarczają tylko standardowe przyciski (odpowiednio *OK* i *OK/Cancel*). Metoda `showConfirmDialog` przyjmuje jeden z czterech typów opcji:

```
DEFAULT_OPTION  
YES_NO_OPTION  
YES_NO_CANCEL_OPTION  
OK_CANCEL_OPTION
```

Metoda `showOptionDialog` pozwala na utworzenie dowolnego zestawu opcji. Opcje podawane są w tablicy obiektów. Każdy element tablicy jest wizualizowany w następujący sposób:

String	Tworzy przycisk, którego etykieta jest łańcuch.
Icon	Tworzy przycisk, którego etykieta jest ikona.
Component	Wyświetla komponent.
Dowolny inny obiekt	Stosuje metodę <code>toString</code> i tworzy przycisk, którego etykieta jest uzyskany łańcuch.

Wartości zwracane przez powyższe funkcje są następujące:

<code>showMessageDialog</code>	Brak
<code>showConfirmDialog</code>	Liczba całkowita reprezentująca wybraną opcję
<code>showOptionDialog</code>	Liczba całkowita reprezentująca wybraną opcję
<code>showInputDialog</code>	Łańcuch wprowadzony lub wybrany przez użytkownika

Metody `showConfirmDialog` i `showOptionDialog` zwracają liczby całkowite reprezentujące kliknięty przez użytkownika przycisk. W przypadku okna dialogowego jest to zwykły indeks wybranej opcji lub wartość `CLOSED_OPTION`, jeśli użytkownik zamknął okno, nie wybierając żadnej opcji. W oknie dialogowym potwierdzenia (ang. *confirmation dialog*) dostępne są następujące wartości zwrotne:

```
OK_OPTION  
CANCEL_OPTION  
YES_OPTION  
NO_OPTION  
CLOSED_OPTION
```

Na pierwszy rzut oka wydaje się, że opcji jest bardzo dużo, ale w praktyce opanowanie ich jest bardzo proste. Należy postępować zgodnie z poniższymi wskazówkami:

1. Wybierz rodzaj okna dialogowego (komunikat, potwierdzenie, opcje, dane wejściowe).
2. Wybierz ikonę (błąd, informacja, ostrzeżenie, pytanie, brak lub własna).
3. Wybierz rodzaj komunikatu (łańcuch, ikona, własny komponent lub stos komponentów).
4. W przypadku okna potwierdzenia wybierz typ opcji (Yes/No, Yes/No/Cancel lub OK/Cancel).

5. W przypadku okna dialogowego opcji wybierz opcje (łańcuchy, ikony lub własne komponenty) i opcję domyślną.
6. W przypadku okna dialogowego danych wejściowych zdecyduj, czy wybrać pole tekstowe, czy listę rozwijalną.
7. Zlokalizuj odpowiednią metodę do wywołania w API JOptionPane.

Wyobraźmy sobie na przykład, że chcemy utworzyć okno dialogowe widoczne na rysunku 9.36. Okno to wyświetla komunikat i prosi użytkownika o zatwierdzenie lub anulowanie. Jest to więc okno potwierdzenia. Jako ikona wyświetlił się znak zapytania. Typ opcji to OK_CANCEL_OPTION. Oto przykładowy kod tworzący takie okno:

```
int selection = JOptionPane.showConfirmDialog(parent,
    "Message", "Tytuł",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```



Łańcuch komunikatu może zawierać znaki nowego wiersza (\n), które powodują, że łańcuch zostanie podzielony na kilka wierszy.

Program, którego klasa ramowa jest przedstawiona na listingu 9.15, wyświetla sześć sekcji z przełącznikami (rysunek 9.37). Klasa tworząca te komponenty znajduje się na listingu 9.16. Naciśnięcie przycisku *Pokaż* powoduje wyświetlenie odpowiedniego okna dialogowego.

Rysunek 9.37.

Program
OptionDialogTest



Listing 9.15. optionDialog/OptionDialogFrame.java

```
package optionDialog;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
```

```

import java.util.*;
import javax.swing.*;

/**
 * Ramka zawierająca ustawienia dotyczące wyboru różnych okien dialogowych opcji
 */
public class OptionDialogFrame extends JFrame
{
    private JPanel typePanel;
    private JPanel messagePanel;
    private JPanel messageTypePanel;
    private JPanel optionTypePanel;
    private JPanel optionsPanel;
    private JPanel inputPanel;
    private String messageString = "Komunikat";
    private Icon messageIcon = new ImageIcon("blue-ball.gif");
    private Object messageObject = new Date();
    private Component messageComponent = new SampleComponent();

    public OptionDialogFrame()
    {
        JPanel gridPanel = new JPanel();
        gridPanel.setLayout(new GridLayout(2, 3));

        typePanel = new JPanel("Typ", "Komunikat", "Potwierdzenie", "Opcja",
        ↳"Dane wejściowe");
        messageTypePanel = new JPanel("Typ komunikatu", "ERROR_MESSAGE",
        ↳"INFORMATION_MESSAGE",
        "WARNING_MESSAGE", "QUESTION_MESSAGE", "PLAIN_MESSAGE");
        messagePanel = new JPanel("Komunikat", "Łańcuch", "Ikona", "Komponent",
        ↳"Inny", "Object[]");
        optionTypePanel = new JPanel("Potwierdzenie", "DEFAULT_OPTION",
        ↳"YES_NO_OPTION",
        "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION");
        optionsPanel = new JPanel("Opcja", "String[]", "Icon[]", "Object[]");
        inputPanel = new JPanel("Dane wejściowe", "Pole tekstowe", "Pole kombi");

        gridPanel.add(typePanel);
        gridPanel.add(messageTypePanel);
        gridPanel.add(messagePanel);
        gridPanel.add(optionTypePanel);
        gridPanel.add(optionsPanel);
        gridPanel.add(inputPanel);

        // Dodanie panelu z przyciskiem Pokaż

        JPanel showPanel = new JPanel();
        JButton showButton = new JButton("Pokaż");
        showButton.addActionListener(new ShowAction());
        showPanel.add(showButton);

        add(gridPanel, BorderLayout.CENTER);
        add(showPanel, BorderLayout.SOUTH);
        pack();
    }

    /**
     * Pobiera aktualnie wybrany komunikat.

```

```

* @return łańcuch, ikona, komponent lub tablica obiektów, w zależności od wyboru w panelu Komunikat
*/
public Object getMessage()
{
    String s = messagePanel.getSelection();
    if (s.equals("Łańcuch")) return messageString;
    else if (s.equals("Ikona")) return messageIcon;
    else if (s.equals("Komponent")) return messageComponent;
    else if (s.equals("Object[]")) return new Object[] { messageString,
        messageIcon,
        messageComponent, messageObject };
    else if (s.equals("Inny")) return messageObject;
    else return null;
}

/**
* Pobiera aktualnie wybrane opcje.
* @return tablica łańcuchów, ikon lub obiektów, w zależności od wyboru w panelu Opcja
*/
public Object[] getOptions()
{
    String s = optionsPanel.getSelection();
    if (s.equals("String[]")) return new String[] { "Żółty", "Niebieski",
        "Czerwony" };
    else if (s.equals("Icon[]")) return new Icon[] { new ImageIcon("yellow-
        ball.gif"),
        new ImageIcon("blue-ball.gif"), new ImageIcon("red-ball.gif") };
    else if (s.equals("Object[]")) return new Object[] { messageString,
        messageIcon,
        messageComponent, messageObject };
    else return null;
}

/**
* Pobiera wybrany komunikat lub typ opcji.
* @param panel Typ komunikatu lub panel Potwierdzenie
* @return wybrana stała XXX_MESSAGE lub XXX_OPTION z klasy JOptionPane
*/
public int getType(ButtonPanel panel)
{
    String s = panel.getSelection();
    try
    {
        return JOptionPane.class.getField(s).getInt(null);
    }
    catch (Exception e)
    {
        return -1;
    }
}

/**
* Słuchacz akcji przycisku Pokaż wyświetla okno dialogowe potwierdzenia, danych wejściowych,
* komunikatu lub opcji w zależności od wyboru typu panelu.
*/
private class ShowAction implements ActionListener
{

```

```

public void actionPerformed(ActionEvent event)
{
    if (typePanel.getSelection().equals("Potwierdzenie"))
        ↪JOptionPane.showConfirmDialog(
            OptionDialogFrame.this, getMessage(), "Tytuł",
            ↪getType(optionTypePanel),
            getType(messageTypePanel));
    else if (typePanel.getSelection().equals("Dane wejściowe"))
    {
        if (inputPanel.getSelection().equals("Pole tekstowe"))
            ↪JOptionPane.showInputDialog(
                OptionDialogFrame.this, getMessage(), "Tytuł",
                ↪getType(messageTypePanel));
        else JOptionPane.showInputDialog(OptionDialogFrame.this, getMessage(),
            ↪"Tytuł",
            getType(messageTypePanel), null, new String[] { "Żółty",
            ↪"Niebieski", "Czerwony" },
            "Niebieski");
    }
    else if (typePanel.getSelection().equals("Komunikat"))
        ↪JOptionPane.showMessageDialog(
            OptionDialogFrame.this, getMessage(), "Tytuł",
            ↪getType(messageTypePanel));
    else if (typePanel.getSelection().equals("Opcja"))
        ↪JOptionPane.showOptionDialog(
            OptionDialogFrame.this, getMessage(), "Tytuł",
            ↪getType(optionTypePanel),
            getType(messageTypePanel), null, getOptions(), getOptions()[0]);
    }
}
}

/**
 * Komponent z pomalowaną powierzchnią
 */
class SampleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        Rectangle2D rect = new Rectangle2D.Double(0, 0, getWidth() - 1,
            ↪getHeight() - 1);
        g2.setPaint(Color.YELLOW);
        g2.fill(rect);
        g2.setPaint(Color.BLUE);
        g2.draw(rect);
    }

    public Dimension getPreferredSize()
    {
        return new Dimension(10, 10);
    }
}

```

Listing 9.16. optionDialog/ButtonPanel.java

```

package optionDialog;

import javax.swing.*;

/**
 * Panel z przełącznikami w ramce z tytułem
 */
public class ButtonPanel extends JPanel
{
    private ButtonGroup group;

    /**
     * Tworzy panel przycisków
     * @param title Tytuł wyświetlny w obramowaniu
     * @param options Tablica etykiet przełączników
     */
    public ButtonPanel(String title, String... options)
    {
        setBorder(BorderFactory.createTitledBorder(BorderFactory.createEtchedBorder(),
            title));
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        group = new ButtonGroup();

        // Utworzenie pojedynczym przełączniku dla każdej opcji
        for (String option : options)
        {
            JRadioButton b = new JRadioButton(option);
            b.setActionCommand(option);
            add(b);
            group.add(b);
            b.setSelected(option == options[0]);
        }
    }

    /**
     * Pobiera aktualnie wybraną opcję
     * @return Zwraca etykietę aktualnie wybranego przełącznika
     */
    public String getSelection()
    {
        return group.getSelection().getActionCommand();
    }
}

```

javax.swing.JOptionPane 1.2

- static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showMessageDialog(Component parent, Object message, String title, int messageType)
- static void showMessageDialog(Component parent, Object message)

- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)
- static void showInternalMessageDialog(Component parent, Object message)

Wyświetla okno dialogowe z komunikatem lub wewnętrzne okno dialogowe (wewnętrzne okno dialogowe w całości zawiera się w swoim oknie nadziednym).

Parametry:	parent	Komponent nadziedny (może być null).
	message	Komunikat, który ma się pojawić w oknie dialogowym (może być łańcuch, ikona, komponent lub tablica tych elementów).
	title	Łańcuch widoczny na pasku tytułu.
	messageType	Jedna z następujących wartości: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE.
	icon	Ikona, która ma się pojawić zamiast standardowej ikony.

- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)
- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)
- static int showConfirmDialog(Component parent, Object message, String title, int optionType)
- static int showConfirmDialog(Component parent, Object message)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)
- static int showInternalConfirmDialog(Component parent, Object message)

Wyświetla okno dialogowe potwierdzenia lub wewnętrzne okno dialogowe potwierdzenia (wewnętrzne okno dialogowe w całości zawiera się w swoim oknie nadziednym). Zwraca wybraną przez użytkownika opcję (OK_OPTION, CANCEL_OPTION, YES_OPTION, NO_OPTION) lub CLOSED_OPTION, jeśli użytkownik zamknie okno.

Parametry:	parent	Komponent nadziedny (może być null).
	message	Komunikat, który ma się pojawić w oknie dialogowym (może być łańcuch, ikona, komponent lub tablica tych elementów).

title	Łańcuch widoczny na pasku tytułu.
messageType	Jedna z następujących wartości: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE.
optionType	Jedna z następujących wartości: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION.
icon	Ikona, która ma się pojawić zamiast standardowej ikony.

- static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)
- static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)

Wyświetla okno dialogowe opcji lub wewnętrzne okno dialogowe opcji (wewnętrzne okno dialogowe w całości zawiera się w swoim oknie nadzędnym). Zwraca indeks wybranej przez użytkownika opcji lub wartość CLOSED_OPTION, jeśli użytkownik anulował okno.

Parametry:	parent	Komponent nadzędny (może być null).
	message	Komunikat, który ma się pojawić w oknie dialogowym (może być łańcuch, ikona, komponent lub tablica tych elementów).
	title	Łańcuch widoczny na pasku tytułu.
	messageType	Jedna z następujących wartości: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE.
	optionType	Jedna z następujących wartości: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION.
	icon	Ikona, która ma się pojawić zamiast standardowej ikony.
	options	Tablica opcji (może zawierać łańcuchy, ikony lub komponenty).
	default	Domyślna opcja, która jest prezentowana użytkownikowi.

- static Object showInputDialog(Component parent, Object message, String title, int message-Type, Icon icon, Object[] values, Object default)
- static String showInputDialog(Component parent, Object message, String title, int message-Type)
- static String showInputDialog(Component parent, Object message)

- static String showInputDialog(Object message)
- static String showInputDialog(Component parent, Object message, Object default) **1.4**
- static String showInputDialog(Object message, Object default) **1.4**
- static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
- static String showInternalInputDialog(Component parent, Object message, String title, int messageType)
- static String showInternalInputDialog(Component parent, Object message)

Wyświetla okno dialogowe przyjmowania danych lub wewnętrzne okno dialogowe przyjmowania danych (wewnętrzne okno dialogowe w całości zawiera się w swoim oknie nadziednym). Zwraca łańcuch znaków wprowadzony przez użytkownika lub wartość `null`, jeśli użytkownik anulował okno.

Parametry:	parent	Komponent nadziedny (może być <code>null</code>).
	message	Komunikat, który ma się pojawić w oknie dialogowym (może być łańcuch, ikona, komponent lub tablica tych elementów).
	title	Łańcuch widoczny na pasku tytułu.
	messageType	Jedna z następujących wartości: <code>ERROR_MESSAGE</code> , <code>INFORMATION_MESSAGE</code> , <code>WARNING_MESSAGE</code> , <code>QUESTION_MESSAGE</code> , <code>PLAIN_MESSAGE</code> .
	icon	Ikona, która ma się pojawić zamiast standardowej ikony.
	values	Tablica wartości, które mają zostać uwzględnione w liście rozwijalnej.
	default	Wartość domyślnie prezentowana użytkownikowi.

9.7.2. Tworzenie okien dialogowych

W poprzednim podrozdziale została zaprezentowana technika tworzenia prostych okien dialogowych za pomocą klasy `JOptionPane`. W tym podrozdziale nauczymy się tworzyć takie okna na własną rękę.

Rysunek 9.38 przedstawia typowe modalne okno dialogowe — zawierające informacje o programie wyświetlane w odpowiedzi na kliknięcie opcji *O programie*.

Implementacja takiego okna polega na rozszerzeniu klasy `JDialog`. Proces ten przebiega w zasadzie tak samo jak w przypadku rozszerzania klasy `JFrame` przy tworzeniu okna głównego aplikacji. Mianowicie:

Rysunek 9.38.

Okno dialogowe typu O programie



- 1 W konstruktorze swojego okna dialogowego wywołaj konstruktor nadklasy `JDialog`.
- 2 Utwórz elementy interfejsu użytkownika okna dialogowego.
3. Utwórz procedury obsługi zdarzeń.
4. Ustaw rozmiar okna.

W konstruktorze nadklasy trzeba podać **ramkę nadziedną**, tytuł okna dialogowego oraz określić **modalność**.

Ramka nadziedna odpowiada za miejsce wyświetlenia okna dialogowego. Wartość `null` powoduje, że okno należy do ukrytej ramki.

Modalność określa, które z pozostałych okien aplikacji będą zablokowane, kiedy wyświetli się to okno. Okno niemodalne nie blokuje innych okien, natomiast okno modalne blokuje wszystkie pozostałe okna (poza swoimi potomkami). Niemodalne okna dialogowe znajdują zastosowanie jako zawsze dostępne zestawy narzędzi. Modalne okno dialogowe można zastosować, w przypadku gdy do kontynuacji działania programu konieczne jest dostarczenie informacji przez użytkownika.



W Java SE 6 dostępne są dwa rodzaje modalności. Okno dialogowe z modalnością dokumentu (ang. *document-modal dialog*) blokuje wszystkie okna należące do tego samego dokumentu. Jako dokument w tym przypadku rozumie się hierarchię okien mających wspólnego przodka, który nie ma właściciela w postaci okna dialogowego. W ten sposób rozwiązano problem z systemami pomocy. Wcześniej użytkownik nie mógł korzystać z pomocy, jeśli było wyświetcone okno modalne. Okno dialogowe z modalnością zestawu narzędzi (ang. *toolkit-modal dialog*) blokuje wszystkie okna należące do tego samego zestawu narzędzi. Zestaw narzędzi to program w Javie, który uruchamia kilka aplikacji, np. silnik appletów w przeglądarce. Więcej informacji na ten temat można znaleźć na stronie www.oracle.com/technetwork/articles/javase/modality-137604.html.

Poniżej znajduje się kod źródłowy tego okna dialogowego:

```
public AboutDialog extends JDialog
{
    public AboutDialog(JFrame owner)
    {
        super(owner, "Test okna O programie", true);
        add(new JLabel(
            "<html><h1><i>Java. Podstawy</i></h1><hr>Cay Horstmann, Gary Cornell</html>"),
            BorderLayout.CENTER);

        JPanel panel = new JPanel();
        JButton ok = new JButton("OK");
    }
}
```

```
ok.addActionListener(new
    ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        setVisible(false);
    }
});
panel.add(ok);
add(panel, BorderLayout.SOUTH);

setSize(250, 150);
}
}
```

Jak widać, konstruktor tworzy elementy interfejsu użytkownika (w tym przypadku etykiety i przycisk) oraz zawiera procedurę obsługi przycisku i ustawia rozmiar okna dialogowego.

Aby wyświetlić okno dialogowe, należy utworzyć nowy obiekt okna dialogowego, a następnie go uwidoczyć:

```
JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);
```

We fragmencie kodu zaprezentowanym poniżej okno dialogowe tworzone jest tylko jeden raz, po czym można go używać za każdym razem, gdy użytkownik kliknie pozycję *O programie*.

```
if (dialog == null) //pierwszy raz
dialog = new AboutDialog(this);
dialog.setVisible(true);
```

Kliknięcie przycisku *OK* powinno zamykać okno. Działanie to zostało zdefiniowane w procedurze obsługi przycisku *OK*:

```
ok.addActionListener(new
    ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        setVisible(false);
    }
});
```

Okno zostanie ukryte także w wyniku kliknięcia przycisku *Zamknij*. Podobnie jak w przypadku ramki JFrame, działanie to można zmienić za pomocą metody setDefaultCloseOperation.

Listing 9.17 przedstawia kod programu testującego omawiane okno dialogowe, a listing 9.18 — klasę AboutDialog.

Listing 9.17. dialog/DialogFrame.java

```
package dialog;

import java.awt.event.*;
import javax.swing.*;
```

```

/**
 * Ramka z menu, którego akcja Plik/O programie wyświetla okno dialogowe
 */
public class DialogFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    private AboutDialog dialog;

    public DialogFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // Tworzenie menu Plik

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu fileMenu = new JMenu("Plik");
        menuBar.add(fileMenu);

        // Tworzenie elementów O programie i Zamknij

        // Element O programie wyświetla okno dialogowe O programie

        JMenuItem aboutItem = new JMenuItem("O programie");
        aboutItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                if (dialog == null) //pierwszy raz
                    dialog = new AboutDialog(DialogFrame.this);
                dialog.setVisible(true); //wyskakujące okno dialogowe
            }
        });
        fileMenu.add(aboutItem);

        // Element Zamknij powoduje zamknięcie programu

        JMenuItem exitItem = new JMenuItem("Zamknij");
        exitItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });
        fileMenu.add(exitItem);
    }
}

```

Listing 9.18. dialog/AboutDialog.java

```

package dialog;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```


/*
 * Przykładowe modalne okno dialogowe wyświetlające komunikat i oczekujące na kliknięcie przycisku Ok
 */
public class AboutDialog extends JDialog
{
    public AboutDialog(JFrame owner)
    {
        super(owner, "Test okna 0 programie", true);

        // Dodanie etykiety HTML

        add(
            new JLabel(
                "<html><h1><i>Core Java</i></h1><hr> By Cay Horstmann and Gary
                ↳Cornell </html>"),
            BorderLayout.CENTER);

        // Przycisk Ok zamkna okno

        JButton ok = new JButton("Ok");
        ok.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                setVisible(false);
            }
        });

        // Dodanie przycisku Ok przy krawędzi południowej

        JPanel panel = new JPanel();
        panel.add(ok);
        add(panel, BorderLayout.SOUTH);

        pack();
    }
}


```

javax.swing.JDialog 1.2

■ `public JDialog(Frame parent, String title, boolean modal)`

Tworzy okno dialogowe. Okno nie jest widoczne, dopóki nie zostanie celowo uwidocznione.

Parametry:	<code>parent</code>	Ramka zawierająca okno
	<code>title</code>	Tytuł okna
	<code>modal</code>	Wartość <code>true</code> , jeśli okno ma być modalne (okno modalne blokuje dostęp do pozostałych okien)

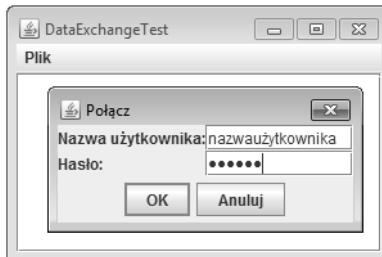
9.7.3. Wymiana danych

Najczęstszym powodem tworzenia okien dialogowych jest chęć uzyskania danych od użytkownika. Wiemy już, że utworzenie obiektu okna dialogowego jest bardzo łatwe (należy podać jego dane początkowe i uwidocznić je za pomocą metody `setVisible(true)`). Teraz zajmiemy się przesyłaniem danych z i do okna.

Przyjrzyjmy się oknu widocznemu na rysunku 9.39, które może służyć do pobierania nazwy i hasła użytkownika pragnącego połączyć się z jakąś usługą internetową.

Rysunek 9.39.

Okno dialogowe z polem hasła



Okno dialogowe powinno posiadać metody ustawiające dane początkowe. Na przykład klasa `PasswordChooser` omawianego programu zawiera metodę o nazwie `setUser`, która wstawia domyślne wartości do pól tekstowych:

```
public void setUser(User u)
{
    username.setText(u.getName());
}
```

Po ustawieniu wartości domyślnych (jeśli jest to konieczne) należy wywołać metodę `setVisible(true)` w celu uwidocznienia okna.

Użytkownik może podać wymagane informacje i kliknąć przycisk *Ok* lub *Anuluj*. Procedury obsługujące zdarzenia każdego z tych przycisków wywołują metodę `setVisible(false)`, która kończy wywołanie metody `setVisible(true)`. Użytkownik może też zamknąć okno. Jeśli nie zdefiniowano żadnego słuchacza zdarzeń okna, zastosowana zostanie standardowa procedura zamkająca polegającą na ukryciu okna, w wyniku którego następuje przerwanie działania metody `setVisible(true)`.

Ważne jest to, że blokada tworzona przez metodę `setVisible(true)` działa do chwili zamknięcia okna. To ułatwia tworzenie modalnych okien dialogowych.

Musimy sprawdzić, czy użytkownik kliknął przycisk *Ok* czy *Anuluj*. W kodzie znacznik `ok` został ustawiony na wartość `false` przed pokazaniem okna. Wartość tę na `true` może zmienić tylko procedura obsługi przycisku *Ok*. W przypadku jego naciśnięcia można pobrać dane wpisane w oknie przez użytkownika.

Prezentowany przykładowy program posiada dodatkowe usprawnienie. Konstruując obiekt typu `JDialog`, trzeba określić ramkę nadziedną. Często jednak zdarza się, że jedno okno dialogowe



Przenoszenie danych z niemodalnego okna dialogowego nie jest takie proste. Wyświetlenie takiego okna nie powoduje założenia blokady przez metodę `setVisible(true)` i program kontynuuje działanie. Kiedy użytkownik kliknie przycisk `Ok` w oknie niemodalnym, musi ono wysłać zdarzenie do jakiegoś obiektu nasłuchującego w programie.

może być wyświetlane w kilku różnych ramkach. Lepiej jest wybrać ramkę nadziedną, **kiedy okno dialogowe jest gotowe do wyświetlenia** niż po utworzeniu obiektu typu `PassswordChooser`.

Sztuka polega na tym, aby sprawić, że klasa `PasswordChooser` będzie rozszerzała klasę `JPanel`, a nie `JDialog`. W tym celu obiekt `JDialog` należy utworzyć w locie, w metodzie `showDialog`:

```
public boolean showDialog(Frame owner, String title)
{
    ok = false;

    if (dialog == null || dialog.getOwner() != owner)
    {
        dialog = new JDialog(owner, true);
        dialog.add(this);
        dialog.pack();
    }

    dialog.setTitle(title);
    dialog.setVisible(true);
    return ok;
}
```

Warto zauważyć, że bezpiecznym ustawieniem dla parametru `owner` jest wartość `null`.

Można to zrobić jeszcze lepiej. Czasami ramka nadziedna nie jest od razu gotowa. Można ją z łatwością uzyskać z dowolnego komponentu parent, np.:

```
Frame owner;
if (parent instanceof Frame)
    owner = (Frame) parent;
else
    owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
```

Usprawnienie to zastosowaliśmy w naszym przykładowym programie. Mechanizm ten wykorzystuje także klasa `JOptionPane`.

Wiele okien dialogowych posiada **przycisk domyślny** (ang. *default button*), który jest automatycznie naciskany, kiedy użytkownik naciśnie klawisz wyzwolenia (ang. *trigger key*) — w większości stylów jest to klawisz *Enter*. Przycisk domyślny jest w jakiś sposób wyróżniony, zazwyczaj grubym obramowaniem.

Przycisk domyślny ustawia się w **panelu głównym okna dialogowego** (ang. *root pane*):

```
dialog.getRootPane().setDefaultButton(okButton);
```

Stosując się do naszych zaleceń dotyczących umieszczenia okna dialogowego w panelu, należy pamiętać, że przycisk domyślny musi być ustawiony po zapakowaniu panelu w oknie. Sam panel nie posiada panelu głównego.

Listing 9.19 przedstawia kod klasy ramowej programu ilustrującego przepływ danych w oknie dialogowym. Klasa tego okna jest pokazana na listingu 9.20.

Listing 9.19. dataExchange/DataExchangeFrame.java

```
package dataExchange;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka z menu, którego akcja Plik/Polacz wyświetla okno dialogowe z polem hasła
 */
public class DataExchangeFrame extends JFrame
{
    public static final int TEXT_ROWS = 20;
    public static final int TEXT_COLUMNS = 40;
    private PasswordChooser dialog = null;
    private JTextArea textArea;

    public DataExchangeFrame()
    {
        // Tworzenie menu Plik

        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        JMenu fileMenu = new JMenu("Plik");
        mbar.add(fileMenu);

        // Tworzenie elementów menu Polacz i Zamknij

        JMenuItem connectItem = new JMenuItem("Połącz");
        connectItem.addActionListener(new ConnectAction());
        fileMenu.add(connectItem);

        // Opcja Zamknij zamkna program

        JMenuItem exitItem = new JMenuItem("Zamknij");
        exitItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });
        fileMenu.add(exitItem);

        textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
        add(new JScrollPane(textArea), BorderLayout.CENTER);
        pack();
    }
}
```

```
/*
 * Akcja Connect wyświetla okno dialogowe z polem hasła
 */

private class ConnectAction implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // Jeśli jest to pierwszy raz, tworzy okno dialogowe

        if (dialog == null) dialog = new PasswordChooser();

        // Ustawianie wartości domyślnych
        dialog.setUser(new User("Twoja nazwa", null));

        // Wyświetlenie okna dialogowego
        if (dialog.showDialog(DataExchangeFrame.this, "Połącz"))
        {
            // Pobranie danych użytkownika w przypadku zatwierdzenia
            User u = dialog.getUser();
            textArea.append("nazwa użytkownika = " + u.getName() + ", hasło = "
                + (new String(u.getPassword())) + "\n");
        }
    }
}
```

Listing 9.20. dataExchange/PasswordChooser.java

```
package dataExchange;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Elementy służące do podania hasła, które widać w oknie dialogowym
 */
public class PasswordChooser extends JPanel
{
    private JTextField username;
    private JPasswordField password;
    private JButton okButton;
    private boolean ok;
    private JDialog dialog;

    public PasswordChooser()
    {
        setLayout(new BorderLayout());

        // Utworzenie panelu z polami nazwy użytkownika i hasła

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(2, 2));
        panel.add(new JLabel("Nazwa użytkownika:"));
        panel.add(username = new JTextField(""));
        panel.add(new JLabel("Hasło:"));

```

```

panel.add(password = new JPasswordField(""));
add(panel, BorderLayout.CENTER);

// Utworzenie przycisków OK i Anuluj, które zamkują okno dialogowe

okButton = new JButton("Ok");
okButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        ok = true;
        dialog.setVisible(false);
    }
});

JButton cancelButton = new JButton("Anuluj");
cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        dialog.setVisible(false);
    }
});

// Dodawanie przycisków w pobliżu południowej krawędzi

JPanel buttonPanel = new JPanel();
buttonPanel.add(okButton);
buttonPanel.add(cancelButton);
add(buttonPanel, BorderLayout.SOUTH);
}

/*
* Ustawia wartości domyślne okna dialogowego
* @param u domyślne informacje użytkownika
*/
public void setUser(User u)
{
    username.setText(u.getName());
}

/*
* Pobiera dane podane w oknie dialogowym
* @return a obiekt typu User, którego stan reprezentuje dane wprowadzone w oknie dialogowym
*/
public User getUser()
{
    return new User(username.getText(), password.getPassword());
}

/*
* Wyświetla panel z elementami przyjmującymi dane od użytkownika w oknie dialogowym
* @param parent komponent w ramce nadzędnej lub wartość null
* @param title tytuł okna dialogowego
*/
public boolean showDialog(Component parent, String title)
{
    ok = false;
}

```

```
// Lokalizacja ramki nadzędnej

Frame owner = null;
if (parent instanceof Frame) owner = (Frame) parent;
else owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);

// Jeśli jest to pierwszy raz lub zmienił się użytkownik, utworzenie nowego okna dialogowego

if (dialog == null || dialog.getOwner() != owner)
{
    dialog = new JDialog(owner, true);
    dialog.add(this);
    dialog.getRootPane().setDefaultButton(okButton);
    dialog.pack();
}

// Ustawienie tytułu i wyświetlenie okna dialogowego

dialog.setTitle(title);
dialog.setVisible(true);
return ok;
}
```

javax.swing.SwingUtilities 1.2

- `Container getAncestorOfClass(Class c, Component comp)`

Zwraca najgłębiej zagnieżdżony kontener nadzędny danego komponentu, który należy do danej klasy lub jednej z jej podklas.

javax.swing.JComponent 1.2

- `JRootPane getRootPane()`

Pobiera panel główny (ang. *root pane*) zawierający dany komponent lub wartość `null`, jeśli komponent ten nie posiada przodka z panelem głównym.

javax.swing.JRootPane 1.2

- `void setDefaultButton(JButton button)`

Ustawia domyślny przycisk dla panelu głównego. Aby dezaktywować ten przycisk, należy wywołać tę metodę z wartością `null`.

javax.swing.JButton 1.2

- `boolean isDefaultButton()`

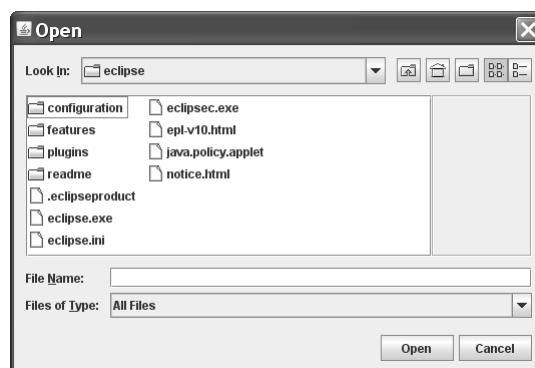
Zwraca wartość `true`, jeśli dany przycisk jest domyślny w swoim panelu głównym.

9.7.4. Okna dialogowe wyboru plików

Wiele aplikacji musi posiadać opcje otwierania i zapisywania plików. Napisanie dobrego okna dialogowego wyświetlającego pliki i katalogi oraz umożliwiającego nawigację po systemie plików nie jest łatwe. Nie chcielibyśmy też ponownie wynajdywać koła. Na szczęście w pakuie Swing dostępna jest klasa `JFileChooser` pozwalająca na tworzenie okien dialogowych wyglądających podobnie do tych, które można spotkać w większości aplikacji rodzinnych. Okna dialogowe `JFileChooser` są zawsze modalne. Należy zauważyć, że klasa `JFileChooser` nie jest podklassą klasy `JDialog`. Zamiast metody `setVisible(true)` do wyświetlania okien dialogowych wyboru pliku służy metoda `showOpenDialog`, a do okien zapisu pliku metoda `showSaveDialog`. Przycisk automatycznie zyskuje etykietę *Open* lub *Save*. Korzystając z metody `showDialog`, można określić własną etykietę. Rysunek 9.40 przedstawia przykładowe okno dialogowe wyboru pliku.

Rysunek 9.40.

Okno dialogowe wyboru pliku



Poniższe punkty opisują procedurę tworzenia okna dialogowego wyboru pliku i odzyskiwania tego, co użytkownik wybrał w polu wyboru.

- 1 Utwórz obiekt typu `JFileChooser`. W przeciwnieństwie do konstruktora klasy `JDialog` w tym przypadku nie trzeba podawać komponentu nadziedzkiego. Dzięki temu okno wyboru pliku można wykorzystać w kilku ramkach.

Na przykład:

```
JFileChooser chooser = new JFileChooser();
```



Wielokrotne użycie jednego obiektu `JFileChooser` jest dobrym pomysłem, ponieważ konstruktor `JFileChooser` może działać powoli, zwłaszcza w systemie Windows, jeśli użytkownik posiada wiele zmapowanych dysków sieciowych.

- 2 Ustaw katalog za pomocą metody `setCurrentDirectory`.

Na przykład poniższa procedura robi użytku z aktualnego katalogu roboczego:

```
chooser.setCurrentDirectory(new File("."));
```

W takim przypadku konieczne jest dostarczenie obiektu typu `File`. Obiekty tego typu zostały szczegółowo omówione w rozdziale 12. Na razie wystarczy nam wiedzieć, że konstruktor `File(String filename)` zamienia plik lub katalog o określonej nazwie na obiekt typu `File`.

3. Jeśli istnieje duże prawdopodobieństwo, że użytkownik wybierze plik o określonej nazwie, nazwę tę należy podać za pomocą metody `setSelectedFile`:

```
chooser.setSelectedFile(new File(filename));
```

4. Aby umożliwić wybór kilku plików w oknie dialogowym, należy użyć metody `setMultiSelectionEnabled`. Jest to oczywiście niezbyt często stosowana opcja.

```
chooser.setMultiSelectionEnabled(true);
```

5. Aby w oknie dialogowym widoczne były wyłącznie pliki określonego typu (na przykład z rozszerzeniem `.gif`), należy utworzyć **filtr plików**. Filtry plików zostały opisane w dalszej części tego rozdziału.

6. Przy standardowych ustawieniach użytkownik może wybierać tylko pliki. Aby umożliwić wybór katalogów, należy użyć metody `setFileSelectionMode`. Jej parametrem powinna być jedna z następujących wartości `JFileChooser.FILES_ONLY` (domyślna), `JFileChooser.DIRECTORIES_ONLY` lub `JFileChooser.FILES_AND_DIRECTORIES`.

7. Do uwidocznienia okna służą metody `showOpenDialog` i `showSaveDialog`. W ich wywołaniach konieczne jest podanie komponentu nadzawanego:

```
int result = chooser.showOpenDialog(parent);
```

lub

```
int result = chooser.showSaveDialog(parent);
```

Jedyna różnica pomiędzy tymi metodami dotyczy przycisku zatwierdzającego, który użytkownik naciska w celu zakończenia operacji. Istnieje też metoda o nazwie `showDialog`, przyjmująca tekst, który ma być wyświetlony na przycisku:

```
int result = chooser.showDialog(parent, "Zaznacz");
```

Wywołania tego typu zwracają wartość tylko wówczas, gdy użytkownik zatwierdzi, anuluje lub zamknie okno dialogowe. Możliwe wartości zwrotne to: `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION` i `JFileChooser.ERROR_OPTION`.

8. Wybrany plik lub pliki pobiera się za pomocą metod `getSelectedFile()` lub `getSelectedFiles`. Zwracają one jeden obiekt typu `File` lub tablicę takich obiektów. Do sprawdzenia nazwy obiektu plikowego można użyć jego metody `getPath`. Na przykład:

```
String filename = chooser.getSelectedFile().getPath();
```

Większość powyższych czynności jest łatwa. Najwięcej problemów z oknem dialogowym wyboru pliku można mieć przy określaniu podzbioru plików, z których użytkownik ma wybierać. Założymy na przykład, że wybierane mogą być obrazy typu GIF. W takiej sytuacji w oknie wyboru powinny być wyświetlane tylko pliki z rozszerzeniem `.gif`. Dodatkowo

powinna być dostępna jakaś informacja, że wyświetlane pliki należą do określonej kategorii, np. *Pliki GIF*. Sytuacja może być jednak bardziej skomplikowana. Jeśli użytkownik może wybierać pliki JPEG, można spotkać dwa różne rozszerzenia — *.jpg* lub *.jpeg*. Zamiast opracowywać technikę kodowania takich złożonych ograniczeń, projektanci okna wyboru pliku zdecydowali się na utworzenie bardziej eleganckiego mechanizmu. Pliki do wyświetlenia są selekcjonowane za pomocą obiektu rozszerzającego abstrakcyjną klasę `javax.swing.JFileChooser`. Przesyłany jest każdy plik, ale wyświetlane zostają tylko te, które akceptuje filtr.

W chwili pisania tej książki dostępne były dwie takie podklasy: domyślny filtr akceptujący wszystkie pliki i filtr akceptujący tylko pliki z określonym rozszerzeniem. Ponadto z łatwością można napisać własny filtr. Wystarczy zaimplementować w nim dwie abstrakcyjne metody nadklasy `FileFilter`:

```
public boolean accept(File f);
public String getDescription();
```

Pierwsza z tych metod sprawdza, czy plik powinien zostać zaakceptowany. Druga zwraca opis typu pliku, który może być wyświetlony w oknie wyboru plików.



W pakiecie `java.io` znajduje się niezwiązany z omawianą klasą interfejs `FileFilter`, który udostępnia jedną metodę `boolean accept(File f)`. Jest ona używana w metodzie `listFiles` klasy `File` do tworzenia listy plików znajdujących się w katalogu. Nie wiadomo, dlaczego projektanci biblioteki Swing nie rozszerzyli tego interfejsu — możliwe, że biblioteka klas Java stała się tak obszerna, że nawet sami programiści z Sun nie potrafią ogarnąć wszystkich standardowych klas i interfejsów.

W przypadku importu pakietów `java.io` i `javax.swing.filechooser` jednocześnie konieczne jest rozwiązywanie konfliktu nazw. Najprostsze rozwiązanie polega na zainportowaniu samej klasy `javax.swing.filechooser.FileFilter` zamiast wszystkich klas tego pakietu — `javax.swing.filechooser.*`.

Po utworzeniu obiektu filtru plików należy go zainstalować w obiekcie okna wyboru pliku (ang. *file chooser*):

```
chooser.setFileFilter(new FileNameExtensionFilter("Pliki obrazów", "gif", "jpg"));
```

W oknie wyboru plików można zainstalować kilka filtrów. Służą do tego poniższe instrukcje:

```
chooser.addChoosableFileFilter(filter1);
chooser.addChoosableFileFilter(filter2);
```

Użytkownik wybiera filtr z listy rozwijalnej znajdującej się na dole okna. Domyślnie filtr *All files* jest zawsze dostępny. Jest to bardzo dobre rozwiązanie, na wypadek gdyby użytkownik chciał wybrać plik o niestandardowym rozszerzeniu. Aby usunąć filtr *All files*, należy użyć poniższej instrukcji:

```
chooser.setAcceptAllFileFilterUsed(false)
```

Okno wyboru plików można ozdobić specjalnymi ikonami i opisami plików. W tym celu należy dostarczyć obiekt klasy rozszerzającej klasę `FileView` z pakietu `javax.swing.filechooser`.



W przypadku wielokrotnego użycia jednego okna wyboru plików do ładowania i zapisywania plików różnego rodzaju należy zastosować poniższą instrukcję

```
chooser.resetChoosableFilters()
```

Czyści ona wszystkie stare filtry przed dodaniem nowych.

Jest to z pewnością zaawansowana technika. W typowych sytuacjach nie ma potrzeby tworzenia widoku plików, ponieważ odpowiada za niego styl. Aby jednak specjalne typy plików miały różne ikony, można utworzyć własny widok plików. Wymaga to rozszerzenia klasy `FileView` i implementacji pięciu metod:

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Następnie do instalacji widoku plików w oknie wyboru plików używamy metody `setFileView`.

Okno wyboru plików wywołuje zaimplementowane metody dla każdego pliku lub katalogu, który chce wyświetlić. Jeśli metoda zwróci wartość `null` dla ikony, nazwy lub opisu, okno wyboru plików odwołuje się do widoku domyślnego w zastosowanym stylu. Zaletą takiego zachowania jest to, że programista musi się zająć tylko tymi typami plików, które go interesują.

Okno wyboru plików podejmuje decyzję, czy otworzyć katalog kliknięty przez użytkownika za pomocą metody `isTraversable`. Pamiętajmy, że metoda ta zwraca obiekt typu `Boolean`, a nie wartość typu `boolean` (logiczna)! Wydaje się to dziwne, ale jest bardzo wygodne — jeśli nie chcemy zmienić domyślnego widoku, wystarczy zwrócić wartość `null`. Wtedy okno wyboru plików zastosuje domyślny widok. Innymi słowy, ta metoda zwraca obiekt typu `Boolean`, który umożliwia wybór jednej z trzech opcji: prawda (`Boolean.TRUE`), fałsz (`Boolean.FALSE`) i bez różnicy (`null`).

Poniższy przykładowy program zawiera prostą klasę widoku plików. Wyświetla ona określoną ikonę, kiedy jakiś plik pasuje do filtra. W tym przypadku wyświetlana jest ikona palety dla wszystkich plików obrazów.

```
class FileIconView extends FileView
{
    public FileIconView(FileFilter aFilter, Icon anIcon)
    {
        filter = aFilter;
        icon = anIcon;
    }

    public Icon getIcon(File f)
    {
        if (!f.isDirectory() && filter.accept(f))
            return icon;
        else return null;
    }
    private FileFilter filter;
    private Icon icon;
}
```

Ten widok plików instalujemy w oknie wyboru plików za pomocą metody `setFileView`:

```
chooser.setFileView(new FileIconView(filter,
    new ImageIcon("palette.gif")));
```

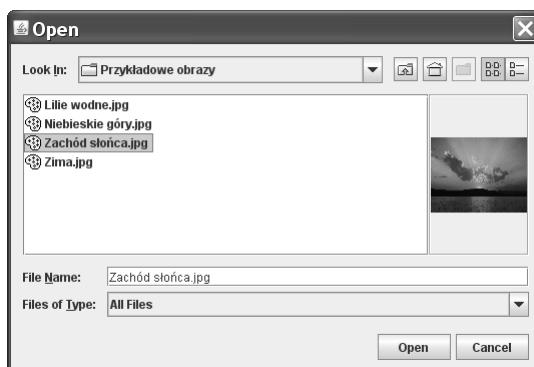
Dzięki temu obok wszystkich plików zaakceptowanych przez filtr będzie widoczna ikona palety, a pozostałe będą wyświetlane w standardowy sposób. Oczywiście używamy tego samego filtra, który utworzyliśmy w oknie wyboru plików.



Bardziej użyteczna przykładowa klasa o nazwie `ExampleFileView` znajduje się w katalogu JDK `demo/jfc/FileChooserDemo`. Pozwala ona na wiązanie ikon i opisów z dowolnymi rozszerzeniami.

W końcu okno dialogowe można wyposażyć w dodatkowe **akcesorium** (ang. *accessory component*). Na przykład rysunek 9.41 przedstawia akcesorium podglądu umieszczone obok listy plików. Wyświetla ono miniaturę wybranego pliku.

Rysunek 9.41.
Okno dialogowe
wyboru pliku
z akcesorium
podglądu



Akcesorium może być każdy komponent Swing. W tym przypadku rozszerzyliśmy klasę `JLabel` i ustawiliśmy jej ikonę na przeskalowaną kopię obrazu graficznego:

```
class ImagePreviewer extends JLabel
{
    public ImagePreviewer(JFileChooser chooser)
    {
        setPreferredSize(new Dimension(100, 100));
        setBorder(BorderFactory.createEtchedBorder());
    }

    public void loadImage(File f)
    {
        ImageIcon icon = new ImageIcon(f.getPath());
        if(icon.getIconWidth() > getWidth())
            icon = new ImageIcon(icon.getImage().getScaledInstance(
                getWidth(), -1, Image.SCALE_DEFAULT));
        setIcon(icon);
        repaint();
    }
}
```

Jest tylko jedna trudność. Kiedy użytkownik kliknie inny plik, podgląd powinien zostać zaktualizowany. Okno wyboru plików wykorzystuje mechanizm JavaBeans do powiadamiania zainteresowanych słuchaczy o zmianach swoich własności. Zaznaczony plik jest własnością, którą można monitorować za pomocą obiektu `PropertyChangeListener`. Bardziej szczegółowo mechanizm ten opisujemy w rozdziale 8. drugiego tomu. Poniżej znajduje się kod przechwytyujący omawiane powiadomienia:

```
chooser.addPropertyChangeListener(new
    PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent event)
    {
        if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
        {
            File newFile = (File) event.getNewValue()
            // Aktualizacja akcesorium
            .
        }
    }
});
```

W przykładowym programie kod znajduje się w konstruktorze `ImagePreviewer`.

Na listingach 9.21 – 9.23 została przedstawiona zmodyfikowana wersja programu *ImageViewer* z rozdziału 2. Dodano w nim niestandardowy widok plików i akcesorium podglądu.

Listing 9.21. `fileChooser/ImageViewerFrame.java`

```
package fileChooser;

import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * Ramka z menu zawierającym opcję Otwórz i obszarem do prezentacji otwartych obrazów
 */
public class ImageViewerFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 400;
    private JLabel label;
    private JFileChooser chooser;

    public ImageViewerFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // Pasek menu
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("Plik");
        menuBar.add(menu);
```

```

JMenuItem openItem = new JMenuItem("Otwórz");
menu.add(openItem);
openItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        chooser.setCurrentDirectory(new File("."));
        // Okno wyboru plików
        int result = chooser.showOpenDialog(ImageViewerFrame.this);

        // Jeśli plik obrazu zostanie zaakceptowany, ustaw go jako ikonę etykiety
        if (result == JFileChooser.APPROVE_OPTION)
        {
            String name = chooser.getSelectedFile().getPath();
            label.setIcon(new ImageIcon(name));
            pack();
        }
    }
});

JMenuItem exitItem = new JMenuItem("Zamknij");
menu.add(exitItem);
exitItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        System.exit(0);
    }
});

// Etykieta do wyświetlania obrazów
label = new JLabel();
add(label);

// Utworzenie akcesorium wyboru plików
chooser = new JFileChooser();

// Akceptuje wszystkie pliki obrazów z rozszerzeniem .jpg, .jpeg, .gif
/*
final ExtensionFileFilter filter = new ExtensionFileFilter();
filter.addExtension("jpg");
filter.addExtension("jpeg");
filter.addExtension("gif");
filter.setDescription("Pliki obrazów");
*/
FileNameExtensionFilter filter = new FileNameExtensionFilter("Pliki obrazów",
    "jpg", "jpeg", "gif");
chooser.setFileFilter(filter);

chooser.setAccessory(new ImagePreviewer(chooser));

chooser.setFileView(new FileIconView(filter, new ImageIcon("palette.gif")));
}
}

```

Listing 9.22. fileChooser/ImagePreviewer.java

```
package fileChooser;

import java.awt.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

/**
 * Akcesoriuム wywietlajace podglad obrazow
 */
public class ImagePreviewer extends JLabel
{
    /**
     * Tworzy obiekt ImagePreviewer
     * @param chooser okno wyboru plikow, którego własność zmienia się, powoduje zmianę obrazu
     * w tym podgladzie
     */
    public ImagePreviewer(JFileChooser chooser)
    {
        setPreferredSize(new Dimension(100, 100));
        setBorder(BorderFactory.createEtchedBorder());

        chooser.addPropertyChangeListener(new PropertyChangeListener()
        {
            public void propertyChange(PropertyChangeEvent event)
            {
                if (event.getPropertyName() ==
                    JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
                {
                    // Uzytkownik wybrał inny plik
                    File f = (File) event.getNewValue();
                    if (f == null)
                    {
                        setIcon(null);
                        return;
                    }

                    // Wczytanie obrazu jako ikony
                    ImageIcon icon = new ImageIcon(f.getPath());

                    // Skalowanie obrazu, jeśli jest zbyt duży na ikonę
                    if (icon.getIconWidth() > getWidth()) icon = new
                    ImageIcon(icon.getImage()
                        .getScaledInstance(getWidth(), -1, Image.SCALE_DEFAULT));

                    setIcon(icon);
                }
            }
        });
    }
}
```

Listing 9.23. fileChooser/FileIconView.java

```

package fileChooser;

import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.*;
import javax.swing.filechooser.FileFilter;

/**
 * Widok plików wyświetlający ikonę obok wszystkich plików zaakceptowanych przez filtr
 */
public class FileIconView extends FileView
{
    private FileFilter filter;
    private Icon icon;

    /**
     * Tworzy obiekt FileIconView
     * @param aFilter filtr plików — wszystkie pliki zaakceptowane przez ten filtr będą miały ikonę
     * @param anIcon — ikona wyświetlana obok wszystkich zaakceptowanych plików
     */
    public FileIconView(FileFilter aFilter, Icon anIcon)
    {
        filter = aFilter;
        icon = anIcon;
    }

    public Icon getIcon(File f)
    {
        if (!f.isDirectory() && filter.accept(f)) return icon;
        else return null;
    }
}

```

javax.swing.JFileChooser 1.2

■ **JFileChooser()**

Tworzy okno dialogowe wyboru plików, którego można używać w wielu ramkach.

■ **void setCurrentDirectory(File dir)**

Ustawia początkowy katalog wyświetlany w oknie dialogowym wyboru plików.

■ **void setSelectedFile(File file)**■ **void setSelectedFiles(File[] file)**

Ustawia domyślny plik w oknie dialogowym wyboru plików.

■ **void setMultiSelectionEnabled(boolean b)**

Ustawia lub wyłącza tryb wyboru wielu plików.

■ **void setFileSelectionMode(int mode)**

Pozwala na wybór tylko plików (domyślnie), tylko katalogów lub jednych i drugich. Parametr mode może mieć jedną z następujących wartości

JFileChooser.FILES_ONLY, JFileChooser.DIRECTORIES_ONLY
lub FileChooser.FILES_AND_DIRECTORIES.

- int showOpenDialog(Component parent)
- int showSaveDialog(Component parent)
- int showDialog(Component parent, String approveButtonText)

Wizualizuje okno dialogowe, w którym przycisk zatwierdzający ma etykietę *Open* bądź *Save* lub w postaci łańcucha approveButtonText. Zwraca wartość APPROVE_OPTION, CANCEL_OPTION (jeśli użytkownik kliknął przycisk anulowania lub zamknął okno) lub ERROR_OPTION (jeśli wystąpił błąd).

- File getFile()
 - File[] getSelectedFiles()
- Pobiera plik lub pliki wybrane przez użytkownika (lub zwraca wartość null, jeśli użytkownik nie wybrał żadnego pliku).
- void setFileFilter(FileFilter filter)

Ustawia maskę pliku dla okna dialogowego wyboru plików. Wyświetlone zostaną wszystkie pliki, dla których filter.accept zwróci wartość true. Ponadto dodaje filtr do listy dostępnych filtrów.

- void addChoosableFileFilter(FileFilter filter)
- Dodaje filtr plików do listy dostępnych filtrów.
- void setAcceptAllFileFilterUsed(boolean b)
- Dodaje lub wyłącza filtr *All files* w liście rozwijalnej.
- void resetChoosableFileFilters()
- Czyści listę dostępnych filtrów plików. Pozostaje tylko filtr *All files*, jeśli nie zostanie jawnie wyłączony.
- void setFileView(FileView view)
- Ustawia widok plików dostarczający informacji o plikach wyświetlanych przez okno wyboru.
- void setAccessory(JComponent component)
- Ustawia komponent akcesorium.

javax.swing.filechooser.FileFilter 1.2

- boolean accept(File f)

Zwraca wartość true, jeśli plik ma być wyświetlany.

- String getDescription()

Zwraca opis filtru plików, na przykład *Pliki obrazów (*.gif, *.jpeg)*.

javax.swing.filechooser.FileNameExtensionFilter 6

- `FileNameExtensionFilter(String description, String ... extensions)`

Tworzy filtr plików z podanymi opisami, akceptującymi wszystkie katalogi i pliki, których nazwy kończą się kropką i podanymi łańcuchami określającymi rozszerzenie.

javax.swing.filechooser.FileView 1.2

- `String getName(File f)`

Zwraca nazwę pliku `f` lub wartość `null`. W normalnych warunkach metoda ta zazwyczaj zwraca `f.getName()`.

- `String getDescription(File f)`

Zwraca możliwy do odczytu opis pliku `f` lub wartość `null`. Jeśli na przykład `f` jest dokumentem HTML, metoda ta może zwrócić jego tytuł.

- `String getTypeDescription(File f)`

Zwraca możliwy do odczytu opis typu pliku `f` lub wartość `null`. Jeśli na przykład `f` jest dokumentem HTML, metoda ta może zwrócić łańcuch `Hypertext document`.

- `Icon getIcon(File f)`

Zwraca ikonę pliku `f` lub wartość `null`. Jeśli na przykład `f` jest plikiem JPEG, metoda ta może zwrócić miniaturę.

- `Boolean isTraversable(File f)`

Zwraca wartość `Boolean.TRUE`, jeśli użytkownik może otworzyć katalog `f`. Metoda ta może zwrócić wartość `false`, jeśli katalog jest z założenia dokumentem złożonym. Podobnie jak wszystkie metody klasy `FileView`, także ta metoda może zwrócić wartość `null`, tym samym odsyłając okno wyboru plików do widoku domyślnego.

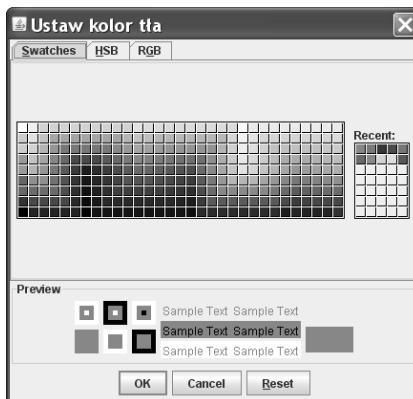
9.7.5. Okna dialogowe wyboru kolorów

Jak przekonaliśmy się w poprzednim podrozdziale, wysokiej jakości okno dialogowe wyboru plików jest skomplikowanym komponentem interfejsu użytkownika, którego zdecydowanie lepiej nie implementować własnoręcznie. Wiele zestawów narzędzi interfejsu użytkownika udostępnia inne często spotykane rodzaje okien dialogowych: wyboru daty i godziny, wartości walutowych, czcionek, kolorów itd. Korzyści są podwójne. Programista może wykorzystać gotową wysoką jakością implementację, zamiast opracowywać własną, a użytkownicy zyskują jednolity styl aplikacji.

W Swing dostępny jest jeszcze tylko jeden dodatkowy komponent wyboru o nazwie `JColorChooser` (rysunki 9.42 do 9.44). Umożliwia on użytkownikowi wybór koloru. Podobnie jak klasa `JFileChooser`, klasa `JColorChooser` jest komponentem, a nie oknem dialogowym. Zawiera natomiast metody służące do tworzenia okien dialogowych z komponentem wyboru koloru.

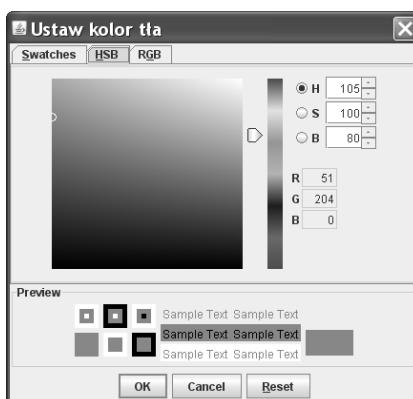
Rysunek 9.42.

Karta Swatches
(próbki)



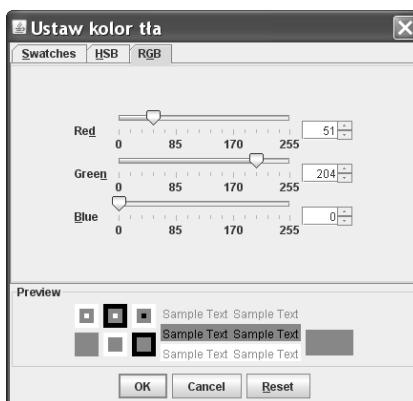
Rysunek 9.43.

Karta HSB



Rysunek 9.44.

Karta RGB



Poniższa instrukcja powoduje wyświetlenie modalnego okna dialogowego z komponentem wyboru koloru:

```
Color selectedColor = JColorChooser.showDialog(parent, title, initialColor);
```

Można też wyświetlić niemodalne okno dialogowe z komponentem wyboru koloru. W takim przypadku należy podać następujące informacje:

- komponent nadzędny,
- tytuł okna dialogowego,
- znacznik ustalający modalność okna,
- komponent wyboru koloru,
- słuchacze przycisków *OK* i *Cancel* (lub `null`, jeśli ma nie być słuchaczy).

Poniższy fragment programu tworzy niemodalne okno dialogowe zmieniające kolor tła w odpowiedzi na kliknięcie przycisku *OK*:

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
    parent,
    "Kolor tła",
    false /* niemodalne */,
    chooser,
    new ActionListener() // Słuchacz przycisku OK.
    {
        public void actionPerformed(ActionEvent event)
        {
            setBackground(chooser.getColor());
        }
    },
    null /* Brak słuchacza przycisku Cancel. */);
```

Można nawet zrobić to lepiej i dodać natychmiastowy podgląd wybranego koloru. Aby monitorować wybierane kolory, należy utworzyć model wyboru komponentu wyboru i dodać słuchacza zmian:

```
chooser.getSelectionModel().addChangeListener(new
    ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        Procedury związane z chooser.getColor();
    }
});
```

W tym przypadku nie ma żadnych korzyści z przycisków *OK* i *Cancel* dostarczanych przez okno wyboru koloru. Komponent wyboru koloru można dodać bezpośrednio do niemodalnego okna dialogowego:

```
dialog = new JDialog(parent, false /* niemodalne */);
dialog.add(chooser);
dialog.pack();
```

Program przedstawiony na listingu 9.24 demonstruje wszystkie trzy wymienione typy okien dialogowych. Kliknięcie przycisku *Modalne* pociąga za sobą konieczność wyboru koloru, zanim można zrobić cokolwiek innego. Kliknięcie przycisku *Niemodalne* daje niemodalne okno dialogowe, ale zmiana koloru następuje dopiero po naciśnięciu przycisku *OK*. Przycisk *Bezpośrednie* wyświetla niemodalne okno dialogowe bez przycisków. Kolor tła panelu jest aktualizowany bezpośrednio po wybraniu koloru w oknie dialogowym.

Listing 9.24. colorChooser/ColorChooserPanel.java

```
package colorChooser;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Panel z przyciskami uruchamiającymi trzy typy okien
 */
public class ColorChooserPanel extends JPanel
{
    public ColorChooserPanel()
    {
        JButton modalButton = new JButton("Modalne");
        modalButton.addActionListener(new ModalListener());
        add(modalButton);

        JButton modelessButton = new JButton("Niemodealne");
        modelessButton.addActionListener(new ModelessListener());
        add(modelessButton);

        JButton immediateButton = new JButton("Bezpośrednie");
        immediateButton.addActionListener(new ImmediateListener());
        add(immediateButton);
    }

    /**
     * Ten słuchacz wyświetla okno modalne.
     */
    private class ModalListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            Color defaultColor = getBackground();
            Color selected = JColorChooser.showDialog(ColorChooserPanel.this, "Ustaw kolor tła",
                defaultColor);
            if (selected != null) setBackground(selected);
        }
    }

    /**
     * Ten słuchacz wyświetla okno niemodealne. Kolor tła panelu zmienia się po kliknięciu przycisku OK.
     */
    private class ModelessListener implements ActionListener
    {
        private JDialog dialog;
        private JColorChooser chooser;

        public ModelessListener()
        {
            chooser = new JColorChooser();
            dialog = JColorChooser.createDialog(ColorChooserPanel.this, "Kolor tła",
                true);
        }
    }
}
```

```

        false /* niemodalne */, chooser, new ActionListener() // Słuchacz
        // przycisku OK
    {
        public void actionPerformed(ActionEvent event)
        {
            setBackground(chooser.getColor());
        }
    }, null /* Brak słuchacza dla przycisku Cancel. */;
}

public void actionPerformed(ActionEvent event)
{
    chooser.setColor(getBackground());
    dialog.setVisible(true);
}
}

/**
 * Ten słuchacz wyświetla okno niemodalne. Kolor tła panelu zmienia się bezpośrednio
 * po wybraniu przez użytkownika koloru.
 */
private class ImmediateListener implements ActionListener
{
    private JDialog dialog;
    private JColorChooser chooser;

    public ImmediateListener()
    {
        chooser = new JColorChooser();
        chooser.getSelectionModel().addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent event)
            {
                setBackground(chooser.getColor());
            }
        });
    }

    dialog = new JDialog((Frame) null, false /* niemodalne */);
    dialog.add(chooser);
    dialog.pack();
}

public void actionPerformed(ActionEvent event)
{
    chooser.setColor(getBackground());
    dialog.setVisible(true);
}
}
}

```

javax.swing.JColorChooser 1.2

■ **JColorChooser()**

Tworzy komponent wyboru koloru z początkowym kolorem białym.

- `Color getColor()`
- `void setColor(Color c)`

Pobiera i ustawia aktualny kolor.

- `static Color showDialog(Component parent, String title, Color initialColor)`

Wyświetla modalne okno dialogowe zawierające komponent wyboru koloru.

Parametry:

<code>parent</code>	Komponent, nad którym ma się pojawić okno.
<code>title</code>	Tytuł okna dialogowego.
<code>initialColor</code>	Początkowy kolor w oknie wyboru koloru.

- `static JDialog createDialog(Component parent, String title, boolean modal, JcolorChooser chooser, ActionListener okListener, ActionListener cancelListener)`

Tworzy okno dialogowe zawierające komponent wyboru koloru.

Parametry:

<code>parent</code>	Komponent, nad którym ma się pojawić okno.
<code>title</code>	Tytuł okna dialogowego.
<code>modal</code>	Wartość <code>true</code> , jeśli wszystkie okna są zablokowane do momentu zamknięcia tego okna.
<code>chooser</code>	Komponent wyboru koloru, który ma być dodany do okna dialogowego.
<code>okListener, cancelListener</code>	Słuchacze przycisków <i>OK</i> i <i>Cancel</i> .

Na tym zakończymy opis elementów interfejsu użytkownika. Informacje zawarte w rozdziałach 7., 8. i 9. pozwalają na tworzenie prostych GUI w Swingu. Bardziej zaawansowane komponenty Swing i techniki graficzne zostały opisane w drugim tomie.

10

Przygotowywanie apletów i aplikacji do użytku

W tym rozdziale:

- Pliki JAR
- Java Web Start
- Aplety
- Zapisywanie ustawień aplikacji

W tej chwili powinniśmy swobodnie posługiwać się większością funkcji języka Java. Mamy też solidne podstawy programowania interfejsów graficznych. Skoro potrafimy tworzyć aplikacje użytkowe, musimy poznać techniki przygotowywania ich do użytku na komputerze użytkownika. W kwestii tej wybór często pada na **aplety** (ang. *applet*), które były powodem ogólnego zainteresowania Javą na początku jej istnienia. Aplet to specjalny rodzaj programu w Javie, który może zostać pobrany przez przeglądarkę z internetu i uruchomiony. Miał on uwolnić użytkowników od problemów związanych z instalacją oprogramowania, które byłoby pobierane na dowolne urządzenie lub komputer obsługujący Javę i podłączony do internetu.

Apletty nie spełniły pokładanych w nich oczekiwania z wielu powodów. Dlatego rozdział ten zaczynamy od technik pakowania aplikacji. Następnie przechodzimy do mechanizmu **Java Web Start**, będącego alternatywą dla dostarczania aplikacji za pomocą internetu, który naprawia niektóre z wad apletów. Na końcu opisujemy aplety, pokazując sytuacje, w których mogą znaleźć zastosowanie.

Piszemy także o sposobach zapisywania danych konfiguracyjnych i preferencji użytkownika w programach.

10.1. Pliki JAR

Pakowanie aplikacji ma na celu utworzenie jednego pliku do wykorzystania przez użytkownika zamiast całej struktury katalogów pełnych plików klas. Do tego celu służą poddawane kompresji ZIP pliki Java Archive (JAR). Mogą one zawierać nie tylko pliki klas, ale również obrazy i pliki dźwiękowe.



W Javie dostępny jest też alternatywny algorytm kompresji o nazwie pack200, który został zoptymalizowany pod kątem bardziej efektywnego zmniejszania rozmiarów plików klas w porównaniu do zwykłego algorytmu ZIP. Według zapewnień firmy Oracle współczynnik kompresji plików klas sięga aż 90%. Więcej informacji na ten temat znajduje się pod adresem <http://docs.oracle.com/javase/1.5.0/docs/guide/deployment/deployment-guide/pack200.html>.

Do tworzenia plików JAR służy narzędzie o nazwie *jar* (w standardowej instalacji JDK znajduje się w katalogu *jdk/bin*). Najczęściej stosowane polecenie tworzące plik JAR ma następującą składnię:

```
jar cvf JARNazwaPliku Plik1 Plik2 . . .
```

Na przykład:

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

Ogólny format polecenia *jar* jest następujący:

```
jar opcje Plik1 Plik2 . . .
```

Tabela 10.1 przedstawia wszystkie opcje narzędzia *jar*. Są one podobne do opcji polecenia *tar* w systemie Unix.

W plikach JAR można pakować aplikacje, komponenty programów (tak zwane beans, o których mowa w rozdziale 8. drugiego tomu) i biblioteki kodu. Na przykład biblioteka wykonawcza JDK jest zawarta w bardzo dużym pliku o nazwie *rt.jar*.

10.1.1. Manifest

Poza klasami, obrazami i innymi plikami źródłowymi każdy plik JAR zawiera plik **manifestu**, który określa specjalne własności archiwum.

Wspomniany plik manifestu ma nazwę *MANIFEST.MF*, a jego lokalizacja to specjalny podkatalog *META-INF* w pliku JAR. Minimalna zawartość takiego pliku nie jest zbyt interesująca:

```
Manifest-Version: 1.0
```

Złożone pliki tego typu mogą zawierać znacznie więcej wpisów pogrupowanych w sekcjach. Pierwsza sekcja nosi nazwę **sekcji głównej** (ang. *main section*) i ma zastosowanie do

Tabela 10.1. Opcje narzędzia jar

Opcja	Opis
c	Tworzy puste archiwum i dodaje do niego pliki. Katalogi są przetwarzane rekursywnie.
C	Zmienia tymczasowo lokalizację. Na przykład polecenie <code>cvf JARfileName.jar -C classes *.class</code> przechodzi do katalogu <code>classes</code> w celu dodania klas.
e	Tworzy punkt startowy w manifeście (zobacz podrozdział 10.1.2, „Wykonywalne pliki JAR”).
f	Określa plik JAR o danej nazwie jako drugi argument wiersza poleceń. Jeśli tego parametru brakuje, <code>jar</code> wyśle wynik do standardowego wyjścia (przy tworzeniu pliku JAR) lub wczyta go ze standardowego wejścia (przy rozpakowywaniu lub tabulacji pliku JAR).
i	Tworzy plik indeksowy (przyspieszający wyszukiwanie w dużych archiwach).
m	Dodaje manifest do pliku JAR. Manifest jest opisem zawartości i pochodzenia pliku archiwum. Każde archiwum ma domyślny manifest, ale można utworzyć własny, który uwierzytelnia zawartość archiwum.
M	Blokuje tworzenie domyślnego pliku manifestu.
t	Wyświetla spis treści.
u	Aktualizuje istniejący plik JAR.
v	Generuje obszerne dane wyjściowe.
x	Wypakowuje pliki. Jeśli podanych zostanie kilka nazw plików, zostaną wypakowane tylko one. W przeciwnym przypadku program wypakuje wszystkie pliki.
0	Wyłącza kompresję ZIP.

całego pliku JAR. Kolejne sekcje określają własności różnych elementów mających nazwy, jak konkretne pliki, pakiety czy adresy URL. Każda z nich musi się zaczynać od słowa `Name`. Sekcje są rozdzielane pustą linią. Na przykład:

`Manifest-Version: 1.0`
`opis całego archiwum`

`Name: Wozzle.class`
`opis jednego pliku`

`Name: com/mycompany/mypkg/`
`opis pakietu`

Aby zmienić zawartość pliku manifestu, należy dokonać niezbędnych zmian i wydać poniższe polecenie:

`jar cfm NazwaPlikuJAR NazwaPlikuManifest . . .`

Na przykład poniższe polecenie tworzy nowy plik JAR z plikiem manifestu:

`jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class`

Aby zaktualizować plik manifestu istniejącego pliku JAR, należy umieścić w pliku tekstowym wpisy, które mają być dodane, i wydać następujące polecenie:

`jar ufm MyArchive.jar manifest-additions.mf`



Więcej informacji na temat plików JAR i manifestu można znaleźć na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/jar/>.

10.1.2. Wykonywalne pliki JAR

Istnieje możliwość określenia **punktu startowego** programu, czyli klasy, od której zaczyna się działanie programu, za pomocą opcji `e` narzędzia `jar`:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass pliki do dodania
```

Klasę główną programu można też określić w manifeście. W tym celu należy do niego dodać instrukcję o następującej postaci:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

Nie dodawaj rozszerzenia `.class` do nazwy klasy głównej.



Na końcu ostatniego wiersza w pliku manifestu musi się znajdować znak nowego wiersza. W przeciwnym przypadku plik zostanie odczytany nieprawidłowo. Błąd polegający na utworzeniu pliku tekstowego zawierającego tylko wiersz `Main-Class` bez znaku końca wiersza jest często spotykany.

W obu wymienionych przypadkach program można uruchomić przy użyciu następującego polecenia:

```
java -jar MyProgram.jar
```

W zależności od konfiguracji systemu operacyjnego może być możliwe uruchomienie aplikacji za pomocą dwukrotnego kliknięcia pliku JAR. Poniżej znajduje się opis zachowania różnych systemów w takiej sytuacji:

- W systemie Windows instalator aplikacji Java tworzy dowiązanie dla plików o rozszerzeniu `.jar`, które uruchamia te pliki za pomocą polecenia `javaw -jar` (polecenie `javaw`, w przeciwnieństwie do `java`, nie otwiera okna wiersza polecień).
- System Solaris rozpoznaje „magiczną liczbę” pliku JAR i uruchamia ją za pomocą polecenia `java -jar`.
- System Mac OS X rozpoznaje rozszerzenie `.jar` i uruchamia programy w Javie w wyniku dwukrotnego kliknięcia pliku JAR.

Jednak programy Javy w plikach JAR to nie to samo co aplikacje rodzime. W systemie Windows można skorzystać z narzędzi innych producentów służących do zamianiania plików JAR na pliki wykonywalne tego systemu. Plik JAR jest opakowywany w plik o rozszerzeniu `.exe`, który lokalizuje i uruchamia maszynę wirtualną Javy (JVM) lub informuje użytkownika, co powinien zrobić, jeśli JVM nie ma. Istnieje kilka komercyjnych i darmowych narzędzi tego typu, np.: JSmooth (<http://jsmooth.sourceforge.net>) i Launch4J (<http://launch4j.sourceforge.net>). Generator instalatorów IzPack (<http://izpack.org>) zawiera także rodzimy program uruchamiający. Więcej informacji na ten temat można znaleźć pod adresem <http://www.javalobby.org/articles/java2exe>.

W komputerach z systemem Mac OS X sytuacja wygląda nieco lepiej. W skład środowiska programistycznego XCode wchodzi narzędzie o nazwie Jar Bundler służące do zamieniania plików JAR na aplikacje Mac.

10.1.3. Zasoby

Klasy używane zarówno w apletach, jak i aplikacjach często wykorzystują pliki danych tego samego typu:

- pliki obrazów i zawierające dźwięk;
- pliki tekstowe zawierające łańcuchy komunikatów i etykiety przycisków;
- pliki z danymi binarnymi, na przykład opisującymi rozkład mapy.

W Javie takie pliki nazywane są **zasobami** (ang. *resources*).



W systemie Windows termin „zasób” (ang. *resource*) ma węższe znaczenie. Zasoby w tym systemie także mogą być plikami obrazów, etykietami przycisków itd., ale są związane z plikami wykonywalnymi z dostępem za pośrednictwem standardowego interfejsu programistycznego. Natomiast pliki zasobów Java są przechowywane osobno, a nie jako części plików klas. Dostęp do zasobów i ich interpretacja zależy od programu.

Weźmy na przykład klasę o nazwie `AboutPanel`, która wyświetla komunikat widoczny na rysunku 10.1.

Rysunek 10.1.
Wyświetlanie
zasobu
z pliku JAR



Wiadomo, że tytuł i rok wydania zostaną zmienione w kolejnym wydaniu książki. Aby ułatwić zmianę, ten tekst należy umieścić w pliku tekstowym, a nie bezpośrednio w kodzie programu.

Powstaje jednak pytanie, gdzie umieścić taki plik jak `about.txt`. Oczywiście najlepiej byłoby, aby znajdował się on razem z pozostałymi plikami programu w pliku JAR.

Program ładowający klasy potrafi znaleźć pliki klas, jeśli znajdują się gdzieś na ścieżce klas, w archiwum lub na serwerze sieciowym. Mechanizm zasobów oferuje podobną funkcjonalność dla plików, które nie są klasami. Poniżej znajduje się spis wymaganych czynności:

1. Utwórz obiekt `Class` klasy, która ma zasób, na przykład `AboutPanel.class`.
2. Jeśli zasobem jest obraz lub plik audio, wywołaj metodę `getResource(filename)` w celu uzyskania lokalizacji zasobu w postaci adresu URL. Następnie odczytaj go za pomocą metody `getImage` lub `getAudioClip`.
3. W przypadku innych zasobów niż obrazy i pliki audio dane z pliku należy wczytywać za pomocą metody `getResourceAsStream`.

Chodzi o to, aby program ładujący klasy potrafił znaleźć klasę i odszukać związane z nią zasoby w tej samej lokalizacji.

Na przykład poniższy fragment kodu tworzy ikonę z pliku `about.gif`:

```
URL url = ResourceTest.class.getResource("about.gif");
Image img = new ImageIcon(url).getImage();
```

Powyższy kod można odczytać następująco: „znajdź plik `about.gif` w tej samej lokalizacji, w której znajduje się klasa `ResourceTest`”.

Poniższe instrukcje wczytują plik `about.txt`:

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
```

Plik zasobu nie musi się znajdować w tym samym katalogu co klasa — może być w jakimś podkatalogu. Można zastosować hierarchiczną nazwę zasobu, jak poniższa:

```
data/text/about.txt
```

Jest to względna nazwa zasobu. Jest ona interpretowana względem pakietu klasy, która ładuje dany zasób. Należy pamiętać, że zawsze trzeba używać separatora /, bez względu na separator katalogów stosowany w systemie, w którym są przechowywane pliki zasobów. Na przykład w systemie plików systemu Windows separatory / są automatycznie zamieniane na \.

Nazwa zasobu zaczynająca się od znaku / jest bezwzględną nazwą zasobu. Jest ona lokalizowana w taki sam sposób jak klasa wewnętrz pakietu. Na przykład zasób:

```
/corejava/title.txt
```

znajduje się w katalogu `corejava` (który może być podkatalogiem ścieżki klas wewnętrz pliku JAR lub, w przypadku appletów, na serwerze sieciowym).

Jedynym przeznaczeniem funkcji ładowania zasobów jest ładowanie plików. Nie istnieją żadne standardowe metody interpretujące zawartość pliku zasobów. Każdy program musi interpretować zawartość swoich plików zasobów na swój własny sposób.

Innym często spotykanym zastosowaniem zasobów jest międzynarodowa lokalizacja programów. W plikach zasobów przechowuje się łańcuchy, które zmieniają się w zależności od języka, czyli komunikaty i etykiety interfejsu użytkownika. Dla każdego języka tworzony jest osobny plik. **API internacjonalizacji**, które zostało opisane w rozdziale 5. drugiego tomu, udostępnia standardową metodę służącą do organizacji i dostępu do plików lokalizacyjnych.

Listing 10.1 przedstawia kod programu demonstrującego ładowanie zasobów. Poniższe polecenia kompilują go, tworzą plik JAR i uruchamiają go:

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

Aby przekonać się, że program pobiera pliki zasobów z archiwum JAR, a nie bieżącego katalogu, można przenieść ten program do innego folderu.

Listing 10.1. resource/ResourceTest.java

```
package resource;

import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

/**
 * @version 1.4 2007-04-30
 * @author Cay Horstmann
 */
public class ResourceTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ResourceTestFrame();
                frame.setTitle("ResourceTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka ładująca zasoby graficzne i tekstowe
 */
class ResourceTestFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    public ResourceTestFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        URL aboutURL = getClass().getResource("about.gif");
        Image img = new ImageIcon(aboutURL).getImage();
        setIconImage(img);
```

```

JTextArea textArea = new JTextArea();
InputStream stream = getClass().getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
while (in.hasNext())
    textArea.append(in.nextLine() + "\n");
add(textArea);
}
}

```

java.lang.Class 1.0

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

Znajduje zasób w tym samym katalogu, w którym jest umieszczona klasa, i zwraca adres URL lub strumień wejściowy, za pomocą którego można ten zasób załadować. Zwraca wartość null, jeśli zasób nie istnieje, dzięki czemu nie powoduje wyjątku dla błędu wejścia-wyjścia.

10.1.4. Pieczętowanie pakietów

W rozdziale 4. wspomnieliśmy o możliwości **pieczętowania** (ang. *seal*) pakietów Javy w celu uniemożliwienia dodawania do nich kolejnych klas. Może być to konieczne w przypadku używania klas, metod i pól o zasięgu pakietowym. Gdyby nie pieczętowanie, inne klasy mogłyby być umieszczane w tym samym pakiecie i dzięki temu uzyskiwać dostęp do elementów pakietowych.

Jeśli na przykład pakiet com.mycompany.util zostanie zapieczętowany, żadna klasa spoza tego zapieczętowanego archiwum nie może być zdefiniowana za pomocą poniższej instrukcji:

```
package com.mycompany.util;
```

W tym celu wszystkie klasy pakietu należy umieścić w pliku JAR. Domyślnie pakiety w pliku JAR nie są zapieczętowane. Można zmienić to domyślne globalne ustawienie, wstawiając wiersz

```
Sealed: true
```

w głównej sekcji pliku manifestu. Aby zapieczętować tylko wybrane pakiety, należy do pliku manifestu w pliku JAR wstawić dodatkowe sekcje:

```
Name: com/mycompany/util/
Sealed: true
```

```
Name: com/mycompany/misc/
Sealed: false
```

Aby zapieczętować pakiet, należy utworzyć plik tekstowy z instrukcjami manifestu. Następnie należy uruchomić narzędzie jar w zwykły sposób:

```
jar cvfm MyArchive.jar manifest.mf pliki do dodania
```

10.2. Java Web Start

Java Web Start jest technologią uruchamiania aplikacji bezpośrednio z internetu. Aplikacje Java Web Start mają następujące cechy:

- Są zazwyczaj dostarczane za pośrednictwem przeglądarki. Po pobraniu aplikacja Java Web Start może być uruchamiana bez użycia przeglądarki.
- Nie rezydują w oknie przeglądarki. Aplikacja działa we własnym oknie ramowym, poza przeglądarką.
- Nie wykorzystują implementacji Javy przeglądarki. Przeglądarka uruchamia tylko zewnętrzną aplikację, podobnie jak w przypadku innych programów, takich jak Adobe Acrobat lub Real Audio.
- Aplikacjom podpisany cyfrowo można nadawać dowolne prawa dostępu. Niepodpisane aplikacje działają w **piaskownicy** (ang. *sandbox*), która nie zezwala na potencjalnie niebezpieczne operacje.

Przygotowywanie aplikacji do dostarczania za pośrednictwem mechanizmu Java Web Start polega na spakowaniu jej do jednego lub większej liczby plików JAR. Następnie należy utworzyć plik deskryptora w formacie JNLP (ang. *Java Network Launch Protocol*). Pliki umieszcza się na serwerze.

Dodatkowo serwer sieciowy musi raportować typ MIME *application/x-java-jnlp-file* dla plików z rozszerzeniem *.jnlp* (typ MIME umożliwia przeglądarkom podjęcie decyzji, który program pomocniczy uruchomić). Szczegółowych informacji na ten temat należy szukać w dokumentacji serwera.



Aby wypróbować mechanizm Java Web Start, zainstaluj Tomcat dostępny na stronie <http://tomcat.apache.org/>. Jest to kontener na serwety i strony JSP, ale serwuje też strony internetowe. Jest wstępnie skonfigurowany do serwowania poprawnego typu MIME dla plików JNLP.

Wypróbowejmy mechanizm Java Web Start na kalkulatorze z rozdziału 9. Wykonaj następujące czynności:

- 1 Skompiluj program.

```
javac -classpath .:/path/to/javaws.jar webstart/*.java
```

- 2 Utwórz plik JAR za pomocą poniższego polecenia:

```
jar cvfe Calculator.jar webstart.Calculator webstart/*.class
```

- 3 Przygotuj plik rozruchowy *Calculator.jnlp* o następującej treści:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://localhost:8080/calculator/" href="Calculator.jnlp">
  <information>
    <title>Przykładowy kalkulator</title>
    <vendor>Cay S. Horstmann</vendor>
```

```

<description>Kalkulator</description>
<offline-allowed/>
</information>
<resources>
  <java version="1.6.0+/">
    <jar href="Calculator.jar"/>
  </resources>
  <application-desc/>
</jnlp>

```

(Należy pamiętać, że numer wersji to 1.6.0, a nie 6.0.)

Format pliku JNLP jest bardzo prosty. Jego pełna specyfikacja znajduje się na stronie <http://www.oracle.com/technetwork/java/javawebstart/index.html>.

4. W przypadku wykorzystania serwera Tomcat należy utworzyć katalog **tomcat/webapps/calculator**, gdzie **tomcat** to katalog główny instalacji Tomcata. Utwórz podkatalog **tomcat/webapps/calculator/WEB-INF** i umieść w nim poniższy plik **web.xml**:

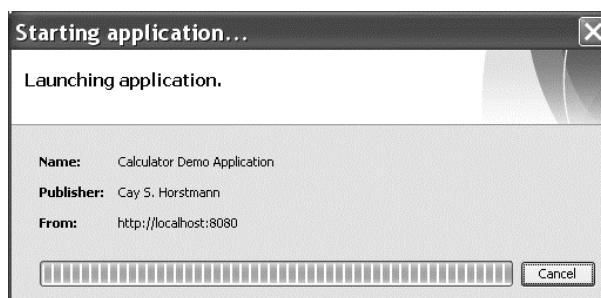
```

<?xml version="1.0" encoding="utf-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">
</web-app>

```

5. Umieść pliki JAR i JNLP na serwerze sieciowym, aby adres URL zgadzał się z wpisem codebase w pliku JNLP. W przypadku serwera Tomcat pliki należy umieścić w katalogu **tomcat/webapps/calculator**.
6. Upewnij się, że masz przeglądarkę skonfigurowaną pod kątem Java Web Start, sprawdzając, czy typ MIME **application/x-java-jnlp-file** jest skojarzony z aplikacją **javaws**. Jeśli zainstalowano pakiet JDK, konfiguracja ta powinna zostać wykonana automatycznie.
7. Uruchom serwer Tomcat.
8. Wpisz w przeglądarce adres pliku JNLP. Na przykład w przypadku użycia serwera Tomcat należy wpisać adres **http://localhost:8080/calculator/Calculator.jnlp**.
9. Powinno się pojawić okno uruchamiania Java Web Start (rysunek 10.2).

Rysunek 10.2.
Uruchamianie aplikacji Java Web Start



10. Chwilę później powinien się pojawić kalkulator z informacją, że jest to aplikacja Javy (rysunek 10.3).

Rysunek 10.3.

Kalkulator otwarty za pośrednictwem Java Web Start



11. Kiedy nastepnym razem spróbujemy uzyskać dostęp do pliku JNLP, program będzie pobierany z pamięci podręcznej. Zawartość tej pamięci można obejrzeć za pomocą panelu kontrolnego w postaci wtyczki Javy (rysunek 10.4). W systemie Windows wtyczki tej należy szukać w *Panelu sterowania*. W systemie Linux należy wykonać polecenie *jdk/jre/bin/ControlPanel*.



Aby nie uruchamiać serwera podczas testowania konfiguracji JNLP, można tymczasowo nadpisać adres URL codebase w pliku JNLP za pomocą poniższego polecenia:

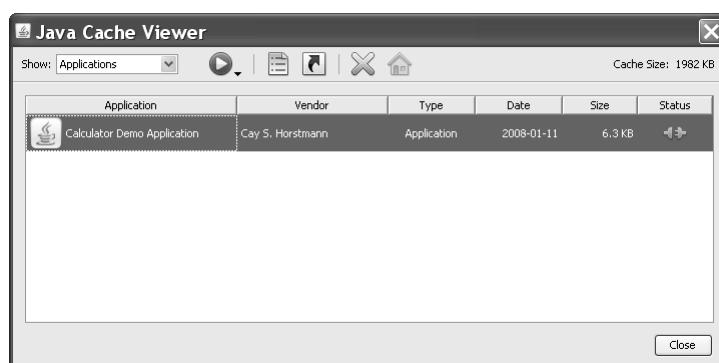
```
javaws -codebase file:///katalogProgramu plikJNL
```

Na przykład w systemie Unix polecenie to można wydać w katalogu zawierającym plik JNLP:

```
javaws -codebase file://`pwd` Calculator.jnlp
```

Rysunek 10.4.

Aplikacje w pamięci podręcznej



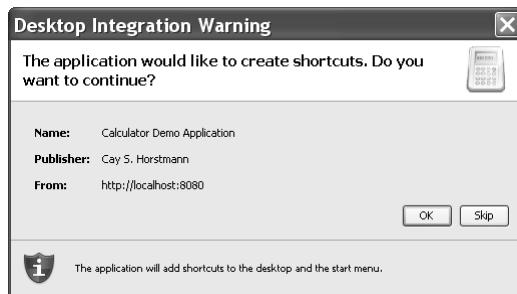
Oczywiście nie chcemy nakłaniać użytkowników do uruchamiania przeglądarki pamięci podręcznej za każdym razem, kiedy chcą uruchomić naszą aplikację. Można sprawić, aby instalator proponował utworzenie skrótów w menu *Start* i na pulpicie. W tym celu należy dodać poniższy kod do pliku JNLP:

```
<shortcut>
  <desktop/>
  <menu submenu="Akcesoria"/>
</shortcut>
```

Podczas pierwszego pobierania aplikacji zostanie wyświetcone ostrzeżenie o dodawaniu skrótów (rysunek 10.5).

Rysunek 10.5.

Ostrzeżenie o integracji



Dodatkowo powinno się dodać ikonę dla skrótu i ekranu uruchomieniowego. Firma Sun zaleca stosowanie ikon o rozmiarach 32×32 i 64×64 piksele. Pliki ikon należy umieścić na serwerze razem z plikami JAR i JNLP. Poniższy kod należy dodać do sekcji `information` pliku JNLP:

```
<icon href="calc_icon32.png" width="32" height="32" />
<icon href="calc_icon64.png" width="64" height="64" />
```

Należy pamiętać, że ikony te nie są związane z ikoną aplikacji. Aby wstawić ikonę dla aplikacji, należy dodać odrębny plik ikony do pliku JAR i wywołać metodę `IconImage` na rzecz klasy ramowej (zobacz listing 10.1).

10.2.1. Piaskownica

Kiedy kod uruchamiany na komputerze jest pobierany ze zdalnego miejsca, sprawą pierwszą staje się zawsze bezpieczeństwo. Aplikację Java Web Start może uruchomić jedno kliknięcie. Wejście na stronę powoduje automatyczne uruchomienie wszystkich znajdujących się na niej appletów. Gdyby kliknięcie odnośnika lub wejście na stronę internetową pozwalało na uruchomienie dowolnego kodu na komputerze użytkownika, przestępcy przeżywaliby złoty wiek wykradania poufnych informacji, danych finansowych i przejmowania komputerów użytkowników w celu rozsyłania spamu.

Technologia Java dysponuje zaawansowanym modelem ochrony, który zapobiega wykorzystywaniu jej do nikczemnych postępów. Model ten został szczegółowo opisany w tomie drugim. **Menedżer zabezpieczeń** (ang. *security manager*) kontroluje dostęp do wszystkich zasobów systemowych. Przy standardowych ustawieniach zezwala tylko na nieszkodliwe operacje. Aby zezwolić na dodatkowe operacje, kod musi być podpisany cyfrowo, a użytkownik musi zatwierdzić podpis certyfikatu.

Co pobierany zdalnie kod **może** robić na wszystkich platformach? Zawsze można wyświetlać obrazy, odtwarzać dźwięki, odpowiadać na naciśnięcia przez użytkownika klawiszy i przycisków myszki oraz wysyłać dane wprowadzone przez użytkownika do hosta, z którego został załadowany kod. Taka funkcjonalność wystarcza do zaprezentowania faktów i liczb oraz pobrania danych od użytkownika składającego zamówienie. Ograniczone środowisko wykonańcze jest często nazywane **piaskownicą** (ang. *sandbox*). Kod działający w piaskownicy nie może nic zmieniać w systemie użytkownika ani go szpiegować.

Programy działające w piaskownicy mają następujące ograniczenia:

- Nie mogą uruchamiać **żadnych** lokalnych programów.
- Nie mogą czytać ani zapisywać plików w lokalnym systemie plików.
- Nie mają dostępu do żadnych informacji o komputerze lokalnym, z wyjątkiem wersji Javy i kilku mało ważnych danych na temat systemu operacyjnego. Kod działający w piaskownicy w szczególności nie ma dostępu do nazwy użytkownika, adresów e-mail itd.
- Programy ładowane z serwera zdalnego nie mogą się komunikować z żadnym hostem poza tym, z którego pochodzą — serwer taki nosi nazwę **serwera pochodzenia** (ang. *originating host*). Dzięki temu użytkownik jest chroniony przed programami, które mogą wykradać wewnętrzne zasoby.
- Wszystkie wyskakujące okna zawierają komunikat ostrzegawczy. Ma on na celu zabezpieczenie przed pomyleniem ich z oknem aplikacji lokalnej. Istnieją obawy, że nie niepodejrzewający użytkownik wejdzie na stronę internetową, podstępem zostanie zmuszony do uruchomienia zdalnego kodu, a następnie wpisze hasło lub numer karty kredytowej, które zostaną przesłane z powrotem do serwera. We wczesnych wersjach JDK komunikat ten brzmiał bardzo złowieszczo: *Untrusted Java Applet Window*. W każdej kolejnej wersji brzmienie to było nieco łagodzone: *Unauthenticated Java Applet Window* czy *Warning: Java Applet Window*. Obecnie jest to *Java Applet Window* lub *Java Application Window*.

10.2.2. Podpisywanie kodu

Ograniczenia piaskownicy są często zbyt rygorystyczne. Na przykład w wewnętrznej sieci firmowej nietrudno spotkać aplikację lub aplet Web Start wymagający dostępu do lokalnych plików. Istnieje możliwość bardzo szczegółowego określenia, jakie prawa dostępu ma każda aplikacja. Technikę tę opisujemy w rozdziale 9. drugiego tomu. Oczywiście aplikacja Web Start może po prostu zażądać wszystkich praw, które ma aplikacja lokalna. Grupa takich aplikacji jest całkiem pokaźna. Aby nadać wszystkie prawa, należy umieścić poniższy fragment kodu w pliku JNLP:

```
<security>
  <all-permissions/>
</security>
```

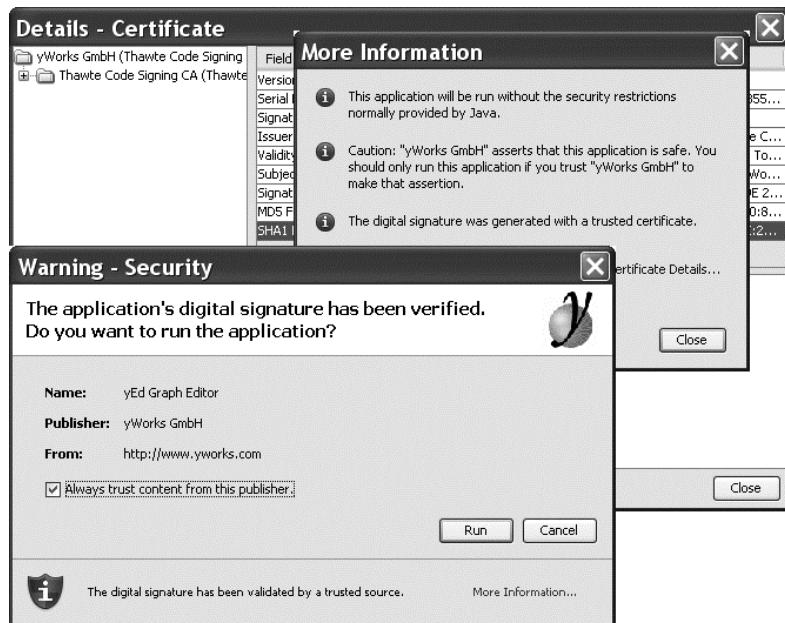
Aby móc działać poza piaskownicą, pliki JAR aplikacji Java Web Start muszą być **podpisane cyfrowo**. Podpisany plik JAR ma certyfikat określający tożsamość tego, kto go podpisał. Techniki kryptograficzne dają pewność, że certyfikat taki nie jest sfałszowany, a wszelkie próby zmiany jego zawartości są natychmiast wykrywane.

Wyobraźmy sobie, że pobieramy aplikację utworzoną i podpisana cyfrowo przez firmę yWorks GmbH z certyfikatem wydanym przez ośrodek certyfikacji Thawte (rysunek 10.6). Odbierając aplikację, mamy pewność, że:

- 1 Kod aplikacji nie został zmieniony w żaden sposób od chwili jej podpisania.
- 2 Podpis rzeczywiście pochodzi od firmy yWorks.

Rysunek 10.6.

Certyfikat bezpieczeństwa



3. Certyfikat naprawdę został wydany przez ośrodek Thawte (mechanizm Java Web Start potrafi sprawdzać certyfikaty wydane przez Thawte i kilka innych instytucji).

Niestety nic więcej nie wiemy. Nie wiemy, czy kod jest na pewno bezpieczny. W rzeczywistości, jeśli klikniemy odnośnik *More Information*, dowiemy się, że aplikacja zostanie uruchomiona bez zwyczajowych ograniczeń. To, czy zainstalować tę aplikację, czy nie, w dużej mierze zależy od tego, czy ufamy firmie yWorks.

Koszty uzyskania certyfikatu od jednego z obsługiwanych dostawców wynoszą kilkaset dolarów rocznie. Wielu deweloperów generuje własne certyfikaty i nimi podpisuje kod. Oczywiście Java Web Start nie ma możliwości sprawdzenia jakości tych certyfikatów. Odbierając taką aplikację, wiadomo, że:

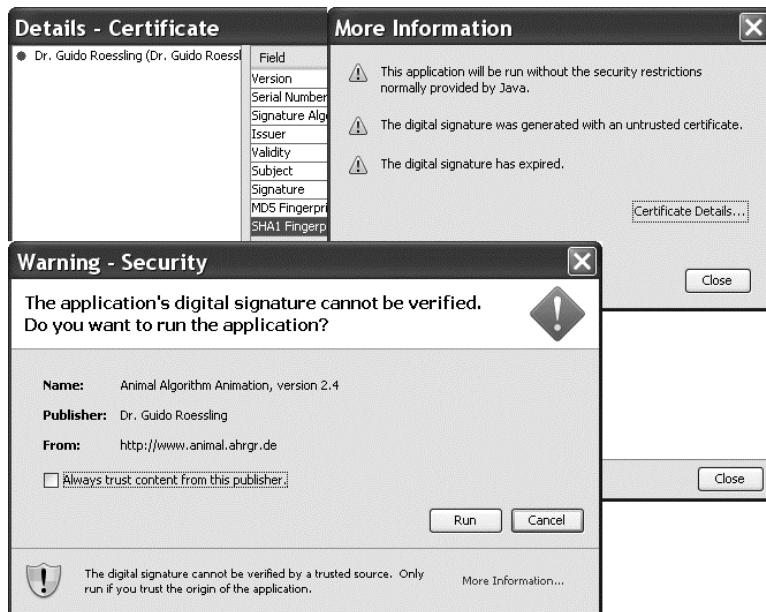
1. Kod wygląda dokładnie tak samo jak wtedy, kiedy został podpisany.
Nikt niepowołany przy nim nie majstrował.
2. Kto podpisał kod, ale Java Web Start nie może sprawdzić kto.

W związku z tym mechanizm ten jest kompletnie bezużyteczny. Każdy mógł zmodyfikować kod, a następnie go podpisać, twierdząc, że jest jego autorem. Niemniej Java Web Start z największą przyjemnością wyświetli certyfikat do zatwierdzenia (zobacz rysunek 10.7). Teoretycznie certyfikat można zweryfikować w jeszcze inny sposób, ale niewielu użytkowników ma wystarczające umiejętności.

OCzywiście każdego dnia mnóstwo ludzi pobiera z internetu i uruchamia aplikacje. Jeśli stwierdzisz, że użytkownicy ufają Twojej aplikacji i infrastrukturze sieciowej, używaj osobiście podpisywanego certyfikatu (zobacz <http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/developersguide/development.html>). W przeciwnym przypadku należy zapewnić użytko-

Rysunek 10.7.

Niebezpieczny certyfikat



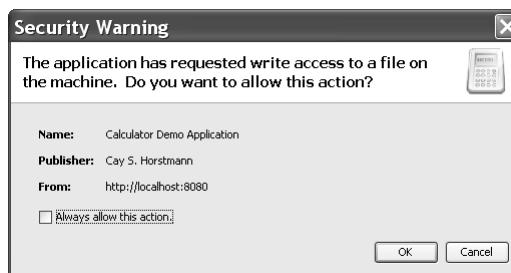
kownikom poczucie bezpieczeństwa i pozostać w piaskownicy. Dzięki API JNLP (o którym mowa w kolejnym podrozdziale) po uzyskaniu zgody użytkownika można dać programowi dostęp tylko do wybranych zasobów.

10.2.3. API JNLP

API JNLP umożliwia aplikacjom działającym w piaskownicy uzyskiwać dostęp do zasobów lokalnych przy zachowaniu odpowiedniego poziomu bezpieczeństwa. Istnieją na przykład usługi polegające na wysyłaniu i zapisywaniu plików. Aplikacja nie ma dostępu do systemu plików i nie może określić nazw plików. W zamian wyświetlane jest okno dialogowe wyboru plików, w którym użytkownik programu wybiera plik. Przed wyświetleniem tego okna pojawia się ostrzeżenie i użytkownik musi wyrazić zgodę na kontynuowanie (rysunek 10.8). Ponadto opisywane API nie daje w rzeczywistości programowi dostępu do obiektu typu File.

Rysunek 10.8.

Ostrzeżenie Java Web Start



W szczególności aplikacja nie może sprawdzić lokalizacji pliku. W ten sposób programista zyskuje narzędzia do implementacji akcji otwierania i zapisywania plików, ale tak dużo informacji systemowych, jak to tylko możliwe, jest ukrytych przed niezaufanymi aplikacjami.

To API udostępnia następujące usługi:

- wysyłanie i zapisywanie plików,
- dostęp do schowka,
- drukowanie,
- pobieranie plików,
- wyświetlanie dokumentów w domyślnej przeglądarce,
- zapisywanie i odczytywanie stałych danych konfiguracyjnych,
- pilnowanie, aby był uruchomiony tylko jeden egzemplarz programu (od Java SE 5.0).

Dostęp do usługi uzyskuje się za pomocą metody `ServiceManager`:

```
FileSaveService service = (FileSaveService)
    ↪ServiceManager.lookup("javax.jnlp.FileSaveService");
```

Powyższa instrukcja spowoduje wyjątek `UnavailableServiceException`, jeśli usługa jest niedostępna.



Aby móc kompilować programy wykorzystujące API JNLP, należy umieścić plik `javaws.jar` na ścieżce klas. Plik ten znajduje się w podkatalogu `jre/lib` w katalogu JDK.

Omówimy najbardziej przydatne usługi JNLP. Aby zapisać plik, trzeba podać ścieżkę startową i rozszerzenia plików wyświetlanych w oknie dialogowym, dane do zapisania oraz sugerowaną nazwę pliku. Na przykład:

```
service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
```

Dane muszą być dostarczone w strumieniu `InputStream`, co nie zawsze jest łatwe. W programie z listingu 10.2 przyjęto następującą strategię działania:

1. Utworzenie strumienia `ByteArrayOutputStream` przechowującego bajty, które mają być zapisane.
2. Utworzenie strumienia `PrintStream` wysyłającego swoje dane do strumienia `ByteArrayOutputStream`.
3. Wydrukowanie informacji, które mają być zapisane, do strumienia `PrintStream`.
4. Utworzenie strumienia `ByteArrayInputStream` odczytującego zapisane bajty.
5. Przekazanie strumienia do metody `saveFileDialog`.

Więcej na temat strumieni piszemy w rozdziale 1. drugiego tomu. Teraz wystarczy tylko побieżnie przejrzeć przykładowy program.

Do odczytu danych z pliku służy klasa `FileOpenService`. Jej metoda `openFileDialog` odbiera sugerowaną ścieżkę początkową i rozszerzenia plików wyświetlanych w oknie dialogowym i zwraca obiekt typu `FileContents`. Następnie można za pomocą metod `getInputStream` i `getOutputStream` odczytać i zapisać dane do pliku. Jeśli użytkownik nie wybrał pliku, metoda `openFileDialog` zwraca wartość `null`.

```

FileOpenService service = (FileOpenService) ServiceManager.lookup("javax.jnlp.
➥FileOpenService");
FileContents contents = service.openFileDialog(".", new String[] { "txt" });
if (contents != null)
{
    InputStream in = contents.getInputStream();
    . .
}

```

Pamiętajmy, że aplikacja nie zna nazwy ani lokalizacji pliku. Aby otworzyć określony plik, można użyć klasy ExtendedService:

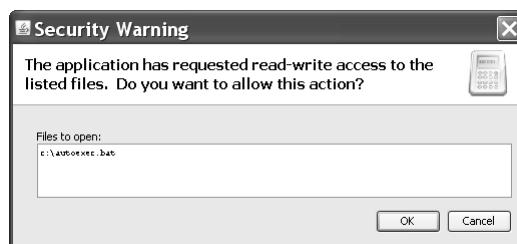
```

ExtendedService service = (ExtendedService) ServiceManager.lookup("javax.jnlp.
➥ExtendedService");
FileContents contents = service.openFile(new File("c:\\autoexec.bat"));
if (contents != null)
{
    OutputStream out = contents.getOutputStream();
    . .
}

```

Użytkownik programu musi się zgodzić na dostęp do tego pliku (rysunek 10.9).

Rysunek 10.9.
Ostrzeżenie
o dostępie
do pliku



Do wyświetlenia dokumentu w domyślnej przeglądarce należy użyć interfejsu BasicService. Zauważmy, że niektóre systemy mogą nie mieć domyślnej przeglądarki.

```

BasicService service = (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
if (service.isWebBrowserSupported())
    service.showDocument(url);
else . .

```

Podstawowy interfejs PersistenceService pozwala aplikacji zapisywać niewielkie ilości danych konfiguracyjnych i odczytywać je po jej ponownym uruchomieniu. Mechanizm ten przypomina nieco pliki cookie HTTP. Ten stały magazyn jako klucze wykorzystuje adresy URL. Adres URL nie musi prowadzić do istniejącego zasobu sieciowego. Są one w tym przypadku wykorzystywane jako wygodny sposób nazewnictwa hierarchicznego. Dla każdego klucza URL aplikacja może zapisać dowolne dane binarne (rozmiar jednego bloku danych w magazynie może być ograniczony).

Odseparowanie od siebie poszczególnych aplikacji jest możliwe dzięki temu, że każdy program może używać tylko takich kluczy, które zaczynają się od określonego podstawowego łańcucha (zgodnie z informacjami w pliku JNLP). Jeśli na przykład aplikacja zostanie pobrana ze strony <http://myserver.com/apps>, może używać tylko kluczy w formacie <http://myserver.com/apps/subkey1/subkey2/>.... Próby dostępu do innych kluczy zakończą się niepowodzeniem.

Aplikacja może sprawdzić podstawę swojego klucza URL za pomocą metody getCodeBase z klasy BasicService.

Do tworzenia kluczy służy metoda create z interfejsu PersistenceService.

```
URL url = new URL(codeBase, "mykey");
service.create(url, maxSize);
```

Aby uzyskać informacje związane z określonym kluczem, należy wywołać metodę get. Zwraca ona obiekt typu FileContents, za pośrednictwem którego można odczytywać i zapisywać dane kluczy. Na przykład:

```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true); // true = nadpisz
```

Niestety nie ma dobrego sposobu na sprawdzenie, czy określony klucz już istnieje, czy trzeba go utworzyć. Można wywołać metodę get. Jeśli klucz nie istnieje, zostanie spowodowany wyjątek FileNotFoundException.

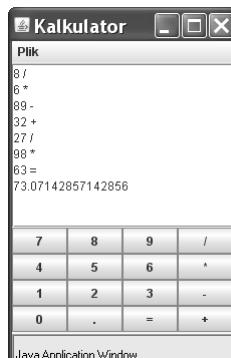


Aplikacje Java Web Start i aplety mogą drukować, wykorzystując normalne API drukowania. Pojawia się tylko okienko, w którym użytkownik jest proszony o zezwolenie na dostęp do drukarki. Więcej informacji na temat API drukowania znajduje się w rozdziale 7. drugiego tomu.

Program na listingu 10.2 jest prostym rozszerzeniem wcześniejszej utworzonego kalkulatora. Ma on wirtualną taśmę, która rejestruje wszystkie obliczenia. Można zapisywać i ładować historię obliczeń. Aplikacja pozwala na ustawienie tytułu ramki, co służy demonstracji trwałego magazynu. Przy ponownym uruchomieniu aplikacja pobierze wybrany przez użytkownika tytuł z trwałego magazynu (rysunek 10.10).

Rysunek 10.10.

Aplikacja
WebStartCalculator



Listing 10.2. webstart/CalculatorFrame.java

```
package webstart;

import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.*;
```

```

import javax.jnlp.*;
import javax.swing.*;

/*
 * Ramka z kalkulatorem i menu do ładowania oraz zapisywania historii obliczeń
 */
public class CalculatorFrame extends JFrame
{
    private CalculatorPanel panel;

    public CalculatorFrame()
    {
        setTitle();
        panel = new CalculatorPanel();
        add(panel);

        JMenu fileMenu = new JMenu("Plik");
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);

        JMenuItem openItem = fileMenu.add("Otwórz");
        openItem.addActionListener(EventHandler.create(ActionListener.class, this,
            "open"));
        JMenuItem saveItem = fileMenu.add("Zapisz");
        saveItem.addActionListener(EventHandler.create(ActionListener.class, this,
            "save"));

        pack();
    }

    /**
     * Pobiera tytuł z magazynu trwałego lub prosi użytkownika o podanie tytułu, jeśli
     * nie ma wcześniejszego wpisu.
     */
    public void setTitle()
    {
        try
        {
            String title = null;

            BasicService basic = (BasicService)
                ServiceManager.lookup("javax.jnlp.BasicService");
            URL codeBase = basic.getCodeBase();

            PersistenceService service = (PersistenceService) ServiceManager
                .lookup("javax.jnlp.PersistenceService");
            URL key = new URL(codeBase, "title");

            try
            {
                FileContents contents = service.get(key);
                InputStream in = contents.getInputStream();
                BufferedReader reader = new BufferedReader(new InputStreamReader(in));
                title = reader.readLine();
            }
            catch (FileNotFoundException e)
            {

```

```
title = JOptionPane.showInputDialog("Podaj tytuł ramki:");
if (title == null) return;

service.create(key, 100);
FileContents contents = service.get(key);
OutputStream out = contents.getOutputStream(true);
PrintStream printOut = new PrintStream(out);
printOut.print(title);
}
setTitle(title);
}
catch (UnavailableServiceException e)
{
JOptionPane.showMessageDialog(this, e);
}
catch (MalformedURLException e)
{
JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
JOptionPane.showMessageDialog(this, e);
}
}

/**
* Otwiera plik historii i aktualizuje zawartość wyświetlacza.
*/
public void open()
{
try
{
FileOpenService service = (FileOpenService) ServiceManager
.lookup("javax.jnlp.FileOpenService");
FileContents contents = service.openFileDialog(".", new String[] { "txt" });

JOptionPane.showMessageDialog(this, contents.getName());
if (contents != null)
{
InputStream in = contents.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(in));
String line;
while ((line = reader.readLine()) != null)
{
panel.append(line);
panel.append("\n");
}
}
catch (UnavailableServiceException e)
{
JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
JOptionPane.showMessageDialog(this, e);
}
}
```

```

    }

    /**
     * Zapisuje historię kalkulatora w pliku.
     */
    public void save()
    {
        try
        {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            PrintStream printOut = new PrintStream(out);
            printOut.print(panel.getText());
            InputStream data = new ByteArrayInputStream(out.toByteArray());
            FileSaveService service = (FileSaveService) ServiceManager
                .lookup("javax.jnlp.FileSaveService");
            service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
        }
        catch (UnavailableServiceException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
        catch (IOException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }
}
}

```

javax.jnlp.ServiceManager

- static String[] getServiceNames()
Zwraca nazwy wszystkich dostępnych usług.
- static Object lookup(string name)
Zwraca usługę o podanej nazwie.

javax.jnlp.BasicService

- URL getCodeBase()
Zwraca katalog zawierający kod aplikacji.
- boolean isWebBrowserSupported()
Zwraca wartość true, jeśli środowisko Web Start może uruchomić przeglądarkę.
- boolean showDocument(URL url)
Podejmuje próbę pokazania danego adresu URL w przeglądarce. Zwraca wartość true, jeśli żądanie kończy się powodzeniem.

javax.jnlp.FileContents

- InputStream getInputStream()
Zwraca strumień wejściowy do odczytu zawartości pliku.

- `OutputStream getOutputStream(boolean overwrite)`

Zwraca strumień wyjściowy do zapisu do pliku. Jeśli parametr `overwrite` ma wartość `true`, aktualna treść pliku jest nadpisywana.

- `String getName()`

Zwraca nazwę pliku (nie pełną ścieżkę katalogową).

- `boolean canRead()`

- `boolean canWrite()`

Zwraca wartość `true`, jeśli dany plik nadaje się do odczytu lub zapisu.

`javax.jnlp.FileOpenService`

- `FileContents openFileDialog(String pathHint, String[] extensions)`

- `FileContents[] openMultiFileDialog(String pathHint, String[] extensions)`

Wyświetla ostrzeżenie dla użytkownika i okno wyboru pliku. Zwraca deskryptory treści pliku lub plików wybranych przez użytkownika bądź wartość `null`, jeśli użytkownik nie wybrał żadnego pliku.

`javax.jnlp.FileSaveService`

- `FileContents saveFileDialog(String pathHint, String[] extensions, InputStream data, String nameHint)`

- `FileContents saveAsFileDialog(String pathHint, String[] extensions, FileContents data)`

Wyświetla ostrzeżenie dla użytkownika i okno wyboru pliku. Zapisuje dane i zwraca deskryptory treści pliku lub plików wybranych przez użytkownika bądź wartość `null`, jeśli użytkownik nie wybrał żadnego pliku.

`javax.jnlp.PersistenceService`

- `long create(URL key, long maxsize)`

Zapisuje dany klucz w pamięci trwałej. Zwraca maksymalny rozmiar przyznawany przez pamięć trwałą.

- `void delete(URL key)`

Usuwa dany klucz.

- `String[] getNames(URL url)`

Zwraca względne nazwy wszystkich kluczy, które zaczynają się od danego adresu URL.

- `FileContents get(URL key)`

Tworzy deskryptor treści, za pośrednictwem którego można modyfikować dane związane z danym kluczem. Jeśli dla danego klucza nie istnieje żaden wpis, zgłoszony jest wyjątek `FileNotFoundException`.

10.3. Aplety

Aplety to programy w języku Java dołączane do stron HTML. Strona HTML musi poinformować przeglądarkę, które aplety ma załadować oraz gdzie mają one być rozmieszczone. Jak nietrudno się domyślić, znacznik służący do wstawiania apletów musi dostarczać informacje dotyczące lokalizacji plików klas oraz samego apletu (jego rozmiaru, lokalizacji itd.). Przeglądarka pobiera pliki klas z internetu (lub katalogu na urządzeniu użytkownika) i automatycznie uruchamia aplet.

Na początku istnienia apletów jedyną przeglądarką, która je obsługiwała, była HotJava firmy Sun. Oczywiście znalazło się niewiele osób, które były skłonne używać oddzielnej przeglądarki dla jednej dodatkowej funkcji. Aplety zyskały prawdziwą popularność z chwilą dołączenia przez firmę Netscape maszyny wirtualnej Javy do przeglądarki Navigator. Niedługo później to samo zrobiła firma Microsoft w przeglądarce Internet Explorer. Niestety Microsoft podciągnęła skrzydła Netscape, opornie dodając obsługę starych wersji Javy w Internet Explorerze, aż w końcu całkiem tego zaniechała.

Problem ten rozwiązuje narzędzi Java Plug-in. Integruje się ono z przeglądarkami jako rozszerzenie, co umożliwia uruchamianie apletów przy wykorzystaniu zewnętrznego środowiska uruchomieniowego Javy.



Aby móc uruchamiać aplety opisywane w tym rozdziale, należy zainstalować najnowszą wersję narzędzia Java Plug-in i upewnić się, że przeglądarka jest połączona z wtyczką. Informacje na temat konfiguracji i pliki do pobrania można znaleźć na stronie <http://java.com>.

10.3.1. Prosty aplet

Aby tradycji stało się zadość, przerobimy program *NotHelloWorld* na aplet. Aplet to zwykła klasa Javy rozszerzająca klasę `java.applet.Applet`. Do implementacji apletów użyjemy Swinga. Wszystkie nasze aplety będą rozszerzały klasę `JApplet`, która jest nadklassą apletów Swing. Jak widać na rysunku 10.11, klasa `JApplet` jest bezpośrednią podklassą klasy `Applet`.



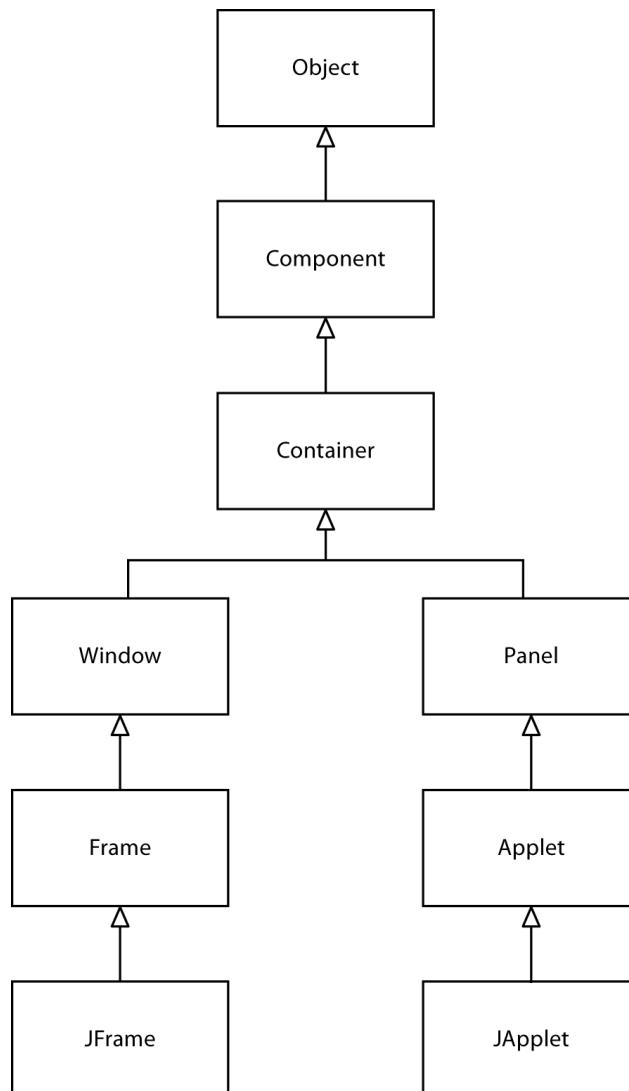
Jeśli aplet zawiera składniki Swing, należy rozszerzyć klasę `JApplet`. Komponenty Swing w zwykłym kontenerze `Applet` nie są poprawnie rysowane.

Na listingu 10.3 przedstawiona jest apletowa wersja programu „nie powitalnego”.

Zwrócić uwagę na duże podobieństwo do wersji z rozdziału 7. Ponieważ aplet działa w oknie przeglądarki, nie trzeba było definiować metody zamkijającej.

Rysunek 10.11.

Diagram dziedziczenia klasy Applet

**Listing 10.3.** applet/NotHelloWorld.java

```
package applet;

import java.awt.*;
import javax.swing.*;

/**
 * @version 1.23 2012-05-14
 * @author Cay Horstmann
 */
public class NotHelloWorld extends JApplet
{
    public void init()
    {
```

```

EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        JLabel label = new JLabel("To nie jest aplet „Witaj, świecie”",
        SwingConstants.CENTER);
        add(label);
    }
});
}
}

```

Uruchomienie powyższego apletu wymaga dwóch czynności:

- 1 Kompilacji plików źródłowych na pliki klas.
- 2 Utworzenia pliku HTML zawierającego informacje o lokalizacji plików klas oraz rozmiarze apletu.

Poniżej znajduje się jego zawartość:

```
<applet code="applet/NotHelloWorld.class" width="300" height="300">
</applet>
```

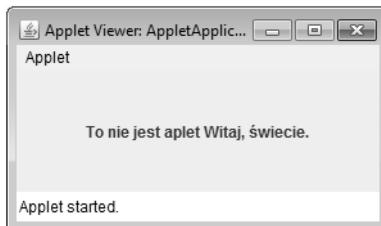
Dobrym pomysłem jest przetestowanie apletu we wchodzącej w skład pakietu JDK **przeglądarce apletów** (ang. *applet viewer*) przed otwarciem go w przeglądarce internetowej. Poniższe polecenie wiersza poleceń otwiera nasz aplet we wspomnianej przeglądarce:

```
appletviewer NotHelloWorldApplet.html
```

Argumentem narzędzia appletviewer jest nazwa pliku HTML, nie pliku klasy. Rysunek 10.12 przedstawia nasz aplet w przeglądarce apletów.

Rysunek 10.12.

Aplet
w przeglądarce
apletów



Apletów można także uruchamiać w środowisku programistycznym. W Eclipse należy kliknąć polecenie *Run/Run As/Java Applet*.

Przeglądarka apletów dobrze sprawdza się jako pierwszy etap testowania. Trzeba jednak w końcu uruchomić aplet w przeglądarce, aby sprawdzić, jak będzie się prezentował użytkownikowi. Przeglądarka apletów pokazuje sam aplet bez otaczającego go kodu HTML. Jeśli strona HTML zawiera kilka znaczników applet, przeglądarka otworzy kilka okien.

Aby obejrzeć swój aplet w przeglądarce, wystarczy załadować w niej odpowiednią stronę HTML (rysunek 10.13). Jeśli aplet nie pojawi się, należy zainstalować narzędzie Java Plug-in.

Rysunek 10.13.

Aplet
w przeglądarce



Jeśli w apletie zostaną wprowadzone jakieś zmiany i zostanie on raz jeszcze podany kompilacji, konieczne jest ponowne uruchomienie przeglądarki, aby załadowała nowe pliki klas. Samo odświeżenie strony nie spowoduje załadowania nowej wersji apletu. Bywa to kłopotliwe przy szukaniu błędów. Można uniknąć ponownego uruchamiania przeglądarki dzięki użyciu konsoli Javy. Należy uruchomić tę konsolę i wydać polecenie `x`, które czyści pamięć programu ładowającego klasy. Wtedy po odświeżeniu strony zostanie załadowana nowa wersja apletu. W systemie Windows należy otworzyć panel kontrolny Java Plug-in znajdujący się w *Panelu sterowania*. W systemie Linux należy użyć polecenia `jcontrol` i zażądać wyświetlenia panelu kontrolnego Javy. Konsola będzie się pojawiać za każdym razem, kiedy ładowany jest aplet.

10.3.1.1. Konwersja programów na aplety

Graficzne aplikacje w Javie można z łatwością przekonwertować na aplety. Cały kod dotyczący interfejsu użytkownika pozostaje bez zmian. Oto lista niezbędnych czynności:

1. Utwórz stronę HTML z odpowiednim znacznikiem wstawiającym aplet.
2. Utwórz podklasę klasy `JApplet`.
3. Usuń z aplikacji metodę `main`. Nie twórz ramki dla aplikacji, ponieważ będzie ona wyświetlana w oknie przeglądarki.
4. Przenieś kod inicjujący z konstruktora ramki do metody `init` apletu. Nie trzeba jawnie konstruować obiektu apletu — przeglądarka robi to automatycznie i wywołuje metodę `init`.
5. Usuń wywołanie metody `setSize`. W apletach za rozmiary odpowiadają parametry HTML `width` i `height`.
6. Usuń wywołanie metody `setDefaultCloseOperation`. Apletu nie można zamknąć — jego działanie kończy się w chwili zamknięcia przeglądarki.

7. Jeśli w programie znajduje się wywołanie metody `setTitle`, należy je usunąć. Apletów nie mają pasków tytułu (można oczywiście nadać tytuł samej stronie HTML za pomocą znacznika `title`).
8. Nie wywołuj metody `setVisible(true)`. Aplet jest wyświetlany automatycznie.

`java.applet.Applet 1.0`

■ `void init()`

Jest wywoływana przy pierwszym ładowaniu apletu. Metodę tę należy przesłonić i umieścić w niej cały kod inicjujący.

■ `void start()`

Należy przesłonić tę metodę i umieścić w niej kod, który ma być wykonywany **za każdym razem**, gdy użytkownik odwiedza stronę zawierającą ten aplet. Do typowych działań należy tu reaktywacja wątku.

■ `void stop()`

Należy przesłonić tę metodę i umieścić w niej kod, który ma być wykonywany **za każdym razem**, gdy użytkownik opuszcza stronę zawierającą ten aplet. Do typowych działań należy tu dezaktywacja wątku.

■ `void destroy()`

Metodę tę należy przedefiniować, wstawiając do niej kod wykonywany w momencie zamknięcia przeglądarki.

■ `void resize(int width, int height)`

Wymusza zmianę rozmiaru apletu. Byłaby to doskonała metoda, gdyby działała na stronach internetowych. Niestety obecnie nie działa w przeglądarkach, ponieważ zakłóca ich mechanizm rozkładu elementów na stronie.

10.3.2. Znacznik applet i jego atrybuty

Znacznik `applet` w najprostszej postaci może wyglądać następująco:

```
<applet code="NotHelloWorld.class" width="300" height="100">
```

Wartością atrybutu `code` jest nazwa pliku klasy, koniecznie z rozszerzeniem `.class`. Atrybuty `width` i `height` określają rozmiar okna apletu w pikselach. Koniec znacznika wyznacza znacznik zamykający `</applet>`. Tekst znajdujący się pomiędzy znacznikami `<applet>` i `</applet>` jest wyświetlany tylko wtedy, gdy przeglądarka nie może wyświetlić apletu. Atrybuty `code`, `width` i `height` są wymagane. Przy braku któregokolwiek z nich przeglądarka nie może wyświetlić apletu.

Wszystkie te informacje powinny się znajdować w kodzie strony internetowej, której minimalna treść może wyglądać następująco:

```
<html>
<head>
<title>NotHelloWorldApplet</title>
```

```
</head>
<body>
    <p>Poniższy wiersz tekstu jest wyświetlany pod patronatem Javy:</p>
    <applet code="NotHelloWorld.class" width="100" height="100">
        Gdyby Twoja przeglądarka obsługiwała Javę, w tym miejscu znajdowałby się applet.
    </applet>
</body>
</html>
```

Znacznik applet ma następujące atrybuty:

■ **width, height**

Atrybuty te są wymagane i określają szerokość i wysokość appletu w pikselach. W przeglądarce appletów wymiary te są traktowane jako początkowe, ale rozmiar każdego okna tej przeglądarki można zmienić. W przeglądarce internetowej **nie ma możliwości** zmiany rozmiaru appletu. Optymalny rozmiar appletu, tak aby wyglądał dobrze u każdego użytkownika, należy określić metodą prób i błędów.

■ **align**

Określa wyrównanie appletu. Wartości tego atrybutu są takie same jak atrybutu align znacznika img.

■ **vspace, hspace**

Określają liczbę pikseli nad i pod appletem (vspace) oraz po jego obu stronach (hspace).

■ **code**

Określa nazwę pliku klasy appletu. Nazwa ta jest traktowana względem katalogu podstawowego (patrz niżej) lub aktualnej strony, jeśli katalog podstawowy nie jest określony.

Ścieżka musi się zgadzać z pakietem, do którego należy klasa. Jeśli na przykład klasa appletu należy do pakietu com.mycompany, atrybut wygląda następująco code="com/mycompany/MyApplet.class". Można też stosować alternatywny zapis w postaci code="com.mycompany.MyApplet.class", ale w takim przypadku nie można stosować ścieżek bezwzględnych. Jeśli plik klasy znajduje się gdzieś indziej, należy użyć atrybutu codebase.

Atrybut code określa tylko nazwę klasy appletu. Oczywiście sam applet może zawierać także inne pliki klas. Kiedy przeglądarka załadowuje klasę zawierającą applet, zorientuje się, że potrzebne są dodatkowe klasy, i je również załadowuje.

Wymagany jest atrybut code lub object (zobacz poniżej).

■ **codebase**

Określa adres URL, pod którym należy szukać plików klas. Adres ten może być bezwzględny i prowadzić nawet do innego serwera. Najczęściej jednak stosuje się adresy względne do podkatalogów na serwerze. Jeśli na przykład struktura plików i katalogów jest następująca:

```

aDirectory/
  └─ MyPage.html
    myApplets/
      └─ MyApplet.class

```

Strona *MyPage.html* powinna zawierać następujący znacznik:

```
<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">
```

■ **archive**

Określa listę plików JAR zawierających klasy i inne zasoby apletu, które są pobierane z serwera przed jego załadowaniem. To znacznie przyspiesza proces ładowania, ponieważ pobranie jednego pliku JAR zawierającego kilka mniejszych innych plików wymaga tylko jednego żądania HTTP. Pliki JAR na liście są rozdzielane przecinkami:

```
<applet code="MyApplet.class"
        archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
        width="100" height="150">
```

■ **object**

Określa nazwę pliku zawierającego **serializowalny** obiekt apletu (**serializacja** obiektu polega na zapisie jego wszystkich pól w pliku — zagadnienie to opisujemy w rozdziale 1. drugiego tomu). Przed wyświetleniem apletu jego obiekt jest poddawany deserializacji z powrotem do pierwotnego stanu. Jeśli jest używany ten atrybut, **nie** jest wywoływana metoda *init* apletu, a metoda *start*.

Przed serializacją obiektu apletu należy wywołać jego metodę *stop*. W ten sposób można utworzyć trwałą przeglądarkę, która automatycznie przeładowuje aplet i przywraca je do takiego samego stanu, w którym były w chwili jej zamykania. Jest to zaawansowana technika, rzadko używana przez projektantów stron internetowych.

W każdym znaczniku *applet* musi się znajdować atrybut *code* lub *object*. Na przykład:

```
<applet object="MyApplet.ser" width="100" height="150">
```

■ **name**

Twórcy skryptów wykorzystują ten atrybut do odwoływania się do apletu w swoich skryptach. Przeglądarki Netscape i Internet Explorer zezwalają na wywoływanie metod apletów na stronie za pośrednictwem JavaScriptu.

Aby uzyskać dostęp do apletu z poziomu JavaScriptu, najpierw należy nadać mu nazwę:

```
<applet code="MyApplet.class" width="100" height="150" name="mine">
</applet>
```

Dzięki temu można się do niego odwoływać za pomocą zapisu *document.applets.nazwa apletu*. Na przykład:

```
var myApplet = document.applets.mine;
```

Dzięki integracji Javy i JavaScriptu w przeglądarkach Netscape i Internet Explorer można wywoływać metody apletu:

```
myApplet.init();
```

Atrybut name ma także kluczowe znaczenie w sytuacjach, kiedy dwa apletty znajdujące się na tej samej stronie mają się ze sobą bezpośrednio komunikować. Należy nadać nazwę każdemu apletowi. Łańcuch ten należy przekazać do metody getApplet z klasy AppletContext. Mechanizm ten, o nazwie **komunikacja między apletami** (ang. *inter-applet communication*), został opisany nieco dalej w tym rozdziale.



Na stronie <http://www.javaworld.com/javatips/jw-javatip80.html> Francis Lu wykorzystuje mechanizm komunikacji Javy z JavaScriptem do rozwiązania odwiecznego problemu zmiany rozmiaru apletu, na stałe ustawnionego przez atrybuty width i height. Jest to dobry przykład integracji tych dwóch języków.

■ alt

Obsługę Javy w przeglądarce można wyłączyć. Jeśli jakiś przewrażliwiony administrator to zrobi, nieszczęśni użytkownicy w miejscu apletu zobaczą tekst zawarty w atrybucie alt.

```
<applet code="MyApplet.class" width="100" height="150"
        alt="Włącz Javę, a zobacysz tutaj mój aplet.">
```

Jeśli przeglądarka w ogóle nie rozpoznaje apletów, czyli pochodzi z czasów prehistorycznych, ignoruje znaczniki applet i param. W takiej sytuacji zostanie wyświetlony tekst znajdujący się pomiędzy znacznikami <applet> i </applet>. Natomiast przeglądarki obsługujące apletty nie wyświetlają tego tekstu. Na przykład:

```
<applet code="MyApplet.class" width="100" height="150">
  Gdyby Twoja przeglądarka obsługiwała Javę, w tym miejscu byłby widoczny
  →mój aplet.
</applet>
```

10.3.3. Znacznik object

Znacznik object wchodzi w skład standardu HTML 4.0 i jest zalecany przez organizację W3C jako zastępnik znacznika applet. Ma on 35 atrybutów, z których większość związana jest z dynamicznym HTML-em (np. onkeydown). Atrybuty pozycjonujące, jak align i height, działają dokładnie tak samo jak w znaczniku applet. Kluczowym atrybutem znacznika object dla apletów jest classid. Określa on lokalizację obiektu. Oczywiście znacznik object może ładować różnego rodzaju obiekty, takie jak apletty Javy, komponenty ActiveX czy nawet Java Plug-in. Typ obiektu określa wartość atrybutu codetype. Na przykład dla apletów Javy wartość ta to application/java. Poniżej znajduje się znacznik object ładowający aplet:

```
<object
  codetype="application/java"
  classid="java:MyApplet.class"
  width="100" height="150">
```

Zauważmy, że za atrybutem classid może się znajdować atrybut codebase, który działa dokładnie tak samo jak w znaczniku applet.

10.3.4. Parametry przekazujące informacje do apletów

Podobnie jak aplikacje wykorzystują informacje podawane za pośrednictwem wiersza poleceń, aplety używają parametrów podawanych w plikach HTML. Służy do tego znacznik param i jego atrybuty. Wyobraźmy sobie, że chcemy, aby na stronie internetowej można było ustawić krój czcionki używanej w apletie. Można do tego użyć następującego kodu HTML:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
</applet>
```

Następnie wartość parametru pobieramy za pomocą metody getParameter z klasy Applet:

```
public class FontParamApplet extends JApplet
{
  public void init()
  {
    String fontName = getParameter("font");
    . . .
  }
  . . .
}
```



Metodę getParameter można wywoływać tylko w metodzie init apletu, nie w konstruktorze, ponieważ w chwili jego wykonywania parametry nie są jeszcze gotowe. Ponieważ układ większości bardziej rozbudowanych apletów jest zdeterminowany przez parametry, zalecamy zaniechanie używania konstruktorów w apletach. Cały kod inicjujący można umieścić w metodzie init.

Parametry są zawsze zwracane jako łańcuchy. Jeśli wymagana jest liczba, łańcuch trzeba przekonwertować na typ liczbowy. Służą do tego standardowe metody, takie jak parseInt z klasy Integer.

Na przykład kod HTML zawierający parametr size, który określa rozmiar czcionki, mógłby wyglądać następująco:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
  <param name="size" value="24"/>
</applet>
```

Poniższy fragment kodu demonstruje sposób odczytu parametru liczbowego:

```
public class FontParamApplet extends JApplet
{
  public void init()
  {
    String fontName = getParameter("font");
    int fontSize = Integer.parseInt(getParameter("size"));
```

```
    . . .
}
```



Przy porównywaniu wartości atrybutu name ze znacznika param z argumentem metody getParameter nie jest rozpoznawana wielkość liter.

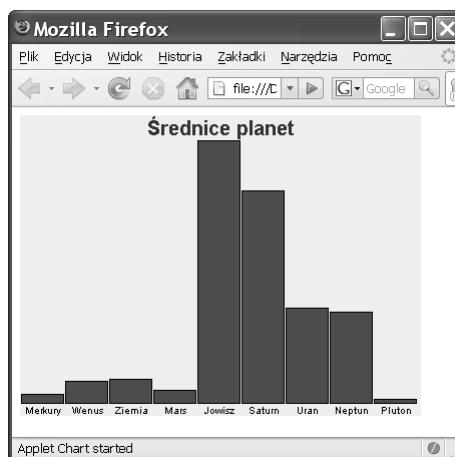
Poza upewnieniem się, że parametry w kodzie pasują, należy sprawdzić, czy parametr size został ustawiony, czy nie. Służy do tego prosty test na obecność wartości null. Na przykład:

```
int fontSize;
String sizeString = getParameter("size");
if (sizeString == null) fontSize = 12;
else fontSize = Integer.parseInt(sizeString);
```

Rysunek 10.14 przedstawia aplet rysujący wykres słupkowy, który w dużym stopniu wykorzystuje parametry.

Rysunek 10.14.

Aplet
wyświetlający
wykres



Aplet ten pobiera etykiety i wysokości słupków z wartości atrybutów znacznika param. Poniżej znajduje się kod HTML strony widocznej na rysunku 10.14.

```
<applet code="Chart.class" width="400" height="300">
  <param name="title" value="Średnice planet"/>
  <param name="values" value="9"/>
  <param name="name.1" value="Merkury"/>
  <param name="name.2" value="Wenus"/>
  <param name="name.3" value="Ziemia"/>
  <param name="name.4" value="Mars"/>
  <param name="name.5" value="Jowisz"/>
  <param name="name.6" value="Saturn"/>
  <param name="name.7" value="Uran"/>
  <param name="name.8" value="Neptun"/>
  <param name="name.9" value="Pluto"/>
  <param name="value.1" value="3100"/>
  <param name="value.2" value="7500"/>
  <param name="value.3" value="8000"/>
```

```

<param name="value.4" value="4200"/>
<param name="value.5" value="88000"/>
<param name="value.6" value="71000"/>
<param name="value.7" value="32000"/>
<param name="value.8" value="30600"/>
<param name="value.9" value="1430"/>
</applet>

```

Można było utworzyć w apletie tablicę łańcuchów i tablicę liczb, ale wykorzystanie mechanizmu parametrów ma dwie zalety. Po pierwsze, na jednej stronie można wyświetlić kilka kopii tego samego apletu, pokazujących różne wykresy — trzeba zastosować kilka znaczników applet z różnymi zestawami parametrów. Po drugie, można zmieniać dane przedstawione na wykresie. Wprawdzie średnice planet jeszcze przez jakiś czas się nie zmienią, ale wyobraźmy sobie, że na stronie przedstawiamy tygodniowy wykres sprzedaży. Stronę internetową łatwo się aktualizuje, ponieważ jest to czysty tekst. Edytowanie i kompilowanie plików Javy wymaga znacznie więcej wysiłku.

Istnieją nawet komercyjne komponenty JavaBean (tak zwane beany), które tworzą o wiele atrakcyjniejsze wykresy. Podając parametry takiemu zakupionemu komponentowi, nie trzeba nawet wiedzieć nic na temat tworzenia wykresów.

Listing 10.4 przedstawia kod źródłowy omawianego apletu rysującego wykres. Należy zauważyć, że metoda `init` pobiera parametry, a metoda `paintComponent` rysuje wykres.

Listing 10.4. chart/Chart.java

```

package chart;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class Chart extends JApplet
{
    public void init()
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                String v = getParameter("values");
                if (v == null) return;
                int n = Integer.parseInt(v);
                double[] values = new double[n];
                String[] names = new String[n];
                for (int i = 0; i < n; i++)
                {
                    values[i] = Double.parseDouble(getParameter("value." + (i + 1)));
                    names[i] = getParameter("name." + (i + 1));
                }
            }
        });
    }
}

```

```
        add(new ChartComponent(values, names, getParameter("title")));
    }
}
}

/**
 * Komponent rysujący wykres słupkowy.
 */
class ChartComponent extends JComponent
{
    private double[] values;
    private String[] names;
    private String title;

    /**
     * Tworzy obiekt typu ChartComponent.
     * @param v tablica wartości wykresu
     * @param n tablica nazw wartości
     * @param t tytuł wykresu
     */
    public ChartComponent(double[] v, String[] n, String t)
    {
        values = v;
        names = n;
        title = t;
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        // Obliczanie wartości minimalnej i maksymalnej.
        if (values == null) return;
        double minValue = 0;
        double maxValue = 0;
        for (double v : values)
        {
            if (minValue > v) minValue = v;
            if (maxValue < v) maxValue = v;
        }
        if (maxValue == minValue) return;

        int panelWidth = getWidth();
        int panelHeight = getHeight();

        Font titleFont = new Font("SansSerif", Font.BOLD, 20);
        Font labelFont = new Font("SansSerif", Font.PLAIN, 10);

        // Obliczanie szerokości tytułu.
        FontRenderContext context = g2.getFontRenderContext();
        Rectangle2D titleBounds = titleFont.getStringBounds(title, context);
        double titleWidth = titleBounds.getWidth();
        double top = titleBounds.getHeight();

        // Rysowanie tytułu.
        double y = -titleBounds.getY(); // wysokość
        double x = (panelWidth - titleWidth) / 2;
        g2.setFont(titleFont);
```

java.applet.Applet 1.0

- `public String getParameter(String name)`

Pobiera wartość parametru zdefiniowanego w znaczniku `param` na stronie internetowej ładującej applet. W łańcuchu `name` rozpoznawane są małe i wielkie litery.

- ## ■ public String getAppletInfo()

Tę metodę wielu programistów przedefiniowuje, aby zwracała łańcuch zawierający informacje o autorze, wersji i prawach autorskich do apletu. Informacje te należy podawać poprzez przesłonięcie tej metody w klasie apletu.

- `public String[][][] getParameterInfo()`

Tę metodę można przedefiniować, aby zwracała tablicę opcji znacznika param, obsługiwanych przez aplet. Każdy wiersz zawiera trzy pozycje: nazwę, typ i opis parametru. Na przykład:

```
"fps", "1-10", "ramek na sekundę"
"repeat", "boolean", "powtórzyć pętlę obrazu?"
"images", "url", "katalog zawierający obrazy"
```

10.3.5. Dostęp do obrazów i plików audio

W aplikacjach można używać obrazów i plików audio. W chwili pisania tego tekstu obsługiwane były następujące formaty plików graficznych: GIF, PNG i JPEG, oraz plików audio: AU, AIFF, WAV i MIDI. Animowane gify są także poprawnie obsługiwane.

Lokalizację obrazów i plików audio określa się za pomocą względnych adresów URL. Bazowy adres URL zazwyczaj sprawdza się za pomocą metody `getDocumentBase` lub `getCodeBase`. Pierwsza z wymienionych metod pobiera adres URL strony HTML zawierającej aplet, a druga adres URL katalogu bazowego kodu.



We wcześniejszych wersjach Javy metody te były źródłem wielu nieporozumień — zobacz błąd #4456393 na stronie <http://bugs.sun.com/bugdatabase/index.jsp>.

Bazowy adres URL i lokalizację pliku należy przekazać do metody `getImage` lub `getAudioClip`. Na przykład:

```
Image cat = getImage(getCodeBase(), "images/cat.gif");
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

Wyświetlania obrazów nauczyliśmy się w rozdziale 7. Jeśli chodzi o pliki audio, wystarczy wywołać metodę `play`. Metodę `play` z klasy `Applet` można nawet wywołać bez uprzedniego załadowania pliku audio.

```
play(getCodeBase(), "audio/meow.au");
```

java.applet.Applet 1.0

- `URL getDocumentBase()`

Pobiera adres URL strony zawierającej aplet.

- `URL getCodeBase()`

Pobiera adres URL katalogu bazowego z kodem, z którego ładowany jest aplet. Jest to albo bezwzględny adres URL katalogu wskazywanego przez atrybut `codebase`, albo adres pliku HTML, jeśli atrybut `codebase` nie istnieje.

- `void play(URL url)`

- `void play(URL url, String name)`

Pierwsza wersja odtwarza plik dźwiękowy znajdujący się pod podanym adresem URL. Druga tworzy ścieżkę względną wobec podanego adresu URL z podanego łańcucha. Jeśli pliku audio nie ma, nic się nie dzieje.

- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`

Pierwsza wersja odtwarza plik dźwiękowy znajdujący się pod podanym adresem URL. Druga tworzy ścieżkę względną wobec podanego adresu URL z podanego łańcucha. Jeśli plik audio nie istnieje, metody te zwracają wartość `null`.

- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`

Zwraca obiekt typu `Image` zawierający obraz wskazywany przez adres URL. Jeśli obraz nie istnieje, zwracana jest natychmiast wartość `null`. W przeciwnym przypadku zostaje uruchomiony osobny wątek ładowania obrazu.

10.3.6. Środowisko działania apletu

Apletty rezydują w przeglądarce internetowej lub przeglądarce apletów. Mogą one wysyłać do przeglądarek żądania dotyczące wykonania określonych działań, typu pobranie pliku audio, wyświetlenie komunikatu w pasku stanu lub otwarcie nowej strony. Przeglądarka może wykonać żądanie lub je zignorować. Jeśli na przykład aplet działający w przeglądarce apletów zgłosi żądanie otwarcia nowej strony, zostanie ono zignorowane.

Do komunikacji z przeglądarką aplet wykorzystuje metodę `getAppletContext`. Zwraca ona obiekt implementujący interfejs typu `AppletContext`. Konkretną implementację tego interfejsu można traktować jako ścieżkę komunikacyjną pomiędzy aplettem a przeglądarką. Poza metodami `getAudioClip` i `getImage` interfejs `AppletContext` zawiera także inne przydatne metody, które opisujemy w kilku kolejnych podrozdziałach.

10.3.6.1. Komunikacja pomiędzy apletami

Na jednej stronie internetowej może się znajdować kilka apletów. Jeśli wszystkie one pochodzą z jednej bazy kodowej, mogą się ze sobą komunikować. Jest to jednak zaawansowana technika, której nie używa się zbyt często.

Jeśli każdy aplet w pliku HTML ma określona nazwę w atrybucie `name`, za pomocą metody `getApplet` z interfejsu `AppletContext` można utworzyć odwołanie do tego apletu. Jeśli na przykład plik HTML zawiera poniższy znacznik:

```
<applet code="Chart.class" width="100" height="100" name="Chart1">
```

odwołanie do apletu daje poniższa instrukcja:

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

Do czego można wykorzystać takie odwołanie? Jeśli klasa `Chart` zawiera metodę przyjmującą nowe dane i ponownie rysującą wykres, metodę tę można wywołać, wykonując odpowiednie rzutowanie.

```
((Chart) chart1).setData(3, "Ziemia", 9000);
```

Listę wszystkich appletów znajdujących się na stronie można wyświetlić bez względu na to, czy mają one atrybut name, czy nie. Metoda getApplets zwraca **obiekt typu wyliczeniowego** (więcej informacji na temat obiektów wyliczeniowych znajduje się w rozdziale 13.). Poniższa pętla drukuje nazwy klas wszystkich appletów znajdujących się na stronie:

```
Enumeration<Applet> e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
    Applet a = e.nextElement();
    System.out.println(a.getClass().getName());
}
```

Applety z różnych stron nie mogą się ze sobą komunikować.

10.3.6.2. Wyświetlanie elementów w przeglądarce

Applety mają dostęp do dwóch miejsc w oknie przeglądarki: paska stanu i obszaru, na którym wyświetlane są strony. W obu przypadkach używa się metod z klasy AppletContext.

Aby wyświetlić tekst w pasku stanu, należy użyć metody showStatus. Na przykład:

```
showStatus("Ładowanie danych . . . proszę czekać");
```



Z naszego doświadczenia wynika, że zastosowanie metody showStatus jest ograniczone. Przeglądarka również używa paska stanu i w większości przypadków następuje dostarczony przez programistę tekst informacji typu *Applet running*. W związku z tym w pasku stanu można wyświetlać komunikaty typu *Ładowanie danych...*, ale nie takie, które są dla użytkownika ważne.

Aby zmusić przeglądarkę do otwarcia nowej strony, należy użyć metody showDocument, którą można wywołać na kilka sposobów. Najprostsze wywołanie polega na podaniu jako argumentu adresu URL strony, która ma zostać otwarta:

```
URL u = new URL("http://java.sun.com/index.html");
getAppletContext().showDocument(u);
```

Jednak takie wywołanie może być problematyczne, ponieważ nowa strona zastępuje aktualną, co powoduje zniknięcie appletu. Aby wrócić do appletu, konieczne jest naciśnięcie przycisku *Wstecz* w przeglądarce.

Aby nowa strona została otwarta w nowym oknie, należy metodzie showDocument podać dodatkowy parametr (zobacz tabela 10.2). Łąćuch _blank wymusza otwarcie dokumentu w nowym oknie. Co ciekawsze, korzystając z ramek HTML, można podzielić okno na kilka części o unikatowych nazwach, w których będą wyświetlane dokumenty żądane przez applet również znajdujący się w jednej z ramek. Przykładowy kod prezentujemy w kolejnym podrozdziale.



Przeglądarka appletów nie wyświetla stron internetowych. Metoda showDocument jest przez nią ignorowana.

Tabela 10.2. Metoda showDocument

Parametr określający cel	Lokalizacja
_self lub brak	Bieżąca ramka
_parent	Ramka nadzędna
_top	Ramka najwyższego rzędu
_blank	Nowe okno najwyższego rzędu, bez nazwy
Dowolny inny łańcuch	Ramka o podanej nazwie; jeśli nie ma ramki o takiej nazwie, otwierane jest nowe okno o takiej nazwie

java.applet.Applet 1.2

- `public AppletContext getAppletContext()`

Tworzy dostęp do środowiska przeglądarki, w której działa aplet. Przy użyciu tych informacji można kontrolować większość przeglądarek.

- `void showStatus(String msg)`

Pokazuje podany łańcuch na pasku stanu przeglądarki internetowej.

java.applet.AppletContext 1.0

- `Enumeration<Applet> getApplets()`

Zwraca wyliczenie (zobacz rozdział 13.) wszystkich apletów znajdujących się w tym samym środowisku, czyli na jednej stronie internetowej.

- `Applet getApplet(String name)`

Zwraca aplet o podanej nazwie znajdujący się w bieżącym kontekście. Zwraca wartość `null`, jeśli aplet nie istnieje. Przeszukiwana jest tylko bieżąca strona internetowa.

- `void showDocument(URL url)`

- `void showDocument(URL url, String target)`

Otwiera nową stronę internetową w przeglądarce. Pierwsza wersja zastępuje starszą stroną nową. Druga otwiera nową stronę w miejscu określonym przez parametr `target` (tabela 10.2).

10.4. Zapisywanie preferencji użytkownika

Użytkownicy oczekują, że wszystkie dokonane przez nich ustawienia zostaną zapisane i zastosowane przy ponownym uruchamianiu aplikacji. Najpierw zajmiemy się prostą techniką opartą na zapisywaniu informacji konfiguracyjnych w plikach własności, które były kiedyś stosowane w Javie. Następnie przejdziemy do opisu niezwykle funkcjonalnego API zarządzania preferencjami.

10.4.1. Mapy własności

Mapy własności (ang. *property map*) to struktury danych przechowujące pary klucz – wartość, które często znajdują zastosowanie jako przechowalnie danych konfiguracyjnych aplikacji. Każda taka mapa ma trzy cechy:

- Klucze i wartości są łańcuchami.
- Można ją łatwo zapisać w pliku i załadować z niego.
- Istnieje druga tabela przechowująca wartości domyślne.

Klasa odpowiedzialna za implementację map własności nosi nazwę `Properties`.

Jak wiadomo, mapy własności znajdują zastosowanie w określaniu opcji konfiguracyjnych programów. Na przykład:

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Witaj, Świecie!");
```

Do zapisania takiej listy własności w pliku należy użyć metody `store`. My zapiszemy naszą mapę w pliku o nazwie `program.properties`. Drugi argument metody `store` to komentarz, który jest dodawany do pliku.

```
FileOutputStream out = new FileOutputStream("program.properties");
settings.store(out, "Ustawienia programu");
```

W pliku zostaną zapisane następujące dane:

```
#Ustawienia programu
#Mon Apr 30 07:22:52 2007
width=200
title=Witaj, Świecie!
```

Do ładowania plików ustawień służą następujące instrukcje:

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

Istnieje zwyczaj przechowywania ustawień programu w podkatalogu głównego katalogu użytkownika. Nazwa tego katalogu zazwyczaj zaczyna się od kropki — w systemie Unix konwencja taka oznacza, że katalog jest katalogiem systemowym ukrytym przed użytkownikiem. W naszym przykładowym programie stosujemy się do tej konwencji.

Do sprawdzenia katalogu głównego użytkownika można wykorzystać metodę `System.getProperty`, która — tak się składa — wykorzystuje obiekt typu `Properties` do zapisu danych systemowych. Katalog główny ma klucz `user.home`. Istnieje także metoda pozwalająca odczytać pojedynczy klucz:

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

Dobrze jest na wszelki wypadek dostarczyć zestaw ustawień domyślnych dla programu. Klasa `Properties` dysponuje dwoma mechanizmami pozwalającymi określić ustawienia domyślne. Po pierwsze, można utworzyć łańcuch, który będzie stosowany domyślnie za każdym razem, kiedy dany klucz nie zostanie znaleziony.

```
String title = settings.getProperty("title", "Domyślny tytuł");
```

Jeśli w mapie właściwości znajduje się właściwość `title`, parametr `title` zostanie ustawiony na jej łańcuch. W przeciwnym przypadku parametr ten przyjmie wartość `Domyślny tytuł`.

Po drugie, jeśli wpisywanie wartości domyślnej w każdym wywołaniu metody `getProperty` okaże się zbyt żmudne, wszystkie ustawienia domyślne można umieścić w drugorzędnnej mapie właściwości dostarczanej następnie w konstruktorze mapy głównej.

```
Properties defaultSettings = new Properties();
defaultSettings.put("width", "300");
defaultSettings.put("height", "200");
defaultSettings.put("title", "Domyślny tytuł");
.
.
Properties settings = new Properties(defaultSettings);
```

Można nawet dostarczyć domyślne ustawienia dla ustawień domyślnych. Wystarczy tylko utworzyć kolejną mapę właściwości i przekazać ją do konstruktora `defaultSettings`. Nie jest to jednak często spotykane rozwiązanie.

Listing 10.5 przedstawia program zapisujący i ładujący ustawienia programu. Zapamiętywane są położenie, rozmiar i tytuł ramki. Wygląd programu można dostosować **według własnego uznania**, edytując plik o nazwie `.corejava/program.properties` znajdujący się w katalogu głównym.

Listing 10.5. properties/PropertiesTest.java

```
package properties;

import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import java.util.Properties;

import javax.swing.*;

/**
 * Program testujący mechanizm właściwości. Ten program zapamiętuje położenie, rozmiar i tytuł ramki.
 * @version 1.00 2007-04-29
 * @author Cay Horstmann
 */
public class PropertiesTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
```

```

        PropertiesFrame frame = new PropertiesFrame();
        frame.setVisible(true);
    }
}
}

<**
 * Ramka pobierająca dane dotyczące położenia i rozmiaru z pliku własności oraz aktualizująca ten plik
 * w momencie zamykania programu
 */
class PropertiesFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private File propertiesFile;
    private Properties settings;

    public PropertiesFrame()
    {
        // Pobranie informacji o położeniu, rozmiarze i tytule z pliku własności

        String userDir = System.getProperty("user.home");
        File propertiesDir = new File(userDir, ".corejava");
        if (!propertiesDir.exists()) propertiesDir.mkdir();
        propertiesFile = new File(propertiesDir, "program.properties");

        Properties defaultSettings = new Properties();
        defaultSettings.put("left", "0");
        defaultSettings.put("top", "0");
        defaultSettings.put("width", "" + DEFAULT_WIDTH);
        defaultSettings.put("height", "" + DEFAULT_HEIGHT);
        defaultSettings.put("title", "");

        settings = new Properties(defaultSettings);

        if (propertiesFile.exists()) try
        {
            FileInputStream in = new FileInputStream(propertiesFile);
            settings.load(in);
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }

        int left = Integer.parseInt(settings.getProperty("left"));
        int top = Integer.parseInt(settings.getProperty("top"));
        int width = Integer.parseInt(settings.getProperty("width"));
        int height = Integer.parseInt(settings.getProperty("height"));
        setBounds(left, top, width, height);

        // Jeśli nie ma tytułu, użytkownik zostanie poproszony o jego podanie

        String title = settings.getProperty("title");
        if (title.equals("")) title = JOptionPane.showInputDialog("Wpisz tytuł ramki:");
    }
}

```

```
if (title == null) title = "";
setTitle(title);

addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent event)
    {
        settings.put("left", "" + getX());
        settings.put("top", "" + getY());
        settings.put("width", "" + getWidth());
        settings.put("height", "" + getHeight());
        settings.put("title", getTitle());
        try
        {
            FileOutputStream out = new FileOutputStream(propertiesFile);
            settings.store(out, "Ustawienia programu");
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
        System.exit(0);
    }
});
```



 Obiekty typu `Properties` są zwykłymi tablicami pozbawionymi struktury hierarchicznej. Programiści często tworzą namiastkę hierarchii, odpowiednio nazywając klucze, np. `window.main.color`, `window.main.title` itd. Jednak klasa `Properties` nie zawiera żadnych metod wspomagających organizację takich hierarchii. Do przechowywania skomplikowanych informacji konfiguracyjnych lepiej używać klasy `Preferences`, która została opisana w kolejnym podrozdziale.

java.util.Properties 1.0

- **Properties()**
Tworzy pustą mapę właściwości.
 - **Properties(Properties defaults)**
Tworzy mapę właściwości z zestawem ustawień domyślnych.
Parametr: `defaults` Wartości domyślne
 - **String getProperty(String key)**
Pobiera mapę właściwości. Zwraca łańcuch skojarzony z kluczem `key` lub łańcuch skojarzony z kluczem `key` w mapie domyślnej, jeśli nie ma go w aktualnej mapie, albo wartość `null`, jeśli nie zostanie on znaleziony także w tej drugiej mapie.
Parametr: `key` Klucz, którego wartość ma zostać pobrana.
 - **String getProperty(String key, String defaultValue)**

Pobiera własność z domyślną wartością, jeśli klucz key nie zostanie znaleziony. Zwraca łańcuch skojarzony z kluczem key lub łańcuch domyślny, jeśli nie ma go w tablicy.

Parametry: key Klucz, którego wartość ma zostać pobrana.
defaultValue Wartość zwracana, jeśli dany klucz nie istnieje.

- void load(InputStream in) throws IOException

Ładuje mapę własności ze strumienia wejściowego.

Parametr: in Strumień wejściowy

- void store(OutputStream out, String header) **1.2**

Zapisuje mapę własności w strumieniu wyjściowym.

Parametry: out Strumień wyjściowy
header Nagłówek umieszczany w pierwszym wierszu zapisywanego pliku

java.lang.System **1.0**

- Properties getProperties()

Pobiera wszystkie właściwości systemowe. Jeśli aplikacja nie ma uprawnień do pobierania wszystkich właściwości systemowych, generowany jest wyjątek zabezpieczeń.

- String getProperty(String key)

Pobiera właściwość systemową opatrzoną podanym kluczem. Jeśli aplikacja nie ma uprawnień do pobierania tej właściwości systemowej, generowany jest wyjątek zabezpieczeń. Poniższe właściwości można zawsze pobierać:

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```



Nazwy wolno dostępnych właściwości systemowych można znaleźć w pliku *security/java.policy* znajdującym się w katalogu środowiska uruchomieniowego Javy.

10.4.2. API Preferences

Klasa `Properties`, mimo iż umożliwia zapisywanie i odczytywanie danych konfiguracyjnych w prosty sposób, ma kilka wad:

- Nie zawsze mamy możliwość zapisania plików konfiguracyjnych w katalogu głównym użytkownika, ponieważ niektóre systemy operacyjne nie znają koncepcji katalogu głównego.
- Nie istnieje standardowa konwencja nazwywania plików konfiguracyjnych, co wiąże się z ryzykiem wystąpienia konfliktów nazw, jeśli użytkownik zainstaluje kilka aplikacji Java.

Niektóre systemy operacyjne mają centralne repozytorium, w którym przechowują dane konfiguracyjne. Najlepszym przykładem takiego repozytorium jest rejestr w systemie Microsoft Windows. Klasa `Preferences` pozwala utworzyć takie repozytorium niezależnie od platformy. W systemie Windows klasa ta wykorzystuje rejestr. W systemie Linux informacje te są zapisywane w lokalnym systemie plików. Oczywiście implementacja repozytorium nie ma tajemnic dla programisty używającego klasy `Preferences`.

Repozytorium `Preferences` ma strukturę drzewiastą z nazwami ścieżek do węzłów typu `/com/mycompany/myapp`. Podobnie jak w przypadku pakietów, konfliktów nazw ścieżek unika się, stosując odwrócone nazwy domen. Projektanci tego API zalecają nawet, aby ścieżkom do węzłów konfiguracyjnych nadawać takie same nazwy jak pakietom używanym w programie.

Każdy węzeł w repozytorium ma oddzielną tablicę par klucz – wartość, w której można przechowywać łańcuchy, liczby i tablice bajtów. Nie ma możliwości zapisywania obiektów serializowanych, ponieważ projektanci uznali, że format ten jest zbyt ulotny do długoterminowego przechowywania. Oczywiście ci, którzy się z tym nie zgadzają, mogą zapisywać serializowane obiekty w tablicach bajtów.

Większą elastyczność zapewniają dodatkowe równoległe drzewa. Każdy użytkownik programu ma własne drzewo i dodatkowe drzewo systemowe, które przechowuje ustawienia wspólne wszystkich użytkowników. Odpowiednie drzewo ustawień jest pobierane przez klasę `Preferences` przy użyciu pojęcia bieżącego użytkownika, rozumianego zgodnie z systemem operacyjnym.

Aby uzyskać dostęp do węzła drzewa, należy zacząć od użytkownika `root` lub katalogu systemowego `root`:

```
Preferences root = Preferences.userRoot();
```

lub

```
Preferences root = Preferences.systemRoot();
```

Następnie uzyskujemy dostęp do węzła. Można ograniczyć się do podania jego ścieżki:

```
Preferences node = root.node("/com/mycompany/myapp");
```

Z pomocą wygodnego skrótu można pobrać węzeł, którego ścieżka jest taka sama jak nazwa pakietu klasy. Wystarczy pobrać obiekt tej klasy i zastosować poniższe wywołanie:

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());  
lub
```

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass());
```

Zazwyczaj obj jest referencją this.

Mając węzeł, można uzyskać dostęp do jego tablicy par klucz – wartość za pomocą następujących metod:

```
String get(String key, String defval)  
int getInt(String key, int defval)  
long getLong(String key, long defval)  
float getFloat(String key, float defval)  
double getDouble(String key, double defval)  
boolean getBoolean(String key, boolean defval)  
byte[] getByteArray(String key, byte[] defval)
```

Należy pamiętać, że przy odczytywaniu informacji trzeba dostarczyć wartość domyślną, na wypadek gdyby w repozytorium brakowało jakichś danych. Wartości domyślne są wymagane z kilku powodów. Danych może brakować, ponieważ użytkownik nigdy nie ustawił danej opcji. Niektóre ograniczone platformy mogą nie mieć repozytorium, a urządzenia przenośne mogą być tymczasowo odłączone od repozytorium.

Do zapisu danych w repozytorium służą metody put:

```
put(String key, String value)  
putInt(String key, int value)
```

i tak dalej.

Listę wszystkich kluczy zapisanych w węźle można uzyskać za pomocą metody String[] keys. Nie ma jednak obecnie sposobu na sprawdzenie typu wartości określonego klucza.

Centralne repozytoria, takie jak rejestr w systemie Windows, zazwyczaj cierpią z dwóch powodów:

- Z czasem zamieniają się w śmietnisko pełne przestarzałych informacji.
- Dane konfiguracyjne płączą się w repozytorium, przez co trudno jest je przenieść na inną platformę.

Klasa Preferences ma rozwiązanie drugiego problemu. Można wyeksportować ustawienia poddrzewa (lub — rzadziej — jednego węzła) za pomocą poniższych metod:

```
void exportSubtree(OutputStream out)  
void exportNode(OutputStream out)
```

Dane są zapisywane w formacie XML. Można je zainportować do innego repozytorium za pomocą następującego wywołania:

```
void importPreferences(InputStream in)
```

Poniżej znajduje się zawartość przykładowego pliku:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
<root type="user">
<map/>
<node name="com">
<map/>
<node name="horstmann">
<map/>
<node name="corejava">
<map>
<entry key="left" value="11"/>
<entry key="top" value="9"/>
<entry key="width" value="453"/>
<entry key="height" value="365"/>
<entry key="title" value="Witaj, Świecie!"/>
</map>
</node>
</node>
</node>
</root>
</preferences>

```

Jeśli program wykorzystuje klasę Preferences, należy umożliwić użytkownikowi import i eksport ustawień, co ułatwia przenoszenie ustawień na inny komputer. Program na listingu 10.6 demonstruje omówioną technikę. Zapisuje on położenie, rozmiar i tytuł głównego okna. Po zamknięciu i ponownym uruchomieniu programu okno będzie wyglądało dokładnie tak samo jak przed zamknięciem programu.

Listing 10.6. preferences/PreferencesTest.java

```

package preferences;

import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import java.util.prefs.*;
import javax.swing.*;

/**
 * Program testujący ustawianie preferencji. Zapamiętuje położenie, rozmiar i tytuł ramki.
 * @version 1.02 2007-06-12
 * @author Cay Horstmann
 */
public class PreferencesTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                PreferencesFrame frame = new PreferencesFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

```

```

        }

    }

    /**
     * Ramka pobierająca dane dotyczące położenia i rozmiaru z preferencji użytkownika oraz aktualizująca
     * preferencje w momencie zamykania programu
     */
    class PreferencesFrame extends JFrame
    {
        private static final int DEFAULT_WIDTH = 300;
        private static final int DEFAULT_HEIGHT = 200;

        public PreferencesFrame()
        {
            // Sprawdzanie położenia, rozmiaru i tytułu w preferencjach

            Preferences root = Preferences.userRoot();
            final Preferences node = root.node("/com/horstmann/corejava");
            int left = node.getInt("left", 0);
            int top = node.getInt("top", 0);
            int width = node.getInt("width", DEFAULT_WIDTH);
            int height = node.getInt("height", DEFAULT_HEIGHT);
            setBounds(left, top, width, height);

            // Jeśli nie ma tytułu, użytkownik zostanie poproszony o jego podanie

            String title = node.get("title", "");
            if (title.equals("")) title = JOptionPane.showInputDialog("Wpisz tytuł
                ramki:");
            if (title == null) title = "";
            setTitle(title);

            // Utworzenie okna wyboru plików wyświetlającego pliki XML

            final JFileChooser chooser = new JFileChooser();
            chooser.setCurrentDirectory(new File("."));

            // Akceptacja wszystkich plików z rozszerzeniem xml
            chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
            {
                public boolean accept(File f)
                {
                    return f.getName().toLowerCase().endsWith(".xml") || f.isDirectory();
                }

                public String getDescription()
                {
                    return "XML files";
                }
            });

            // menu
            JMenuBar menuBar = new JMenuBar();
            setJMenuBar(menuBar);
            JMenu menu = new JMenu("Plik");
            menuBar.add(menu);

            JMenuItem exportItem = new JMenuItem("Eksport ustawień");

```

```
menu.add(exportItem);
exportItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        if (chooser.showSaveDialog(PreferencesFrame.this) ==
            JFileChooser.APPROVE_OPTION)
        {
            try
            {
                OutputStream out = new
                    FileOutputStream(chooser.getSelectedFile());
                node.exportSubtree(out);
                out.close();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}):
JMenuItem importItem = new JMenuItem("Import ustawień");
menu.add(importItem);
importItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        if (chooser.showOpenDialog(PreferencesFrame.this) ==
            JFileChooser.APPROVE_OPTION)
        {
            try
            {
                InputStream in = new
                    FileInputStream(chooser.getSelectedFile());
                Preferences.importPreferences(in);
                in.close();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}):
JMenuItem exitItem = new JMenuItem("Zamknij");
menu.add(exitItem);
exitItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        node.putInt("left", getX());
        node.putInt("top", getY());
        node.putInt("width", getWidth());
        node.putInt("height", getHeight());
```

```
        node.put("title", getTitle());
        System.exit(0);
    }
}
}
```

java.util.prefs.Preferences 1.4

- Preferences userRoot()

Zwraca węzeł preferencji root użytkownika programu.

- Preferences systemRoot()

Zwraca węzeł preferencji root systemu.

- Preferences node(String path)

Zwraca węzeł, do którego można uzyskać dostęp z bieżącego węzła za pośrednictwem podanej ścieżki path. Jeśli ścieżka jest bezwzględna (czyli zaczyna się od znaku /), szukanie węzła zaczyna się od korzenia drzewa zawierającego ten węzeł preferencji. Jeśli węzeł w podanej ścieżce nie istnieje, zostanie utworzony.

- Preferences userNodeForPackage(Class c1)

- Preferences systemNodeForPackage(Class c1)

Zwraca węzeł w bieżącym drzewie użytkownika lub drzewo systemowe, którego ścieżka bezwzględna węzła odpowiada nazwie pakietu klasy c1.

- String[] keys()

Zwraca wszystkie klucze należące do węzła.

- String get(String key, String defval)

- int getInt(String key, int defval)

- long getLong(String key, long defval)

- float getFloat(String key, float defval)

- double getDouble(String key, double defval)

- boolean getBoolean(String key, boolean defval)

- byte[] getByteArray(String key, byte[] defval)

Zwraca wartość skojarzoną z danym kluczem lub podaną wartość domyślną, jeśli z kluczem nie jest skojarzona żadna wartość lub skojarzona wartość jest nieprawidłowego typu, lub magazyn preferencji nie jest dostępny.

- void put(String key, String value)

- void putInt(String key, int value)

- void putLong(String key, long value)

- void putFloat(String key, float value)

- `void putDouble(String key, double value)`
- `void putBoolean(String key, boolean value)`
- `void putByteArray(String key, byte[] value)`

Zapisuje parę klucz – wartość w węźle.
- `void exportSubtree(OutputStream out)`

Zapisuje preferencje węzła i jego potomków w określonym strumieniu.
- `void exportNode(OutputStream out)`

Zapisuje preferencje węzła (ale nie jego potomków) w określonym strumieniu.
- `void importPreferences(InputStream in)`

Importuje preferencje zawarte w określonym strumieniu.

Na tym zakończymy opis technik przygotowywania aplikacji w Javie do użytku. W następnym rozdziale nauczyszmy się wykorzystywać wyjątki do określania zachowania programu w sytuacjach awaryjnych. Dodatkowo opisujemy techniki testowania i znajdowania błędów, które pozwalają pozbyć się wielu usterek jeszcze przed uruchomieniem programu.

11

Wyjątki, dzienniki, asercje i debugowanie

W tym rozdziale:

- Obsługa błędów
- Obsługa wyjątków
- Wskazówki dotyczące stosowania wyjątków
- Asercje
- Dzienniki
- Wskazówki dotyczące debugowania
- Wskazówki dotyczące debugowania programów z GUI
- Praca z debuggerem

Gdybyśmy żyli w idealnym świecie, użytkownicy zawsze wprowadzaliby prawidłowe dane, wybierane pliki zawsze by istniały, a kod byłby pozbawiony wszelkich błędów. Programy prezentowane do tej pory są zbudowane tak, jakbyśmy żyli w takim właśnie wyimaginowanym miejscu. Nadeszła jednak pora na poznanie technik programistycznych służących do pracy w świecie rzeczywistym, pełnym niepoprawnych danych i błędnego kodu.

Jeśli podczas pracy z programem wystąpi błąd, który zniweczy jakąś część pracy użytkownika, może on już nigdy nie wrócić do tego programu. Te nieprzyjemne sytuacje mogą być spowodowane niedopatrzeniem programisty lub rozmaitymi czynnikami zewnętrznymi. W takiej sytuacji program musi przynajmniej:

- poinformować użytkownika o błędzie,
- zapisać dotyczczącową pracę,
- pozwolić użytkownikowi na eleganckie zakończenie pracy.

Sytuacje wyjątkowe, które mogą wywołać awarię programu (jak na przykład podanie nieprawidłowych danych na wejściu), są w Javie rozwiązywane za pomocą techniki **obsługi wyjątków**. Obsługa wyjątków w Javie wygląda podobnie jak w językach C++ i Delphi. Pierwsza część tego rozdziału opisuje wyjątki w Javie.

Testowanie polega na przeprowadzaniu wielu różnych testów, mających na celu sprawdzenie, czy program działa zgodnie z przewidywaniami. Testy te mogą jednak zabierać dużo czasu, a po zakończeniu testowania zazwyczaj są niepotrzebne. Można je wtedy usunąć i w razie potrzeby wkleić z powrotem, kiedy znów będą potrzebne. Jest to jednak żmudne zajęcie. Druga część rozdziału została poświęcona selektywnemu uruchamianiu testów za pomocą asercji.

Komunikacja z użytkownikiem w wyjątkowej sytuacji nie zawsze jest możliwa. Często też nie można normalnie zamknąć programu. Wtedy dobrym pomysłem jest zapisanie danych zdarzenia do późniejszej analizy. Trzecia część tego rozdziału została poświęcona technikom zapisu danych do dziennika.

Na zakończenie dajemy kilka wskazówek na temat wydobywania pożytecznych informacji z działającego programu oraz używania debugera w zintegrowanym środowisku programistycznym.

11.1. Obsługa błędów

Wyobraźmy sobie, że w trakcie działania programu wystąpił błąd. Mógł zostać spowodowany przez nieprawidłowe dane w pliku, wadliwe połączenie sieciowe lub (musimy to powiedzieć) użycie nieprawidłowego indeksu tablicy bądź referencji, która nie została jeszcze przypisana do żadnego obiektu. Użytkownicy oczekują, że w razie wystąpienia błędu ich programy będą się zachowywać racjonalnie. Jeśli program nie może ukończyć zadania z powodu błędu, powinien wykonać jedną z dwóch czynności:

- powrócić do bezpiecznego stanu i pozwolić użytkownikowi wydać inne polecenia;
- pozwolić użytkownikowi zapisać całą pracę i zamknąć program.

To może być trudne, ponieważ procedury wykrywające (lub powodujące) błędy zazwyczaj znajdują się z dala od kodu, który mógłby przywrócić dane do bezpiecznego stanu, lub procedury mogących zapisać pracę i zamknąć program. Obsługa wyjątków polega na przekazywaniu kontroli z miejsca wystąpienia błędu do procedur, które mogą rozwiązać ten problem. Aby obsłużyć sytuacje wyjątkowe w programie, należy przewidzieć, jakiego rodzaju błędy i problemy mogą wystąpić. Co trzeba rozważyć?

- **Błędy danych wejściowych.** Poza robieniem literówek użytkownicy czasami nie trzymają się ogólnych zaleceń i chodzą własnymi ścieżkami. Założymy na przykład, że użytkownik chce przejść pod adres URL o niepoprawnej składni. Program powinien to sprawdzić, a jeśli tego nie zrobi, wystąpią problemy z warstwą sieciową.
- **Błędy urządzeń.** Urządzenia nie zawsze działają tak, jak powinny. Drukarka może być wyłączona, a strona internetowa, z której drukujemy, może być chwilowo niedostępna. Urządzenia często zawodzą w trakcie pracy. Na przykład w drukarce w czasie drukowania może się skończyć papier.

- **Ograniczenia fizyczne.** Może się skończyć miejsce na dysku.
- **Błędy w kodzie.** Jedna z metod może nie działać prawidłowo. Może na przykład zwracać niepoprawne wyniki albo nieprawidłowo używać innych metod. Inne przykłady błędów spowodowanych przez kod to nieprawidłowe obliczenie indeksu w tablicy i próba dostępu do nieistniejącego elementu tablicy mieszającej czy próba pobrania elementu z pustego stosu.

Typową reakcją na błąd w metodzie jest zwrot specjalnego kodu błędu, który następnie jest przekazywany do analizy metodzie wywołującej. Na przykład metody odczytujące dane z plików często zwracają wartość `-1` zamiast standardowego znaku końca pliku. Jest to doskonały sposób na poradzenie sobie z wieloma sytuacjami wyjątkowymi. Inna często spotykana wartość zwrotna oznaczająca błąd to referencja `null`.

Niestety nie zawsze da się zwrócić kod błędu. Czasami odróżnienie poprawnych danych od niepoprawnych może być trudne. Metoda zwracająca liczby całkowite nie może zwrócić wartości `-1` jako błąd, ponieważ wartość ta może być poprawna.

W związku z tym, jeśli metoda nie może normalnie ukończyć swojego działania, może wybrać alternatywny sposób wybrnięcia z sytuacji (pisaliśmy o tym w rozdziale 5.). Wtedy nie zwraca żadnej wartości, tylko **wyrzuca** obiekt zawierający informacje o błędzie. Zauważmy, że metoda jestkończona natychmiast — nie zwraca żadnej wartości. Ponadto działanie programu nie jest wznowiane od miejsca, w którym metoda została wywołana. Zamiast tego mechanizm obsługi wyjątków zaczyna szukać **procedury obsługi błędów**, która potrafi rozwiązać dany problem.

Wyjątki mają własną składnię oraz wchodzą w skład specjalnej hierarchii dziedziczenia. Najpierw zajmiemy się tą składnią, a następnie udzielimy kilku wskazówek dotyczących efektywnego wykorzystania tej właściwości języka.

11.1.1. Klasifikacja wyjątków

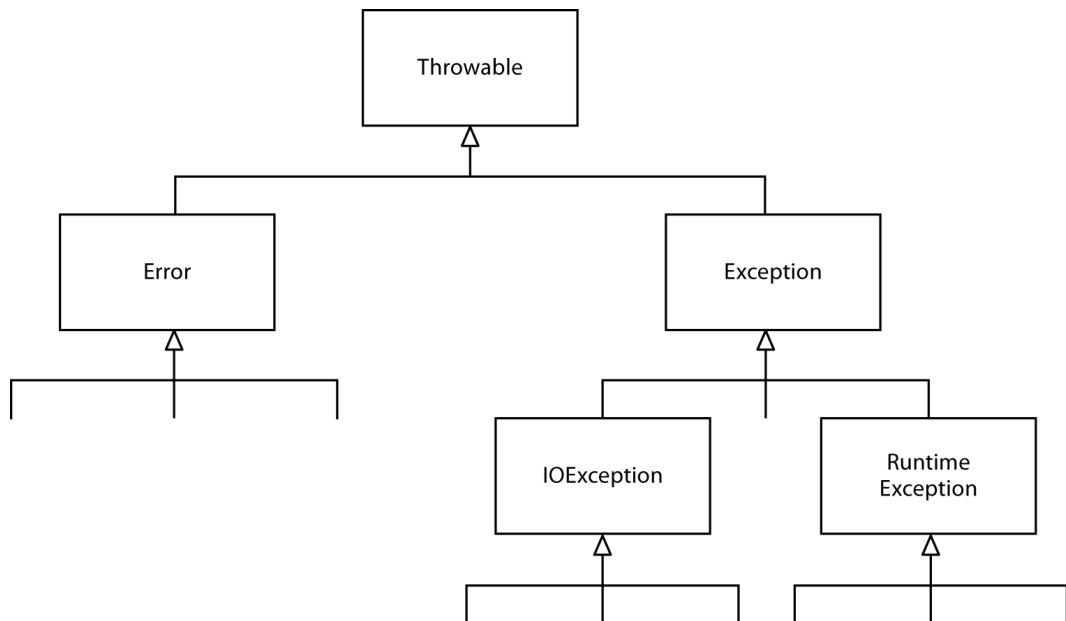
Typ obiektu wyjątku w Javie jest zawsze pochodną klasy `Throwable`. Jak się niebawem przekonamy, można pisać własne klasy wyjątków, jeśli te dostępne standardowo nie wystarczają.

Rysunek 11.1 przedstawia uproszczony diagram hierarchii wyjątków.

Należy zauważyć, że wszystkie wyjątki są potomkami klasy `Throwable`, chociaż hierarchia jej potomków dzieli się na dwie gałęzie: `Error` i `Exception`.

Klasy będące potomkami klasy `Error` odpowiadają błędom wewnętrzny i wyczerpaniu zasobów w środowisku uruchomieniowym. Nie należy wyrzucać obiektów tego typu, ponieważ jeśli wystąpi błąd wewnętrzny, niewiele można zrobić poza powiadomieniem użytkownika i zamknięciem programu. Tego typu sytuacje występują niezbyt często.

Programiści Javy o wiele więcej czasu poświęcają klasie `Exception`. Ona także dzieli się na dwie gałęzie: wyjątki związane z wykonywaniem, tak zwane wyjątki wykonawcze (`Runtime` → `Exception`), i pozostałe. Ogólna zasada głosi: wyjątki typu `RuntimeException` są powodowane przez błędy programisty. Wszystkie pozostałe mają związek z innymi niepożądanymi zdarzeniami, takimi jak błędy wejścia-wyjścia, które zaszły w poza tym dobrym programie.



Rysunek 11.1. Hierarchia wyjątków w Javie

Do wyjątków dziedziczących po klasie `RuntimeException` należą:

- niepoprawne rzutowanie,
- dostęp do nieistniejącego elementu tablicy,
- dostęp do pustego wskaźnika.

Do wyjątków, które nie dziedziczą po `RuntimeException`, należą:

- próba odczytu za końcem pliku,
- próba otwarcia niepoprawnego adresu URL,
- próba znalezienia obiektu typu `Class` dla łańcucha, który nie zgadza się z żadną z istniejących klas.

Zasada „jeśli wystąpił wyjątek `RuntimeException`, znaczy, że popełniłeś błąd” sprawdza się doskonale. Wyjątku `ArrayIndexOutOfBoundsException` można uniknąć, porównując dany indeks tablicy z jej rozmiarem. Wyjątek `NullPointerException` nie miałby miejsca, gdyby przed użyciem zmiennej sprawdzono, czy nie ma wartości `null`.

Jak wygląda sprawa z adresami URL? Czy nie można sprawdzić ich składni przed użyciem? Problem w tym, że różne przeglądarki obsługują adresy różnego rodzaju. Na przykład przeglądarka Firefox poradzi sobie z adresem typu `mailto:`, podczas gdy przeglądarka appletów nie. W związku z tym pojęcie niepoprawnej składni może mieć różne znaczenie w różnych środowiskach.

W specyfikacji Javy wyjątki typu Error lub RuntimeException nazywane są **wyjątkami niekontrolowanymi** (ang. *unchecked exception*). Wszystkie pozostałe to **wyjątki kontrolowane** (ang. *checked exception*). My również stosujemy tę terminologię. Kompilator sprawdza, czy dostarczono procedury obsługi dla wszystkich wyjątków kontrolowanych.



Nazwa „wyjątek wykonawczy” (RuntimeException) jest nieco nieprecyzyjna, ponieważ wszystkie błędy, które opisujemy, występują w czasie działania programu.



Osoby zaznajomione ze znacznie uboższą hierarchią wyjątków w standardowej bibliotece C++ mogą być zdziwione. Język ten dysponuje dwiema podstawowymi klasami wyjątków o nazwie `runtime_error` i `logic_error`. Ta druga jest odpowiednikiem klasy `RuntimeException` w Javie, czyli także określa błędy logiczne. Klasa `runtime_error` jest podstawą wyjątków powodowanych przez nieprzewidywalne zdarzenia. Odpowiada wyjątkom, które w Javie nie są typu `RuntimeException`.

11.1.2. Deklarowanie wyjątków kontrolowanych

Metoda w Javie może spowodować wyjątek, jeśli napotka sytuację, z którą nie potrafi sobie poradzić. Zasada jest prosta: metoda nie tylko informuje kompilator, jakie wartości może zwrócić, lecz **również co może się nie udać**. Na przykład instrukcja odczytująca dane z pliku wie, że pliku tego może nie być lub może być pusty. W związku z tym procedura przetwarzająca informacje z pliku musi powiadomić kompilator, że może spowodować wyjątek wejścia-wyjścia.

Informację, że metoda może spowodować wyjątek, należy umieścić w jej nagłówku. Treść nagłówka może się zmieniać w zależności od tego, jakie wyjątki kontrolowane metoda może spowodować. Poniżej znajduje się na przykład deklaracja jednego z konstruktorów klasy `FileInputStream` z biblioteki standardowej (więcej na temat strumieni piszemy w rozdziale 12.).

```
public FileInputStream(String name) throws FileNotFoundException
```

Ta deklaracja informuje, że konstruktor ten tworzy obiekt typu `FileInputStream` z parametru typu `String`, ale może także spowodować wyjątek `FileNotFoundException`. Jeśli taka przykry sytuacja będzie miała miejsce, konstruktor nie utworzy nowego obiektu typu `FileInputStream`, tylko wyrzuci obiekt typu `FileNotFoundException`. Wtedy system wykonawczy rozpoczęcie poszukiwanie procedury obsługi wyjątków, która potrafi obsłużyć obiekt typu `FileNotFoundException`.

Pisząc metodę, nie trzeba informować o każdym możliwym wyjątku, który może przez nią zostać zgłoszony. Aby zrozumieć, co i kiedy należy uwzględnić w klauzuli `throws`, trzeba pamiętać, że wyjątki są zgłaszane w następujących sytuacjach:

- Wywołanie metody, która zgłasza wyjątek kontrolowany, na przykład konstruktor `FileInputStream`.
- Wykrycie błędu i zgłoszenie wyjątku kontrolowanego za pomocą instrukcji `throw` (instrukcję `throw` opisujemy w kolejnym podrozdziale).

- Błąd programisty, typu `a[-1] = 0`, który powoduje wyjątek niekontrolowany, np. `ArrayIndexOutOfBoundsException`.
- Wystąpienie błędu wewnętrznego w maszynie wirtualnej lub bibliotece wykonawczej.

Jeśli ma miejsce któryś z dwóch pierwszych scenariuszy, trzeba poinformować programistów, którzy będą używać naszej metody, o możliwości wystąpienia wyjątku. Dlaczego? Każda metoda, która może zgłosić wyjątek, jest potencjalną pułapką. Jeśli żadna procedura obsługi nie przechwyci tego wyjątku, bieżący wątek wykonywania zostanie zamknięty.

Deklarując metodę, która może spowodować wyjątek, należy — podobnie jak w metodach należących do standardowych klas Javy — w nagłówku umieścić **specyfikację wyjątku** informującą, że metoda ta może zgłosić wyjątek.

```
class MyAnimation
{
    .
    .
    public Image loadImage(String s) throws IOException
    {
        .
    }
}
```

Jeśli metoda może zgłosić wyjątki kontrolowane różnych typów, w jej nagłówku musi się znaleźć ich lista rozdzielona przecinkami:

```
class MyAnimation
{
    .
    .
    public Image loadImage(String s) throws FileNotFoundException, EOFException
    {
        .
    }
}
```

Nie trzeba natomiast informować o wyjątkach wewnętrznych Javy, czyli tych, które dziedziczą po klasie Error. Ich źródłem może być każdy fragment kodu, przez co nie ma możliwości sprawowania nad nimi kontroli.

Podobnie nie należy informować o wyjątkach dziedziczących po klasie `RuntimeException`:

```
class MyAnimation
{
    .
    .
    void drawImage(int i) throws ArrayIndexOutOfBoundsException //zły styl
    {
        .
    }
}
```

Wyjątki tego typu mogą być w pełni kontrolowane przez programistę. Jeśli istnieje ryzyko wystąpienia błędów związanych z indeksami w tablicy, należy poświęcić chwilę czasu na ich naprawę, zamiast informować, że mogą wystąpić.

Podsumowując, każda metoda musi deklarować wszystkie wyjątki **kontrolowane** (ang. *checked*), które może zgłosić. Wyjątki niekontrolowane są albo poza zasięgiem programisty

(Error), albo powstają w takich sytuacjach, do których programista nie powinien dopuścić (RuntimeException). Jeśli metoda nie deklaruje wszystkich wyjątków kontrolowanych, kompilator wyświetli komunikat o błędzie.

Oczywiście, jak przekonaliśmy się wcześniej, zamiast deklarować wyjątek, można go przechwycić. Wtedy nie zostanie on wyrzucony z metody, a więc nie jest potrzebna specyfikacja throws. Dalej nauczymy się podejmować decyzję, czy przechwycić wyjątek, czy pozwolić zrobić to komuś innemu.



Przesłaniając metodę, należy pamiętać, że wyjątki kontrolowane deklarowane przez metodę w podklasie nie mogą być bardziej ogólne niż w metodzie nadklasy (metoda podklasy może zgłaszać mniej ogólnie wyjątki lub nie zgłaszać żadnych). Jeśli metoda w nadklasie nie zgłasza żadnych wyjątków kontrolowanych, metoda w podklasie również nie może tego robić. Jeśli na przykład przesłoniemy metodę `JComponent.paintComponent`, metoda `paintComponent` w podklasie nie może zgłaszać żadnych wyjątków kontrolowanych, ponieważ nie robi tego metoda z nadklasy.

Jeśli metoda deklaruje zgłaszanie wyjątków należących do określonej klasy, może zgłaszać wyjątki tej klasy lub dowolnej z jej podklas. Na przykład konstruktor `FileInputStream` może deklarować zgłaszanie wyjątków typu `IOException`. W takim przypadku nie wiadomo, o jaki typ wyjątku `IOException` konkretnie może chodzić. Może to być czysty typ `IOException` lub jeden z jego podtypów, na przykład `FileNotFoundException`.



Specyfikator `throws` w Javie ma prawie identyczne zastosowanie jak `throw` w C++. Jedyna różnica polega na tym, że w C++ specyfikatory `throw` są stosowane w czasie działania programu, a nie kompilacji. Oznacza to, że kompilator C++ nie zwraca uwagi na specyfikacje wyjątków. Jeśli jednak wyjątek wystąpi w funkcji nieznajdującej się na liście `throw`, wywoływana jest funkcja `unexpected` i program zostaje zamknięty.

Ponadto w C++, jeśli funkcja nie posiada żadnej specyfikacji `throw`, może zgłosić każdy rodzaj wyjątku. W Javie metoda bez specyfikatora `throws` nie może w ogóle zgłaszać wyjątków kontrolowanych.

11.1.3. Zgłaszanie wyjątków

Wyobraźmy sobie, że w naszym kodzie miało miejsce straszne zdarzenie. Mamy metodę o nazwie `readData`, która wczytuje plik z następującym nagłówkiem:

`Content-length: 1024`

Znak końca pliku pojawia się jednak po 733 znakach. Decydujemy, że sytuacja ta jest na tyle nienormalna, że trzeba zgłosić wyjątek.

Należy podjąć decyzję, jakiego typu wyjątek to ma być. Dobrym wyborem wydaje się jakiś rodzaj wyjątku `IOException`. W dokumentacji API można znaleźć typ `EOFException`, którego opis brzmi „Sygnalizuje niespodziewane napotkanie końca pliku w czasie wczytywania danych”. Doskonale. Zgłaszymy go następująco:

`throw new EOFException();`

lub

```
EOFException e = new EOFException();
throw e;
```

Cała metoda wygląda tak:

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (. . .)
    {
        if (!in.hasNext()) // napotkano koniec pliku
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

Klasa `EOFException` posiada jeszcze jeden konstruktor, który przyjmuje argument w postaci łańcucha. Można z niego zrobić dobry użytku, dostarczając bardziej szczegółowy opis sytuacji wyjątkowej.

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

Jak widać, zgłaszanie wyjątków jest proste, jeśli istnieje możliwość wykorzystania jednej z istniejących klas. W takim przypadku należy:

1. Znaleźć odpowiednią klasę wyjątków.
2. Utworzyć obiekt tej klasy.
3. Zgłosić go.

Kiedy metoda zgłosi wyjątek, nie zwraca wartości do wywołującego. Oznacza to, że nie ma konieczności zajmowania się domyślną wartością zwrotną lub kodem błędu.



Zgłaszanie wyjątków w C++ i Javie wygląda prawie tak samo. Jedyna różnica polega na tym, że w Javie można generować wyłącznie obiekty należące do podklas klasy `Throwable`. W C++ można zgłaszać wartości dowolnego typu.

11.1.4. Tworzenie klas wyjątków

Czasami w programie może wystąpić problem, do którego nie pasuje żadna ze standardowych klas wyjątków. W takim przypadku można utworzyć własną klasę, która powinna dziedziczyć po klasie `Exception` lub jednej z jej podklas, na przykład `IOException`. Istnieje zwyczaj polegający na dostarczeniu zarówno konstruktora domyślnego, jak i konstruktora pobierającego szczegółowe dane (metoda `toString` w nadklasie `Throwable` drukuje szczegółowe dane, które mogą być przydatne w czasie debugowania).

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

Teraz można generować własne typy wyjątków:

```
String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // napotkano koniec pliku
        {
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

java.lang.Throwable 1.0

■ **Throwable()**

Tworzy nowy obiekt typu `Throwable`, niezawierający żadnych szczegółowych informacji.

■ **Throwable(String message)**

Tworzy obiekt typu `Throwable` z opisem określonym przez parametr `message`. Zgodnie z konwencją wszystkie pochodne klasy wyjątków posiadają zarówno konstruktor domyślny, jak i konstruktor ze szczegółowym komunikatem.

■ **String getMessage()**

Zwraca szczegółowy opis wyjątku.

11.2. Przechwytywanie wyjątków

Potrafimy już zgłaszać wyjątki. To proste zadanie polega na wygenerowaniu wyjątku — potem można o nim zapomnieć. Oczywiście niektóre partie kodu muszą przechwytywać wyjątki, a to wymaga dłuższego planowania.

Jeśli wygenerowany wyjątek nie zostanie przechwycony, program zostanie zamknięty, a w konsoli pojawi się komunikat informujący o typie wyjątku i dane ze śledzenia stosu. Programy z graficznym interfejsem (aplety i aplikacje) przechwytyują wyjątki, drukując komunikaty ze

śledzenia stosu wywołań i wracają do pętli przetwarzającej interfejs (dobrym rozwiązaniem podczas usuwania błędów z programu z graficznym interfejsem użytkownika jest ustawienie konsoli w widocznym miejscu).

Do przechwytywania wyjątków służy blok try-catch. Najprostsza forma tego bloku wygląda następująco:

```
try
{
    kod
    więcej kodu
    i jeszcze trochę kodu
}
catch (TypWyjątku e)
{
    procedura obsługi określonego typu wyjątków
}
```

Jeśli którykolwiek z fragmentów kodu w bloku try zgłosi wyjątek typu określonego w klauzuli catch, to:

- 1 Program pominie resztę kodu w bloku try.
- 2 Program wykona procedurę obsługi znajdująca się w klauzuli catch.

Jeśli żaden z fragmentów kodu w bloku try nie zgłosi wyjątku, klauzula catch zostaje pominięta.

Jeśli w którejkolwiek z metod zostanie wygenerowany wyjątek innego typu niż określony w klauzuli catch, program natychmiast z tej metody wychodzi (może gdzieś wyżej w stosie wywołań znajduje się klauzula catch przeznaczona dla tego typu wyjątku).

Powyzsze informacje zilustrujemy typowym przykładem procedury odczytującej dane:

```
public void read(String filename)
{
    try
    {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            przetwarzanie danych wejściowych
        }
    }
    catch (IOException exception)
    {
        exception.printStackTrace();
    }
}
```

Jak widać, większość kodu w klauzuli try jest prosta — odczytuje i przetwarza bajty aż do napotkania końca pliku. Rzut oka do API Javy pozwala się zorientować, że metoda read może zgłosić wyjątek IOException. W takim przypadku wychodzimy z pętli while, wchodzimy do

klauzuli catch i generujemy dane ze śledzenia stosu. W przypadku niewielkiego programu wydaje się to rozsądny sposobem obsługi tego wyjątku. Jakie są jednak inne wyjścia z tej sytuacji?

Często najlepiej jest nic nie robić, tylko przekazać wyjątek do wywołującego. Jeśli błąd powstanie w metodzie read, najlepiej, by zajęła się nim procedura, która wywołała tę metodę! Takie podejście do problemu wymaga dodania informacji, że metoda może zgłosić wyjątek IOException.

```
public void read(String filename) throws IOException
{
    InputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1)
    {
        przetwarzanie danych wejściowych
    }
}
```

Należy pamiętać, że kompilator ściśle trzyma się specyfikatorów throws. Jeśli wywołujemy metodę, która generuje wyjątek kontrolowany, musi on zostać obsłużony lub przekazany dalej.

Które z tych dwóch rozwiązań jest lepsze? Ogólna zasada nakazuje przechwytywać te wyjątki, które można obsłużyć, i odsyłać te, których nie potrafimy obsłużyć.

Odsyłając wyjątek, trzeba dodać specyfikator throws, aby ostrzec wywołującego, że może zostać wygenerowany wyjątek.

Informacje na temat typów wyjątków zgłaszanych przez konkretne metody można znaleźć w dokumentacji API. Dysponując tymi informacjami, można zdecydować, czy je obsłużyć, czy dodać do listy throws. Wybór tej drugiej możliwości nie stanowi dla programisty żadnej ujmy. Lepiej przekazać wyjątek do fachowej obsługi, niż obsłużyć go źle.

Jest jeden wyjątek od tej reguły, o którym już wspominaliśmy. Pisząc metodę przesłaniającą metodę z nadklasy, która nie zgłasza żadnych wyjątków (np. paintComponent z klasy JComponent), każdy kontrolowany wyjątek **musimy** przechwycić w kodzie tej metody. Nie można dodać więcej specyfikatorów throws do metody w podklasie, niż jest w metodzie w nadklasie.



Przechwytywanie wyjątków w Javie wygląda prawie tak samo jak w C++. Kod:

```
catch (Exception e) // Java
```

jest analogiczny do kodu:

```
catch (Exception& e) // C++
```

Nie istnieje instrukcja analogiczna do catch(...). Nie jest ona potrzebna w Javie, ponieważ wszystkie wyjątki pochodzą od wspólnej nadklasy.

11.2.1. Przechwytywanie wielu typów wyjątków

W jednym bloku try można przechwycić kilka typów wyjątków i każdy z nich obsłużyć w inny sposób. Dla każdego typu należy napisać oddzielną klauzulę catch, np.:

```
try
{
    kod, który może generować wyjątki
}
catch (FileNotFoundException e)
{
    działania dotyczące nieprawidłowego adresu URL
}
catch (UnknownHostException e)
{
    działania dotyczące nieznanych hostów
}
catch (IOException e)
{
    działania dotyczące wszystkich pozostałych błędów wejścia-wyjścia
}
```

Obiekt wyjątku może zawierać informacje o naturze wyjątku. Aby dowiedzieć się więcej o danym obiekcie, należy użyć następującego wywołania:

```
e.getMessage()
```

Poniższa instrukcja zwraca szczegółowe dane na temat błędu (jeśli istnieja) lub rzeczywisty typ obiektu wyjątku:

```
e.getClass().getName()
```

Od Java SE 7 można przechwytywać różne typy wyjątków w jednej klauzuli catch. Przypuśćmy na przykład, że działanie w przypadku braku pliku i nieznanego hosta jest takie samo. Wówczas można połączyć klauzule catch w następujący sposób:

```
try
{
    kod, który może spowodować wyjątki
}
catch (FileNotFoundException | UnknownHostException e)
{
    działania awaryjne w przypadku braku plików lub nieznanego hosta
}
catch (IOException e)
{
    działania awaryjne dotyczące pozostałych problemów wejścia i wyjścia
}
```

Jest to potrzebne tylko wtedy, gdy typ jednego z przechwytywanych wyjątków nie jest podklassą innego.



Gdy przechwytywanych jest kilka wyjątków, zmienną wyjątku jest `final`. Przykładowo zmiennej `e` w klauzuli nie można przypisać innej wartości:

```
catch (FileNotFoundException | UnknownHostException e) { ... }
```



Przechwytywanie po kilka wyjątków naraz nie tylko sprawia, że kod jest bardziej przejrzysty, ale również powoduje jego szybsze działanie. W wygenerowanym kodzie bajtowym znajduje się tylko jeden blok dla wspólnej klauzuli `catch`.

11.2.2. Powtórne generowanie wyjątków i budowanie łańcuchów wyjątków

Wyjątek można wygenerować także w klauzuli `catch`. Zazwyczaj robi się to w celu zmiany jego typu. W podsystemie, z którego korzystają inni programiści, warto utworzyć taki typ wyjątku, który wskazuje na awarię tego podsystemu. Przykładem takiego typu wyjątku jest `ServletException`. Kod wykonujący serwlet może nie potrzebować wszystkich szczegółów na temat tego, co zawiodło. Zdecydowanie natomiast musi wiedzieć, że serwer miał awarię.

Poniższy fragment kodu przedstawia przechwycenie wyjątku i powtórne jego wygenerowanie:

```
try
{
    dostęp do bazy danych
}
catch (SQLException e)
{
    throw new ServletException("Błąd bazy danych: " + e.getMessage());
}
```

W tym przypadku do obiektu `ServletException` został dodany komunikat.

Można jednak zrobić coś lepszego, czyli ustawić pierwotny wyjątek jako powód (ang. *cause*) nowego wyjątku:

```
try
{
    dostęp do bazy danych
}
catch (SQLException e)
{
    Throwable se = new ServletException("database error");
    se.initCause(e);
    throw se;
}
```

Po przechwyceniu tego wyjątku można odzyskać dane dotyczące pierwotnego wyjątku:

```
Throwable e = se.getCause();
```

Ta technika opakowywania jest szczególnie polecana. Pozwala ona na zgłoszanie wyjątków wysokiego poziomu w podsystemach bez utraty szczegółów dotyczących pierwotnej awarii.



Technika opakowywania jest przydatna również wtedy, gdy wyjątek kontrolowany występuje w metodzie, która nie może zgłaszać wyjątków kontrolowanych. Można go przechwycić i opakować w obiekt wyjątku wykonawczego.

Czasami trzeba tylko zarejestrować informację o wyjątku i zgłosić go ponownie bez zmianiania:

```
try
{
    dostęp do bazy danych
}
catch (Exception e)
{
    logger.log(level, message, e);
    throw e;
}
```

Przed pojawiением się Java SE 7 metoda ta była problematyczna. Wyobraźmy sobie, że przedstawiony kod znajduje się w metodzie:

```
public void updateRecord() throws SQLException
```

Kompilator najpierw znajdował instrukcję `throw` w bloku `catch`, następnie sprawdzał typ `e` i narzekał, że metoda ta może zgłaszać dowolny typ wyjątku, nie tylko `SQLException`. Zostało to już naprawione. Kompilator bierze pod uwagę, że `e` pochodzi z bloku `try`. Biorąc pod uwagę, że w bloku tym jedynymi wyjątkami kontrolowanymi są egzemplarze klasy `SQLException`, oraz uwzględniając fakt, że `e` nie zmienia się w bloku `catch`, można powiedzieć, że otaczająca metodą zgłasza wyjątki typu `SQLException`.

11.2.3. Klauzula `finally`

Kiedy zostaje zgłoszony wyjątek, następuje zatrzymanie wykonywania pozostałego kodu w metodzie i wyjście z niej. Może to powodować problem, jeśli metoda ta pobrała jakieś zasoby, o których wie tylko ona, a które muszą zostać wyczyszczone. Rozwiążaniem tego problemu może być przechwycenie i ponowne wygenerowanie wszystkich wyjątków. Jest to jednak mało wydajna metoda, ponieważ konieczne jest czyszczenie zasobów w dwóch miejscach — w normalnym kodzie i w kodzie wyjątku.

W Javie dostępne jest lepsze rozwiązywanie polegające na użyciu klauzuli `finally`. Poniżej przedstawiamy, jak prawidłowo pozbyć się obiektu `Graphics`. Tych samych technik należy używać do zamknięcia połączeń z bazą danych. W rozdziale 4. drugiego tomu wyjaśniamy, dlaczego zamknięcie wszystkich połączeń z bazą danych jest bardzo ważne, nawet kiedy wystąpią wyjątki.

Kod w klauzuli `finally` jest wykonywany bez względu na to, czy wyjątek zostanie przechwycony, czy nie. W poniższym przykładzie kontekst graficzny zostanie usunięty **bez względu na okoliczności**.

```
InputStream in = new FileInputStream(...);
try
{
```

```

// 1
potencjalne źródło wyjątków
// 2
}
catch (IOException e)
{
    // 3
    wyświetlanie okna dialogowego błędu
    // 4
}
finally
{
    // 5
    in.close();
}
// 6

```

Przeanalizujmy trzy możliwe sytuacje, w których program wykona zawartość klauzuli `finally`:

1. Kod nie generuje żadnego wyjątku. W tym przypadku najpierw wykonywany jest kod w klauzuli `try`. Następnie wykonywana jest klauzula `finally`. Dalej sterowanie przekazywane jest do pierwszej instrukcji za klauzulą `finally`. Innymi słowy, przepływ sterowania jest następujący: 1, 2, 5 i 6.
2. Kod generuje wyjątek, w tym przypadku `IOException`, przechwytywany w klauzuli `catch`. W tym przypadku kod w bloku `try` jest wykonywany do punktu, w którym został wygenerowany wyjątek. Reszta kodu w tym bloku zostaje pominięta. Następnie wykonywany jest kod z pasującej klauzuli `catch` i kod z klauzuli `finally`.

Jeśli kod nie wygeneruje wyjątku, wykonany zostanie pierwszy wiersz kodu za klauzulą `finally`. Tym razem przepływ sterowania jest taki: 1, 3, 4, 5, 6.

Jeśli klauzula `catch` wygeneruje wyjątek, zostaje on zwrócony do metody, która wywołała tę metodę, i przepływ sterowania przechodzi przez punkty 1, 3 i 5.

3. Kod zgłasza wyjątek, którego nie przechwytuje żadna klauzula `catch`. Wykonywany jest kod w bloku `try` do momentu wygenerowania wyjątku. Następnie wykonywany jest kod w klauzuli `finally` i wyjątek jest zwracany do metody, która wywołała tę metodę. Sterowanie przechodzi tylko przez punkty 1 i 5.

Klauzuli `finally` można użyć bez klauzuli `catch`. Przyjrzyjmy się poniższej instrukcji `try`:

```

InputStream in = ...;
try
{
    potencjalne źródło wyjątków
}
finally
{
    in.close();
}

```

Instrukcja `in.close()` w klauzuli `finally` zostanie wykonana bez względu na to, czy w bloku `try` zostanie zgłoszony wyjątek. Oczywiście, jeśli wyjątek zostanie wygenerowany, jest on zgłoszany ponownie i musi zostać przechwycony w innej klauzuli `catch`.

W poniższej wskazówce wyjaśniamy, dlaczego naszym zdaniem dobrym pomysłem jest użycie w ten sposób klauzuli `finally` do zamykania zasobów.



Zdecydowanie zalecamy oddzielenie od siebie bloków `try-catch` i `try-finally`. Dzięki temu kod jest znacznie bardziej przejrzysty. Na przykład:

```
InputStream in = ...;
try
{
    try
    {
        potencjalne źródło wyjątków
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    wyświetlenie informacji o błędzie
}
```

Wewnętrzny blok `try` ma tylko jedno zadanie: pilnuje, czy strumień wejściowy został zamknięty. Zewnętrzny blok `try` ma również tylko jedno zadanie: pilnuje, aby błędy były wyciszane. Rozwiążanie to jest nie tylko bardziej przejrzyste, ale też bardziej funkcjonalne — błędy w klauzuli `finally` są zgłaszane.



Klauzula `finally` może generować nieprawidłowe wyniki, jeśli zawiera instrukcję `return`. Założymy, że wychodzimy ze środka bloku `try` za pomocą instrukcji `return`. Przed zwróceniem przez metodę wartości wykonywana jest zawartość bloku `finally`. Jeśli blok ten również zawiera instrukcję `return`, przysłania oryginalną wartość zwrotną. Przestudiujmy poniższy sztuczny przykład:

```
public static int f(int n)
{
    try
    {
        int r = n * n;
        return r;
    }
    finally
    {
        if (n == 2) return 0;
    }
}
```

Jeśli wywołamy metodę `f(2)`, blok `try` wyliczy wartość `r = 4` i wykona instrukcję `return`. Jednak przed zwróceniem tej wartości zostanie wykonana klauzula `finally`, która spowoduje zwrot wartości 0 zamiast spodziewanej 4.

Czasami klauzula `finally` powoduje poważne problemy, zwłaszcza kiedy metoda sprzątająca także może zgłosić wyjątek. Wyobraźmy sobie, że chcemy, aby w chwili wystąpienia wyjątku w kodzie przetwarzającym strumień został on zamknięty.

```
InputStream in = ...;
try
{
    potencjalne źródło wyjątków
}
finally
{
    in.close();
}
```

Następnie wyobraźmy sobie, że kod w bloku `try` zgłasza wyjątek **innego typu niż** `IOException`, który leży w sferze zainteresowań wywołującego ten kod. Zostaje wykonana klauzula `finally` i następuje wywołanie metody `close`. Ta metoda sama może wygenerować wyjątek `IOException`. Jeśli tak się stanie, oryginalny wyjątek zostaje utracony i zamiast niego zgłoszany jest wyjątek `IOException`.

Stanowi to problem, ponieważ pierwszy wyjątek może być bardziej interesujący. Jeśli chcesz działać zgodnie ze sztuką i ponownie zgłosić pierwotny wyjątek, to musisz bardzo skomplikować kod. Oto przykład:

```
InputStream in = ...;
Exception ex = null;
try
{
    try
    {
        potencjalne źródło wyjątków
    }
    catch (Exception e)
    {
        ex = e;
        throw e;
    }
}
finally
{
    try
    {
        in.close();
    }
    catch (Exception e)
    {
        if (ex == null) throw e;
    }
}
```

Na szczęście w Java SE 7 znacznie ułatwiono zamykanie zasobów, o czym przekonasz się w następnym podrozdziale.

11.2.4. Instrukcja try z zasobami

W Java SE 7 możliwe jest zastosowanie wygodnego skrótu dla konstrukcji typu:

```
otwarcie zasobu
try
{
    praca z zasobem
}
finally
{
    zamknięcie zasobu
}
```

Zasób musi jedynie należeć do klasy implementującej interfejs AutoCloseable. Interfejs ten ma tylko jedną metodę:

```
void close() throws Exception
```



Istnieje też interfejs podrzędny AutoCloseable o nazwie Closeable, który również zawiera tylko metodę close. Metoda ta jednak zgłasza wyjątki typu IOException.

W najprostszej postaci instrukcja try z zasobami wygląda tak:

```
try (Resource res = ...)
{
    praca z zasobem
}
```

Jeśli blok try istnieje, metoda res.close() jest wywoływana automatycznie. Oto typowy przykład — odczyt wszystkich słów z pliku:

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")))
{
    while (in.hasNext())
        System.out.println(in.next());
}
```

Gdy działanie bloku zakończy się normalnie lub wystąpi wyjątek, nastąpi wywołanie metody in.close(), jak gdyby zdefiniowana była klauzula finally.

Można też określić kilka zasobów, np.:

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")),
PrintWriter out = new PrintWriter("out.txt"))
{
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

Niezależnie od sposobu zakończenia wykonywania bloku zasoby in i out zostaną zamknięte. Gdybyśmy mieli to zaprogramować ręcznie, musielibyśmy użyć dwóch zagnieżdzonych konstrukcji try-finally.

Jak widziałeś w poprzednim podrozdziale, problemy powstają w momencie, gdy zarówno blok try, jak i metoda close zgłaszą wyjątek. Dzięki instrukcji try z zasobami można to elegancko rozwiązać. Oryginalny wyjątek jest zgłaszany ponownie, a wszystkie wyjątki zgłoszone przez metody close są „łumione”. Zostają automatycznie przechwycone i dodane do oryginalnego wyjątku przy użyciu metody addSuppressed. Jeśli Cię interesują, możesz uzyskać tablicę wyjątków metody close, wywołując metodę getSuppressed.

Nie musisz się męczyć. Jeśli musisz zamknąć jakiś zasób, zawsze używaj instrukcji try z zasobami.



Instrukcja try z zasobami też może zawierać klauzule catch i finally, które są wykonywane po zamknięciu zasobów. W praktyce jednak lepiej jest nie ładować tak dużo do jednej instrukcji try.

11.2.5. Analiza danych ze śledzenia stosu

Dane ze **śledzenia stosu** (ang. *stack trace*) przedstawiają listę wszystkich oczekujących wywołań metod w określonym momencie wykonywania programu. Prawie każdy widział taką listę — jest wyświetlana zawsze, gdy program w Javie zostaje zamknięty z powodu nieprzechwyconego wyjątku.

Opis stosu można uzyskać za pomocą metody printStackTrace z klasy Throwable:

```
Throwable t = new Throwable();
ByteArrayOutputStream out = new ByteArrayOutputStream();
t.printStackTrace(out);
String description = out.toString();
```

Bardziej elastyczna w działaniu jest metoda getStackTrace, zwracająca tablicę obiektów klasy StackTraceElement, które można przeanalizować w programie, np.:

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
    analiza ramek
```

Klasa StackTraceElement posiada metody służące do sprawdzania nazwy pliku i numeru, a także nazwy klasy i metody wykonywanego wiersza kodu. Metoda `toString` tworzy sformatowany łańcuch zawierający wszystkie te informacje.

Metoda `Thread.getAllStackTraces` generuje dane ze śledzenia stosu dla wszystkich wątków. Poniżej znajduje się przykład jej użycia:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
    StackTraceElement[] frames = map.get(t);
    analiza ramek
}
```

Więcej informacji na temat interfejsu Map i wątków znajduje się w rozdziałach 13. i 14.

Listing 11.1 drukuje stos wywołań rekurencyjnej funkcji obliczającej silnię. Na przykład wywołanie factorial(3) zwróci następujący wynik:

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.main(StackTraceTest.java:34)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6
```

Listing 11.1. stackTrace/StackTraceTest.java

```
package stackTrace;

import java.util.*;

/**
 * Program wyświetlający stos wywołań wywołania rekurencyjnej metody.
 * @version 1.01 2004-05-10
 * @author Cay Horstmann
 */
public class StackTraceTest
{
    /**
     * Oblicza silnię liczby.
     * @param n nieujemna liczba całkowita
     * @return n! = 1 * 2 * ... * n
     */
    public static int factorial(int n)
    {
        System.out.println("factorial(" + n + "):");
        Throwable t = new Throwable();
        StackTraceElement[] frames = t.getStackTrace();
        for (StackTraceElement f : frames)
            System.out.println(f);
        int r;
        if (n <= 1) r = 1;
        else r = n * factorial(n - 1);
        System.out.println("return " + r);
        return r;
    }

    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Wpisz n: ");
        int n = in.nextInt();
```

```

        factorial(n);
    }
}

```

java.lang.Throwable **1.0**

- Throwable(Throwable cause) **1.4**

- Throwable(String message, Throwable cause) **1.4**

Tworzy obiekt typu Throwable z określonym powodem cause.

- Throwable initCause(Throwable cause) **1.4**

Ustawia powód cause dla obiektu lub zgłasza wyjątek, jeśli obiekt ten ma już powód. Zwraca this.

- Throwable getCause() **1.4**

Zwraca obiekt wyjątku, który został ustawiony jako powód tego obiektu, lub wartość null, jeśli żaden powód nie został ustawiony.

- StackTraceElement[] getStackTrace() **1.4**

Pobiera dane ze śledzenia stosu w czasie, kiedy został utworzony obiekt.

- void addSuppressed(Throwable t) **7**

Dodaje „stłumiony” wyjątek do wyjątku. Ma to miejsce w instrukcji try z zasobami, gdzie t jest wyjątkiem zgłoszonym przez metodę close.

- Throwable[] getSuppressed() **7**

Pobiera wszystkie „stłumione” wyjątki wyjątku. Najczęściej są to wyjątki zgłoszone przez metodę close w instrukcji try z zasobami.

java.lang.Exception **1.0**

- Exception(Throwable cause) **1.4**

- Exception(String message, Throwable cause)

Tworzy obiekt typu Exception z określonym powodem.

java.lang.RuntimeException **1.0**

- RuntimeException(Throwable cause) **1.4**

- RuntimeException(String message, Throwable cause) **1.4**

Tworzy wyjątek typu RuntimeException z określonym powodem.

java.lang.StackTraceElement **1.4**

- String getFileName()

Zwraca nazwę pliku źródłowego zawierającego punkt wykonania elementu lub wartość null, jeśli informacja ta jest niedostępna.

- `int getLineNumber()`

Zwraca numer wiersza w pliku źródłowym zawierającym punkt wykonania elementu lub wartość -1, jeśli informacja ta jest niedostępna.

- `String getClassName()`

Zwraca nazwę klasy z pełnym kwalifikatorem, zawierającej punkt wykonania elementu.

- `String getMethodName()`

Zwraca nazwę metody zawierającej punkt wykonania elementu. Nazwa konstruktora to `<init>`. Nazwa statycznego inicjatora to `<clinit>`. Nie można odróżnić przeładowanych metod o tej samej nazwie.

- `boolean isNativeMethod()`

Zwraca wartość `true`, jeśli punkt wykonania elementu znajduje się w metodzie rodzimej.

- `String toString()`

Zwraca sformatowany łańcuch zawierający nazwę metody i klasy oraz nazwę pliku i numer wiersza, jeśli dane te są dostępne.

11.3. Wskazówki dotyczące stosowania wyjątków

Zdania na temat prawidłowego stosowania wyjątków są podzielone. Niektórzy programiści uważają wszystkie wyjątki kontrolowane za niepotrzebny ciężar, inni zaś sprawiają wrażenie, jakby nie mogli się nimi nacieszyć. Naszym zdaniem wyjątki (nawet kontrolowane) mają swój obszar zastosowań. Poniższe wskazówki dotyczą ich poprawnego wykorzystania.

1. Obsługa wyjątków nie może zastąpić prostych testów.

Aby to udowodnić, napisaliśmy program, który próbuje 10 000 000 razy pobrać element z pustego stosu. Najpierw sprawdziliśmy, czy stos jest pusty.

```
if (!s.empty()) s.pop();
```

Następnie pobraliśmy element bez względu na warunki. W wyniku tego przechwyciliśmy wyjątek informujący nas, że nie powinniśmy byli tego robić.

```
try()
{
    s.pop();
}
catch (EmptyStackException e)
{
}
```

Zmierzyliśmy czas obu tych operacji i uzyskaliśmy następujące wyniki: wersja z `isEmpty` została wykonana w 646 milisekund, natomiast wersja przechwytyująca wyjątek `EmptyStackException` działała 21 739 milisekund.

Jak widać, przechwycenie wyjątku zajęło znacznie więcej czasu niż przeprowadzenie prostego testu. Wniosek: wyjątków używaj wyłącznie w sytuacjach wyjątkowych.

2. Nie rozdrabniaj się.

Wielu programistów każdą instrukcję umieszcza w osobnym bloku try.

```
OutputStream out;
Stack s;

for (i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch (EmptyStackException s)
    {
        // stos był pusty
    }
    try
    {
        out.writeInt(n);
    }
    catch (IOException e)
    {
        // problem z zapisem w pliku
    }
}
```

Taki sposób programowania drastycznie zwiększa ilość kodu. Miej na uwadze cel, który chcesz osiągnąć. W tym przypadku chcemy pobrać 100 liczb ze stosu i zapisać je w pliku (nie zastanawiaj się po co — to tylko dla zabawy). Jeśli pojawi się jakiś problem, nic nie możemy zrobić. Jeśli stos jest pusty, to się nagle nie zapełni. Jeśli plik zawiera błąd, to błąd ten się magicznie nie naprawi. W takim razie rozsądnie by było umieścić **cały** kod tego zadania w jednym bloku try. Jeśli któreś z działań się nie powiedzie, można zatrzymać całe zadanie.

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch (IOException e)
{
    // problem z zapisem w pliku
}
catch (EmptyStackException s)
{
    // stos był pusty
}
```

Ten kod jest znacznie bardziej przejrzysty i spełnia jedną z zasad obsługi błędów — **oddziela** normalny tok działania od procedur obsługi błędów.

3. Właściwie wykorzystuj hierarchię wyjątków.

Nie generuj zawsze wyjątku `RuntimeException`. Znajdź odpowiednią podklasę lub utwórz własną.

Nie przechwytyuj tylko typu `Throwable`, ponieważ przez to kod jest bardziej do odczytania i utrzymania.

Respektuj różnicę między wyjątkami kontrolowanymi a niekontrolowanymi. Te pierwsze są z natury uciążliwe i nie należy ich generować dla błędów logicznych (na przykład problem z tym ma biblioteka refleksyjna, w której wywołujący często musi przechwytywać wyjątki, choć wiadomo, że one nigdy nie powstają).

Nie wahaj się zamienić wyjątku jednego typu na inny typ, który jest bardziej odpowiedni. Jeśli na przykład przetwarzasz liczbę całkowitą w pliku, przechwyty wyjątku `NumberFormatException` i zamień jego typ na podkласę klasy `IOException` lub swoją własną.

4. Nie ukrywaj wyjątków.

W Javie bardzo silna jest pokusa wyciszania wyjątków. Napisaliśmy na przykład metodę wywołującą inną metodę, która z kolei z bardzo małym prawdopodobieństwem może spowodować wyjątek. Kompilator podnosi alarm, ponieważ nie umieściliśmy tego wyjątku na liście `throws` tej metody. Nie chcemy jednak tego robić, ponieważ wtedy kompilator będzie marudzić o pozostałych metodach, które wywołują tę metodę. W związku z tym wyciszamy ten wyjątek:

```
public Image loadImage(String s)
{
    try
    {
        kod mogący spowodować wyjątek kontrolowany
    }
    catch (Exception e)
    {} //po kłopocie
}
```

Dzięki temu kod przejdzie komplikację bez problemu. Będzie działał jak należy, dopóki nie wystąpi wyjątek, który zostanie zignorowany. Jeśli uważasz, że wyjątki są ważne, poświęć nieco czasu, aby je prawidłowo obsługiwać.

5. Nie bądź pobłażliwy.

Niektórzy programiści czują obawy przed generowaniem wyjątków w odpowiedzi na błędy. Myślą, że może lepiej byłoby w zamian zwrócić jakąś wartość, kiedy metoda zostanie wywołana z nieprawidłowymi parametrami. Czy na przykład metoda `Stack.pop` nie mogłaby zwrócić wartości `null` zamiast wyjątku, kiedy stos jest pusty? Naszym zdaniem lepiej jest zgłosić wyjątek `EmptyStackException` w punkcie wystąpienia awarii, niż później odebrać wyjątek `NullPointerException`.

6. Przekazywanie wyjątków nie stanowi ujmy.

Wielu programistów czuje się zobowiązanych do przechwytcenia wszystkich zgłoszonych wyjątków. Kiedy wywołują metodę, która może spowodować wyjątek,

np. konstruktor `FileInputStream` lub metoda `readLine`, starają się przechwycić wszystkie wyjątki, które mogą wystąpić. Często jednak lepiej jest taki wyjątek przekazać:

```
public void readStuff(String filename) throws IOException // To nie jest powód
// do wstydu!
{
    InputStream in = new FileInputStream(filename);
    ...
}
```

Metody wyższego poziomu często dysponują lepszymi mechanizmami informowania użytkownika o błędach lub porzucania nieudanych poleceń.



Zasady 5. i 6. można podsumować słowami „generuj wcześnie, przechwytyj późno”.

11.4. Asercje

Asercje są powszechnie stosowaną techniką programowania zachowawczego. Założymy, że mamy pewność, iż określony warunek, na którym polegamy w programie, jest spełniony. Możemy na przykład wykonywać następujące działanie:

```
double y = Math.sqrt(x);
```

Jesteśmy pewni, że zmienna `x` nie ma wartości ujemnej. Może ona być wynikiem innego działania, które nie może zwracać ujemnych wyników, lub jest parametrem metody, która przyjmuje tylko wartości dodatnie. Mimo to wolimy jednak sprawdzić dwa razy, niż pozwolić wakrać się do swoich obliczeń nieprawidłowym wartościom. Oczywiście jednym z wyjścia jest wygenerowanie wyjątku:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

Niestety ten kod pozostanie w programie nawet po skończeniu testów. Jeśli takich miejsc jest dużo, program będzie działał zauważalnie wolniej, niż powinien.

Lepszym wyjściem z tej sytuacji są asercje, ponieważ można je automatycznie usunąć po zakończeniu testowania.

W Javie dostępne jest słowo kluczowe `assert`. Istnieją dwa podstawowe sposoby jego stosowania:

```
assert warunek;
i
assert warunek : wyrażenie
```

Każda z tych wersji sprawdza warunek i zgłasza błąd `AssertionError`, a jeśli warunek nie zostanie spełniony — wartość `false`. W drugiej wersji wyrażenie jest przekazywane do konstruktora obiektu `AssertionError` i zamieniane na łańcuch komunikatu.



Jedynym przeznaczeniem **wyrażenia** jest utworzenie łańcucha komunikatu. Obiekt `AssertionError` nie przechowuje rzeczywistej wartości wyrażenia, a więc nie można jej z niego później wydobyć. W dokumentacji JDK napisano w protekcyjnym tonie, że „mogłoby to skłaniać programistów do próbowania rozwiązania problemów w asercjach, co przeczyłoby sensowi istnienia samego narzędzia”.

Aby zapewnić, że zmienna `x` nie jest ujemna, używamy poniższej asercji:

```
assert x >= 0;
```

Możemy też przekazać rzeczywistą wartość `x` do obiektu `AssertionError`, aby ją później wyświetlić:

```
assert x >= 0;
```



C++ Makro `assert` w języku C zamienia warunek asercji w łańcuch, który jest drukowany w przypadku niespełnienia warunku asercji. Jeśli na przykład warunek `assert(x>=0)` nie zostanie spełniony, zostanie wydrukowane, że warunek, który nie został spełniony, to `x >= 0`. W Javie warunek nie staje się automatycznie częścią komunikatu o błędzie. Aby go zobaczyć, trzeba go przekazać jako łańcuch do obiektu `AssertionError`: `x>=0 : "x>=0"`.

11.4.1. Włączanie i wyłączanie asercji

Przy standardowych ustawieniach asercje są wyłączone. Aby je włączyć, należy przy uruchamianiu programu użyć opcji `-enableassertions`, w skrócie `-ea`:

```
java -enableassertions MyApp
```

Pamiętajmy, że, aby włączyć lub wyłączyć asercje, nie trzeba ponownie kompilować programu. Funkcja ta należy do **modułu ładującego klasy** (ang. *class loader*). Kiedy asercje są wyłączone, moduł ten usuwa ich kod, aby nie spowalniały programu.

Asercje można nawet włączyć tylko w wybranej klasie lub pakiecie. Na przykład:

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

Powyższe polecenie włącza asercje w klasie `MyClass` i wszystkich klasach należących do pakietu `com.mycompany.mylib` oraz jego podpakietów. Opcja `-ea...` włącza asercje we wszystkich klasach pakietu domyślnego.

Można też wyłączyć asercje w wybranych klasach i pakietach. Służy do tego opcja `-disableassertions`, w skrócie `-da`:

```
java -ea:... -da:MyClass MyApp
```

Niektóre klasy nie są ładowane przez moduł ładujący, a bezpośrednio przez maszynę wirtualną. Za pomocą opisywanych opcji można włączać i wyłączać wybrane asercje także w tych klasach.

Opcje `-ea` i `-da`, które włączają i wyłączają wszystkie asercje, nie działają w klasach biblioteki systemowej. Do włączania asercji w tych klasach służy opcja `-enablesystemassertions`, w skrócie `-esa`.

Stan asercji modułów ładujących można także kontrolować z poziomu programu. Więcej informacji na ten temat znajduje się w wyciągach z API na końcu tego podrozdziału.

11.4.2. Zastosowanie asercji do sprawdzania parametrów

W Javie dostępne są trzy mechanizmy działania w sytuacjach awaryjnych:

- generowanie wyjątków,
- dzienniki,
- asercje.

Kiedy należy stosować asercje:

- W przypadkach awarii, których nie da się naprawić.
- Asercje są włączane wyłącznie w fazie rozwoju i testów (czasami żartuje się, że przypomina to zakładanie kapoka, będąc w pobliżu brzegu, i zrzucanie go po wypłynięciu na pełne morze)

W związku z tym asercji nie należy stosować do sygnalizowania innym częściom programu błędów, które można naprawić, lub informowania użytkownika o problemach. Asercji należy używać wyłącznie do lokalizacji błędów wewnętrznych powstających w fazie testowej.

Przeanalizujmy typowy scenariusz — sprawdzanie parametrów metody. Czy do sprawdzenia niedozwolonych indeksów lub referencji `null` powinniśmy użyć asercji? Aby odpowiedzieć na to pytanie, trzeba zajrzeć do dokumentacji tej metody. Założymy, że implementujemy metodę sortującą.

```
/*
Sortuje określony zakres danych wskazanej tablicy w rosnącej kolejności liczbowej.
Początek sortowanego zakresu wyznacza parametr fromIndex (włącznie), a koniec — parametr toIndex
(wyłącznie).
@param a tablica do posortowania
@param fromIndex indeks pierwszego elementu (włącznie) zakresu do posortowania
@param toIndex indeks ostatniego elementu zakresu do posortowania (wyłącznie)
@throws IllegalArgumentException, jeśli fromIndex > toIndex
@throws ArrayIndexOutOfBoundsException, jeśli fromIndex < 0 lub toIndex > a.length
*/
static void sort(int[] a, int fromIndex, int toIndex)
```

Według dokumentacji metoda ta zgłasza wyjątek, jeśli wartości indeksów są nieprawidłowe. Działanie to stanowi część umowy zawartej pomiędzy metodą a wywołującym tę metodę. Implementując ją, musimy stosować się do postanowień tej umowy i zgłaszać wymienione wyjątki. Nie byłoby właściwe użycie w zamian asercji.

Czy powinniśmy utworzyć asercję zapewniającą, że a nie ma wartości `null`? Nie. Dokumentacja milczy, jeśli chodzi o zachowanie metody, kiedy a ma wartość `null`. Wywołujący mogą się spodziewać, że w takim przypadku zwróci ona wartość, a nie zgłosi błąd niespełnienia warunku asercji.

Zobaczmy jednak, jak by wyglądała sytuacja, gdyby umowa była nieco inna:

```
@param a tablica do posortowania (nie może być null)
```

Teraz wywołujący metodę wie, że tej metody nie można wywoływać na rzecz pustej tablicy. W związku z tym może się ona zaczynać od asercji:

```
assert a != null;
```

W informatyce ten rodzaj umowy nazywa się **warunkiem wstępny** (ang. *precondition*). Pierwotna wersja metody nie posiadała żadnych warunków wstępnych dotyczących parametrów — zapewniała przewidywalne zachowanie we wszystkich sytuacjach. Zmodyfikowana wersja zawiera jeden warunek wstępny — a nie może mieć wartości `null`. Jeśli wywołujący nie spełni tego warunku, wszystkie założenia przestają działać i metoda może zrobić, co zechce. W rzeczywistości, kiedy zastosujemy asercję, działanie metody w przypadku nieprawidłowego wywołania jest nieprzewidywalne. Może zgłosić błąd asercji albo wyjątek `NullPointerException`, w zależności od ustawień modułu ładującego klasy.

11.4.3. Zastosowanie asercji do dokumentowania założeń

Wielu programistów używa komentarzy do dokumentowania swoich założeń. Spójrzmy na przykład z pliku <http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html>:

```
if (i % 3 == 0)
    .
    .
else if (i % 3 == 1)
    .
    .
else // (i % 3 == 2)
    .
    .
```

W tym przypadku rozsądnie byłoby skorzystać z asercji:

```
if (i % 3 == 0)
    .
    .
else if (i % 3 == 1)
    .
    .
else
{
    assert i % 3 == 2;
    .
}
```

Oczywiście jeszcze lepiej byłoby przemyśleć to dokładniej. Jakie są możliwe wartości działania `i % 3`? Jeśli `i` jest liczbą dodatnią, reszta może wynosić 0, 1 lub 2. Jeśli `i` jest liczbą ujemną, reszta może mieć wartość -1 lub -2. W związku z tym prawdziwe założenie mówi, że `i` nie może być liczbą ujemną. Lepsza byłaby następująca asercja przed instrukcją `if`:

```
assert i >= 0;
```

W każdym razie ten przykład pokazuje dobry sposób użycia asercji do sprawdzenia samego siebie przez programistę. Jak widać, asercje są taktycznym narzędziem używanym w testowaniu i debugowaniu. Natomiast rejestracja danych w dzienniku jest narzędziem strategicznym używanym w całym cyklu życia programu. Dziennikami zajmiemy się w kolejnym podrozdziale.

java.lang.ClassLoader **1.0**

■ void setDefaultAssertionStatus(boolean b) **1.4**

Włącza lub wyłącza asercje we wszystkich klasach ładowanych przez moduł ładowający. Ustawienie to może zostać przesłonięte na poziomie konkretnego pakietu lub konkretnej klasy.

■ void setClassAssertionStatus(String className, boolean b) **1.4**

Włącza lub wyłącza asercje w danej klasie i jej klasach wewnętrznych.

■ void setPackageAssertionStatus(String packageName, boolean b) **1.4**

Włącza lub wyłącza asercje we wszystkich klasach w danym pakiecie i ich podklasach.

■ void clearAssertionStatus() **1.4**

Usuwa wszystkie ustawienia klasowe i pakietowe stanu asercji oraz wyłącza wszystkie asercje w klasach ładowanych przez moduł.

11.5. Dzienniki

Żadnemu programiście nie jest obce wstawianie instrukcji `System.out.println` do kodu sprawiającego problemy w celu uzyskania wglądu w działanie programu. Po odkryciu źródła problemów instrukcję taką usuwamy, aby użyć jej ponownie, gdy wystąpi kolejny problem. API Logging ma za zadanie zlikwidować tę niedogodność. Poniżej znajduje się lista jego najważniejszych zalet:

- Można łatwo wyłączyć rejestrację wszystkich lub wybranych poziomów rekordów w dzienniku. Ich ponowne włączenie jest również łatwe.
- Wyłączanie rejestracji danych jest bardzo mało kosztowne, co znaczy, że pozostawiony w programie kod rejestrujący powoduje minimalne opóźnienia.
- Rekordy dziennika można wysyłać do różnych procedur obsługi, które wyświetlgą go w konsoli, zapiszą w pliku itd.
- Zarówno rejestratory (ang. *logger*), jak i procedury obsługi mogą filtrować rekordy. Filtr odsiewa niepotrzebne dane przy użyciu kryteriów podanych przez programistę.
- Rekordy w dzienniku mogą być formatowane na różne sposoby, na przykład jako czysty tekst lub XML.

- Aplikacja może posiadać wiele obiektów typu Logger (rejestratorów) o hierarchicznej strukturze nazw, na przykład com.mycompany.myapp, przypominającej nazwy pakietów.
- Domyślnie opcje konfiguracyjne rejestracji są zapisywane w pliku konfiguracyjnym. W razie potrzeby istnieje możliwość zmiany tego mechanizmu w aplikacji.

11.5.1. Podstawy zapisu do dziennika

Zapisu w dzienniku najprościej dokonać przy użyciu globalnego rejestratora i jego metody `info`:

```
Logger.global.info("Plik->Otwórz - wybrany element menu");
```

Domyślnie rekord jest drukowany następująco:

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen
INFO: Plik->Otwórz - wybrany element menu
```

Natomiast wywołanie:

```
Logger.global.setLevel(Level.OFF);
```

umieszczone w odpowiednim miejscu (na przykład na początku metody `main`) całkowicie wyłącza zapis do dziennika.



Dopóki nie zostanie poprawiony błąd 7184195, globalny rejestrator należy włączać za pomocą wywołania `Logger.getGlobal().setLevel(Level.INFO)`.

11.5.2. Zaawansowane techniki zapisu do dziennika

Znając podstawy, możemy przejść do bardziej zaawansowanych technik. W profesjonalnej aplikacji nie należy zapisywać wszystkich danych w jednym globalnym rejestratorze. Zamiast tego definiujemy własne rejestratory.

Aby utworzyć lub pobrać rejestrator, należy użyć metody `getLogger`:

```
private static final Logger myLogger =
Logger.getLogger("com.mycompany.myapp");
```

Podobnie jak nazwy pakietów, nazwy rejestratorów wykazują strukturę hierarchiczną. W rzeczywistości są one **bardziej** hierarchiczne niż pakiety. Pomiędzy pakietem a jego przodkiem nie ma żadnych związków semantycznych. Natomiast rejestratory potomne dzielą pewne właściwości ze swoimi przodkami. Jeśli na przykład ustawimy priorytet informacji zapisywanych przez rejestrator `com.mycompany`, rejestratory potomne odziedziczą ten priorytet.

Istnieje siedem poziomów ważności komunikatów:

- SEVERE,
- WARNING,

- INFO,
- CONFIG,
- FINE,
- FINER,
- FINEST.

Domyślnie włączone są trzy pierwsze poziomy. Aby ustawić inny poziom, można użyć poniższej instrukcji:

```
logger.setLevel(Level.FINE);
```

Od tej pory włączona jest rejestracja wszystkich poziomów od FINE w góre.

Aby włączyć zapis wszystkich poziomów, należy napisać Level.ALL, a aby całkowicie wyłączyć zapis do dziennika, należy napisać Level.OFF.

Każdy z poziomów posiada swoje metody, np.:

```
logger.warning(message);
logger.fine(message);
```

Można też użyć metody log z podanym poziomem:

```
logger.log(Level.FINE, message);
```



Przy ustawieniach domyślnych zapisywane są wszystkie rekordy poziomu INFO lub wyższego. W związku z tym dla komunikatów dotyczących debugowania należy używać poziomów CONFIG, FINE, FINER i FINEST, które przydają się w diagnostyce, ale są bezużyteczne dla użytkownika.



Zmieniając poziom zapisu na niższy od INFO, należy pamiętać o wprowadzeniu zmian w konfiguracji procedury obsługi dziennika. Domyślna procedura obsługi dziennika tłumii wszystkie komunikaty poniżej poziomu INFO. Szczegóły znajdują się w kolejnym podrozdziale.

W domyślnym rekordzie dziennika znajduje się nazwa klasy i metody zawierającej wywołanie rejestratora, zgodnie z danymi pobranymi ze stosu wywołań. Jeśli jednak maszyna wirtualna zoptymalizuje wykonywanie, precyzyjne informacje mogą być niedostępne. Aby uzyskać dokładne informacje o lokalizacji klasy i metody odpowiedzialnych za to wywołanie, można użyć metody logp. Jej sygnatura jest następująca:

```
void logp(Level l, String className, String methodName, String message)
```

Istnieją metody służące do śledzenia przepływu wykonywania:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

Na przykład:

```
int read(String file, String pattern)
{
    logger.entering("com.mycompany.mylib.Reader", "read",
        new Object[] { file, pattern });

    . . .

    logger.exiting("com.mycompany.mylib.Reader", "read", count);
    return count;
}
```

Te wywołania generują rekordy dziennika na poziomie FINER, które zaczynają się od łańcuchów ENTRY i RETURN.



W przeszłości metody rejestrujące z parametrem `Object[]` będą obsługiwały zmienne listy parametrów (tzw. `varargs`). Wtedy będzie można tworzyć wywołania typu `logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)`.

Zapis do dziennika jest często wykorzystywany w celu rejestrowania nieprzewidywalnych wyjątków. Są dwie metody, które zapisują w dzienniku opis wyjątku.

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Typowe zastosowania:

```
if (. . .)
{
    IOException exception = new IOException(" . . .");
    logger.throwing("com.mycompany.mylib.Reader", "read", exception);
    throw exception;
}

try
{
    . . .
}
catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

Metoda `throwing` zapisuje rekordy na poziomie FINER i komunikaty zaczynające się od słowa THROW.

11.5.3. Zmiana konfiguracji menedżera dzienników

Ustawienia systemu rejestrującego można zmienić w specjalnym pliku konfiguracyjnym. Domyślny plik konfiguracyjny to `jre/lib/logging.properties`.

Aby użyć innego pliku, należy ustawić właściwość `java.util.logging.config.file` na ten plik, uruchamiając aplikację za pomocą następującego polecenia:

```
java -Djava.util.logging.config.file=configFile MainClass
```



Menedżer dzienników jest inicjowany razem z maszyną wirtualną, zanim zostanie wykonana metoda `main`. Jeśli stosujesz wywołanie `System.setProperty("java.util.logging.config.file", file)` w metodzie `main`, to dodatkowo wywołaj metodę `LogManager.readConfiguration()`, aby ponownie zainicjować menedżer.

Aby zmienić domyślny poziom rejestracji, należy w pliku konfiguracyjnym zmienić poniższy wiersz:

```
.level=INFO
```

Poziomy rejestracji można określić dla każdego rejestratora osobno:

```
com.mycompany.myapp.level=FINE
```

Należy dodać przyrostek `.level` do nazwy rejestratora.

Jak przekonamy się dalej, rejestratory nie wysyłają komunikatów do konsoli, ponieważ jest to zadanie dla obiektów typu `Handler`. Obiekty te również mają swoje poziomy. Aby w konsoli były wyświetlane komunikaty poziomu `FINE`, należy zastosować następujące ustawienie:

```
java.util.logging.ConsoleHandler.level=FINE
```



Ustawienia menedżera dzienników **nie** są ustawieniami systemowymi. Uruchomienie aplikacji za pomocą polecenia `-Dcom.mycompany.myapp.level=FINE` nie ma żadnego wpływu na rejestrator.



Przynajmniej do Java SE 7 w dokumentacji klasy `LogManager` jest napisane, że właściwości `java.util.logging.config.class` i `java.util.logging.config.file` można ustawiać za pośrednictwem API `Preferences`. To nieprawda — zobacz <http://bugs.sun.com/bugdatabase/>.



Plik konfiguracji zapisu do dziennika jest przetwarzany przez klasę `java.util.logging.LogManager`. Można wybrać inny menedżer dzienników, ustawiając właściwość systemową `java.util.logging.manager` na nazwę jakiejś podklasy. Alternatywnie można zatrzymać standardowy menedżer, omijając inicjację z pliku konfiguracji zapisu do dziennika. Należy ustawić właściwość systemową `java.util.logging.config.class` na klasę, która ustawia właściwości menedżera dzienników w inny sposób. Więcej informacji można znaleźć w dokumentacji klasy `LogManager`.

Poziomy rejestracji można także zmieniać w działającym programie za pomocą programu `jconsole`. Informacje na ten temat można znaleźć pod adresem www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl.

11.5.4. Lokalizacja

Czasami konieczna jest lokalizacja komunikatów dziennika, aby były zrozumiałe dla użytkowników z różnych krajów. Lokalizacji poświęciliśmy rozdział 5. drugiego tomu. Tutaj krótko opisujemy, o czym trzeba pamiętać przy lokalizacji komunikatów dziennika.

Dane lokalizacyjne aplikacji są przechowywane w tak zwanych **pakietach lokalizacyjnych** (ang. *resource bundle*). Pakiet taki zawiera odwzorowania dla poszczególnych lokalizacji (jak Stany Zjednoczone czy Niemcy). Na przykład pakiet lokalizacyjny może odwzorowywać łańcuch `reading-File` na łańcuchy `Reading file` po angielsku i `Achtung! Datei wird eingelesen` po niemiecku.

Program może zawierać kilka pakietów lokalizacyjnych, np. jeden dla menu i jeden dla komunikatów dziennika. Każdy pakiet ma swoją nazwę (np. `com.mycompany.logmessages`). Dodając odwzorowania do pakietu lokalizacyjnego, należy dostarczyć plik dla każdej lokalizacji. Dane dotyczące języka angielskiego znajdują się w pliku o nazwie `com/mycompany/logmessages_en.properties`, a języka niemieckiego w pliku `com/mycompany/logmessages_de.properties` (`en` i `de` to kody językowe). Pliki te powinny się znajdować w tym samym miejscu co pliki klas aplikacji, aby klasa `ResourceBundle` mogła je automatycznie odszukać. Format tych plików to czysty tekst, a ich struktura wygląda następująco:

```
readingFile=Achtung! Datei wird eingelesen  
renamingFile=Datei wird umbenannt  
...
```

Pakiet lokalizacyjny można określić w chwili uzyskiwania dostępu do rejestratora:

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

Następnie należy podać klucz pakietu lokalizacyjnego (a nie rzeczywisty łańcuch komunikatu) dla komunikatu dziennika.

```
logger.info("readingFile");
```

Do lokalizowanych komunikatów często trzeba dodawać argumenty. W takich sytuacjach komunikat powinien zawierać symbole zastępcze `{0}`, `{1}` itd. Aby na przykład do komunikatu dziennika dodać nazwę pliku, należy wstawić następujący symbol zastępczy:

```
Reading file {0}.  
Achtung! Datei {0} wird eingelesen.
```

Następnie wartości do symboli zastępczych wstawiamy za pomocą jednego z poniższych wywołań:

```
logger.log(Level.INFO, "readingFile", fileName);  
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

11.5.5. Obiekty typu Handler

Domyślnie rejestratory wysyłają rekordy do obiektu typu `ConsoleHandler`, który drukuje je w strumieniu `System.err`. Ścisłej rzecz biorąc, rejestrator wysyła rekord do nadrzędnego obiektu `Handler`, a przodek najwyższego poziomu (o nazwie "") ma obiekt typu `ConsoleHandler`.

Podobnie jak rejestratory, obiekty Handler mają poziomy rejestracji. Aby rekord został zapisany, jego poziom musi być wyższy niż poziom **zarówno** rejestratora, jak i obiektu Handler. Plik konfiguracyjny menedżera dzienników ustawia poziom rejestracji domyślnego obiektu Handler konsoli następująco:

```
java.util.logging.ConsoleHandler.level=INFO
```

Aby rejestrować rekordy poziomu FINE, należy zmienić poziom domyślnego rejestratora i obiektu Handler w konfiguracji. Istnieje też możliwość całkowitego pominięcia pliku konfiguracyjnego i instalacji własnego obiektu Handler.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

Domyślnie rejestrator wysyła rekordy zarówno do swoich własnych obiektów Handler, jak i obiektów Handler swojego przodka. Nasz rejestrator jest potomkiem pierwotnego rejestratora (o nazwie ""), który wysyła wszystkie rekordy poziomu INFO lub wyższego do konsoli. Nie chcemy jednak oglądać tych rekordów dwa razy. Dlatego ustawiamy własność useParentHandlers na false.

Aby wysłać rekordy dziennika gdzieś indziej, trzeba dodać jeszcze jeden obiekt Handler. API dzienników udostępnia dwa przydatne typy obiektów Handler : FileHandler i SocketHandler. Ten drugi wysyła rekordy do określonego hosta i portu. Nas bardziej interesuje FileHandler, który zapisuje rekordy w plikach.

Aby wysłać rekordy do domyślnego obiektu Handler zapisującego do plików, można użyć poniższego kodu:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

Rekordy są wysyłane do pliku o nazwie *javan.log* znajdującego się w katalogu głównym użytkownika — *n* to liczba odróżniająca poszczególne pliki. Jeśli w systemie (np. Windows 95/98/Me) nie ma czegoś takiego jak katalog główny, plik jest zapisywany w domyślnej lokalizacji, np. *C:\Windows*. Domyślnym formatem rekordów jest XML. Typowy rekord w dzienniku wygląda następująco:

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

Domyślne zachowanie obiektu Handler zapisującego do pliku można zmienić za pomocą różnych ustawień w konfiguracji menedżera dzienników (tabela 11.1) lub przy użyciu innego konstruktora (zobacz wyciągi z API na końcu tego podrozdziału).

Tabela 11.1. Parametry konfiguracyjne obiektu Handler zapisującego do pliku

Właściwość konfiguracyjna	Opis	Wartość domyślna
java.util.logging.FileHandler.level	Poziom rejestracyjny obiektu Handler	Level.ALL
java.util.logging.FileHandler.append	O określa, czy handler powinien dopisywać dane do istniejącego pliku, czy dla każdego uruchomionego programu otwierać nowy plik	false
java.util.logging.FileHandler.limit	Przybliżona maksymalna liczba bajtów, którą można zapisać w pliku, zanim zostanie otwarty kolejny (0 oznacza brak limitu)	0 (brak limitu) w klasie FileHandler, 50000 w konfiguracji domyślnego menedżera dzienników
java.util.logging.FileHandler.pattern	Wzorzec nazwy pliku dziennika. Zmienne wzorców zostały opisane w tabeli 11.2	%h/java%u.log
java.util.logging.FileHandler.count	Liczba dzienników w cyklu rotacyjnym	1 (brak rotacji)
java.util.logging.FileHandler.filter	Klasa filtru, która ma zostać zastosowana	Brak filtru
java.util.logging.FileHandler.encoding	Kodowanie znaków	Kodowanie platformy
java.util.logging.FileHandler.formatter	Formater rekordów	java.util.logging. ➥XMLFormatter

Zazwyczaj programista nie chce używać domyślnej nazwy pliku dziennika. W tym celu należy zastosować inny wzorzec, jak `%h/myapp.log` (zmienne wzorców zostały przedstawione w tabeli 11.2).

Jeśli kilka aplikacji (lub kilka kopii jednej aplikacji) używa tego samego dziennika, należy włączyć znacznik append lub użyć zmiennej `%u` we wzorcu nazwy pliku, aby każda aplikacja tworzyła unikalną kopię dziennika.

Dobrym pomysłem jest też włączenie rotacji plików. Pliki dzienników są przechowywane w kolejności rotacyjnej — `myapp.log.0`, `myapp.log.1`, `myapp.log.2` itd. Kiedy plik przekroczy dopuszczalny rozmiar, najstarszy dziennik jest usuwany, pozostałe pliki mają zmieniane nazwy i tworzony jest nowy plik z numerem `0`.

Istnieje możliwość zdefiniowania własnego typu Handler poprzez rozszerzenie klasy `Handler` lub `StreamHandler`. Przykład takiego typu prezentujemy w programie demonstracyjnym zamieszczonym na końcu tego podrozdziału. Wyświetla on rekordy w oknie (zobacz rysunek 11.2).

Tabela 11.2. Zmienne wzorców nazw plików dziennika

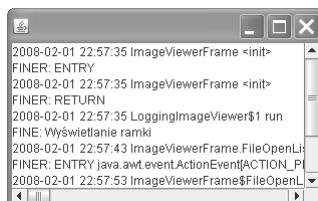
Zmienna	Opis
%h	Wartość właściwości user.home
%t	Katalog tymczasowy systemu
%u	Unikalna liczba pozwalająca uniknąć konfliktów
%g	Numer pokolenia dzienników (przyrostek .%g jest używany, jeśli rotacja jest włączona, a wzorzec nie zawiera zmiennej %g)
%%	Znak %



Wielu programistów wykorzystuje dzienniki jako pomoc dla obsługi technicznej. Jeśli program źle działa, użytkownik może odesłać dzienniki do sprawdzenia. W takim przypadku należy włączyć znacznik append bądź używać dzienników rotacyjnych albo jedno i drugie.

Rysunek 11.2.

Klasa typu Handler wyświetlająca rekordy w oknie



Nasza klasa rozszerza klasę StreamHandler i instaluje strumień, którego metody write drukują dane wyjściowe w obszarze tekstowym.

```

class WindowHandler extends StreamHandler
{
    public WindowHandler()
    {

        final JTextArea output = new JTextArea();
        setOutputStream(new
            OutputStream()
        {
            public void write(int b) {} //nie jest wywoływana
            public void write(byte[] b, int off, int len)
            {
                output.append(new String(b, off, len));
            }
        });
    }
}

```

Z metodą tą związany jest tylko jeden problem — obiekt Handler buforuje rekordy i wysyła je do strumienia dopiero po zapełnieniu bufora. Dlatego przeddefiniowaliśmy metodę publish, aby opróżniała bufor po każdym rekordzie.

```
class WindowHandler extends StreamHandler
{
    ...
    public void publish(LogRecord record)
    {
        super.publish(record);
        flush();
    }
}
```

Aby napisać bardziej niezwykłą klasę Handler dla strumieni, można rozszerzyć klasę Handler i zdefiniować metody publish, flush oraz close.

11.5.6. Filtry

Domyślnie rekordy są filtrowane zgodnie z ich priorytetami. Jednak każdy rejestrator i obiekt Handler może posiadać dodatkowy filtr rekordów. Definicja filtru polega na zaimplementowaniu interfejsu Filter i zdefiniowaniu poniższej metody:

```
boolean isLoggable(LogRecord record)
```

Po przeanalizowaniu rekordu dziennika, stosując dowolne kryteria, zwracamy wartość true dla tych rekordów, które powinny zostać dodane do dziennika. Na przykład można utworzyć filtr akceptujący tylko komunikaty wygenerowane przez metody entering i exiting. Filtr ten powinien zatem wywoływać metodę record.getMessage() i sprawdzać, czy komunikat zaczyna się od słowa ENTRY lub RETURN.

Instalacja filtru w rejestratorze lub obiekcie Handler sprowadza się do wywołania metody setFilter. Pamiętajmy, że można używać tylko jednego filtru naraz.

11.5.7. Formatory

Klasy ConsoleHandler i FileHandler tworzą rekordy dzienników w formacie tekstowym lub XML. Można jednak zdefiniować własny format. W tym celu należy rozszerzyć klasę Formatter i przededefiniować poniższą metodę:

```
String format(LogRecord record)
```

Informacje zawarte w rekordzie formatujemy w dowolny sposób i zwracamy powstały łańcuch. We własnej metodzie format możemy wywołać poniższą metodę:

```
String formatMessage(LogRecord record)
```

Ta metoda formatuje komunikat zawarty w rekordzie, zastępując parametry i stosując lokalizację.

Wiele formatów plików (np. XML) wymaga, aby formatowane rekordy miały jakiś nagłówek i stopkę. W takim przypadku należy przesłonić poniższe metody:

```
String getHead(Handler h)
String getTail(Handler h)
```

Na koniec wywołujemy metodę `setFormatter`, aby zainstalować formater w obiekcie `Handler`.

11.5.8. Przepis na dziennik

Przy tak dużej liczbie opcji związanych z dziennikami łatwo zapomnieć o podstawach. Poniższy przepis podsumowuje najczęstsze operacje.

1. W prostej aplikacji użyj jednego rejestratora. Dobrym pomysłem jest nadanie temu rejestratorowi takiej samej nazwy jak ogólnemu pakietowi aplikacji, np. `com.mycompany.myprog`. Rejestrator można zawsze utworzyć za pomocą poniższej instrukcji:

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

Dla wygody do klas, które robią dużo zapisów, można dodać pola statyczne:

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

2. Przy domyślnych ustawieniach wszystkie komunikaty poziomu `INFO` lub wyższego są zapisywane w konsoli. Można zmienić to domyślne ustawienie, ale — jak się przekonaliśmy — wymaga to nieco pracy. W związku z tym lepiej jest zastosować w aplikacji bardziej rozsądne ustawienie domyślne.

Poniższy kod zapewnia, że wszystkie komunikaty są zapisywane do pliku właściwego dla danej aplikacji. Należy go umieścić w metodzie `main`.

```
if (System.getProperty("java.util.logging.config.class") == null
    && System.getProperty("java.util.logging.config.file") == null)
{
    try
    {
        Logger.getLogger("").setLevel(Level.ALL);
        final int LOG_ROTATION_COUNT = 10;
        Handler handler = new FileHandler("%h/myapp.log", 0, LOG_ROTATION_COUNT);
        Logger.getLogger("").addHandler(handler);
    }
    catch (IOException e)
    {
        logger.log(Level.SEVERE, "Nie można utworzyć handlера pliku dziennika", e);
    }
}
```

3. Teraz jesteśmy gotowi do zapisywania danych w dzienniku. Pamiętajmy, że wszystkie komunikaty poziomów `INFO`, `WARNING` i `SEVERE` są drukowane w konsoli. Należy zatem zarezerwować je dla komunikatów mających znaczenie dla użytkowników programu. Poziom `FINE` doskonale nadaje się do komunikatów przeznaczonych dla programistów.

Wszędzie, gdzie kusi użycie metody `System.out.println`, lepiej utworzyć komunikat dziennika:

```
logger.fine("Okno dialogowe wyboru pliku zostało anulowane");
```

Dobrym pomysłem jest też zapisywanie w dzienniku niespodziewanych wyjątków. Na przykład:

```
try
{
    ...
}
catch (SomeException e)
{
    logger.log(Level.FINE, "objaśnienie", e);
}
```

Listing 11.2 pokazuje zastosowanie powyższego przepisu w praktyce z jedną modyfikacją: komunikaty dziennika są dodatkowo wyświetlane w oknie.

Listing 11.2. *logging/LoggingImageViewer.java*

```
package logging;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.logging.*;
import javax.swing.*;

/**
 * Zmodyfikowana przeglądarka grafiki, która zapisuje w dzienniku informacje o różnych zdarzeniach
 * @version 1.02 2007-05-31
 * @author Cay Horstmann
 */
public class LoggingImageViewer
{
    public static void main(String[] args)
    {
        if (System.getProperty("java.util.logging.config.class") == null
            && System.getProperty("java.util.logging.config.file") == null)
        {
            try
            {
                Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
                final int LOG_ROTATION_COUNT = 10;
                Handler handler = new FileHandler("%h/LoggingImageViewer.log", 0,
                    LOG_ROTATION_COUNT);
                Logger.getLogger("com.horstmann.corejava").addHandler(handler);
            }
            catch (IOException e)
            {
                Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
                    "Nie można utworzyć obiektu obsługi pliku dziennika", e);
            }
        }

        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                Handler windowHandler = new WindowHandler();
                windowHandler.setLevel(Level.ALL);
                Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);
            }
        });
    }
}
```

```

JFrame frame = new ImageViewerFrame();
frame.setTitle("LoggingImageViewer");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Logger.getLogger("com.horstmann.corejava").fine("Wyświetlanie ramki");
frame.setVisible(true);
}

}

}

/***
 * Ramka zawierająca obraz
 */
class ImageViewerFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 400;

    private JLabel label;
    private static Logger logger = Logger.getLogger("com.horstmann.corejava");

    public ImageViewerFrame()
    {
        logger.entering("ImageViewerFrame", "<init>");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // Pasek menu
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("Plik");
        menuBar.add(menu);

        JMenuItem openItem = new JMenuItem("Otwórz");
        menu.add(openItem);
        openItem.addActionListener(new FileOpenListener());

        JMenuItem exitItem = new JMenuItem("Zamknij");
        menu.add(exitItem);
        exitItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                logger.fine("Zamykanie.");
                System.exit(0);
            }
        });
    }

    // Etykieta
    label = new JLabel();
    add(label);
    logger.exiting("ImageViewerFrame", "<init>");
}

private class FileOpenListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        logger.fine("FileOpenListener.actionPerformed");
    }
}

```

```

{
    logger.entering("ImageViewerFrame.FileOpenListener", "actionPerformed",
    event);

    // Okno wyboru plików
    JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    // Akceptowanie wszystkich plików z rozszerzeniem .gif
    chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
    {
        public boolean accept(File f)
        {
            return f.getName().toLowerCase().endsWith(".gif") ||
            f.isDirectory();
        }

        public String getDescription()
        {
            return "Obrazy GIF";
        }
    });

    // Wyświetlanie okna dialogowego wyboru plików
    int r = chooser.showOpenDialog(ImageViewerFrame.this);

    // Jeśli plik obrazu został zaakceptowany, jest on ustawiany jako ikona etykiety
    if (r == JFileChooser.APPROVE_OPTION)
    {
        String name = chooser.getSelectedFile().getPath();
        logger.log(Level.FINE, "Wczytywanie pliku {0}", name);
        label.setIcon(new ImageIcon(name));
    }
    else logger.fine("Anulowano okno otwierania pliku.");
    logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
}
}

/*
 * Klasa obslugi wyświetlania rekordów dziennika w oknie
 */
class WindowHandler extends StreamHandler
{
    private JFrame frame;

    public WindowHandler()
    {
        frame = new JFrame();
        final JTextArea output = new JTextArea();
        output.setEditable(false);
        frame.setSize(200, 200);
        frame.add(new JScrollPane(output));
        frame.setFocusableWindowState(false);
        frame.setVisible(true);
        setOutputStream(new OutputStream()
        {
            public void write(int b)


```

```

    {
    } // nie jest wywoływanego

    public void write(byte[] b, int off, int len)
    {
        output.append(new String(b, off, len));
    }
};

public void publish(LogRecord record)
{
    if (!frame.isVisible()) return;
    super.publish(record);
    flush();
}
}

```

java.util.logging.Logger 1.4

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

Zwraca rejestrator o podanej nazwie. Jeśli rejestrator ten nie istnieje, tworzy go.

Parametry: `loggerName` Hierarchiczna nazwa rejestratora,
np. `com.mycompany.myapp`

`bundleName` Nazwa pakietu zasobu, w którym można szukać
zlokalizowanych tekstów

- `void severe(String message)`
- `void warning(String message)`
- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

Zapisuje rekord z poziomem określonym przez nazwę metody i podanym komunikatem.

- `void entering(String className, String methodName)`
- `void entering(String className, String methodName, Object param)`
- `void entering(String className, String methodName, Object[] param)`
- `void exiting(String className, String methodName)`
- `void exiting(String className, String methodName, Object result)`

Zapisuje rekord opisujący uruchamianie lub zamykanie metody o podanych parametrach lub wartości zwrotnej.

- `void throwing(String className, String methodName, Throwable t)`

Zapisuje rekord opisujący generowanie danego obiektu wyjątku.

- `void log(Level level, String message)`

- `void log(Level level, String message, Object obj)`

- `void log(Level level, String message, Object[] objs)`

- `void log(Level level, String message, Throwable t)`

Zapisuje rekord z podanym poziomem i komunikatem oraz opcjonalnie dodaje obiekty typu `Throwable`. Aby obiekty zostały dodane, komunikat musi zawierać symbole zastępcze formatu — `{0}, {1}` itd.

- `void logp(Level level, String className, String methodName, String message)`

- `void logp(Level level, String className, String methodName, String message, Object obj)`

- `void logp(Level level, String className, String methodName, String message, Object[] objs)`

- `void logp(Level level, String className, String methodName, String message, Throwable t)`

Zapisuje rekord o danym poziomie, szczegółowe dane wywołującego i komunikat oraz opcjonalnie dołącza obiekty lub obiekt `Throwable`.

- `void logrb(Level level, String className, String methodName, String bundleName, String message)`

- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`

- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`

- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`

Zapisuje rekord o danym poziomie, szczegółowe dane wywołującego, nazwę pakietu lokalizacyjnego, komunikat oraz opcjonalnie dołącza obiekty lub obiekt `Throwable`.

- `Level getLevel()`

- `void setLevel(Level l)`

Pobiera i ustawia poziom rejestratora.

- `Logger getParent()`

- `void setParent(Logger l)`

Pobiera i ustawia rejestrator nadzędny rejestratora.

- `Handler[] getHandlers()`

Zwraca wszystkie obiekty `Handler` rejestratora.

- void addHandler(Handler h)
 - void removeHandler(Handler h)
- Dodaje lub usuwa obiekt Handler rejestratora.
- boolean getUseParentHandlers()
 - void setUseParentHandlers(boolean b)
- Sprawdza bądź ustawia właściwość use parent handler. Jeśli właściwość ta ma wartość true, rejestrator przesyła wszystkie zapisane rekordy do obiektów Handler swojego przodka.
- Filter getFilter()
 - void setFilter(Filter f)
- Sprawdza i ustawia filtr dla rejestratora.

java.util.logging.Handler 1.4

- abstract void publish(LogRecord record)
- Wysyła rekord w odpowiednie miejsce.
- abstract void flush()
- Opróżnia bufor.
- abstract void close()
- Opróżnia bufor i zwalnia wszystkie powiązane zasoby.
- Filter getFilter()
 - void setFilter(Filter f)
- Sprawdza i ustawia filtr dla obiektu Handler.
- Formatter getFormatter()
 - void setFormatter(Formatter f)
- Sprawdza i ustawia formater obiektu Handler.
- Level getLevel()
 - void setLevel(Level l)
- Sprawdza i ustawia poziom obiektu Handler.

java.util.logging.ConsoleHandler 1.4

- ConsoleHandler()
- Tworzy nowy obiekt Handler konsoli.

java.util.logging.FileHandler 1.4

- FileHandler(String pattern)
- FileHandler(String pattern, boolean append)

- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`

Tworzy obiekt `Handler` zapisujący do plików.

Parametry:	<code>pattern</code>	Wzorzec nazwy dziennika. Zmienne tego wzorca zostały opisane w tabeli 11.2
	<code>limit</code>	Przybliżona maksymalna liczba bajtów, która musi być zapisana przed otwarciem kolejnego pliku
	<code>count</code>	Liczba plików w cyklu rotacyjnym
	<code>append</code>	Wartość <code>true</code> oznacza, że nowo utworzony obiekt <code>Handler</code> zapisu do plików powinien dodawać dane do istniejącego pliku dziennika

java.util.logging.LogRecord 1.4

- `Level getLevel()`
Zwraca poziom rekordu.
- `String getLoggerName()`
Zwraca nazwę rejestratora, który zarejestrował dany rekord.
- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`
Zwraca pakiet lokalizacyjny, który ma zostać użyty do lokalizacji komunikatu, bądź jego nazwę albo wartość `null`, jeśli nie dostarczono żadnego pakietu.
- `String getMessage()`
Pobiera surowy komunikat przed lokalizacją lub formatowaniem.
- `Object[] getParameters()`
Zwraca obiekty parametrów lub wartość `null`, jeśli nie ma żadnego parametru.
- `Throwable getThrown()`
Zwraca wyrzucony obiekt lub wartość `null`, jeśli nie ma takiego obiektu.
- `String getSourceClassName()`
- `String getSourceMethodName()`
Zwraca lokalizację procedury, która zarejestrowała rekord. Informacja ta może zostać dostarczona przez procedurę rejestrującą lub pobrana bezpośrednio ze stosu wywołań. Może być nieprawidłowa, jeśli procedura rejestrująca poda nieprawidłową wartość lub uruchomiony fragment kodu został zoptymalizowany, przez co dokładnej lokalizacji nie da się sprawdzić.
- `long getMillis()`
Zwraca czas tworzenia w milisekundach, od 1970 roku.

- `long getSequenceNumber()`

Zwraca unikalny numer sekwencji rekordu.

- `int getThreadID()`

Zwraca unikalny identyfikator ID wątku, w którym został utworzony rekord. Identyfikatory te są przypisywane przez klasę `LogRecord` i nie mają żadnego związku z identyfikatorami innych wątków.

java.util.logging.Filter 1.4

- `boolean isLoggable(LogRecord record)`

Zwraca wartość `true`, jeśli dany rekord dziennika powinien zostać zapisany.

java.util.logging.Formatter 1.4

- `abstract String format(LogRecord record)`

Zwraca łańcuch powstały w wyniku sformatowania danego rekordu.

- `String getHead(Handler h)`
- `String getTail(Handler h)`

Zwraca łańcuchy, które powinny się znajdować w nagłówku i na dole dokumentu zawierającego rekordy dziennika. Zgodnie z definicją w nadklasie `Formatter` metody te zwracają pusty łańcuch. W razie potrzeby należy je przesłonić.

- `String formatMessage(LogRecord record)`

Zwraca zlokalizowany i sformatowany komunikat zawarty w rekordzie.

11.6. Wskazówki dotyczące debugowania

Wyobraźmy sobie, że napisaliśmy solidny program, w którym przechwyciliśmy i odpowiednio obsłużyliśmy wszystkie wyjątki. Uruchamiamy go i okazuje się, że nie działa zgodnie z oczekiwaniami. Co teraz? (Osoby, które nigdy nie miały takiego problemu, mogą pominąć tę część rozdziału).

Oczywiście najlepiej postarać się o dobry i wszechstronny debugger. Każde profesjonalne środowisko programistyczne, takie jak Eclipse czy NetBeans, ma wbudowany debugger. Narzędziem tym zajmiemy się dalej, a teraz przedstawiamy kilka wskazówek, które można wypróbować przed jego wyłączeniem.

- 1 Wartość każdej zmiennej można wydrukować lub zarejestrować, używając poniższego kodu:

```
System.out.println("x=" + x);
```

lub

```
Logger.global.info("x=" + x);
```

Jeśli `x` jest liczbą, zostanie przekonwertowany na łańcuch. Jeśli jest obiektem, zostanie wywołana na jego rzecz metoda `toString()`. Aby sprawdzić stan obiektu parametru niejawnego, należy wydrukować stan obiektu `this`.

```
Logger.global.info("this=" + this);
```

Większość klas w bibliotece Javy bardzo sumiennie przesłania metodę `toString()`, aby móc zwrócić przydatne informacje o danej klasie. Jest to prawdziwy skarb w trakcie debugowania. Należy to samo robić we własnych klasach.

2. Jedna sztuczka — wydaje się, że mało znana — polega na umieszczeniu w każdej klasie osobnej metody `main`. W metodzie tej można umieścić namiastkę testu jednostkowego umożliwiającą przetestowanie klasy w odosobnieniu.

```
public class MyClass
{
    metody i pola
    ...
    public static void main(String[] args)
    {
        procedury testowe
    }
}
```

Utwórz kilka obiektów, wywołaj wszystkie metody i sprawdź, czy każda z nich zwraca prawidłową wartość. Aby uruchomić testy, należy każdy z plików wykonać w maszynie wirtualnej osobno. Przy uruchamianiu apletu żadna z tych metod nie jest w ogóle wywoływana. Przy uruchamianiu aplikacji maszyna wirtualna wywołuje tylko metodę `main` klasy uruchomieniowej.

3. Osoby, którym spodobała się poprzednia wskazówka, powinny zainteresować się narzędziem JUnit ze strony <http://junit.org>. Jest to bardzo popularny framework testowy, który ułatwia organizację zestawów przypadków testowych. Testy powinno się przeprowadzać po wprowadzeniu jakichś zmian w klasie, a po znalezieniu błędu należy dodać nowe testy.
4. **Rejestrujący obiekt pośredni** (ang. *logging proxy*) to obiekt podklasy, który przechwytuje wszystkie wywołania metod, rejestruje je, a następnie wywołuje nadklasę. Jeśli na przykład mamy problem z metodą `nextDouble` z klasy `Random`, możemy utworzyć obiekt pośredni będący egzemplarzem podklasy anonimowej:

```
Random generator = new
{
    Random()
    {
        public double nextDouble()
        {
            double result = super.nextDouble();
            Logger.getGlobal().info("nextDouble: " + result);
            return result;
        }
    };
}
```

Przy każdym wywołaniu metody `getDouble` tworzony jest rekord w dzienniku.

Aby dowiedzieć się, kto wywołał metodę, należy sprawdzić stos wywołań.

5. Stos można uzyskać z każdego obiektu wyjątku za pomocą metody printStackTrace z klasy Throwable. Poniższy fragment programu przechwytuje wszystkie wyjątki, drukuje ich obiekty i ślad stosu oraz ponownie je generuje, aby mogły odnaleźć swoją procedurę obsługi.

```
try
{
    ...
}
catch (Throwable t)
{
    t.printStackTrace();
    throw t;
}
```

Aby wygenerować ślad stosu, nie trzeba nawet przechwytywać wyjątku. Wystarczy poniższa instrukcja w dowolnym miejscu programu:

```
Thread.dumpStack();
```

6. Normalnie dane ze śledzenia stosu są wysyłane do strumienia System.out. Można użyć metody void printStackTrace(PrintWriter s), aby zapisać je w pliku. Aby zarejestrować lub wyświetlić dane ze śledzenia stosu, można użyć poniższego kodu:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
new Throwable().printStackTrace(out);
String description = out.toString();
```

7. Często wygodnym rozwiązaniem jest zapisanie błędów programu w pliku. Są one jednak wysyłane do strumienia System.out, a nie System.out, przez co nie można ich zapisać przy użyciu poniższego polecenia:

```
java MyProgram > errors.txt
```

Zamiast tego należy przechwycić strumień błędów jako:

```
java MyProgram 2> errors.txt
```

Aby zapisać zarówno strumień System.out, jak i System.out w jednym pliku, należy użyć następującego polecenia:

```
java MyProgram >& errors.txt
```

Sposób ten działa w powłoce bash i Windows.

8. Zapisywanie danych ze śledzenia stosu nieprzechwyconych wyjątków w strumieniu System.out nie jest idealnym rozwiązaniem. Komunikaty te wprowadzają zamęt u użytkowników końcowych, którzy je przez przypadek zobaczą, i nie są dostępne do celów diagnostycznych, kiedy są potrzebne. Lepiej jest zapisywać je w pliku. Można też zmienić procedurę obsługi nieprzechwyconych wyjątków za pomocą metody Thread.setDefaultUncaughtExceptionHandler:

```
Thread.setDefaultUncaughtExceptionHandler(
    new Thread.UncaughtExceptionHandler()
    {
        public void uncaughtException(Thread t, Throwable e)
    }
}
```

```
        zapis informacji w pliku dziennika
    };
```

9. Aby zobaczyć, jakie klasy są ładowane, należy uruchomić maszynę wirtualną przy użyciu znacznika `-verbose`. Zostaną wydrukowane informacje podobne do poniższych:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
...
...
```

Metoda ta może czasami pomóc w diagnozowaniu problemów ze ścieżką klas.

10. Opcja `-Xlint` znajduje najczęstsze problemy z kodem. Jeśli na przykład program skompilujemy za pomocą poniższego polecenia:

```
javac -Xlint:fallthrough
```

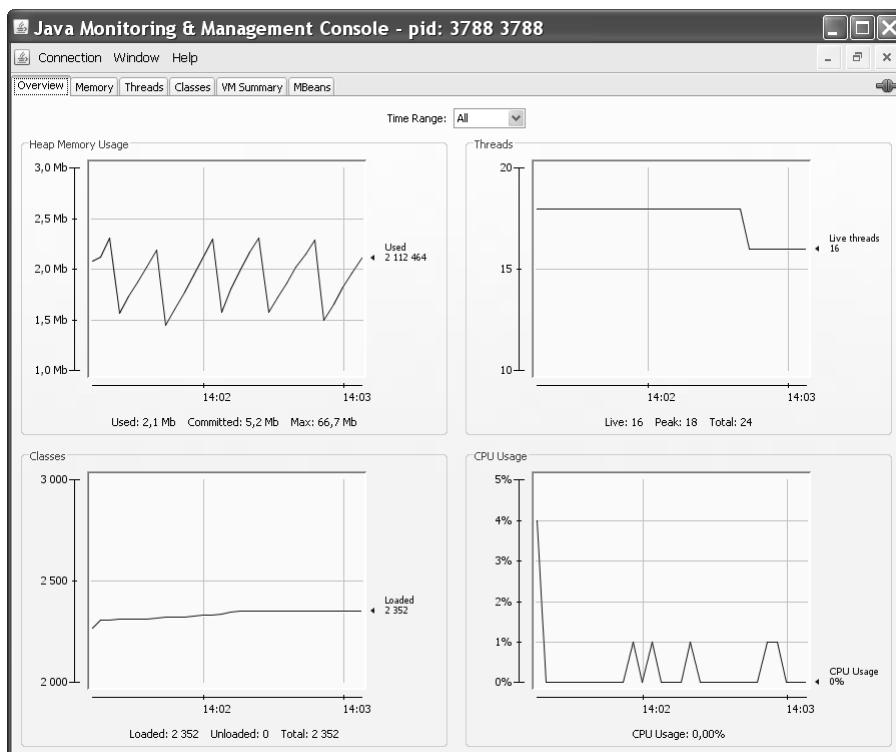
kompilator zgłosi brakujące instrukcje `break` w instrukcjach `switch` (mianem *lint* pierwotnie określano narzędzie służące do znajdowania potencjalnych problemów w programach w języku C, a obecnie nazwę tę stosuje się do narzędzi, które oznaczają konstrukcje budzące wątpliwości, ale nie niedozwolone).

Dostępne są następujące opcje:

<code>-Xlint</code> lub <code>-Xlint:all</code>	Przeprowadza wszystkie testy.
<code>-Xlint:deprecation</code>	Działa tak samo jak opcja <code>-deprecation</code> — wyszukuje odradzane metody.
<code>-Xlint:fallthrough</code>	Szuka brakujących instrukcji <code>break</code> w instrukcjach <code>switch</code> .
<code>-Xlint:finally</code>	Ostrzega o klauzulach <code>finally</code> , które nie mogą się normalnie zakończyć.
<code>-Xlint:none</code>	Nie przeprowadza żadnego testu.
<code>-Xlint:path</code>	Sprawdza, czy wszystkie katalogi na ścieżce klas istnieją.
<code>-Xlint:serial</code>	Ostrzega o serializowalnych klasach bez <code>serialVersionUID</code> (zobacz rozdział 1. drugiego tomu).
<code>-Xlint:unchecked</code>	Ostrzega przed niebezpiecznymi konwersjami pomiędzy typami uogólnionymi a surowymi (zobacz rozdział 12.).

11. Maszyna wirtualna Javy może też monitorować aplikacje i zarządzać nimi. Polega to na instalacji agentów w maszynie wirtualnej, które śledzą zużycie pamięci, wykorzystanie wątków, ładowanie klas itd. Funkcja ta jest szczególnie przydatna w dużych i długo działających aplikacjach, takich jak serwery. Demonstracją tych możliwości jest dostępne w JDK narzędzie graficzne o nazwie *jconsole*, które wyświetla statystyki dotyczące działania maszyny wirtualnej (rysunek 11.3). Należy znaleźć identyfikator procesu, który w systemie operacyjnym obsługuje maszynę wirtualną. W systemach Unix/Linux należy w tym celu użyć narzędzia *ps*, w systemie Windows trzeba skorzystać z menedżera zadań¹. Następnie uruchamiamy program *jconsole*:

jconsole IDprocesu



Rysunek 11.3. Program *jconsole*

Konsola dostarcza mnóstwo informacji na temat uruchomionego programu. Więcej informacji można znaleźć na stronie www.oracle.com/technetwork/articles/java/jconsole-1564139.html.

¹ W systemie Windows po uruchomieniu okna *Menedżer zadań* za pomocą klawiszy *Ctrl+Alt+Delete* należy przejść do karty *Procesy* i z menu *Widok* wybrać opcję *Wybierz kolumny* oraz zaznaczyć pole *PID* (*identyfikator procesu*) — przyp. tłum.

- 12.** Za pomocą narzędzia *jmap* można sprawdzić, jakie obiekty znajdują się na stercie. Służą do tego następujące polecenia:

```
jmap -dump:format=b,file=nazwaPlikuZrzutu IDprocesu
jhat NazwaPlikuZrzutu
```

Następnie w przeglądarce należy wpisać adres *localhost:7000*. Umożliwia to dostęp do zawartości sterty w czasie zrzutu.

- 13.** Uruchomienie maszyny wirtualnej ze znacznikiem *-Xprof* powoduje wywołanie prostego narzędzia profilującego, które śledzi najczęściej uruchamiane metody w kodzie. Informacje te są wysyłane do strumienia *System.out* i informują między innymi o tym, które metody zostały skompilowane przez kompilator JIT.



Opcje *-X* kompilatora nie są oficjalnie obsługiwane i mogą być niedostępne w niektórych wersjach JDK. Aby sprawdzić, jakie niestandardowe opcje są dostępne, należy użyć polecenia *java -X*.

11.7. Wskazówki dotyczące debugowania aplikacji z GUI

W tej części rozdziału zamieściliśmy kilka dodatkowych porad na temat znajdowania błędów w programach z graficznym interfejsem użytkownika. Później pokażemy, jak zautomatyzować testy GUI za pomocą robota AWT.

- 1.** Jeśli kiedykolwiek zdarzyło Ci się spojrzeć na okno Swinga i zastanawiać, jak jego twórca uzyskał tak dobry efekt, możesz uzyskać nieco informacji. Naciśnij kombinację klawiszy *Ctrl+Shift+F1*, aby wyświetlić wydruk wszystkich komponentów w hierarchii:

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,.....
javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.JRootPane$RootLayout,.....
javax.swing.JPanel[null,glassPane,0,0,292x173,hidden,layout=java.awt.FlowLayout,.....
javax.swing.JLayeredPane[null,layeredPane,0,0,292x173,.....
javax.swing.JPanel[null,contentPane,0,0,292x173,layout=java.awt.GridBagLayout,.....
javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null,.....
javax.swing.CellRendererPane[,0,0,0x0,hidden]
javax.swing.DefaultListCellRenderer$UIResource[-73,-19,0x0,.....
javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.OverlayLayout,.....
javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.OverlayLayout,.....
javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null,.....
javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null,.....
javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=null,.....
```

- 2.** Osoby mające problemy z prawidłową prezentacją komponentów Swing polubią narzędzie **Swing graphics debugger**. Nawet jeśli nie piszemy własnych klas komponentów, ciekawe i kształcące jest podejrzenie, jak dokładnie rysowana jest

zawartość komponentu. Aby włączyć debugowanie komponentu Swing, należy użyć metody `setDebugGraphicsOptions` z klasy `JComponent`. Dostępne są następujące opcje:

<code>DebugGraphics.FLASH_OPTION</code>	Zabarwia na chwilę na czerwono każdą linię, prostokąt i fragment tekstu przed ich narysowaniem.
<code>DebugGraphics.LOG_OPTION</code>	Drukuje komunikat dotyczący każdej operacji rysowania.
<code>DebugGraphics.BUFFERED_OPTION</code>	Wyświetla operacje, które są wykonywane w buforze pozaekranowym.
<code>DebugGraphics.NONE_OPTION</code>	Wyłącza debugowanie grafiki.

Odkryliśmy, że aby opcja `FLASH` działała, trzeba wyłączyć `double buffering`, funkcję Swing mającą na celu redukcję migotania podczas aktualizacji okna. Magiczne zaklęcie włączające opcję `FLASH` brzmi następująco:

```
RepaintManager.currentManager(getRootPane()).setDoubleBufferingEnabled(false);
((JComponent) getContentPane()).setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

Kod ten należy umieścić na końcu konstruktora ramki. W czasie działania programu panel główny będzie się zapełniał w zwolnionym tempie. Bardziej precyzyjne debugowanie można wykonać, wywołując metodę `setDebugGraphicsOptions` na rzecz jednego komponentu. Można ustawić czas, liczbę i kolor błysków — szczegóły na ten temat można znaleźć w dokumentacji internetowej metody `DebugGraphics`.

3. Jeśli chcesz otrzymać rejestr wszystkich zdarzeń AWT wygenerowanych w aplikacji z GUI, możesz w każdym komponencie emitującym zdarzenia zainstalować słuchacza. Dzięki refleksji można to łatwo zautomatyzować. Na listingu 11.3 przedstawiony jest kod źródłowy klasy `EventTracer`.

Aby śledzić komunikaty, należy dodać komponent, którego zdarzenia mają być śledzone, do obiektu śledzącego zdarzenia:

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

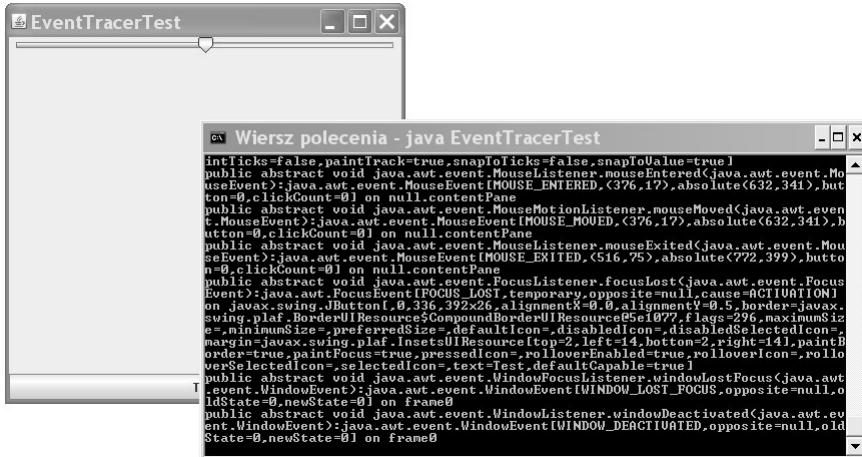
Spowoduje to wydrukowanie opisu tekstowego wszystkich zdarzeń (rysunek 11.4).

Listing 11.3. eventTracer/EventTracer.java

```
package eventTracer;

import java.awt.*;
import java.beans.*;
import java.lang.reflect.*;

/**
 * @version 1.31 2004-05-10
 * @author Cay Horstmann
 */
public class EventTracer
{
```



Rysunek 11.4. Klasa EventTracer w akcji

```

private InvocationHandler handler;

public EventTracer()
{
    // Handler wszystkich obiektów pośrednich zdarzeń
    handler = new InvocationHandler()
    {
        public Object invoke(Object proxy, Method method, Object[] args)
        {
            System.out.println(method + ":" + args[0]);
            return null;
        }
    };
}

/**
 * Dodawanie obiektów śledzących zdarzenia dla wszystkich zdarzeń, których ten komponent i jego
 * potomkowie mogą nasłuchiwać
 * @param c a komponent
 */
public void add(Component c)
{
    try
    {
        // Pobranie wszystkich zdarzeń, których ten komponent może nasłuchiwać
        BeanInfo info = Introspector.getBeanInfo(c.getClass());

        EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
        for (EventSetDescriptor eventSet : eventSets)
            addListener(c, eventSet);
    }
    catch (IntrospectionException e)
    {
    }
}
// W razie wystąpienia wyjątku nie dodawać słuchaczy

if (c instanceof Container)

```

```

    {
        // Pobranie wszystkich potomków i rekurencyjne wywołanie metody add
        for (Component comp : ((Container) c).getComponents())
            add(comp);
    }
}

/**
 * Dodanie słuchacza do danego zbioru zdarzeń
 * @param c a komponent
 * @param eventSet deskryptor interfejsu nasłuchującego
 */
public void addListener(Component c, EventSetDescriptor eventSet)
{
    // Utworzenie obiektu pośredniego dla tego typu słuchaczy i przekazanie wszystkich wywołań
    // do handlera

    Object proxy = Proxy.newProxyInstance(null, new Class[] {
        eventSet.getListenerType() },
        handler);

    // Dodanie obiektu pośredniego jako słuchacza do komponentu
    Method addListenerMethod = eventSet.getAddListenerMethod();
    try
    {
        addListenerMethod.invoke(c, proxy);
    }
    catch (ReflectiveOperationException e)
    {
    }
    // W razie wystąpienia wyjątku nie dodawać słuchaczy
}
}

```

11.7.1. Zaprzeganie robota AWT do pracy

Klasa Robot umożliwia wysyłanie zdarzeń naciśnięcia klawiszy i kliknięcia przyciskiem myszy do dowolnego programu AWT. Przeznaczeniem tej klasy jest umożliwienie automatycznego testowania interfejsów użytkownika.

Aby utworzyć robota, najpierw trzeba utworzyć obiekt klasy GraphicsDevice. Aby uzyskać domyślne urządzenie ekranowe, można użyć poniższych instrukcji:

```
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen = environment.getDefaultScreenDevice();
```

Następnie tworzymy robota:

```
Robot robot = new Robot(screen);
```

Aby wysłać zdarzenie naciśnięcia klawisza, należy kazać robotowi, aby zasymulował naciśnięcie i zwolnienie klawisza:

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

W przypadku myszy najpierw należy nią poruszyć, a potem nacisnąć i zwolnić przycisk:

```
robot.mouseMove(x, y); // x i y to bezwzględne współrzędne piksela na ekranie.
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

Wszystko sprowadza się do symulowania zdarzeń klawiatury i myszy oraz robienia zrzutów ekranu, aby sprawdzić, czy aplikacja zachowała się w odpowiedni sposób. Do robienia zrzutów ekranu służy metoda `createScreenCapture`:

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

Współrzędne prostokąta (ang. *rectangle*) są także bezwzględne.

Pomiędzy kolejnymi instrukcjami robota powinna być krótka przerwa, aby aplikacja mogła nadążyć. Do tego celu można użyć metody `delay`, której parametrem jest liczba milisekund opóźnienia. Na przykład:

```
robot.delay(1000); // opóźnienie o 1000 milisekund
```

Program przedstawiony na listingu 11.4 demonstruje sposób wykorzystania robota. Ten robot testuje program `ButtonTest` z rozdziału 8. Najpierw naciśnięcie spacji aktywuje pierwszy przycisk. Następnie robot oczekuje dwie sekundy, aby można było zobaczyć, co się stało. Następnie symuluje klawisz *Tab* i kolejne naciśnięcie spacji powodujące naciśnięcie kolejnego przycisku. Na zakończenie symulujemy kliknięcie przyciskiem myszy trzeciego przycisku (możliwe, że będzie trzeba dostosować współrzędne x i y w programie, aby przycisk był rzeczywiście wciskany). Program kończy się zrobieniem zrzutu ekranu i wyświetleniem go w dodatkowej ramce (rysunek 11.5).



Robot powinien działać w osobnym wątku, jak w przykładowym kodzie. Więcej informacji o wątkach znajduje się w rozdziale 14.

Przykład ten pokazuje jasno, że klasa `Robot` sama w sobie nie jest wygodnym sposobem testowania interfejsu użytkownika. Może natomiast służyć jako podstawa narzędzia testującego. Profesjonalne narzędzie powinno przechwytywać, zapisywać i powtarzać scenariusze zachowania użytkownika oraz znajdować lokalizacje komponentów na ekranie, dzięki czemu miejsca kliknięć myszą nie są wybierane na drodze prób i błędów.

Listing 11.4. robot/RobotTest.java

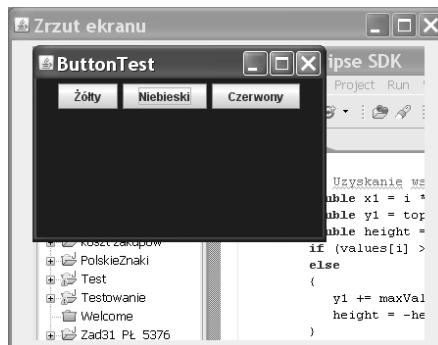
```
package robot;

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * @version 1.04 2012-05-17
 * @author Cay Horstmann
 */
public class RobotTest
```

Rysunek 11.5.

Zrzut ekranu zrobiony przez robota AWT



```

{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                // Ramka z panelem zawierającym przycisk

                JFrame frame = new JFrame();
                frame.setTitle("ButtonTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    // Powiązanie robota z ekranem

    GraphicsEnvironment environment =
    ↵GraphicsEnvironment.getLocalGraphicsEnvironment();
    GraphicsDevice screen = environment.getDefaultScreenDevice();

    try
    {
        final Robot robot = new Robot(screen);
        robot.waitForIdle();
        new Thread()
        {
            public void run()
            {
                runTest(robot);
            }
        }.start();
    }
    catch (AWTException e)
    {
        e.printStackTrace();
    }
}

/**
 * Uruchamia procedurę testową
 * @param robot robot związany z ekranem
 */

```

```

public static void runTest(Robot robot)
{
    // Symulacja naciśnięcia spacji
    robot.keyPress(' ');
    robot.keyRelease(' ');

    // Symulacja naciśnięcia klawisza Tab i spacji
    robot.delay(2000);
    robot.keyPress(KeyEvent.VK_TAB);
    robot.keyRelease(KeyEvent.VK_TAB);
    robot.keyPress(' ');
    robot.keyRelease(' ');

    // Symulacja kliknięcia myszą prawego przycisku w oknie
    robot.delay(2000);
    robot.mouseMove(220, 40);
    robot.mousePress(InputEvent.BUTTON1_MASK);
    robot.mouseRelease(InputEvent.BUTTON1_MASK);

    // Zrobienie zrzutu ekranu i wyświetlenie obrazu
    robot.delay(2000);
    BufferedImage image = robot.createScreenCapture(new Rectangle(0, 0, 400, 300));

    ImageFrame frame = new ImageFrame(image);
    frame.setVisible(true);
}

}

/**
 * Ramka zawierająca wyświetlany obraz
 */
class ImageFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 450;
    private static final int DEFAULT_HEIGHT = 350;

    /**
     * @param image obraz do wyświetlenia
     */
    public ImageFrame(Image image)
    {
        setTitle("Zrzut ekranu");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JLabel label = new JLabel(new ImageIcon(image));
        add(label);
    }
}

```

java.awt.GraphicsEnvironment **1.2**

- static GraphicsEnvironment getLocalGraphicsEnvironment()

Zwraca lokalne środowisko graficzne.

- `GraphicsDevice getDefaultScreenDevice()`

Zwraca domyślne urządzenie z ekranem. Należy pamiętać, że w komputerach, do których podłączono kilka monitorów, na każdy ekran przypada jedno urządzenie graficzne — do utworzenie tablicy wszystkich urządzeń ekranowych można użyć metody `getScreenDevices()`.

`java.awt.Robot 1.3`

- `Robot(GraphicsDevice device)`

Tworzy robota, który może współpracować z danym urządzeniem.

- `void keyPress(int key)`

- `void keyRelease(int key)`

Symuluje naciśnięcie lub zwolnienie klawisza.

Parametr: `key` Kod klawisza. Więcej informacji na temat kodów klawiszy można znaleźć w opisie klasy `KeyStroke`.

- `void mouseMove(int x, int y)`

Symuluje ruch myszką.

Parametry: `x, y` Położenie kurSORA myszy wyrażone współrzędnymi bezwzględnymi.

- `void mousePress(int eventMask)`

- `void mouseRelease(int eventMask)`

Symuluje naciśnięcie i zwolnienie przycisku myszy.

Parametr: `eventMask` Maska zdarzenia opisująca przyciski myszy. Więcej informacji na temat masek zdarzeń znajduje się w opisie klasy `InputEvent`.

- `void delay(int milliseconds)`

Opóźnia działanie robota o określoną liczbę milisekund.

- `BufferedImage createScreenCapture(Rectangle rect)`

Robi zrzut określonej części ekranu.

Parametr: `rect` prostokąt, który ma zostać objęty w zrzucie ekranu. Współrzędne są bezwzględne.

11.8. Praca z debugerem

Debugowanie polegające na wstawianiu instrukcji drukujących nie należy do największych przyjemności w życiu. Trzeba bez przerwy wstawiać i usuwać instrukcje, a następnie ponownie kompilować program. Lepszym rozwiązaniem jest użycie debugera. Narzędzie to uruchamia

program w normalnym tempie i zatrzymuje się w punkcie wstrzymania, w którym programista może obejrzeć wszystkie interesujące go dane.

Listing 11.5 przedstawia celowo uszkodzoną wersję programu ButtonTest z rozdziału 8. Klikanie przycisków w oknie niczego nie zmienia. Spójrzmy na kod — kliknięcie przycisku powinno ustawać w tle kolor określony w nazwie przycisku.

Listing 11.5. debugger/BuggyButtonTest.java

```
package debugger;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.22 2007-05-14
 * @author Cay Horstmann
 */
public class BuggyButtonTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new BuggyButtonFrame();
                frame.setTitle("BuggyButtonTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

class BuggyButtonFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public BuggyButtonFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // Dodanie panelu do ramki
        BuggyButtonPanel panel = new BuggyButtonPanel();
        add(panel);
    }
}

class BuggyButtonPanel extends JPanel
{
    public BuggyButtonPanel()
    {
```

```

    ActionListener listener = new ButtonListener();

    JButton yellowButton = new JButton("Żółty");
    add(yellowButton);
    yellowButton.addActionListener(listener);

    JButton blueButton = new JButton("Niebieski");
    add(blueButton);
    blueButton.addActionListener(listener);

    JButton redButton = new JButton("Czerwony");
    add(redButton);
    redButton.addActionListener(listener);
}

private class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String arg = event.getActionCommand();
        if (arg.equals("żółty")) setBackground(Color.yellow);
        else if (arg.equals("niebieski")) setBackground(Color.blue);
        else if (arg.equals("czerwony")) setBackground(Color.red);
    }
}
}

```

W tak krótkim programie znalezienie błędu może być możliwe dzięki samemu przeczytaniu kodu źródłowego. Założymy jednak, że analiza kodu źródłowego nie jest praktycznym rozwiązaniem. Nauczysz się teraz lokalizować błędy za pomocą debugera dostępnego w środowisku Eclipse.



Jeśli używasz niezależnego debugera, takiego jak *JSwat* (<http://code.google.com/p/jswat/>), czy sędziwego i niezwykle niezdarnego *jdb*, musisz skompilować swój program z opcją `-g`. Na przykład:

```
javac -g BuggyButtonTest.java
```

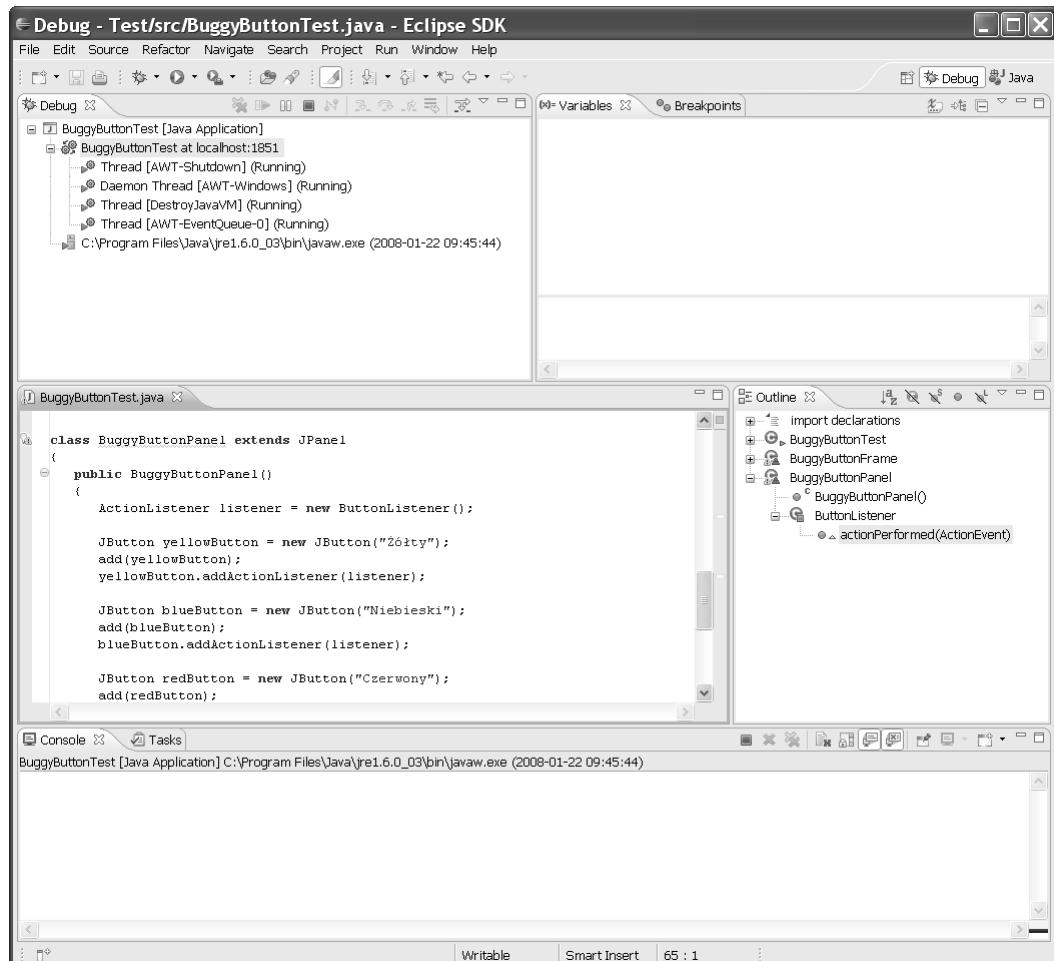
W środowisku zintegrowanym odbywa się to automatycznie.

W Eclipse debugger uruchamia opcja menu *Run/Debug As/Java Application*.

Ustawimy punkt wstrzymania (ang. *breakpoint*) w pierwszym wierszu metody `actionPerformed`. Kliknij prawym przyciskiem myszy na marginesie znajdującym się po lewej stronie wybranego wiersza kodu i z menu podręcznego wybierz opcję *Toggle Breakpoint*.

Punkt wstrzymania zostanie osiągnięty w chwili, gdy Java dojdzie do metody `actionPerformed`. Aby tak się stało, kliknij przycisk *Żółty*. Debugger zatrzymuje się na początku metody `actionPerformed` — rysunek 11.6.

Do uruchamiania programu krok po kroku służą dwa polecenia. Opcja *Step Into* wchodzi do każdego wywołania metody. Opcja *Step Over* przechodzi do następnego wiersza bez



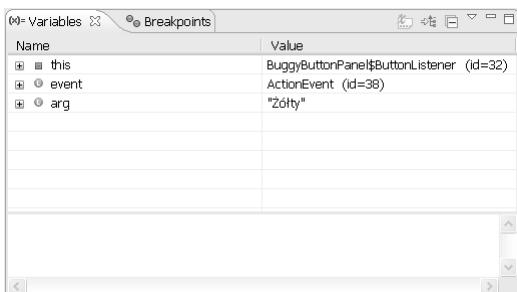
Rysunek 11.6. Zatrzymanie w punkcie wstrzymania

wchodzenia do żadnych dalszych metod. W Eclipse opcje te to *Run/Step Into* i *Run/Step Over*, a odpowiadające im skróty klawiaturowe to *F5* i *F6*. Zastosuj dwukrotnie opcję *Step Over* i sprawdź, gdzie jesteśmy.



Planowaliśmy, że program zrobi coś innego — wywoła metodę `setColor(Color.yellow)` i z niej wyjdzie.

Sprawdźmy w panelu zmiennych lokalnych wartość zmiennej `arg`:



Wiadomo już, gdzie tkwi błąd. Zmienna `arg` miała wartość `żółty`, z wielką literą `ż` na początku, a program do porównywania używał słowa `żółty` pisanego małą literą:

```
if (arg.equals("żółty"))
```

Aby wyłączyć debugger, należy w menu *Run* kliknąć opcję *Terminate*.

Eclipse posiada o wiele więcej opcji debugera, ale te podstawowe, które zostały omówione, dają już całkiem duże pole do popisu. Inne debugery, na przykład NetBeans, udostępniają bardzo podobne zestawy opcji.

Ten rozdział stanowi wprowadzenie do obsługi wyjątków oraz testowania i debugowania. Kolejne dwa rozdziały poświęciliśmy programowaniu uogólnionemu i najważniejszemu sposobowi jego zastosowania — kolekcjom.

12

Programowanie ogólne

W tym rozdziale:

- Dlaczego programowanie ogólne
- Definicja prostej klasy ogólnej
- Metody ogólne
- Ograniczenia zmiennych typowych
- Kod generyczny a maszyna wirtualna
- Ograniczenia i braki
- Zasady dziedziczenia dla typów ogólnych
- Typy wieloznaczne
- Refleksja a typy ogólne

Typy ogólne (także generyczne lub parametryzowane) stanowią największą zmianę w języku Java od chwili jego zaistnienia. Mimo że funkcjonalność tę wprowadzono dopiero w Java SE 5.0, o jej dodanie postulowano w jednym z pierwszych dokumentów JSR (ang. *Java Specification Requests*), dokładniej mówiąc JSR 14 z 1999 roku. Prace nad specyfikacją i testowaniem implementacji zajęły ekspertom około pięciu lat.

Zaletą programowania ogólnego jest możliwość pisania bezpieczniejszego i łatwiejszego do odczytu kodu w porównaniu do programów usianych zmiennymi `Object` i konwersjami typów. Typy ogólne są szczególnie przydatne w klasach kolekcyjnych, jak wszelobylska `ArrayList`.

Typy parametryzowane, przynajmniej na pierwszy rzut oka, przypominają szablony w C++. Szablony w tym języku, podobnie jak typy parametryzowane w Javie, zostały wprowadzone ze względu na kolekcje ze ścisłą kontrolą typów. Jednak z czasem odkryto dla nich wiele nowych zastosowań. Niewykluczone, że po przeczytaniu tego rozdziału odkryjesz własne nowatorskie zastosowania dla typów ogólnych w swoich programach.

12.1. Dlaczego programowanie ogólne

Programowanie ogólne (ang. *generic programming*) oznacza pisanie kodu, za pomocą którego można tworzyć obiekty wielu różnych typów. Na przykład nie chcemy tworzyć osobnych klas do gromadzenia obiektów typu `String` oraz `File` i nie musimy — klasa `ArrayList` pozwala gromadzić obiekty dowolnego typu. Jest to przykład programowania ogólnego.

Zanim w Javie pojawiły się klasy ogólne, programowanie ogólne polegało na wykorzystaniu **dziedziczenia**. Klasa `ArrayList` po prostu przechowywała tablicę referencji do elementów typu `Object`:

```
public class ArrayList //przed klasami ogólnymi
{
    private Object[] elementData;

    public Object get(int i) { . . . }
    public void add(Object o) { . . . }

    .
}
```

Metoda ta ma dwie wady. Po pierwsze, każda pobierana wartość musi zostać poddana rzutowaniu:

```
ArrayList files = new ArrayList();
.
String filename = (String) names.get(0);
```

Po drugie, nie ma żadnego mechanizmu sprawdzania błędów. Można wstawić wartości dowolnego typu:

```
files.add(new File("..."));
```

Powyższa instrukcja przejdzie z powodzeniem komplikację i program będzie normalnie działać. Natomiast w innej części programu konwersja wartości zwróconej przez metodę `get` na typ `String` może spowodować błąd.

Programowanie ogólne oferuje lepsze rozwiązanie: **parametry typowe** (ang. *type parameters*). Obecnie klasa `ArrayList` ma parametr typowy, który określa typ jej elementów:

```
ArrayList<String> files = new ArrayList<String>();
```

Dzięki temu kod jest mniej zawarty. Od razu widać, że dana lista tablicowa przechowuje obiekty typu `String`.



Jak już wspominaliśmy, w Java SE 7 i nowszych typ ogólny w konstruktorze można opuścić:

```
private Object[] elementData;
```

Typ ten jest dedukowany z typu zmiennej.

Pozytki płynące z tych informacji mogą zostać wykorzystane także przez kompilator. Wywołanie metody `get` nie pociąga za sobą konieczności wykonywania konwersji, ponieważ kompilator wie, że zostanie zwrócony typ `String`, a nie `Object`:

```
String filename = files.get(0);
```

Ponadto kompilator wie, że metoda `add` klasy `ArrayList<String>` ma parametr typu `String`, który jest znacznie bardziej bezpieczny niż `Object`. Teraz kompilator może sprawdzić, czy nie są wstawiane obiekty nieprawidłowego typu. Na przykład poniższa instrukcja spowoduje błąd komplikacji:

```
files.add(new File("...")); // Do ArrayList<String> można wstawić tylko obiekty typu String.
```

O wiele korzystniej jest spowodować błąd kompilatora niż wyjątek konwersji w trakcie działania programu.

W tym tkwi atrakcyjność parametrów typowych: pozwalają uprościć kod i są bezpieczniejsze.

12.1.1. Dla kogo programowanie ogólne

Klasy ogólne (ang. *generic class*; także generyczne lub parametryzowane), jak `ArrayList`, są łatwe w użyciu. Większość programistów używa takich typów jak `ArrayList<String>`, tak jakby były one częścią języka, podobnie jak tablice `String[]` (oczywiście listy tablicowe są lepsze, ponieważ mogą się automatycznie powiększać).

Natomiast implementacja klasy ogólnej jest trudniejsza. Programiści używający takiej konstrukcji w miejsce parametru typu mogą wstawić najróżniejsze klasy. Oczekując, że nie będzie żadnych uciążliwych ograniczeń ani niejasnych komunikatów o błędach. W związku z tym twórca klasy ogólnej musi przewidzieć wszystkie potencjalne sposoby użycia jego produktu.

Jak duże trudności może to sprawić? Projektanci biblioteki standardowej musieli poradzić sobie z następującym problemem. Klasa `ArrayList` zawiera metodę `addAll`, która dodaje do listy wszystkie elementy znajdujące się w innej kolekcji. Programista może chcieć wstawić wszystkie elementy z kolekcji `ArrayList<Manager>` do kolekcji `ArrayList<Employee>`. Oczywiście operacja w przeciwną stronę powinna być zabroniona. Jak sprawić, by jeden rodzaj operacji był możliwy, a drugi nie? Projektanci Javy wykazali się pomysłowością i rozwiązał ten problem, opracowując nową koncepcję **typu wieloznacznego** (ang. *wildcard type*). Jest to twór abstrakcyjny, ale umożliwia twórcy biblioteki tworzenie maksymalnie ogólnych metod.

W programowaniu ogólnym wyróżniamy trzy poziomy zaawansowania. Na poziomie podstawowym programista używa tylko klas ogólnych — z reguły takich kolekcji jak `ArrayList` — nie wiedząc nic o zasadzie ich działania. Większość programistów pozostaje na tym poziomie, dopóki nie natknie się na jakieś problemy. Można na przykład odebrać niejasne komunikaty o błędach, jeśli pomiesza się różne klasy ogólne lub użyje starego kodu, który został napisany w czasach, kiedy nie istniało programowanie ogólne. W takiej sytuacji konieczne jest zdobycie wiedzy na temat typów ogólnych, co pozwoli na systematyczne, a nie wyrywkowe rozwiązywanie problemów. Wreszcie, niektórzy mogą zechcieć pisać własne klasy i metody ogólne.

Twórcy aplikacji raczej nie muszą pisać dużo kodu generycznego. Większość trudnej pracy została wykonana przez programistów z firmy Sun, którzy dostarczyli parametrów typowych dla wszystkich klas kolekcyjnych. Ogólna reguła jest taka, że na użyciu parametrów typowych skorzystać mogą tylko programy zawierające dużo konwersji z bardzo ogólnych typów (jak Object czy interfejs Comparable).

Rozdział ten zawiera wszystkie informacje potrzebne do pisania własnych procedur ogólnych. Przewidujemy jednak, że większość Czytelników wykorzysta tę wiedzę przede wszystkim do rozwiązywania problemów oraz zaspokojenia własnej ciekawości na temat zasad działania parametryzowanych klas kolekcyjnych.

12.2. Definicja prostej klasy ogólnej

Klasa ogólna (ang. *generic class*) to taka, która ma co najmniej jedną zmienną typową. W tym rozdziale jako przykład przedstawiamy prostą klasę o nazwie `Pair`. Pozwoli nam ona skoncentrować się na typach ogólnych bez rozpraszanego się na zagadnienia związane z przechowywaniem danych. Oto kod ogólnej klasy `Pair`:

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

Klasa ta ma zmienną typu `T` umieszczoną w nawiasach ostrych `< >` za nazwą klasy. Klasa ogólna (parametryzowana) może mieć więcej zmiennych typowych. Na przykład klasa `Pair` mogłaby mieć różne typy dla pierwszego i drugiego pola:

```
public class Pair<T, U> { . . . }
```

Zmienne typowe używane w definicji klasy określają typy zwrotne metod oraz typy pól i zmiennych lokalnych. Na przykład:

```
private T first; //używa zmiennej typowej
```



Przyjęta powszechnie zasada nakazuje nadawanie zmiennym typowym krótkich nazw zaczynających się wielką literą. W bibliotece Java zmienna `E` oznacza typ elementu kolekcji, zmienne `K` i `V` oznaczają typy kluczy i wartości tablic, a `T` (oraz w razie potrzeby sąsiednie litery `U` i `S`) — dowolny typ.

Aby utworzyć egzemplarz typu ogólnego, należy zastąpić zmienne typowe rzeczywistymi typami, na przykład:

```
Pair<String>
```

Wynik tego działania można traktować jako zwykłą klasę z konstruktorami:

```
Pair<String>()
Pair<String>(String, String)
```

i metodami:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Innymi słowy, klasa ogólna jest jakby fabryką zwykłych klas.

Program z listingu 12.1 przedstawia klasę `Pair` w działaniu. Statyczna metoda `minmax` przemierza tablicę i jednocześnie znajduje najmniejszą i największą wartość. Znalezione wyniki zwraca w postaci obiektu klasy `Pair`. Przypomnijmy, że metoda `compareTo` porównuje dwa łańcuchy i zwraca wartość 0, jeśli są one identyczne, liczbę całkowitą ujemną, jeśli pierwszy łańcuch występuje przed drugim w kolejności słownikowej, oraz liczbę dodatnią całkowitą w pozostałych przypadkach.

Listing 12.1. pair1/PairTest1.java

```
package pair1;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class PairTest1
{
    public static void main(String[] args)
    {
        String[] words = { "Ala", "ma", "kota", "i", "psa" };
        Pair<String> mm = ArrayAlg.minmax(words);
        System.out.println("min = " + mm.getFirst());
        System.out.println("max = " + mm.getSecond());
    }
}

class ArrayAlg
{
    /**
     * Pobiera najmniejszą i największą wartość z tablicy łańcuchów
     * @param a tablica łańcuchów
     * @return złożona z najmniejszej i największej wartości lub null, jeśli tablica „a” jest null bądź pusta
     */
    public static Pair<String> minmax(String[] a)
    {
        if (a == null || a.length == 0) return null;
        String min = a[0];
        String max = a[0];
        for (int i = 1; i < a.length; i++)
```

```

    {
        if (min.compareTo(a[i]) > 0) min = a[i];
        if (max.compareTo(a[i]) < 0) max = a[i];
    }
    return new Pair<>(min, max);
}
}

```



Powierzchniowe klasy ogólne w Javie są podobne do klas szablonowych w C++. Jedyna rzucająca się w oczy różnica polega na tym, że w Javie nie ma specjalnego słowa kluczowego `template`. W dalszej części tego rozdziału przekonasz się jednak, że różnic tych jest więcej.

12.3. Metody ogólne

W poprzednim podrozdziale nauczyliśmy się definiować klasy ogólne. Istnieje także możliwość tworzenia metod z parametrami typowymi.

```

class ArrayAlg
{
    public static <T> T getMiddle(T[] a)
    {
        return a[a.length / 2];
    }
}

```

Ta metoda jest zdefiniowana w zwykłej, nieogólnej klasie. Jednak nawiasy ostre i zmienna typowa jednoznacznie wskazują, że jest to metoda ogólna. Należy zauważyc, że zmienne typu znajdują się za modyfikatorami (w tym przypadku `public static`), a przed typem zwrotnym.

Metody ogólne (parametryzowane) można definiować zarówno w zwykłych klasach, jak i klasach ogólnych.

Rzeczywisty typ w wywołaniu metody ogólnej należy podać w nawiasach ostrych przed nazwą metody podczas wywołania:

```
String middle = ArrayAlg.<String>getMiddle("Jan", "S.", "Kowalski");
```

W tym przypadku (i w większości innych) parametr typu `<String>` można opuścić. Komplator i tak ma wystarczająco danych, aby wywołać odpowiednią metodę. Porównuje typ zmiennej `names` (czyli `String[]`) z typem generycznym `T[]` i dedukuje, że `T` musi być typu `String`. Oznacza to, że można użyć poniższego prostszego wywołania:

```
String middle = ArrayAlg.getMiddle("Jan", "S.", "Kowalski");
```

Wnioskowanie o typie metod generycznych w większości sytuacji doskonale się sprawdza. Może się jednak zdarzyć, że kompilator popełni błąd, a wtedy konieczne jest rozszerzanie zwróconego komunikatu o błędzie. Przeanalizujmy poniższy przykład:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

Powyższa instrukcja spowoduje wyświetlenie trudnego do rozszyfrowania komunikatu, który w każdym kompilatorze może być nieco inny, informującego, że kod można zinterpretować na dwa równoważne sposoby. Rozumienia deklaracji typu `found` nauczysz się nieco dalej. W największym skrócie — kompilator automatycznie opakował parametry w obiekty typu `Double` i `Integer`, a następnie spróbował znaleźć wspólny nadtyp dla obu tych klas. Znalazł dwa: `Number` i interfejs `Comparable`, który sam jest typem ogólnym. Ten problem można rozwiązać, zapisując wszystkie parametry jako typ `double`.



Aby sprawdzić, jaki typ kompilator zastosuje dla wywołania metody parametryzowanej, można zastosować sztuczkę opracowaną przez Petera von der Ahé. Należy celowo popełnić błąd i przestudiować zgłoszony komunikat o błędzie. Weźmy na przykład instrukcję `ArrayAlg.getMiddle("Witaj, ". 0, null)`. Przypiszmy jej wynik do komponentu `JButton`, co nie może się udać. Zostanie wyświetlony komunikat:

```
found:
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>
```

Krótko mówiąc, wynik ten można przypisać do typu `Object`, `Serializable` lub `Comparable`.



C++ W C++ parametry typowe umieszcza się za nazwą metody, co może prowadzić do niezbyt przyjemnych dwuznaczności. Na przykład `g(f<a, b>(c))` może oznaczać: „wywołaj `g` z wynikiem zwróconym przez `f<a, b>(c)`” lub „wywołaj `g` z dwiema wartościami logicznymi `f<a` i `b>(c)`”.

12.4. Ograniczenia zmiennych typowych

Czasami konieczne jest nałożenie pewnych ograniczeń na zmienne typowe klas i metod. Chcemy znaleźć najmniejszy element w tablicy:

```
class ArrayAlg
{
    public static <T> T min(T[] a) //prawie dobrze
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}
```

Jest jednak jeden problem. Zajrzyjmy do kodu metody `min`. Zmienna `smallest` jest typu `T`, co oznacza, że może być obiektem dowolnej klasy. Skąd wiadomo, że klasa, do której należy `T`, ma metodę `compareTo`?

Rozwiązywanie polega na ograniczeniu T do klas, które implementują interfejs `Comparable` — standardowy interfejs zawierający tylko jedną metodę o nazwie `compareTo`. W tym celu należy zdefiniować ograniczenie dla zmiennej typowej T :

```
public static <T extends Comparable> T min(T[] a) . . .
```

W rzeczywistości sam interfejs `Comparable` jest typem generycznym. Na razie zignorujemy wynikające z tego dodatkowe problemy i ostrzeżenia zgłasiane przez kompilator. Prawidłowy sposób użycia parametrów typowych z interfejsem `Comparable` został opisany w podrozdziale 12.8, „Typy wieloznaczne”.

Od tej pory ogólną metodę `min` można wywoływać wyłącznie z parametrami tablic klas implementujących interfejs `Comparable`, na przykład `String`, `Date` itd. Wywołanie metody `min` z parametrem tablicy typu `Rectangle` spowoduje błąd kompilacji, ponieważ klasa `Rectangle` nie implementuje interfejsu `Comparable`.



W języku C++ nie można ograniczać typów parametrów szablonów. Utworzenie egzemplarza szablonu przy użyciu złego typu powoduje zgłoszenie w kodzie szablonu (często niejasnego) komunikatu o błędzie.

Zagadką może być, czemu w tej sytuacji używa się słowa kluczowego `extends` zamiast `implements` — przecież `Comparable` to interfejs. Notacja:

`<T extends typ graniczny>`

oznacza, że T musi być **podtypem** typu granicznego. Zarówno T , jak i typ graniczny mogą być klasą lub interfejsem. Projektanci Javy wybrali słowo `extends`, ponieważ jest ono bliskie koncepcji podtypów, a poza tym nie chcieli wprowadzać nowego słowa kluczowego, na przykład `sub`.

Zmienna typowa lub typ wieloznaczny mogą mieć wiele ograniczeń. Na przykład:

`T extends Comparable & Serializable`

Znakiem rozdzielającym poszczególne typy graniczne jest ampersand (`&`), ponieważ przecinek oddziela zmienne typowe.

Tak samo jak w przypadku dziedziczenia, wśród nadtypów może istnieć dowolna liczba interfejsów, ale tylko jedna klasa. Jeśli wśród typów granicznych znajduje się klasa, musi być ona umieszczona na początku listy.

W kolejnym przykładowym programie (listing 12.2) przerobiliśmy metodę `minmax` na metodę ogólną. Znajduje ona najmniejszą i największą wartość tablicy parametryzowanej i zwraca obiekt typu `Pair<T>`.

Listing 12.2. pair2/PairTest2.java

```
package pair2;
import java.util.*;
```

```


    /**
     * @version 1.01 2012-01-26
     * @author Cay Horstmann
     */
    public class PairTest2
    {
        public static void main(String[] args)
        {
            GregorianCalendar[] birthdays =
            {
                new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
                new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
                new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
                new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
            };
            Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
            System.out.println("min = " + mm.getFirst().getTime());
            System.out.println("max = " + mm.getSecond().getTime());
        }
    }

    class ArrayAlg
    {
        /**
         * Pobiera najmniejszą i największą wartość z tablicy obiektów typu T.
         * @param a tablica obiektów typu T
         * @return para złożona z najmniejszej i największej wartości lub wartość null, jeśli tablica „a” jest
         * null bądź pusta
         */
        public static <T extends Comparable> Pair<T> minmax(T[] a)
        {
            if (a == null || a.length == 0) return null;
            T min = a[0];
            T max = a[0];
            for (int i = 1; i < a.length; i++)
            {
                if (min.compareTo(a[i]) > 0) min = a[i];
                if (max.compareTo(a[i]) < 0) max = a[i];
            }
            return new Pair<>(min, max);
        }
    }
}


```

12.5. Kod ogólny a maszyna wirtualna

Maszyna wirtualna nie przetwarza obiektów typów ogólnych — wszystkie obiekty należą do zwykłych klas. Wcześniej wersja implementacji typów ogólnych umożliwiała komplikację programu wykorzystującego typy parametryzowane na pliki klas, które mogły być wykonywane nawet przez pierwsze maszyny wirtualne! W późniejszej fazie prac nad typami ogólnymi zrezygnowano jednak z tej zgodności wstępnej.

Każdemu typowi ogólnemu odpowiada typ **surowy** o takiej samej nazwie, ale z usuniętymi parametrami typu. W miejsce parametrów typowych **wstawiane** są typy graniczne (lub typ `Object` w przypadku zmiennych bez ograniczeń).

Na przykład typ surowy odpowiadający typowi `Pair<T>` wygląda następująco:

```
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }

    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
}
```

Ponieważ `T` jest nieograniczoną zmienną typową, w jej miejsce został wstawiony typ `Object`.

W wyniku powstała zwykła klasa, która mogłaby zostać utworzona przed wprowadzeniem typów ogólnych do Javy.

Program może zawierać różnego rodzaju typy `Pair`, na przykład `Pair<String>` i `Pair<GregorianCalendar>`, ale usunięcie parametrów zawsze zamienia je w surowy typ `Pair`.



Pod tym względem typy ogólne Javy znacznie różnią się od szablonów w C++. Drugi z języków dla każdej instancji szablonu tworzy inny typ. Nazywa się to „puchnięciem kodu szablonów” (ang. *template code bloat*). W języku Java problem ten nie występuje.

Zmienne typów w typie surowym są zastępowane pierwszą klasą graniczną lub typem `Object`, jeśli nie podano żadnych ograniczeń. Na przykład zmienna typowa w klasie `Pair<T>` nie ma ograniczeń, dlatego w jej typie surowym parametr `T` został zastąpiony przez typ `Object`. Zadeklarujmy teraz nieco inny typ:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    private T lower;
    private T upper;
    . . .

    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
}
```

Surowy typ `Interval` wygląda następująco:

```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    . . .

    public Interval(Comparable first, Comparable second) { . . . }
}
```



Można się zastanawiać, co by było, gdyby zamieniono kolejność ograniczeń — `class Interval<Serializable & Comparable>`. W takim przypadku w typie surowym parametr `T` zostałby zastąpiony typem `Serializable`, a kompilator wykonywałby konwersje na typ `Comparable` tam, gdzie to potrzebne. Dlatego ze względu na wydajność interfejsy niezawierające żadnych metod należy umieszczać na samym końcu listy ograniczeń.

12.5.1. Translacja wyrażeń generycznych

W przypadku wyczyszczania ze zmiennych typowych typu zwrotnego w wywołaniu metody ogólnej kompilator stosuje rzutowanie. Przeanalizujmy na przykład poniższe instrukcje:

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

Metoda `getFirst` po wymazaniu typów zwraca typ `Object`. Kompilator automatycznie wstawia rzutowanie do typu `Employee`. Oznacza to, że konwertuje wywołanie metody na dwie instrukcje maszyny wirtualnej:

- odwołanie do surowej metody `Pair.getFirst`;
- rzutowanie zwróconego typu `Object` na typ `Employee`.

Rzutowanie jest także stosowane przy uzyskiwaniu dostępu do pól generycznych. Założymy, że pola `first` i `second` klasy `Pair` są publiczne (nie jest to dobry styl programowania, ale dozwolony w Javie). Wtedy w kodzie bajtowym poniższego wyrażenia również zostanie wstawione rzutowanie:

```
Employee buddy = buddies.first;
```

12.5.2. Translacja metod ogólnych

Wymazywanie typów zdarza się także w metodach ogólnych. Programiści często wyobrażają sobie metody ogólne typu:

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

jako całe rodziny metod. Ale po wymazaniu typów zostaje tylko jedna metoda:

```
public static Comparable min(Comparable[] a)
```

Należy zauważyc, że parametr `T` został usunięty, a pozostawiono tylko jego typ graniczny `Comparable`.

Z czyszczeniem metod z typów generycznych wiążą się pewne komplikacje. Przyjrzyjmy się poniższemu fragmentowi programu:

```
class DateInterval extends Pair<Date>
{
    public void setSecond(Date second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    ...
}
```

Obiekt klasy `DateInterval` zawiera parę obiektów klasy `Date`. Przesłonimy metody, aby mieć pewność, że druga wartość nie będzie nigdy mniejsza od pierwszej. Po wymazaniu typów powyższa klasa przyjmie następującą postać:

```
class DateInterval extends Pair //po wymazaniu typów
{
    public void setSecond(Date second) { . . . }
    ...
}
```

Zaskakujące może być to, że istnieje jeszcze jedna metoda `setSecond`, odziedziczona po klasie `Pair`:

```
public void setSecond(Object second)
```

Jest to z pewnością inna metoda, o czym świadczy inny typ parametru — `Object` zamiast `Date`, a **nie powinna** być inna. Przyjrzyjmy się poniższym instrukcjom:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; //OK — przypisanie do nadklasy
pair.setSecond(aDate);
```

Spodziewamy się polimorficznego wywołania właściwej metody `setSecond`. Ponieważ `pair` jest referencją do obiektu klasy `DateInterval`, powinna to być metoda `DateInterval.setSecond`. Problem polega na tym, że wymazywanie typów zakłóca polimorfizm. Dlatego kompilator stara się uniknąć problemu, generując **metodę pomostową** (ang. *bridge method*) w klasie `DateInterval`:

```
public void setSecond(Object second) { setSecond((Date) second); }
```

Aby zrozumieć, jak to działa, szczegółowo przeanalizujemy wykonywanie poniższej instrukcji:

```
pair.setSecond(aDate)
```

Zmienna `pair` jest typu `Pair<Date>`, który ma tylko jedną metodę `setSecond(Object)`. Maszyna wirtualna wywołuje tę metodę na rzecz obiektu wskazywanego przez zmienną `pair`. Obiekt ten jest typu `DateInterval`. W związku z tym wywoływana jest metoda `DateInterval.setSecond(Object)`, która jest zsyntetyzowaną metodą pomostową. Wywołuje ona metodę `DateInterval.setSecond(Date)`, czyli tę, którą chcemy.

Metody pomostowe mogą być nawet jeszcze dziwniejsze. Przypuśćmy, że w klasie Date
 ↵Interval przesłonięto także metodę getSecond:

```
class DateInterval extends Pair<Date>
{
    public Date getSecond() { return (Date) super.getSecond().clone(); }
    ...
}
```

W klasie DateInterval znajdują się dwie metody getSecond:

```
Date getSecond()      // w klasie DateInterval
Object getSecond()    // przesłania metodę zdefiniowaną w klasie Pair, aby wywoływała pierwszą metodę
```

W Javie nie można pisać takiego kodu, ponieważ dwie metody nie mogą mieć takich samych typów parametrów — w tym przypadku nie mają ich w ogóle. Natomiast w maszynie wirtualnej metodę identyfikują typy parametrów i **typ zwrotny**. Dlatego kompilator może utworzyć kod bajtowy dwóch metod różniących się tylko typem zwrotnym, który zostanie prawidłowo obsłużony przez maszynę wirtualną.



Metody pomostowe nie ograniczają się tylko do typów ogólnych. W rozdziale 5. zauważyliśmy, że metoda przesłaniająca inną metodę może mieć bardziej ograniczony typ zwrotny. Na przykład:

```
public class Employee implements Cloneable
{
    public Employee clone() throws CloneNotSupportedException { ... }
```

Mówiąc, że metody Object.clone i Employee.clone mają **kowariantne typy zwrotne**.

W rzeczywistości klasa Employee zawiera **dwie** metody clone:

```
Employee clone()    // zdefiniowana powyżej
Object clone()      // zsytetyzowana metoda pomostowa przesłania metodę Object.clone
```

Zsytetyzowana metoda pomostowa wywołuje nowo utworzoną metodę.

Podsumowując, należy zapamiętać następujące fakty dotyczące translacji typów ogólnych:

- W maszynie wirtualnej nie ma typów ogólnych, tylko zwykłe klasy i metody.
- Parametry typowe są zastępowane odpowiadającymi im typami granicznymi.
- Metody pomostowe są syntetyzowane w celu zachowania polimorfizmu.
- Rzutowanie jest wstawiane w razie potrzeby w celu zachowania bezpieczeństwa typów.

12.5.3. Używanie starego kodu

Gdy projektowano moduł typów ogólnych w Javie, jednym z najważniejszych celów inżynierów było zapewnienie zgodności nowego kodu ze starym. Spójrzmy na konkretny przykład. Do ustawiania etykiet komponentów JSlider używa się poniższej metody:

```
void setLabelTable(Dictionary table)
```

W tym przypadku `Dictionary` jest surowym typem, ponieważ komponent `JSlider` został zaimplementowany przed typami ogólnymi. Jednak do wstawiania wartości do słownika należy używać typu ogólnego:

```
Dictionary<Integer, Component> labelTable = new Hashtable<>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
```

Gdy do metody `setLabelTable` przekażemy obiekt typu `Dictionary<Integer, Component>`, kompilator zgłosi ostrzeżenie:

```
slider.setLabelTable(labelTable); // ostrzeżenie
```

Kompilator nie ma pewności, co metoda `setLabelTable` może zrobić z obiektem typu `Dictionary`. Może na przykład zamienić wszystkie klucze na łańcuchy, co złamałoby gwarancję, że klucze są typu `Integer`. To z kolei mogłoby doprowadzić do wyjątków rzutowania w kolejnych operacjach.

Z ostrzeżeniem tym nie można wiele zrobić. Co najwyżej możemy spróbować dowiedzieć się, co `JSlider` najprawdopodobniej może zrobić z danym obiektem `Dictionary`. W tym przypadku jest oczywiste, że `JSlider` tylko odczytuje dane, a więc ostrzeżenie można zignorować.

Teraz wyobraźmy sobie odwrotną sytuację, w której otrzymujemy obiekt surowego typu ze starej klasy. Możemy przypisać go do zmiennej typu ogólnego, ale wtedy kompilator zgłosi ostrzeżenie. Na przykład:

```
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // ostrzeżenie
```

Przeglądamy ostrzeżenie i sprawdzamy, czy tablica etykiet rzeczywiście zawiera obiekty typu `Integer` i `Component`. Oczywiście nigdy nie można mieć całkowitej pewności. Może się zdarzyć, że złośliwy użytkownik zainstaluje inny obiekt `Dictionary` w suwaku. Sytuacja ta nie jest jednak gorsza niż przed pojawiением się typów ogólnych. W najgorszym wypadku program wygeneruje wyjątek.

Po przeanalizowaniu ostrzeżenia można się go pozbyć za pomocą **adnotacji** (ang. *annotation*). Adnotację można umieścić przed lokalną zmienną:

```
@SuppressWarnings("unchecked")
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // brak ostrzeżenia
```

Można też dodać adnotację dla całej metody:

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

Ta adnotacja wyłącza sprawdzanie błędów dla całej metody.

12.6. Ograniczenia i braki

W tej sekcji omawiamy ograniczenia, które trzeba wziąć pod uwagę, decydując się na użycie typów ogólnych. Większość z nich ma swoje źródło w wymazywaniu typów.

12.6.1. Nie można podawać typów prostych jako parametrów typowych

Parametru typowego nie można zastąpić typem prostym. Dlatego nie da się utworzyć klasy `Pair<double>`, ale `Pair<Double>`. Powodem jest tu oczywiście wymazywanie typów. Po oczyszczeniu klasy `Pair` ma pola typu `Object`, w których nie można przechowywać wartości typu `double`.

Mimo że zasada ta jest denerwująca, jest ona zgodna z zasadą odrębności typów prostych w języku Java. Nie jest to wielka strata — typów prostych jest tylko osiem i zawsze można je obsłużyć za pomocą osobnych klas i metod, jeśli typy opakowujące nie mogą być użyte.

12.6.2. Sprawdzanie typów w czasie działania programu jest możliwe tylko dla typów surowych

Obiekty w maszynie wirtualnej zawsze należą do konkretnego typu nieogólnego. Dlatego operacja sprawdzania typu zawsze zwraca typ surowy. Na przykład instrukcja:

```
if (a instanceof Pair<String>) // BŁĄD
```

sprawdza tylko, czy `a` jest obiektem jakiegośkolwiek klasy `Pair`. To samo dotyczy poniższego testu:

```
if (a instanceof Pair<T>) // BŁĄD
```

i rzutowania:

```
Pair<String> p = (Pair<String>) a; // OSTRZEŻENIE — można tylko sprawdzić, czy a jest typu Pair
```

Każde użycie operatora `instanceof` lub zastosowanie rzutowania związane z typami ogólnymi będzie skutkowało zgłoszeniem przez kompilator ostrzeżenia, mającego na celu powiadomienie programisty o ryzykowności operacji.

Z tego samego powodu metoda `getClass` zawsze zwraca typ surowy, na przykład:

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // są równe
```

Wynikiem porównania jest wartość `true`, ponieważ oba wywołania metody `getClass` zwracają `Pair.class`.

12.6.3. Nie można tworzyć tablic typów ogólnych

Nie można tworzyć tablic typów parametryzowanych, na przykład:

```
Pair<String>[] table = new Pair<String>[10]; // błąd
```

Dlaczego powyższy kod jest zły? Po wymazaniu parametrów zmienna `table` jest typu `Pair[]`. Można ją przekonwertować na typ `Object`:

```
Object[] objarray = table;
```

Tablica pamięta typ przechowywanych elementów i w przypadku zapisu w niej elementu złego typu generuje wyjątek `ArrayStoreException`:

```
objarray[0] = "Witaj. "; // błąd — typ elementów to Pair
```

Wymazywanie typów powoduje jednak, że mechanizm ten nie działa w przypadku typów ogólnych. Poniższa instrukcja przeszłaby z powodzeniem kontrolę tablicy, ale nadal powodowałaby błąd typu. Dlatego zabroniono tworzenia tablic typów ogólnych.

```
objarray[0] = new Pair<Employee>();
```

Należy podkreślić, że zabronione jest tylko tworzenie tych tablic. Można zadeklarować zmienną typu `Pair<String>`, ale nie można jej zainicjować przy użyciu instrukcji `new Pair<String>[10]`.



Można deklarować tablice typów wieloznacznych, a następnie poddawać je rzutowaniu:

```
Pair<String>[] table = (Pair<String>[]) new Pair<?>[10];
```



Wynik nie jest bezpieczny. Jeśli zapiszesz `Pair<Employee>` w elemencie `table[0]`, a następnie wywołasz metodę klasy `String` na `table[0].getFirst()`, otrzymasz wyjątek `ClassCastException`.



Aby utworzyć kolekcję obiektów typów ogólnych, należy użyć klasy `ArrayList`: zapis `ArrayList<Pair<String>>` jest bezpieczny i efektywny.

12.6.4. Ostrzeżenia dotyczące zmiennej liczby argumentów

W poprzednim podrozdziale dowiedziałeś się, że Java nie obsługuje tablic typów ogólnych. W tej części rozdziału znajduje się omówienie podobnego problemu: przekazywania egzemplarzy ogólnego typu do metody ze zmienną liczbą argumentów.

Spójrz na poniższą prostą metodę:

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
    for (t : ts) coll.add(t);
}
```

Przypomnijmy, że parametr `ts` jest tablicą zawierającą wszystkie dostarczone argumenty.

Teraz spójrz na poniższe wywołanie:

```
Collection<Pair<String>> table = ...;
Pair<String> pair1 = ...;
Pair<String> pair2 = ...;
addAll(table, pair1, pair2);
```

Aby wywołać tę metodę, maszyna wirtualna Javy musi utworzyć tablicę `Pair<String>`, co jest wbrew zasadom. Reguły zostały jednak rozluźnione dla takich przypadków i zamiast błędu zostanie zgłoszone tylko ostrzeżenie.

Ostrzeżenie to można stłumić na dwa sposoby. Można dodać adnotację `@SuppressWarnings("unchecked")` do metody zawierającej wywołanie `addAll` lub (od Java SE 7) adnotację `@SafeVarargs` do metody `addAll`:

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

Teraz metodę tę można wywoływać z typami ogólnymi. Adnotację tę można stosować do wszystkich metod, które tylko odczytują elementy z tablicy parametrów, i jest to z pewnością jej najczęstsze zastosowanie.



Dzięki adnotacji `@SafeVarargs` można obejść ograniczenie dotyczące tworzenia ogólnych tablic. Można w tym celu użyć poniższej metody:

```
@SafeVarargs static <E> E[] array(E... array) { return array; }
```

Teraz można zastosować poniższe wywołanie:

```
Pair<String>[] table = array(pair1, pair2);
```

Wydaje się to wygodne, ale kryje się tu niebezpieczeństwo. Kod:

```
Object[] objarray = table;
objarray[0] = new Pair<Employee>();
```

zostanie wykonany bez wyjątku `ArrayStoreException` (ponieważ tablica sprawdza tylko typ wymazywany), a wyjątek otrzymamy w innym miejscu, w którym użyjemy elementu `table[0]`.

12.6.5. Nie wolno tworzyć egzemplarzy zmiennych typowych

Zmiennych typowych nie można używać w wyrażeniach typu `new T(...)`, `new T[...]` czy `T.class`. Na przykład poniższy konstruktor `Pair<T>` jest niedozwolony:

```
public Pair() { first = new T(); second = new T(); } // błąd
```

W wyniku wymazywania typów parametr `T` został zamieniony na typ `Object`, a z pewnością naszym zamiarem nie jest utworzenie wywołania `new Object()`.

Można sobie z tym poradzić, konstruując obiekty ogólne poprzez refleksję i wywołując metodę `Class.newInstance`.

Niestety istnieją pewne komplikacje. Nie można użyć takiego wywołania:

```
first = T.class.newInstance(); // błąd
```

Wyrażenie `T.class` jest złe. W zamian trzeba tak zaprojektować API, aby otrzymać obiekt klasy `Class`, na przykład:

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try { return new Pair<T>(cl.newInstance(), cl.newInstance()); }
    catch (Exception ex) { return null; }
}
```

Tę metodę można wywołać następująco:

```
Pair<String> p = Pair.makePair(String.class);
```

Należy zauważyć, że klasa `Class` sama jest generyczna. Na przykład `String.class` jest egzemplarzem (i to jedynym) klasy `Class<String>`. Dlatego metoda `makePair` może wywnioskować typ pary, którą tworzy.

Nie można utworzyć tablicy generycznej:

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm = new T[2]; . . . } // błąd
```

Wymazywanie typów spowodowałoby, że metoda ta zawsze tworzyłaby tablicę `Comparable[2]`.

Jeśli tablica jest używana wyłącznie jako prywatne pole egzemplarza klasy, można ją zadeklarować jako `Object[]` i przy pobieraniu elementów stosować rzutowanie. Na przykład klasę `ArrayList` można zaimplementować następująco:

```
public class ArrayList<E>
{
    private Object[] elements:
    . . .
    @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
    public void set(int n, E e) { elements[n] = e; } // nie jest potrzebne rzutowanie
}
```

Rzeczywista implementacja nie jest taka prosta:

```
public class ArrayList<E>
{
    private E[] elements:
    public ArrayList() { elements = (E[]) new Object[10]; }
    . . .
}
```

W tym przypadku rzutowanie na typ `E[]` jest złe, ale przez wymazywanie typów jest to nie do wykrycia.

Technika ta nie zadziała w naszej metodzie `minmax`, ponieważ zwracamy tablicę `T[]` i podanie jej złego typu spowoduje błąd wykonawczy. Założymy, że mamy następującą implementację:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
    Object[] mm = new Object[2];
```

```

    . . .;
    return (T[]) mm; // kompilator zgłosi ostrzeżenie
}

```

Poniższe wywołanie przejdzie komplikację bez żadnego ostrzeżenia:

```
String[] ss = minmax("Tomasz", "Darek", "Henryk");
```

Wyjątek `ClassCastException` jest generowany, kiedy referencja do typu `Object[]` jest rzutowana na typ `Comparable[]` podczas zwracania wartości przez metodę.

W takiej sytuacji można skorzystać z refleksji i wywołać metodę `Array.newInstance`:

```

public static <T extends Comparable> T[] minmax(T[] a)
{
    T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    . . .
}

```

Metoda `toArray` z klasy `ArrayList` nie ma tyle szczęścia. Musi utworzyć tablicę `T[]`, ale nie zna typu elementów. Dlatego istnieją jej dwa warianty:

```
Object[] toArray()
T[] toArray(T[] result)
```

Druga wersja pobiera parametr w postaci tablicy. Jeśli tablica ta jest wystarczająco duża, zostanie użyta. W przeciwnym razie tworzona jest nowa tablica o odpowiednim rozmiarze, a typem jej komponentów jest `result`.

12.6.6. Zmiennych typowych nie można używać w statycznych kontekstach klas ogólnych

Do zmiennych typowych nie można odwoływać się w polach i metodach statycznych. Na przykład:

```

public class Singleton<T>
{
    private static T singleInstance; // błąd

    public static T getSingleInstance() // błąd
    {
        if (singleInstance == null) utwórz nowy egzemplarz T
        return singleInstance;
    }
}

```

Gdyby to było możliwe, można by było zadeklarować klasę `Singleton<Random>` dla liczb losowych i klasę `Singleton<JFileChooser>` do tworzenia okien wyboru pliku. Jest to jednak niemożliwe, ponieważ dzięki wymazywaniu typów istnieje tylko jedna klasa `Singleton` i tylko jedno pole `singleInstance`. Dlatego pola i metody statyczne ze zmiennymi typowymi są zabronione.

12.6.7. Obiektów klasy ogólnej nie można generować ani przechwytywać

Nie można generować ani przechwytywać obiektów klas ogólnych. Niedozwolone jest nawet rozszerzanie klasy `Throwable` przez klasę generyczną. Na przykład poniższa definicja spowoduje błąd komilacji:

```
public class Problem<T> extends Exception { /* . . . */ } // błąd — nie można rozszerzać
// klasy Throwable
```

Zmiennych typowych nie można używać w klauzulach `catch`. Dlatego poniższa metoda spowoduje błąd komilacji:

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        procedury
    }
    catch (T e) // błąd — nie można przechwycić zmiennej typowej
    {
        Logger.global.info(...)
    }
}
```

Można natomiast używać zmiennych typowych w specyfikacjach wyjątków. Poniższa metoda jest poprawna:

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
    try
    {
        procedury
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}
```

12.6.7.1. Można wyłączyć sprawdzanie wyjątków kontrolowanych

Podstawową zasadą obsługi wyjątków w Javie jest utworzenie procedury obsługi dla każdego kontrolowanego wyjątku. Za pomocą typów ogólnych można to obejść. Najważniejsza w realizacji tego celu jest poniższa metoda:

```
@SuppressWarnings("unchecked")
public static <T extends Throwable> void throwAs(Throwable e) throws T
{
    throw (T) e;
}
```

Przypuśćmy, że metoda ta znajduje się w klasie `Block`. Gdy zastosujemy poniższe wywołanie, kompilator uzna, że `t` staje się wyjątkiem niekontrolowanym.

```
Block.<RuntimeException>throwAs(t);
```

Poniższa konstrukcja zamienia wszystkie wyjątki w takie, które dla kompilatora są niekontrolowane:

```
try
{
    instrukcje
}
catch (Throwable t)
{
    Block.<RuntimeException>throwAs(t);
}
```

Zapakujemy to w abstrakcyjną klasę. Użytkownik przesłoni metodę body, aby dostarczyć konkretną akcję. Wywołując metodę toThread, otrzymasz obiekt klasy Thread, którego metoda run nie zwraca uwagi na niekontrolowane wyjątki.

```
public abstract class Block
{
    public abstract void body() throws Exception;

    public Thread toThread()
    {
        return new Thread()
        {
            public void run()
            {
                try
                {
                    body();
                }
                catch (Throwable t)
                {
                    Block.<RuntimeException>throwAs(t);
                }
            }
        };
    }

    @SuppressWarnings("unchecked")
    public static <T extends Throwable> void throwAs(Throwable e) throws T
    {
        throw (T) e;
    }
}
```

Przykładowo poniższy program uruchamia wątek zgłoszający niekontrolowany wyjątek.

```
public class Test
{
    public static void main(String[] args)
    {
        new Block()
        {
            public void body() throws Exception
            {
                Scanner in = new Scanner(new File("quux"));
                while (in.hasNext())
                    System.out.println(in.next());
            }
        };
    }
}
```

```
        }
        .toThread().start();
    }
}
```

Gdy uruchomisz ten program, otrzymasz dane stosu z wyjątkiem `FileNotFoundException` (oczywiście pod warunkiem że nie podałeś pliku o nazwie `quux`).

Co w tym takiego niezwykłego? Standardowo trzeba przechwytywać wszystkie kontrolowane wyjątki w metodzie `run` wątku i **opakowywać je** w wyjątkach niekontrolowanych — metoda `run` nie zgłasza wyjątków kontrolowanych.

Tutaj jednak nie stosujemy opakowywania. Po prostu zgłaszamy wyjątek, tak by kompilator uznał, że nie jest to wyjątek kontrolowany.

Przy użyciu klas ogólnych, wymazywania typów i adnotacji `@SuppressWarnings` obeszliśmy bardzo ważną część systemu typów Javy.

12.6.8. Uważaj na konflikty, które mogą powstać po wymazaniu typów

Nie można tworzyć warunków powodujących konflikty po usunięciu typów generycznych. Założymy na przykład, że do klasy `Pair` dodajemy metodę `equals`:

```
public class Pair<T>
{
    public boolean equals(T value) { return first.equals(value) &&
        second.equals(value); }

    ...
}
```

Weźmy klasę `Pair<String>`. W zasadzie ma ona dwie metody `equals`:

```
boolean equals(String) // zdefiniowana w klasie Pair<T>
boolean equals(Object) // odziedziczona po klasie Object
```

Tym razem jednak nasza intuicja zawodzi. Metoda `boolean equals(T)` po wymazaniu typów ma postać `boolean equals(Object)` i wchodzi w konflikt z metodą `Object.equals`.

Aby sobie z tym poradzić, należy oczywiście zmienić nazwę metody sprawiającej problem.

W specyfikacji typów ogólnych opisano jeszcze inną regułę: „Aby translacja poprzez wymazywanie typów była możliwa, klasa lub zmienna typowa nie może być w jednym czasie podtypem dwóch interfejsów będących różnymi wersjami parametrycznymi tego samego interfejsu”. Na przykład poniższy kod jest zły:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
{ . . . } // błąd
```

W takiej sytuacji klasa `GregorianCalendar` implementowałaby interfejsy `Comparable<Calendar>` i `Comparable<GregorianCalendar>`, które są różnymi wersjami parametrycznymi tego samego interfejsu.

Związek tego ograniczenia z wymazywaniem typów nie jest oczywisty. Niegeneryczna wersja jest dozwolona:

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

Powód tkwi znacznie głębiej. Konflikt wystąpiłby pomiędzy zsyntetyzowanymi metodami pomostowymi. Metoda pomostowa klasy implementującej interfejs Comparable<X> jest następująca:

```
public int compareTo(Object other) { return compareTo((X) other); }
```

Nie można mieć dwóch takich metod dla różnych typów X.<<F1-k>>

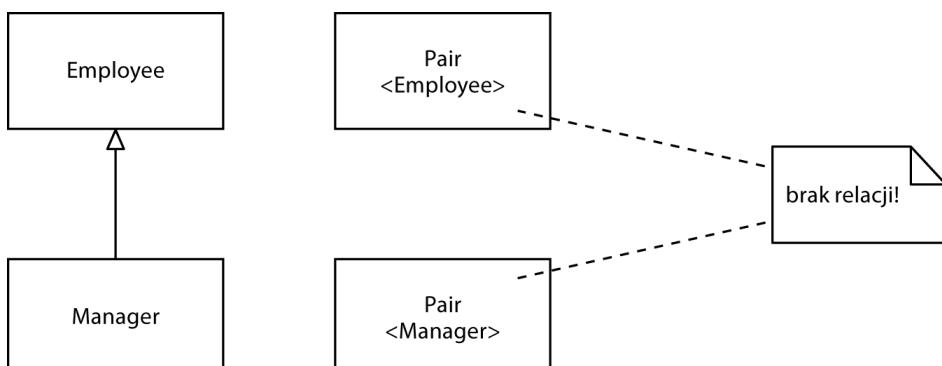
12.7. Zasady dziedziczenia dla typów ogólnych

Aby móc pracować z klasami ogólnymi, trzeba znać kilka reguł dotyczących dziedziczenia i podtypów. Zaczniemy od kwestii, która wielu programistom sprawia problemy. Wyobraźmy sobie klasę i jej podklasę, na przykład Employee i Manager. Czy klasa Pair<Manager> jest podklassą klasy Pair<Employee>? Może się to wydawać zaskakujące, ale nie. Na przykład nie można skompilować poniższego fragmentu kodu:

```
Manager[] topHonchos = . . . ;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // błąd
```

Metoda minmax zwraca typ Pair<Manager>, a nie Pair<Employee>, nie można jej też przypisać jednego z tych typów do drugiego.

Ogólna zasada jest taka, że pomiędzy typami Pair<S> i Pair<T> **nie** ma żadnego związku, bez względu na to, co łączy S i T (zobacz rysunek 12.1).



Rysunek 12.1. Brak relacji dziedziczenia pomiędzy klasami par

Ograniczenie to wydaje się okrutne, ale jest konieczne ze względu na bezpieczeństwo typów. Wyobraźmy sobie, że możemy przekonwertować typ Pair<Manager> na typ Pair<Employee>. Spójrzmy na poniższy kod:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies;           // niedozwolone, ale założymy, że tak
employeeBuddies.setFirst(lowlyEmployee);
```

Ostatnia instrukcja jest bez wątpienia poprawna, ale zmienne employeeBuddies i managerBuddies odwołują się do **tego samego obiektu**. W ten sposób w jednej parze umieściliśmy osobę z kierownictwa i zwykłego pracownika. Operacja ta nie powinna być możliwa przy użyciu typu Pair<Manager>.



Właśnie została opisana ważna różnica pomiędzy typami ogólnymi i tablicami w Javie. Tablicę Manager[] można przypisać do zmiennej typu Employee[]:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

Tablice jednak mają specjalną ochronę. Próba zapisu zwykłego pracownika w employeeBuddies[] zakończy się wygenerowaniem przez maszynę wirtualną wyjątku ArrayStoreException.

Typ parametryzowany zawsze można przekonwertować na typ surowy. Na przykład Pair<Employee> jest podtypem typu surowego Pair. Taka konwersja jest potrzebna ze względu na zgodność ze starym kodem.

Czy można dokonać konwersji na typ surowy, a następnie spowodować błąd typu? Niestety tak. Spójrzmy na poniższy przykład:

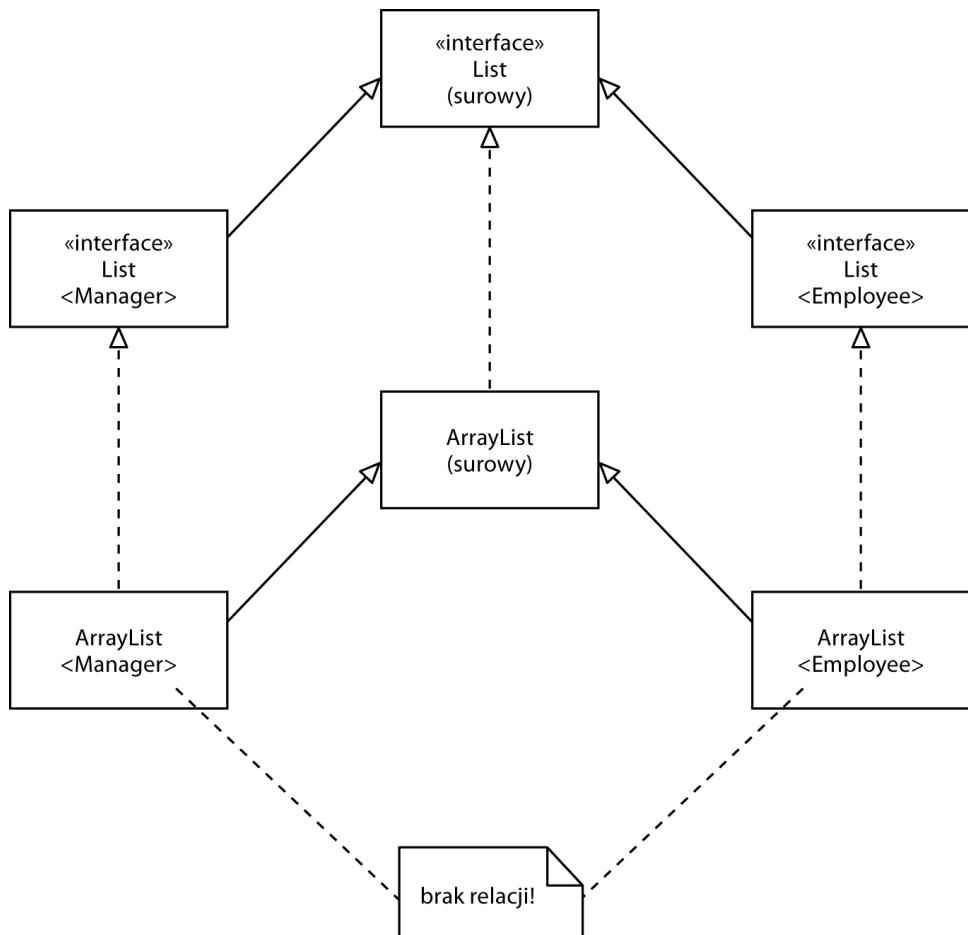
```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies;           // OK
rawBuddies.setFirst(new File("..."));        // tylko ostrzeżenie kompilatora
```

To wydaje się straszne, ale należy pamiętać, że nie jest gorzej niż w starszych wersjach Javy. Bezpieczeństwo maszyny wirtualnej nie wchodzi w grę. Jeśli metoda `getFirst` pobierze obcy obiekt i przypisze go do zmiennej typu Manager, zostanie wygenerowany wyjątek ClassCastException.

Ostatecznie klasy ogólne mogą rozszerzać lub implementować inne klasy ogólne. Pod tym względem nie różnią się niczym od zwykłych klas. Na przykład klasa `ArrayList<T>` implementuje interfejs `List<T>`. Oznacza to, że typ `ArrayList<Manager>` można przekonwertować na typ `List<Manager>`. Jak już jednak wiemy, `ArrayList<Manager>` to **nie** `ArrayList<Employee>` ani `List<Employee>`. Relacje te przedstawia rysunek 12.2.

12.8. Typy wieloznaczne

Naukowcy zajmujący się systemami typów już od dłuższego czasu wiedzieli, że używanie sztywnych systemów typów ogólnych nie należy do przyjemności. Projektanci Javy wpadli na pomysłowe (ale i bezpieczne) rozwiązanie tego problemu — **typ wieloznaczny** (ang. *wildcard type*). Na przykład poniższy typ wieloznaczny oznacza dowolny typ ogólny `Pair`, którego parametr typowy jest podklassą klasy `Employee`, na przykład `Pair<Manager>`, ale nie `Pair<String>`.



Rysunek 12.2. Relacje pomiędzy generycznymi typami listowymi

Pair<? extends Employee>

Założymy, że chcemy napisać metodę drukującą pary pracowników:

```

public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " i " + second.getName() + " to kumple.");
}
  
```

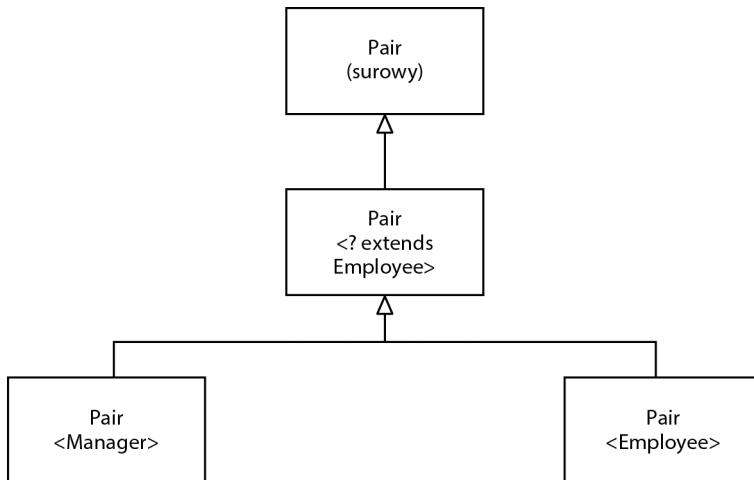
Jak wiadomo z poprzedniego podrozdziału, do metody tej nie można przekazać typu `Pair<Manager>`, co stanowi spore ograniczenie. Jest jednak proste rozwiązanie w postaci typów wieloznacznych:

```
public static void printBuddies(Pair<? extends Employee> p)
```

Typ `Pair<Manager>` jest podtypem `Pair<? extends Employee>` (zobacz rysunek 12.3).

Rysunek 12.3.

Relacje klasowe przy zastosowaniu typów wieloznacznych



Czy za pomocą typu wieloznacznego można uszkodzić typ `Pair<Manager>` poprzez referencję typu `Pair<? extends Employee>`?

```

Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // błąd kompilacji
  
```

Uszkodzenie jest niemożliwe. Wywołanie metody `setFirst` spowodowało błąd nieprawidłowego typu. Aby dowiedzieć się dlaczego, szczegółowo przeanalizujemy typ `Pair<? extends Employee>`. Jego metody są następujące:

```

? extends Employee getFirst()
void setFirst(? extends Employee)
  
```

To uniemożliwia wywołanie metody `setFirst`. Kompilator wie tylko, że potrzebny jest jakiś podtyp `Employee`, ale nie wie jaki. Nie zgadza się na przekazanie żadnego konkretnego typu, ponieważ dzoker (?) może do niego nie pasować.

Ten problem nie istnieje w przypadku metody `getFirst`. Wartość zwrotną tej metody można z powodzeniem przypisać do referencji typu `Employee`.

Jest to kluczowa cecha typów wieloznacznych z ograniczeniami. Teraz dysponujemy możliwością rozróżnienia bezpiecznych metod akcesora i niebezpiecznych metod mutatora.

12.8.1. Ograniczenia nadtypów typów wieloznacznych

Ograniczenia typów wieloznacznych są podobne do ograniczeń zmiennych typowych, ale mają jedną dodatkową cechę — można określić **ograniczenia nadtypów**:

```
? super Manager
```

Ten typ wieloznaczny jest ograniczony do wszystkich nadtypów typu `Manager` (projektanci mieli dużo szczęścia, że istniejące słowo kluczowe `super` tak precyzyjnie określa ten rodzaj relacji).

Do czego może się to przydać? Typy wieloznaczne z ograniczeniami nadtypów są przeciwieństwem typów wieloznacznych opisanych w podrozdziale 12.8., „Typy wieloznaczne”. Można przekazywać do metod parametry, ale nie można używać ich wartości zwrotnych. Na przykład klasa `Pair<? super Manager>` zawiera następujące metody:

```
void setFirst(? super Manager)
? super Manager getFirst()
```

Kompilator nie wie, jaki dokładnie typ ma metoda `setFirst`, ale może ją wywołać na rzecz każdego obiektu typu `Manager`, `Employee` i `Object`, jednak nie na rzecz obiektów należących do jej podtypów, jak na przykład `Executive`. Dlatego wywołującą metodę `getFirst` nie wie na pewno, jakiego typu obiekt ona zwróci. W związku z tym wartość tę można przypisać tylko do typu `Object`.

Oto typowy przykład takiej sytuacji. Mamy tablicę obiektów `Manager`. Chcemy, aby kierowcy z największą i najmniejszą premią znaleźli się w jednym obiekcie `Pair`. Jakiego rodzaju ma to być para? Może to być `Pair<Employee>` albo `Pair<Object>` (zobacz rysunek 12.4). Poniższa metoda przyjmie każdy odpowiedni obiekt `Pair`:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

Podsumowując, typy wieloznaczne z ograniczeniami nadtypów pozwalają na zapis w obiektach ogólnych, a typy wieloznaczne z ograniczeniami podtypów pozwalają na odczyt z obiektów ogólnych.

Oto jeszcze jeden przykład zastosowania ograniczeń nadtypów. Interfejs `Comparable` jest typem generycznym. Jego deklaracja jest następująca:

```
public interface Comparable<T>
{
    public int compareTo(T other);
```

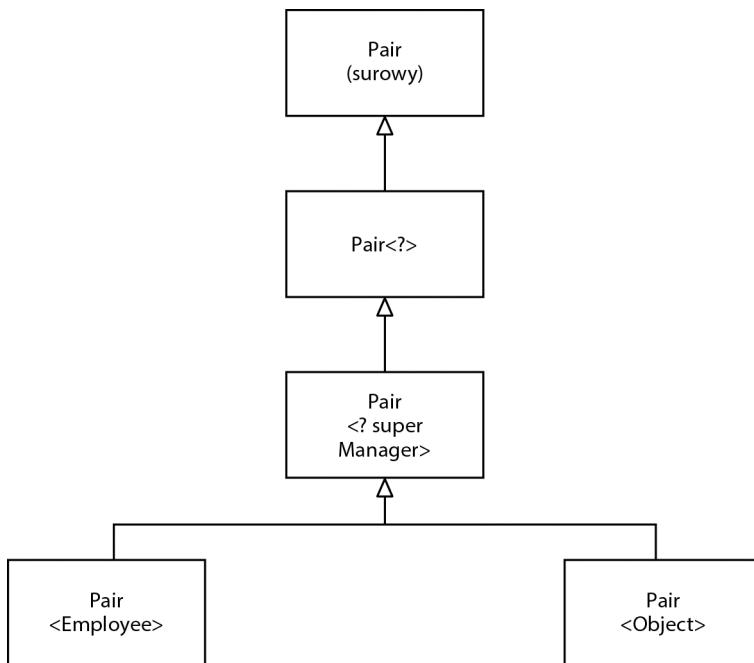
Tutaj zmienna typowa określa typ parametru `other`. Na przykład klasa `String` implementuje interfejs `Comparable<String>`, a deklaracja jej metody `compareTo` jest następująca:

```
public int compareTo(String other)
```

Jest to zgrabne rozwiązanie, ponieważ parametr jawnym ma odpowiedni typ. Przed Java SE 5.0 parametr `other` był typu `Object` i implementacja metody musiała zawierać operację konwersji typów.

Rysunek 12.4.

Typ wieloznaczny z ograniczeniem nadtypów



Ponieważ interfejs Comparable jest typem ogólnym, metodę `min` z klasy `ArrayAlg` można napisać nieco lepiej. Jej deklaracja może wyglądać następująco:

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

Taki zapis wydaje się bardziej staranny niż `T extends Comparable`, dzięki czemu może nadawać się dla wielu klas. Jeśli na przykład chcemy znaleźć najmniejszą wartość w tablicy łańcuchów, parametr `T` będzie typu `String`, a `String` jest podtypem `Comparable<String>`. Pojawia się jednak problem przy przetwarzaniu tablicy obiektów typu `GregorianCalendar`. Tak się składa, że klasa `GregorianCalendar` jest podklassą klasy `Calendar`, która z kolei implementuje interfejs `Comparable<Calendar>`. Dlatego klasa `GregorianCalendar` implementuje interfejs `Comparable<Calendar>`, a nie `Comparable<GregorianCalendar>`.

W takiej sytuacji na ratunek przychodzą nadtypy:

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Teraz metoda `compareTo` przyjmie następującą formę:

```
int compareTo(? super T)
```

Może ona przyjmować obiekty typu `T` — jeśli `T` jest na przykład `GregorianCalendar`, albo nadtypu `T`. Bez względu na wszystko przekazanie obiektu typu `T` do tej metody jest bezpieczne.

Deklaracje typu `<T extends Comparable<? super T>>` dla niedoświadczonego programisty wyglądają przytłaczająco. Jak na ironię, celem tej deklaracji jest pomoc programistom poprzez usunięcie niepotrzebnych ograniczeń parametrów wywołania. Ci, którzy nie są zainteresowani typami generycznymi, szybko uczą się nie zwracać szczególnej uwagi na te deklaracje i przyjmować, że programiści biblioteki się nie pomyliili. Programiści bibliotek natomiast

muszą się przyzwyczaić do typów wieloznacznych, jeśli nie chcą, aby użytkownicy przeklinali ich, kiedy zostaną zmuszeni do wykonywania losowych konwersji, aż program w końcu się skompiluje.

12.8.2. Typy wieloznaczne bez ograniczeń

Typy wieloznaczne można stosować nawet bez żadnych ograniczeń, na przykład `Pair<?>`. Na pierwszy rzut oka zapis ten wydaje się identyczny z surowym typem `Pair`. W rzeczywistości między tymi typami są bardzo duże różnice. Typ `Pair<?>` ma takie metody:

```
? getFirst()
void setFirst(?)
```

Wartość zwrotną metody `getFirst` można przypisać tylko do typu `Object`. Metody `setFirst` nie można w ogóle wywołać, **nakreślając typem** `Object`. Na tym polega główna różnica pomiędzy typami `Pair<?>` i `Pair`: metodę `setObject` surowej klasy `Pair` można wywołać z **dowolnym** obiektem typu `Object`.



Można zastosować wywołanie `setFirst(null)`.

Do czego może się przydać taki typ? Znajduje on zastosowanie w bardzo prostych działaniach. Na przykład poniższa metoda sprawdza, czy para zawiera dany obiekt. Nie potrzebuje znać rzeczywistego typu.

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
```

Stosowania typu wieloznaczniego można uniknąć, zamieniając metodę na generyczną:

```
public static <T> boolean hasNulls(Pair<T> p)
```

jednak wersja z typem wieloznaczonym wydaje się mniej zawiła.

12.8.3. Chwytanie typu wieloznaczniego

Napiszmy metodę przestawiającą elementy w parze:

```
public static void swap(Pair<?> p)
```

Dżoker nie jest zmienną typową, dlatego nie można go używać jako typu w programie. Innymi słowy, nie można napisać poniższego kodu:

```
? t = p.getFirst(); // błąd
p.setFirst(p.getSecond());
p.setSecond(t);
```

Mamy problem, ponieważ musimy przechować tymczasowo pierwszy element, aby wykonać zamianę. Na szczęście istnieje pewne ciekawe rozwiązanie tego problemu. Możemy napisać metodę pomocniczą, na przykład o nazwie swapHelper:

```
public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}
```

Należy zauważyć, że metoda swapHelper jest ogólna, podczas gdy mająca stały parametr typu Pair<?> swap nie.

Metodę swapHelper możemy wywołać w metodzie swap:

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

W tym przypadku parametr T metody swapHelper **chwyta typ wieloznaczny**. Nie wiadomo, jaki typ określa dżoker, ale jest to typ określony, dzięki czemu definicja <T>swapHelper jest w pełni prawidłowa, jeśli T określa tamten typ.

Oczywiście w tym przypadku nie musieliśmy używać typu wieloznaczniego. Można było bezpośrednio zaimplementować metodę <T> void swap(Pair<T> p) jako ogólną, nie używając dżokerów. Spójrzmy jednak na poniższy fragment kodu, w którym typ wieloznaczny ma swoje naturalne miejsce w obliczeniach:

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swapHelper(result); // OK — metoda swapHelper chwyta typ wieloznaczny
}
```

W tym przypadku mechanizmu chwytania typu wieloznaczniego nie można było pominąć.

Chwytanie typu wieloznaczniego jest dozwolone tylko w ścisłe określonych warunkach. Kompilator musi być w stanie zagwarantować, że symbol wieloznaczny reprezentuje jeden określony typ. Na przykład T w ArrayList<Pair<T>> nie może uchwycić typu wieloznaczniego w ArrayList<Pair<?>>. Lista ta może zawierać dwa typy Pair<?>, a symbol ? w każdym z nich może reprezentować inny typ.

Program przedstawiony na listingu 12.3 demonstruje zastosowanie w praktyce omówionych do tej pory technik.

Listing 12.3. pair3/PairTest3.java

```
package pair3;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class PairTest3
{
    public static void main(String[] args)
```

```

{
    Manager ceo = new Manager("Stanisław Skąpy", 800000, 2003, 12, 15);
    Manager cfo = new Manager("Piotr Podstępny", 600000, 2003, 12, 15);
    Pair<Manager> buddies = new Pair<>(ceo, cfo);
    printBuddies(buddies);

    ceo.setBonus(1000000);
    cfo.setBonus(500000);
    Manager[] managers = { ceo, cfo };

    Pair<Employee> result = new Pair<>();
    minmaxBonus(managers, result);
    System.out.println("pierwszy: " + result.getFirst().getName()
        + ", drugi: " + result.getSecond().getName());
    maxminBonus(managers, result);
    System.out.println("pierwszy: " + result.getFirst().getName()
        + ", drugi: " + result.getSecond().getName());
}
}

public static void printBuddies(Pair<? extends Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " i " + second.getName() + "
    są kumplami.");
}

public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}

public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swapHelper(result); // metoda swapHelper chwyta typ wieloznaczny
}
}

class PairAlg
{
    public static boolean hasNulls(Pair<?> p)
    {
        return p.getFirst() == null || p.getSecond() == null;
    }

    public static void swap(Pair<?> p) { swapHelper(p); }

    public static <T> void swapHelper(Pair<T> p)
{

```

```

    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}
}

```

12.9. Refleksja a typy ogólne

Klasa `Class` jest obecnie ogólna. Na przykład `String.class` jest obiektem (w rzeczywistości jedynym obiektem) klasy `Class<String>`.

Przydatność parametru typowego polega na tym, że pozwala on na dokładniejsze określenie typów zwrotnych metod klasy `Class<T>`. Poniższe metody klasy `Class<T>` korzystają z parametru typowego:

```

T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)

```

Metoda `newInstance` zwraca egzemplarz tej klasy utworzony za pomocą konstruktora domyślnego. Jej typ zwrotny można teraz określić jako `T`, czyli taki sam jak klasy `Class<T>`. W ten sposób unikamy rzutowania.

Metoda `cast` zwraca przekazany do niej obiekt rzutowany na typ `T`, jeśli jego typ jest podtypem `T`. W przeciwnym razie generuje wyjątek `BadCastException`.

Metoda `getEnumConstants` zwraca `null`, jeśli klasa nie jest klasą enum, lub tablicę wartości wyliczenia, które wiadomo, że są typu `T`.

Metody `getConstructor` i `getDeclaredConstructor` zwracają obiekt typu `Constructor<T>`. Klasa `Constructor` również jest już ogólna, dlatego jej metoda `newInstance` ma odpowiedni typ zwrotny.

java.lang.Class<T> 1.0

■ T newInstance() 5.0

Zwraca egzemplarz utworzony za pomocą konstruktora domyślnego.

■ T cast(Object obj) 5.0

Zwraca `obj`, jeśli jest `null`, lub może być przekonwertowany na typ `T`, w przeciwnym przypadku generuje wyjątek `BadCastException`.

■ T[] getEnumConstants() 5.0

Zwraca tablicę zapełnioną wartościami, jeśli `T` jest typem wyliczeniowym, lub `null` w przeciwnym przypadku.

- `Class<? super T> getSuperclass()` **5.0**

Zwraca nadklasę tej klasy lub `null`, jeśli `T` nie jest w ogóle klasą lub jest klasą `Object`.

- `Constructor<T> getConstructor(Class... parameterTypes)` **5.0**

- `Constructor<T> getDeclaredConstructor(Class... parameterTypes)` **5.0**

Zwraca konstruktor publiczny lub konstruktor mający parametry o podanych typach.

`java.lang.reflect.Constructor<T>` **1.1**

- `T newInstance(Object... parameters)` **5.0**

Tworzy nowy egzemplarz powstały przy użyciu podanych parametrów.

12.9.1. Zastosowanie parametrów `Class<T>` do dopasowywania typów

Czasami dobrze jest dopasować zmienną typową parametru `Class<T>` w metodzie generycznej. Oto kanoniczny przykład:

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
    IllegalAccessException
{
    return new Pair<T>(c.newInstance(), c.newInstance());
}
```

Jeśli wywołamy poniższą metodę:

```
makePair(Employee.class)
```

`Employee.class` będzie obiektem typu `Class<Employee>`. Parametr typu `T` metody `makePair` odpowiada `Employee`, dzięki czemu kompilator może wywnioskować, że metoda ta zwróci typ `Pair<Employee>`.

12.9.2. Informacje o typach generycznych w maszynie wirtualnej

Jedną z godnych odnotowania cech typów ogólnych w Javie jest wymazywanie typów w maszynie wirtualnej. Może to brzmieć zaskakująco, ale oczyszczone klasy pamiętają o tym, że były ogólne. Na przykład surowa klasa `Pair` wie, że powstała z parametryzowanej klasy `Pair<T>`, mimo że obiekt typu `Pair` nie ma informacji, czy został utworzony jako `Pair<String>`, czy `Pair<Employee>`.

Spójrzmy na poniższą metodę:

```
public static Comparable min(Comparable[] a)
```

Powstała ona w wyniku oczyszczenia poniższej metody ogólnej:

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

Za pomocą rozszerzeń API refleksji można zdobyć następujące informacje:

- Metoda generyczna ma parametr typu o nazwie T .
- Parametr typu ma ograniczenie podtypu, które samo jest typem generycznym.
- Typ ograniczający ma parametr wieloznaczny.
- Parametr wieloznaczny ma ograniczenie nadtypu.
- Metoda ogólna ma parametr w postaci tablicy ogólnej.

Innymi słowy, można zdobyć wszystkie dane dotyczące klas i metod parametryzowanych, które zostały podane w ich deklaracjach. Nie ma natomiast sposobu na dowiedzenie się, jak parametry typowe zostały zastąpione w konkretnych obiektach lub wywołaniach metod.

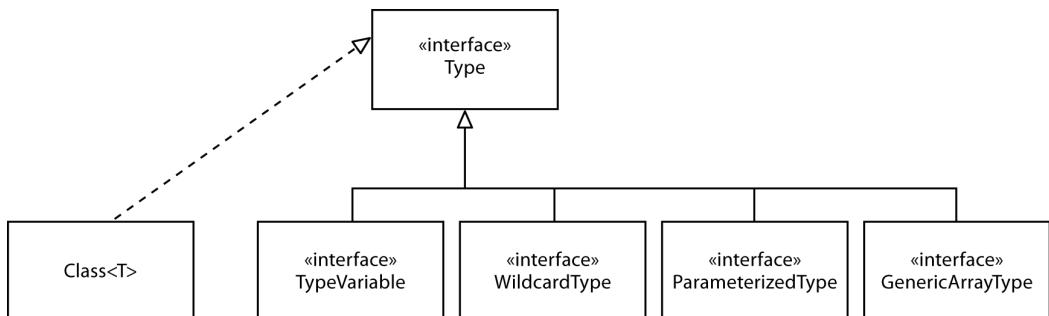


Informacje o typach zawarte w plikach klas, które umożliwiają stosowanie refleksji względem typów ogólnych, nie są zgodne ze starszymi wersjami maszyny wirtualnej.

W pakiecie `java.lang.reflect` znajduje się interfejs o nazwie `Type`, którego celem jest informowanie o deklaracjach typów ogólnych. Interfejs ten ma następujące podtypy:

- klasa `Class` opisująca typy konkretne;
- interfejs `TypeVariable` opisujący zmienne typowe (jak `T extends Comparable<? super T>`);
- interfejs `WildcardType` opisujący typy wieloznaczne (jak `? super T`);
- interfejs `ParameterizedType` opisujący typy klas ogólnych lub interfejsowe (na przykład `Comparable<? super T>`);
- interfejs `GenericArrayType` opisujący tablice ogólne (na przykład `T[]`).

Rysunek 12.5 przedstawia hierarchię dziedziczenia tego interfejsu. Należy zauważać, że cztery ostatnie podtypy są interfejsami — maszyna wirtualna tworzy egzemplarze odpowiednich klas implementujących te interfejsy.



Rysunek 12.5. Interfejs `Type` i jego potomkowie

Program przedstawiony na listingu 12.4 wykorzystuje API refleksji ogólnej do drukowania informacji na temat klas. Dla klasy `Pair` zwrócił następujący raport:

```
class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
```

Listing 12.4. genericReflection/GenericReflectionTest.java

```
package genericReflection;

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

/**
 * @version 1.10 2007-05-15
 * @author Cay Horstmann
 */
public class GenericReflectionTest
{
    public static void main(String[] args)
    {
        // Wczytanie nazwy klasy z argumentów wiersza poleceń lub danych wprowadzonych przez użytkownika
        String name;
        if (args.length > 0) name = args[0];
        else
        {
            Scanner in = new Scanner(System.in);
            System.out.println("Wpisz nazwę klasy (np. java.util.Collections): ");
            name = in.nextLine();
        }

        try
        {
            // Wydrukuj dane generyczne o klasie i jej metodach publicznych
            Class<?> cl = Class.forName(name);
            printClass(cl);
            for (Method m : cl.getDeclaredMethods())
                printMethod(m);
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
    }

    public static void printClass(Class<?> cl)
    {
        System.out.print(cl);
        printTypes(cl.getTypeParameters(), "<", " ", " ", ">", true);
        Type sc = cl.getGenericSuperclass();
        if (sc != null)
        {
            System.out.print(" extends ");
            printType(sc, false);
        }
        printTypes(cl.getGenericInterfaces(), " implements ", " ", "", false);
        System.out.println();
    }
}
```

```
}

public static void printMethod(Method m)
{
    String name = m.getName();
    System.out.print(Modifier.toString(m.getModifiers()));
    System.out.print(" ");
    printTypes(m.getTypeParameters(), "<", " ", " ", "> ", true);

    printType(m.getGenericReturnType(), false);
    System.out.print(" ");
    System.out.print(name);
    System.out.print("(");
    printTypes(m.getGenericParameterTypes(), "", " ", " ", "", false);
    System.out.println(")");
}

public static void printTypes(Type[] types, String pre, String sep, String suf,
    boolean isDefinition)
{
    if (pre.equals(" extends ") && Arrays.equals(types, new Type[]
        ↳{ Object.class })) return;
    if (types.length > 0) System.out.print(pre);
    for (int i = 0; i < types.length; i++)
    {
        if (i > 0) System.out.print(sep);
        printType(types[i], isDefinition);
    }
    if (types.length > 0) System.out.print(suf);
}

public static void printType(Type type, boolean isDefinition)
{
    if (type instanceof Class)
    {
        Class<?> t = (Class<?>) type;
        System.out.print(t.getName());
    }
    else if (type instanceof TypeVariable)
    {
        TypeVariable<?> t = (TypeVariable<?>) type;
        System.out.print(t.getName());
        if (isDefinition)
            printTypes(t.getBounds(), " extends ", " & ", "", false);
    }
    else if (type instanceof WildcardType)
    {
        WildcardType t = (WildcardType) type;
        System.out.print("?");
        printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
        printTypes(t.getLowerBounds(), " super ", " & ", "", false);
    }
    else if (type instanceof ParameterizedType)
    {
        ParameterizedType t = (ParameterizedType) type;
        Type owner = t.getOwnerType();
        if (owner != null)
```

```
        {
            printType(owner, false);
            System.out.print(".");
        }
        printType(t.getRawType(), false);
        printTypes(t.getActualTypeArguments(), "<", " ", " ", ">", false);
    }
    else if (type instanceof GenericArrayType)
    {
        GenericArrayType t = (GenericArrayType) type;
        System.out.print("");
        printType(t.getGenericComponentType(), isDefinition);
        System.out.print("[]");
    }
}
```

Jeśli uruchomimy go na rzecz klasy `ArrayAlg` w katalogu `PairTest2`, raport będzie zawierał następujące dane:

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

Metody użyte w tym programie zostały opisane w wyciągu z API na końcu podrozdziału.

java.lang.Class<T> **1.0**

- ## ■ TypeVariable[] getTypeParameters() 5.0

Zwraca zmienne typowe typu ogólnego, jeśli typ ten został zadeklarowany jako ogólny, lub tablicę o długości 0 w przeciwnym przypadku.

- ## ■ Type getGenericSuperclass() 5.0

Zwraca typ ogólny nadklasy, która została zadeklarowana dla tego typu, lub `null`, jeśli typem tym jest `Object` albo typ niebędący klasą.

- ## ■ Type[] getGenericInterfaces() 5.0

Zwraca typy ogólne interfejsów, które zostały zadeklarowane dla tego typu, przy zachowaniu kolejności z deklaracji, lub tablicę o długości 0, jeśli typ ten nie implementuje interfejsów.

java.lang.reflect.Method 1.1

- ## ■ TypeVariable[] getTypeParameters() 5.0

Zwraca zmienne typowe typu ogólnego, jeśli metoda ta została zadeklarowana jako ogólna, lub tablicę o długości 0 w przeciwnym przypadku.

- ## ■ Type getGenericReturnType() 5.0

Zwraca generyczny typ zwrotny, z którym została zadeklarowana ta metoda.

- ## ■ Type[] getGenericParameterTypes() 5.0

Zwraca generyczne parametry typowe, z którymi została zadeklarowana ta metoda. Jeśli metoda ta nie ma żadnych parametrów, zwracana jest tablica o długości 0.

java.lang.reflect.TypeVariable 5.0

- `String getName()`
Zwraca nazwę zmiennej typowej.
- `Type[] getBounds()`
Zwraca ograniczenia podklasy zmiennej typowej lub tablicę o długości 0, jeśli zmienna nie ma ograniczeń.

java.lang.reflect.WildcardType 5.0

- `Type[] getLowerBounds()`
Zwraca ograniczenia (extends) podklasy zmiennej typowej lub tablicę o długości 0, jeśli nie ma żadnych ograniczeń podklasowych.
- `Type[] getUpperBounds()`
Zwraca ograniczenia (super) nadklasy zmiennej typowej lub tablicę o długości 0, jeśli nie ma żadnych ograniczeń nadklasowych.

java.lang.reflect.ParameterizedType 5.0

- `Type getRawType()`
Zwraca typ surowy typu parametryzowanego.
- `Type[] getActualTypeArguments()`
Zwraca parametry typu, które zostały użyte w deklaracji typu parametryzowanego.
- `Type getOwnerType()`
Zwraca typ klasy zewnętrznej, jeśli jest wywołana na rzecz klasy wewnętrznej, lub `null` w przypadku wywołania na rzecz klasy najwyższego poziomu.

java.lang.reflect.GenericArrayType 5.0

- `Type getGenericComponentType()`
Zwraca generyczny typ komponentu, który został użyty w deklaracji tego typu tablicy.

Potrafimy już używać klas ogólnych i tworzyć własne klasy oraz metody tego typu. Nie mniej ważną umiejętnością zdobytą w tym rozdziale jest zdolność rozumienia deklaracji typów ogólnych, które można spotkać w dokumentacji API i komunikatach o błędach. Wyczerpującym źródłem wiedzy na temat typów ogólnych w Javie jest, sporządzona przez Angelikę Langer, znakomita lista często (i niezbyt często) zadawanych pytań na ten temat: <http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>.

W kolejnym rozdziale zobaczymy, jak z typów generycznych korzystają kolekcje.

13

Kolekcje

W tym rozdziale:

- Interfejsy kolekcyjne
- Konkretne klasy kolekcyjne
- Architektura kolekcji
- Algorytmy
- Stare kolekcje

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

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

13.1. Interfejsy kolekcyjne

Początkowo w Javie dostępnych było tylko kilka klas implementujących najbardziej przydatne struktury danych: `Vector`, `Stack`, `Hashtable`, `BitSet` oraz interfejs `Enumerable` odpowiadający za abstrakcyjny mechanizm dostępu do elementów w każdym z tych kontenerów.

Było to z pewnością mądro posunięcie ze strony projektantów języka — opracowanie pełnej biblioteki klas kolekcyjnych wymaga czasu i doświadczenia.

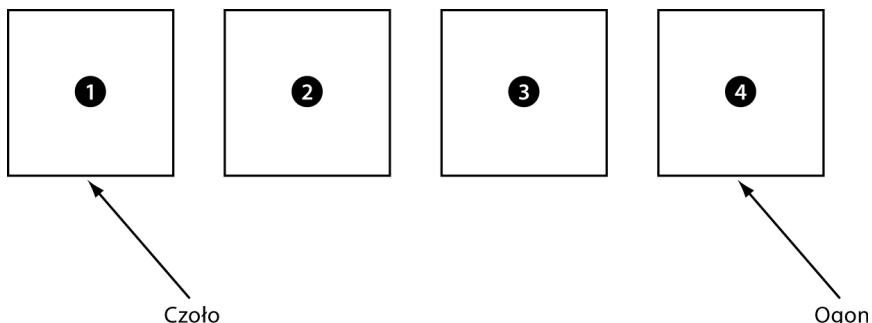
Projektanci doszli do wniosku, że czas na zaprezentowanie pełnego zestawu struktur danych nadszedł w Java 1.2. Musieli dokonać wielu trudnych wyborów, ponieważ chcieli stworzyć bibliotekę mającą niewielkie rozmiary i łatwą do opanowania dla programistów. Starali się uniknąć poziomu złożoności charakteryzującego bibliotekę C++ STL (ang. *Standard Template Library*), jednocześnie próbując skorzystać z dobrodziejstw algorytmów uogólnionych, których pionierem była właśnie wymieniona biblioteka. Stare klasy musiały pasować do nowej architektury. Jak to zwykle bywa przy projektowaniu bibliotek kolekcji, kilkakrotnie stawano przed trudnym wyborem, co zaowocowało kilkoma osobliwymi decyzjami projektowymi. W tym podrozdziale przedstawiamy podstawową strukturę architektury kolekcji Javy, pokazujemy sposoby jej wykorzystania oraz wyjaśniamy, czym kierowali się projektanci, podejmując niektóre bardziej kontrowersyjne decyzje.

13.1.1. Oddzielenie warstwy interfejsów od warstwy klas konkretnych

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

Interfejs Queue pozwala na dodawanie elementów na końcu kolejki, usuwanie ich z początku struktury danych oraz sprawdzanie liczby elementów w kolejce. Ta struktura danych znajduje zastosowanie przy tworzeniu zbiorów obiektów, z których elementy są pobierane zgodnie z zasadą „pierwszy przyjdzie, pierwszy wyjdzie” (zobacz rysunek 13.1).

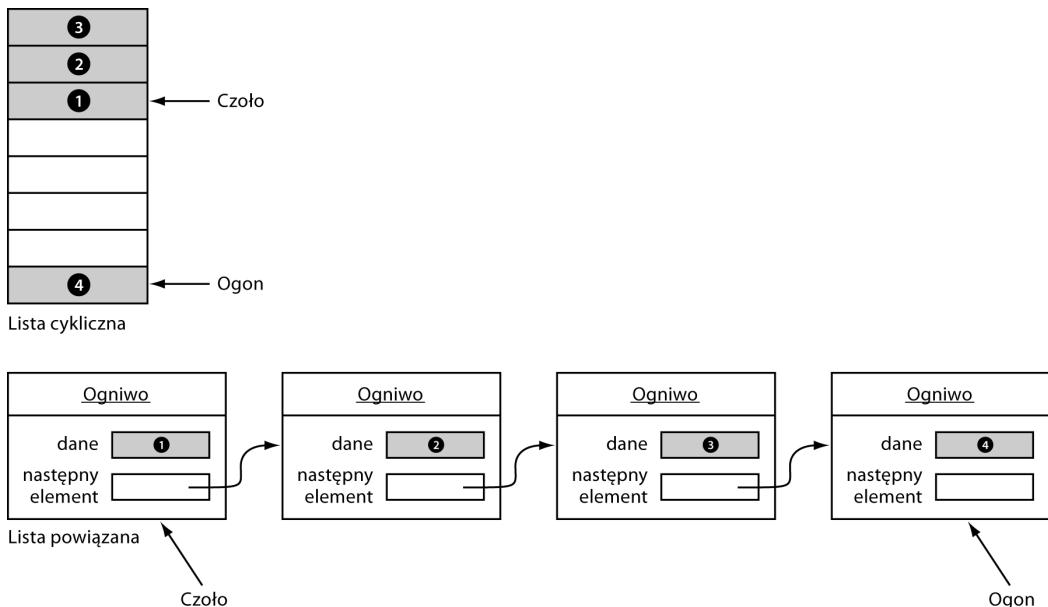
Rysunek 13.1.
Kolejka



Interfejs Queue w swojej uproszczonej formie może wyglądać następująco:

```
interface Queue<E> // uproszczona wersja interfejsu z biblioteki standardowej
{
    void add(E element);
    E remove();
    int size();
}
```

Sam interfejs nie zawiera żadnych danych na temat implementacji kolejki. Z dwóch najczęściej spotykanych implementacji kolejki jedna działa na zasadzie listy cyklicznej, a druga listy powiązanej (rysunek 13.2).



Rysunek 13.2. Implementacje kolejki

Każda z tych implementacji można zrealizować za pomocą klasy implementującej interfejs Queue.

```
class CircularArrayQueue<E> implements Queue<E> // klasa niebiblioteczna
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private E[] elements;
    private int head;
    private int tail;
}

class LinkedListQueue<E> implements Queue<E> // klasa niebiblioteczna
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private Link head;
    private Link tail;
}
```

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



W bibliotece Javy nie ma klas o nazwach `CircularQueue` i `LinkedListQueue`. Używamy ich do wyjaśnienia różnicy pomiędzy interfejsami kolekcyjnymi a implementacjami. Do utworzenia listy cyklicznej można użyć wprowadzonej w Java SE 6 klasy `ArrayDeque`. Listy powiązane tworzy klasa `LinkedList`, która implementuje interfejs `Queue`.

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

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

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

Co sprawia, że wybieramy jedną implementację zamiast drugiej? Interfejs nie dostarcza żadnych informacji na temat wydajności. Lista cykliczna jest nieco szybsza od listy powiązanej, a więc — ogólnie rzecz biorąc — jest preferowana. Jednak jak zawsze jest coś za coś.

Lista cykliczna jest kolekcją **ograniczoną**, czyli ma ograniczoną pojemność. Jeśli nie określmy limitu obiektów przechowywanych w takiej liście, niewykluczone, że lepiej byśmy na tym wyszli, gdybyśmy zastosowali listę powiązaną.

W dokumentacji API można znaleźć jeszcze jeden zestaw klas, których nazwy zaczynają się od słowa `Abstract`, na przykład `AbstractQueue`. Klasy te są przeznaczone dla twórców bibliotek. W razie (mało prawdopodobnej) potrzeby zaimplementowania własnej klasy kolejki łatwiej rozszerzyć klasę `AbstractQueue`, niż zaimplementować wszystkie metody interfejsu `Queue`.

13.1.2. Interfejsy Collection i Iterator

Interfejsem o kluczowym znaczeniu w hierarchii kolekcji jest interfejs `Collection`. Dwie z jego metod mają fundamentalne znaczenie:

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

Pozostałe metody tego interfejsu omawiamy dalej.

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

Metoda iterator zwraca obiekt implementujący interfejs Iterator. Za pomocą tego obiektu można odwiedzić kolejno wszystkie znajdujące się w kolekcji elementy.

13.1.2.1. Iteratory

W interfejsie Iterator występują trzy metody:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

Wywołując wielokrotnie metodę next, można kolejno odwiedzić wszystkie elementy kolekcji. Jeśli metoda ta napotka koniec kolekcji, zgłosi wyjątek NoSuchElementException. Dlatego przed nią należy zawsze wywoływać metodę hasNext, która zwraca wartość true, jeśli są jeszcze jakieś elementy. Aby przejrzeć wszystkie elementy kolekcji, należy utworzyć obiekt typu Iterator i wywoływać metodę next tak długo, jak metoda hasNext zwraca wartość true. Na przykład:

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

W Java SE 5.0 wprowadzono zgrabniejszą wersję tej pętli. Jej bardziej zwięzły zapis realizuje się za pomocą pętli for each:

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

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

Pętlę for each można stosować do wszystkich obiektów implementujących interfejs Iterable, w którym znajduje się tylko jedna metoda:

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

Interfejs Collection rozszerza interfejs Iterable. Dzięki temu pętli for each można używać do działań na wszystkich kolekcjach ze standardowej biblioteki.

Kolejność odwiedzania elementów zależy od rodzaju kolekcji. W przypadku listy ArrayList iterator zaczyna od indeksu o numerze 0 i zwiększa ten numer w każdym kolejnym powtórzeniu (iteracji). Natomiast dostęp do elementów w zbiorze HashSet odbywa się w zasadzie

losowo. Pewne jest, że przemierzając tę kolekcję, uzyskamy dostęp do każdego jej elementu, ale nie wiadomo, w jakiej kolejności będzie się to odbywać. Nie sprawia to zazwyczaj problemu, ponieważ w działaniach typu obliczanie sumy lub zliczanie elementów pasujących do wzorca kolejność jest nieistotna.



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

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

Iteratory w Javie należy wyobrażać sobie jako obiekty znajdujące się **pomiędzy elementami**. W chwili wywołania metody `next` iterator **przeskakuje** kolejny element i zwraca referencję do elementu, który właśnie przeskoczył (zobacz rysunek 13.3).



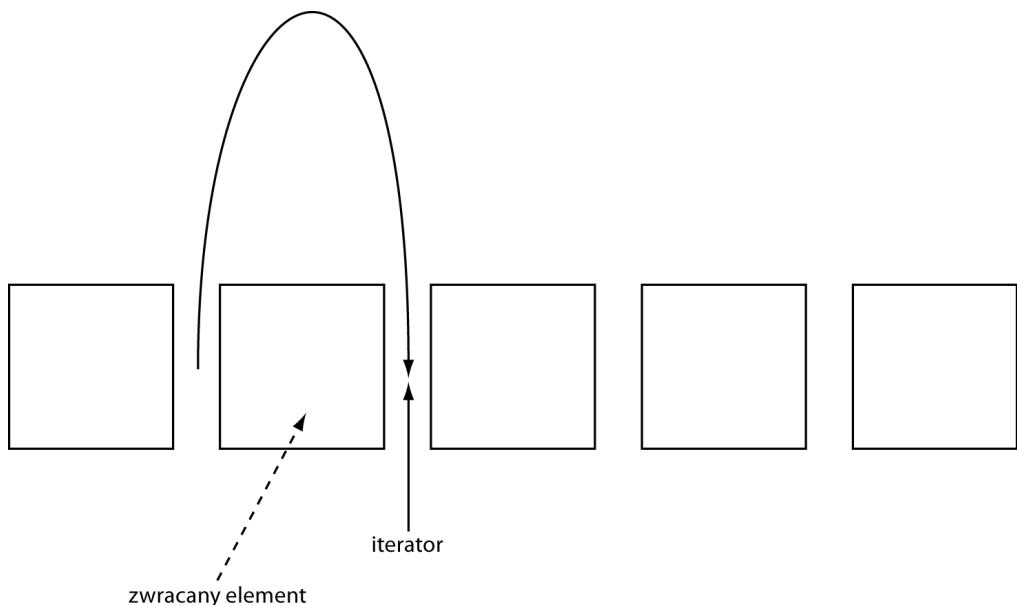
Istnieje jeszcze inna ciekawa analogia. Instrukcję `Iterator.next` można traktować jako odpowiednik instrukcji `InputStream.read`. Odczyt bajta ze strumienia automatycznie oznacza jego „połknienie”. Kolejne wywołanie metody `read` powoduje „połkniecie” i zwrócenie następnego bajta z danych wejściowych.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

13.1.2.2. Usuwanie elementów

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

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

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



Rysunek 13.3. Przesuwanie iteratora

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

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

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

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

13.1.2.3. Uogólnione metody użytkowe

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

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

Twórcy biblioteki standardowej doszli do wniosku, że niektóre z tych metod są tak przydatne, iż powinny być dostępne w bibliotece. Dzięki temu użytkownicy tego zbioru klas nie muszą wielokrotnie wynajdywać koła. Jedna z tych metod nosi nazwę `contains`.

W rzeczywistości w interfejsie Collection znajduje się spora liczba przydatnych metod, które muszą być udostępniane przez wszystkie implementujące go klasy. Należą do nich:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

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

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

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

    public boolean contains(Object obj)
    {
        for (E element : c) // wywołuje metodę iterator()
            if (element.equals(obj))
                return true;
        return false;
    }
    ...
}
```

Konkretna klasa kolekcyjna może być rozszerzeniem klasy AbstractCollection. W klasie konkretnej konieczne jest zdefiniowanie metody iterator, ale metoda contains jest już w niej dostępna, ponieważ zostaje odziedziczona po nadklasie abstrakcyjnej AbstractCollection. Jeśli jednak w podklasie istnieje możliwość zdefiniowania efektywniejszej metody contains, nic nie stoi na przeszkodzie, aby to zrobić.

Jest to bardzo dobre podejście do projektowania architektury klas. Użytkownicy kolekcji mają do dyspozycji bogaty zestaw metod, a twórcy struktur danych nie są obciążani implementacją wszystkich rutynowych metod.

`java.util.Collection<E> 1.2`

■ `Iterator<E> iterator()`

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

■ `int size()`

Zwraca liczbę elementów przechowywanych w kolekcji.

■ `boolean isEmpty()`

Zwraca wartość true, jeśli kolekcja nie zawiera żadnych elementów.

■ `boolean contains(Object obj)`

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

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

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

■ `boolean add(Object element)`

Dodaje element do kolekcji. Zwraca wartość true, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

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

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

■ `boolean remove(Object obj)`

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

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

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

■ `void clear()`

Usuwa wszystkie elementy z kolekcji.

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

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

■ `Object[] toArray()`

Zwraca tablicę obiektów zapełnioną elementami z kolekcji.

■ `<T> T[] toArray(T[] arrayToFill)`

Zwraca tablicę zapełnioną obiektami z kolekcji. Jeśli tablica arrayToFill jest odpowiedniej długości, zostaje zapełniona elementami kolekcji. Dodatkowe miejsca są zapełniane wartościami null. W przeciwnym przypadku tworzona jest nowa tablica takiego samego typu jak arrayToFill i o takiej samej długości jak rozmiar kolekcji.

java.util.Iterator<E> **1.2**

■ boolean hasNext()

Zwraca wartość true, jeśli są jeszcze elementy do odwiedzenia.

■ E next()

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

■ void remove()

Usuwa ostatnio odwiedzony obiekt. Wywołanie tej metody musi następować bezpośrednio po odwiedzeniu elementu. Jeśli kolekcja zmieniła się od ostatniego odwiedzenia elementu, metoda ta wyrzuci wyjątek IllegalStateException.

13.2. Konkretnie klasy kolekcyjne

Zamiast zbytnio zagłębiać się w tajniki wszystkich interfejsów, postanowiliśmy opisać najpierw konkretnie struktury danych. Po dokładnym zapoznaniu się z klasami konkretnymi wróćmy do tematyki abstrakcyjnej, a zwłaszcza przyjrzymy się organizacji architektonicznej tych klas. Tabela 13.1 przedstawia zestawienie kolekcji dostępnych w standardowej bibliotece oraz ich krótkie opisy (dla uproszczenia pomineliśmy kolekcje bezpieczne dla wątków, które zostały opisane w rozdziale 14.). Wszystkie klasy prezentowane w tabeli, z wyjątkiem tych, których nazwy kończą się słowem Map, implementują interfejs Collection. Mapy implementują interfejs Map i zostały opisane w podrozdziale 13.2.8, „Mapy”.

13.2.1. Listy powiązane

W wielu prezentowanych do tej pory przykładach kodu wykorzystywaliśmy tablice i ich dynamicznego krewniaka, czyli klasę ArrayList. Niestety tablice i listy tablicowe mają jedną poważną wadę — usuwanie elementów z ich środka jest mało efektywne, ponieważ czynność ta wymaga przesunięcia wszystkich elementów znajdujących się za tym usuwanym w stronę początku (zobacz rysunek 13.4). To samo dotyczy wstawiania elementów.

Problem ten pomaga rozwiązać inna powszechnie znana struktura danych, nazywana **listą powiązaną** (ang. *linked list*). Podczas gdy obiekty w tablicy są zapisywane w kolejnych komórkach pamięci, w liście powiązanej znajdują się one w osobnych **ogniwach** (ang. *link*). Każde ognivo przechowuje także referencję do następnego ogniva w szeregu. W Javie praktycznie wszystkie listy powiązane są **listami dwukierunkowymi** (ang. *doubly linked list*), co oznacza, że każde ognivo przechowuje także referencję do ogniva je poprzedzającego (zobacz rysunek 13.5).

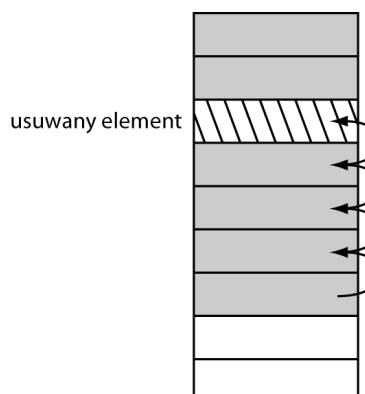
Usuwanie elementów z środka listy powiązanej charakteryzuje się krótkim czasem wykonania, ponieważ aktualizowane są tylko elementy znajdujące się po obu stronach usuwanego obiektu (zobacz rysunek 13.6).

Tabela 13.1. Kolekcje konkretne w bibliotece Javy

Typ kolekcji	Opis
ArrayList	Indeksowana lista o dynamicznie zmieniających się rozmiarach
LinkedList	Uporządkowana lista pozwalająca na szybkie wstawianie i usuwanie elementów w dowolnej lokalizacji
ArrayDeque	Nieposiadająca ani początku, ani końca lista cykliczna
HashSet	Nieuporządkowana kolekcja, w której wszystkie obiekty muszą być unikatowe
TreeSet	Uporządkowany zbiór
EnumSet	Zbiór wartości typu wyliczeniowego
LinkedHashSet	Zbiór pamiętający kolejność wstawianych do niego elementów
PriorityQueue	Kolekcja pozwalająca na szybkie usunięcie najmniejszego elementu
HashMap	Struktura danych przechowująca pary klucz – wartość
TreeMap	Mapa sortująca klucze
EnumMap	Mapa, w której klucze są typami wyliczeniowymi
LinkedHashMap	Mapa pamiętająca kolejność wstawianych do niej elementów
WeakHashMap	Mapa, której wartości mogą zostać usunięte przez system zbierania nieużytków, jeśli nie są używane gdzieś indziej
IdentityHashMap	Mapa przechowująca klucze porównywane za pomocą operatora == zamiast equals

Rysunek 13.4.

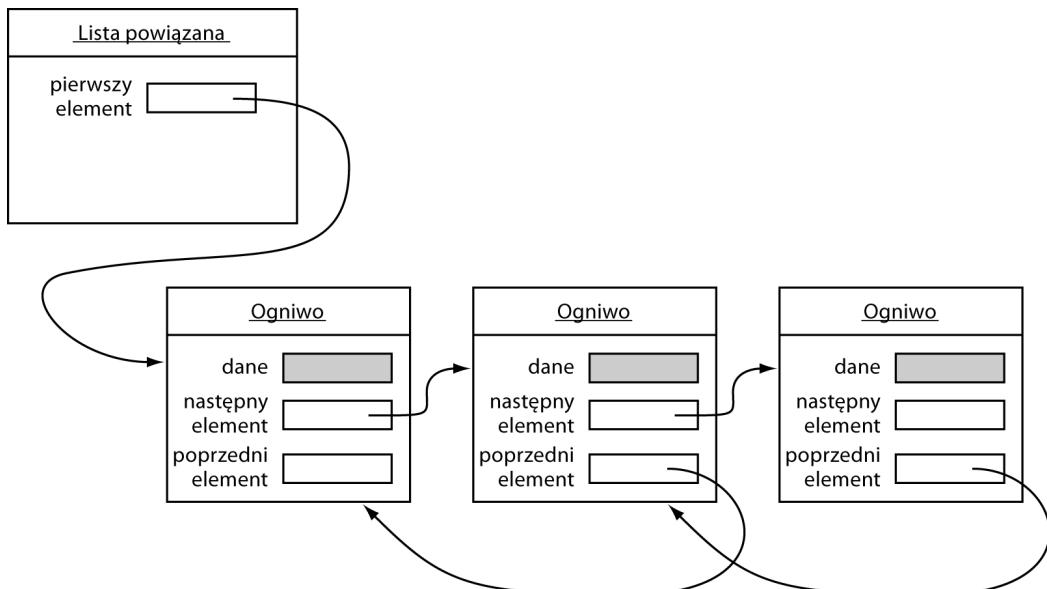
Usuwanie elementu z tablicy



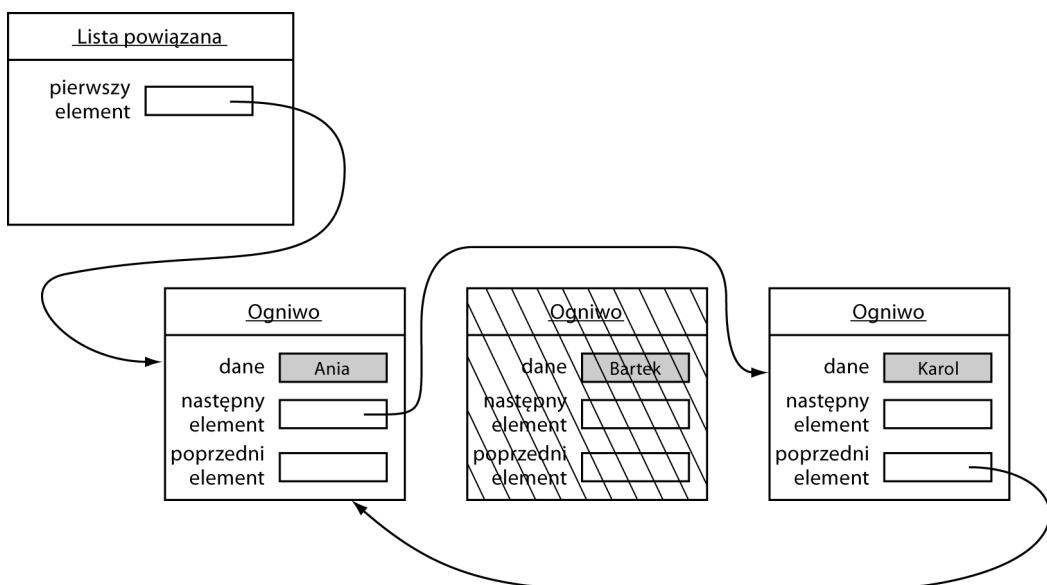
Wiele osób ukończyło kurs struktur danych, na którym uczy się implementacji list powiązanych. Niektórzy mogą mieć złe wspomnienia związane z płatanią połączeń przy usuwaniu lub dodawaniu elementów do list powiązanych. Dla tych osób mamy dobrą wiadomość — w bibliotece kolekcji Javy znajduje się gotowa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

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

```
List<String> staff = new LinkedList<String>(); // Klasa LinkedList implementuje interfejs List
staff.add("Ania");
staff.add("Bartek");
```



Rysunek 13.5. Lista dwukierunkowa



Rysunek 13.6. Usuwanie elementu z listy powiązanej

```

staff.add("Karol");
Iterator iter = staff.iterator();
String first = iter.next();
String second = iter.next();
iter.remove();

```

// dojście do pierwszego elementu
// dojście do drugiego elementu
// usunięcie ostatnio odwiedzonego elementu

Pomiędzy listami powiązanymi a kolekcjami uogólnionymi istnieje pewna istotna różnica. Lista powiązana jest **kolekcją uporządkowaną**, w której położenie obiektów ma znaczenie. Metoda `LinkedList.add` dodaje obiekt na końcu listy. Często jednak elementy są wstawiane do środka listy. Metoda `add` związana z położeniem elementu znajduje się w zestawie obowiązków iteratora, ponieważ iteratory służą do opisu położenia elementów w kolekcjach. Zastosowanie iteratorów do dodawania elementów jest uzasadnione tylko w uporządkowanych kolekcjach. Na przykład omawiane w następnej kolejności zbiory są strukturami danych, które nie porządkują w żaden sposób swoich elementów. Dlatego w interfejsie `Iterator` nie ma metody `add`. Umieszczonej ją za to w interfejsie `ListIterator`:

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

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

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

```
E previous()
boolean hasPrevious()
```

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

Metoda `listIterator` z klasy `LinkedList` zwraca obiekt iteratora implementujący interfejs `ListIterator`.

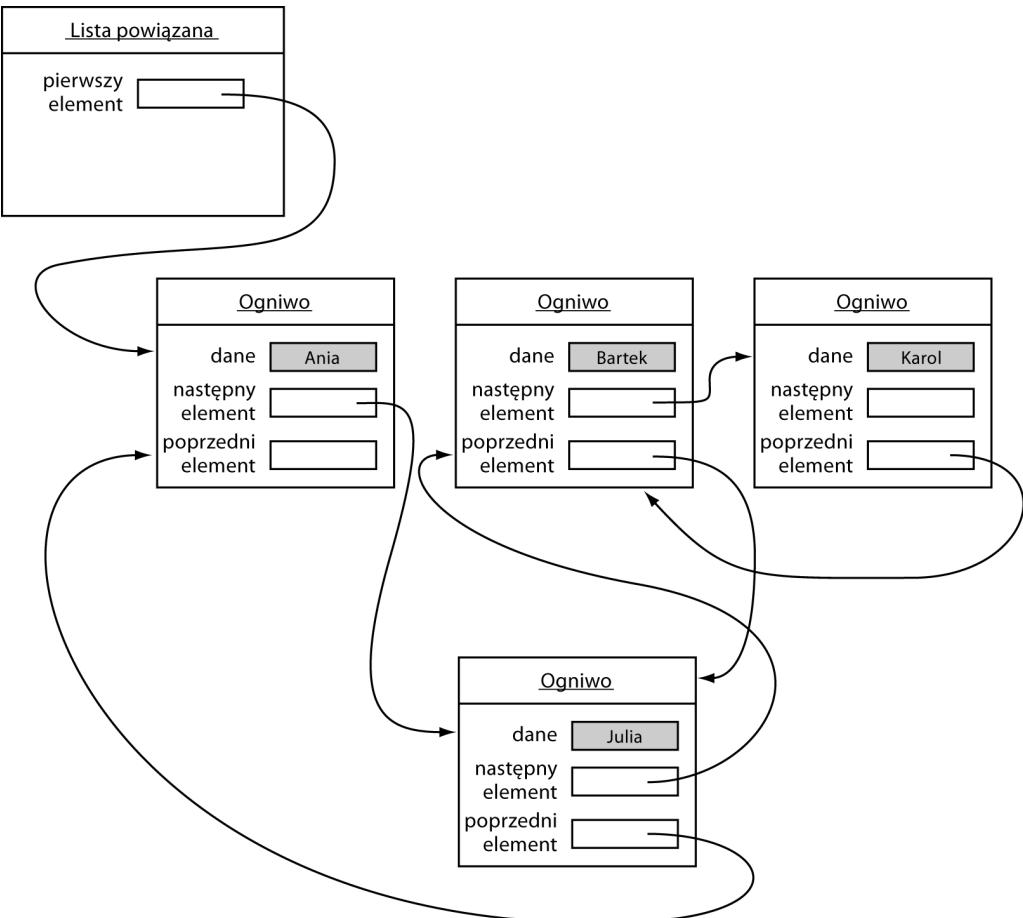
```
ListIterator<String> iter = staff.listIterator();
```

Metoda `add` dodaje element **przed** iteratorem. Na przykład poniższe instrukcje pomijają pierwszy element na liście powiązanej i dodają element `Julia` przed drugim elementem (zobacz rysunek 13.7):

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

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

Jeśli metoda `add` zostanie użyta z nieużywanym jeszcze iteratorem utworzonym przez metodę `listIterator`, wskazującym początek listy powiązanej, nowy element zostanie dodany na samym początku listy. Jeśli iterator przejdzie za ostatni element listy (czyli znajdzie się w miejscu, w którym metoda `hasNext` zwraca wartość `false`), dodawany element będzie stanowił nowy ogon listy. W liście o długości n istnieje $n+1$ miejsc, w których można wstawiać elementy. Ta liczba odpowiada liczbie położen, w których może się znaleźć iterator.



Rysunek 13.7. Dodawanie elementu do listy powiązanej

Jeśli na przykład lista powiązana zawiera trzy elementy A, B i C, to istnieją w niej cztery miejsca (oznaczone znakiem |), w których można wstawić nowy element:

ABC
A|BC
AB|C
ABC|



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

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

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

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

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

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

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

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

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



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

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

Wiemy już z poprzedniego podrozdziału, że wiele przydatnych metod do działań na listach powiązanych znajduje się w interfejsie `Collection`. Większość z nich jest zdefiniowana w nad-klasie `AbstractCollection` klasy `LinkedList`. Na przykład metoda `toString` wywołuje metodę `toString` na rzecz każdego elementu i tworzy jeden długi łańcuch w formacie `[A, B, C]`, co

przydaje się podczas debugowania. Aby sprawdzić, czy na liście znajduje się określony element, należy użyć metody `contains`. Na przykład instrukcja `staff.contains("Henryk")` zwróci wartość `true`, jeśli lista powiązana zawiera łańcuch `Henryk`.

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

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

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

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

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

```
for (int i = 0; i < list.size(); i++)
    operacje na elemencie list.get(i);
```

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



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

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

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

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

Program przedstawiony na listingu 13.1 demonstruje praktyczne zastosowanie list powiązanych. Tworzy on dwie listy, scalą je, usuwa co drugi element z drugiej z nich, a na końcu testuje działanie metody `removeAll`. Zachęcamy do prześledzenia przepływu sterowania w tym programie i przyjrzenia się iteratorm. Pomocne mogą się okazać rysunki przedstawiające położenie iteratatorów:

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

Listing 13.1. `linkedList/LinkedListTest.java`

```
package linkedList;

import java.util.*;

/**
 * Program demonstrujący działania na listach powiązanych
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class LinkedListTest
{
    public static void main(String[] args)
    {
        List<String> a = new LinkedList<>();
        a.add("Ania");
        a.add("Karol");
        a.add("Eryk");

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

        // Scalenie list a i b

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

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

        System.out.println(a);

        // Usunięcie co drugiego słowa z listy b

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

```

```

    {
        bIter.next(); // Opuszczenie następnego elementu
        bIter.remove(); // Usunięcie elementu
    }
}

System.out.println(b);

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

a.removeAll(b);

System.out.println(a);
}
}

```

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

java.util.List<E> 1.2

- `ListIterator<E> listIterator()`
Zwraca iterator listy.
- `ListIterator<E> listIterator(int index)`
Zwraca iterator listy, którego pierwsze wywołanie metody `next` zwróci element znajdujący się pod podanym indeksem.
- `void add(int i, E element)`
Dodaje element w określonym miejscu.
- `void addAll(int i, Collection<? extends E> elements)`
Dodaje wszystkie elementy z kolekcji w określonym miejscu.
- `E remove(int i)`
Usuwa i zwraca element znajdujący się w określonym miejscu.
- `E get(int i)`
Zwraca element znajdujący się w określonym miejscu.
- `E set(int i, E element)`
Zastępuje element znajdujący się w określonym miejscu nowym elementem i zwraca stary element.
- `int indexOf(Object element)`
Zwraca położenie pierwszego wystąpienia elementu `element` lub wartość `-1`, jeśli element nie zostanie znaleziony.
- `int lastIndexOf(Object element)`
Zwraca położenie ostatniego wystąpienia elementu `element` lub wartość `-1`, jeśli element nie zostanie znaleziony.

java.util.ListIterator<E> **1.2**

- void add(E newElement)
Dodaje element przed bieżącą pozycją.
- void set(E newElement)
Zastępuje ostatni element odwiedzony przez metodę next lub previous nowym elementem. Zgłasza wyjątek IllegalStateException, jeśli lista została zmodyfikowana od czasu ostatniego wywołania metody next lub previous.
- boolean hasPrevious()
W trakcie przemierzania listy wstecz zwraca wartość true, jeśli są jeszcze nieodwiedzone elementy.
- E previous()
Zwraca poprzedni obiekt. W przypadku osiągnięcia początku listy zgłasza wyjątek NoSuchElementException.
- int nextIndex()
Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody next.
- int previousIndex()
Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody previous.

java.util.LinkedList<E> **1.2**

- LinkedList()
Tworzy pustą listę powiązaną.
- LinkedList(Collection<? extends E> elements)
Tworzy listę powiązaną i dodaje do niej wszystkie elementy z określonej kolekcji.
- void addFirst(E element)
- void addLast(E element)
Dodaje element na początku lub końcu listy.
- E getFirst()
- E getLast()
Zwraca element znajdujący się na początku lub końcu listy.
- E removeFirst()
- E removeLast()
Usuwa element z początku lub końca listy.

13.2.2. Listy tablicowe

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



Wielu doświadczonych programistów używało do tej pory klasy Vector jako dynamicznej tablicy. Czemu używać klasy ArrayList zamiast Vector? Z jednego powodu — wszystkie metody klasy Vector są **synchronizowane**, dzięki czemu można bezpiecznie uzyskać dostęp do jednego obiektu tej klasy z dwóch wątków. Jeśli natomiast pracujemy tylko w jednym wątku — co zdarza się nieporównywalnie częściej — tracimy bardzo dużo czasu na synchronizację. Natomiast metody klasy ArrayList nie są synchronizowane. Zalecamy stosowanie klasy ArrayList zamiast Vector zawsze wtedy, gdy nie jest potrzebna synchronizacja.

13.2.3. Zbiór HashSet

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

Powszechnie znaną strukturą danych pozwalającą szybko wyszukiwać obiekty jest **tablica miesząca** (ang. *hash table*; nazywana też haszową). Tablica ta oblicza liczbę całkowitą, zwaną **kodem mieszącym** (ang. *hash code*), dla każdego obiektu. Kod mieszący powstaje w jakiś sposób z pól obiektu, najlepiej w taki, aby obiekty zawierające różne dane dawały różne kody. Tabela 13.2 zawiera kilka przykładowych kodów mieszących zwróconych przez metodę hashCode z klasy String.

Tabela 13.2. Kody mieszące zwrócone przez metodę hashCode

Łańcuch	Kod mieszący
Lee	76268
lee	107020
eel	100300

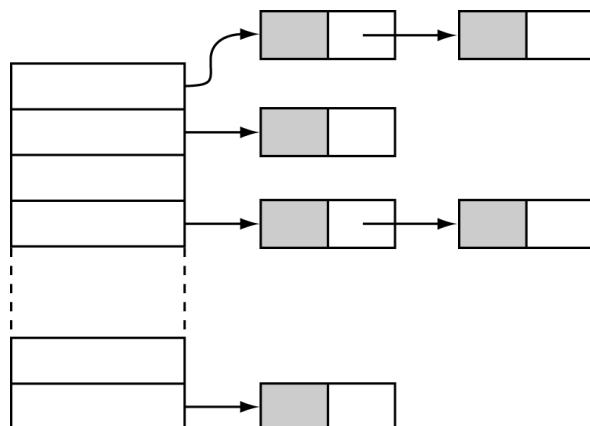
Definiując własne klasy, należy zaimplementować własną metodę `hashCode` — więcej na ten temat można znaleźć w rozdziale 5. Implementacja ta musi być zgodna z metodą `equals` — jeśli `a` jest równe `b` (`a.equals(b)`), to `a` i `b` muszą mieć ten sam kod mieszający.

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

Tablice mieszające w Javie są zaimplementowane jako tablice list powiązanych. Listy w tej sytuacji noszą miano **komórek** lub **kubelków** (ang. *bucket*; zobacz rysunek 13.8). Aby określić miejsce do przechowywania obiektu w tablicy, należy obliczyć jego kod mieszający i znaleźć resztę z dzielenia tej liczby przez liczbę wszystkich komórek. Jeśli na przykład obiekt ma kod mieszający 76268, a wszystkich komórek jest 128, zostanie on umieszczony w komórce 108 (ponieważ reszta z dzielenia 76 268 przez 128 wynosi 108). Jeśli mamy szesczęście i nie ma w tej komórce żadnego innego elementu, umieszczamy tam nasz obiekt. Oczywiście nie da się uniknąć trafienia komórki ze znajdującym się wewnątrz obiektem. Sytuację taką nazywamy **kolizją**. Wtedy porównujemy nowy obiekt z wszystkimi obiektami w tej komórce, aby sprawdzić, czy go tam jeszcze nie ma. Jeśli kody mieszające są rozsądnie rozłożone losowo, a liczba komórek wystarczająco duża, powinno być konieczne przeprowadzenie tylko kilku porównań.

Rysunek 13.8.

Tablica mieszająca



Aby zyskać większą kontrolę nad działaniem tablicy, można określić początkowy licznik komórek. Licznik ten określa liczbę komórek, w których przechowywane są obiekty o identycznych wartościach mieszających. Jeśli do tablicy wstawi się zbyt wiele elementów, zwiększa się liczba kolizji i pogarsza szybkość znajdowania elementów.

Jeśli znana jest przybliżona ostateczna liczba elementów tablicy, to można ustawić licznik komórek. Zazwyczaj ustawia się go na 75 do 150% spodziewanej liczby elementów. Niektórzy badacze utrzymują, że dobrze jest ustawić ten licznik na liczbę pierwszą, co pozwoli uniknąć grupowania się kluczy, jednak nie ma na to przekonujących dowodów. Biblioteka standardowa stosuje liczniki komórek będące potęgami liczby 2, a domyślna liczba to 16 (każda wartość podawana jako wielkość tablicy jest automatycznie zaokrąglana do najbliższej potęgi dwójkowej).

Oczywiście nie zawsze wiadomo, ile elementów będzie trzeba przechować. Czasami można też podać za małą liczbę. Jeśli tablica zostanie przepelniona, konieczna jest jej **reorganizacja**. Polega to na utworzeniu nowej tablicy z większą liczbą komórek i przeniesieniu wszystkich danych do tej nowej tablicy. Stara tablica zostaje usunięta. O reorganizacji tablicy decyduje **współczynnik zapelnienia** (ang. *load factor*). Jeśli na przykład współczynnik ten wynosi 0,75 (wartość domyślna), a tablica zostanie zapelniona w ponad 75 procentach, następuje jej automatyczna reorganizacja, w wyniku której tworzona jest tablica o dwukrotnie większej liczbie komórek. W większości zastosowań najlepiej pozostawić współczynnik 0,75 bez zmian.

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

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

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

Przykładowy program zaprezentowany na końcu tego podrozdziału (listing 13.2) wczytuje słowa ze strumienia `System.in`, dodaje je do zbioru, a następnie drukuje je na ekranie. Można na przykład wprowadzić tekst *Alice in Wonderland* (który można pobrać ze strony <http://www.gutenberg.net>) za pomocą poniższego polecenia:

```
java SetTest < alice30.txt
```

Program wczyta wszystkie słowa ze strumienia wejściowego i wstawi je do zbioru `HashSet`. Następnie przejdzie przez wszystkie unikatowe słowa i wydrukuje ich liczbę (książka *Alice in Wonderland* zawiera 5909 unikatowych słów, wliczając informację o prawach autorskich zamieszczoną na początku pliku). Słowa są wyświetlane w losowej kolejności.

Listing 13.2. set/SetTest.java

```
package set;

import java.util.*;

/**
 * Program drukujący wszystkie słowa ze strumienia wejściowego przy użyciu zbioru
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class SetTest
{
    public static void main(String[] args)
    {
        Set<String> words = new HashSet<>(); // Klasa HashSet implementuje interfejs Set
        long totalTime = 0;
```

```

Scanner in = new Scanner(System.in);
while (in.hasNext())
{
    String word = in.next();
    long callTime = System.currentTimeMillis();
    words.add(word);
    callTime = System.currentTimeMillis() - callTime;
    totalTime += callTime;
}

Iterator<String> iter = words.iterator();
for (int i = 1; i <= 20 && iter.hasNext(); i++)
    System.out.println(iter.next());
System.out.println("... ");
System.out.println(words.size() + " niepowtarzających się słów. " + totalTime
+ " milisekund.");
}
}

```



Należy zachować ostrożność przy modyfikowaniu elementów zbioru. Jeśli kod mieszący elementu zmieni się, element ten będzie się znajdował w niewłaściwym miejscu w strukturze danych.

java.util.HashSet<E> **1.2**

- **HashSet()**

Tworzy pusty obiekt HashSet.

- **HashSet(Collection<? extends E> elements)**

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

- **HashSet(int initialCapacity)**

Tworzy pusty obiekt HashSet o określonej pojemności (liczbie komórek).

- **HashSet(int initialCapacity, float loadFactor)**

Tworzy obiekt HashSet o określonej pojemności i z określonym współczynnikiem zapełnienia (liczba od 0,0 do 1,0 określa poziom zapełnienia zbioru, przy którym jest on reorganizowany na zbiór o większej pojemności).

java.lang.Object **1.0**

- **int hashCode()**

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

13.2.4. Zbiór TreeSet

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

```
SortedSet<String> sorter = new TreeSet<String>(); // Klasa TreeSet implementuje klasę
SortedSet.
sorter.add("Bartek");
sorter.add("Ania");
sorter.add("Karol");
for (String s : sorter) System.println(s);
```

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

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

W związku z tym dodawanie elementów do zbioru TreeSet odbywa się nieco wolniej niż do zbioru HashSet (tabela 13.3 przedstawia dane porównawcze) ale TreeSet automatycznie sortuje elementy.

Tabela 13.3. Dodawanie elementów do zbiorów HashSet i TreeSet

Dokument	Liczba wszystkich słów	Liczba unikatowych słów	HashSet	TreeSet
<i>Alice in Wonderland</i>	28 195	5909	5 sekund	7 sekund
<i>The Count of Monte Cristo</i>	46 630	37 545	75 sekund	98 sekund

java.util.TreeSet<E> **1.2**

■ `TreeSet()`

Tworzy pusty zbiór TreeSet.

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

Tworzy zbiór TreeSet i wstawia do niego wszystkie elementy z określonej kolekcji.

13.2.5. Porównywanie obiektów

Skąd zbiór TreeSet wie, w jaki sposób posortować elementy? Standardowo zakłada, że wstawiane elementy implementują interfejs Comparable, w którym znajduje się jedna metoda:

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

Instrukcja `a.compareTo(b)` musi zwrócić wartość 0, jeśli obiekty a i b są równe, całkowitą liczbę ujemną, jeśli a znajduje się przed b, lub dodatnią liczbę całkowitą, jeśli a występuje za b. Sama wartość nie ma znaczenia, liczy się tylko jej znak (>0 , 0 lub <0). Interfejs Comparable jest implementowany przez kilka standardowych klas biblioteki Javy. Jedną z nich jest klasa String. Jej metoda `compareTo` porównuje łańcuchy według **porządku leksyko-graficznego**.

Aby móc wstawiać własne obiekty, należy we własnym zakresie zdefiniować porządek sortowania, implementując interfejs Comparable. W klasie Object nie istnieje domyślna implementacja metody `compareTo`.

Poniższa procedura na przykład sortuje obiekty typu Item według numeru części:

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    ...
}
```

Porównywanie dwóch całkowitych liczb **dodatnich**, takich jak numery seryjne, można zaimplementować, wykorzystując ich różnicę. Jeśli różnica jest ujemna, pierwszy element powinien się znajdować przed drugim, zero oznacza, że elementy są identyczne, a liczba dodatnia pozostałe możliwości.



Sztuczka ta działa wyłącznie dla niezbyt dużych liczb. Jeśli x jest bardzo dużą liczbą dodatnią, a y jest bardzo dużą liczbą ujemną, różnica x-y może przekroczyć zakres typu.

Niestety zastosowanie interfejsu Comparable do definicji kolejności sortowania jest obwarcione ograniczeniami. Jedna klasa może implementować go tylko jeden raz. Co w takim razie zrobić, jeśli w jednej kolekcji elementy muszą być posortowane według numerów seryjnych, a w innej według opisów? Dodatkową trudnością są obiekty klas, których twórcy nie zadali sobie trudu implementacji interfejsu Comparable.

W tego rodzaju sytuacjach należy zmusić zbiór TreeSet do użycia różnych metod porównujących, przekazując obiekt Comparator do konstruktora TreeSet. Interfejs Comparator zawiera metodę `compare` z dwoma parametrami jawnymi:

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

Metoda `compare`, podobnie jak `compareTo`, zwraca ujemną liczbę całkowitą, jeśli `a` występuje przed `b`, zero, jeśli `a` i `b` są identyczne, lub dodatnią liczbę całkowitą w pozostałych przypadkach.

Aby posortować elementy według ich opisów, wystarczy zdefiniować klasę implementującą interfejs `Comparable`:

```
class ItemComparator implements Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
}
```

Następnie obiekt tej klasy należy przekazać do konstruktora zbioru `TreeSet`:

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

Elementy drzewa utworzonego przy użyciu komparatora (obiektu typu `Comparator`) są porównywane za pomocą tego komparatora.

Należy zauważyć, że komparator ten nie zawiera żadnych danych, a jedynie metodę porównującą. Tego typu obiekty czasami nazywane są **obiektami funkcyjnymi**.

Obiekty funkcyjne są często definiowane w locie jako egzemplarze anonimowych klas wewnętrznych:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {
            String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
        }
    });
}
```



W rzeczywistości interfejs `Comparator<T>` zawiera dwie metody: `compare` i `equals`. Oczywiście każda klasa udostępnia metodę `equals`, przez co wydaje się, że dodanie jej do tego interfejsu nie daje wielkiego pozytku. Dokumentacja API wyjaśnia, że nie trzeba przesłaniać tej metody, ale w niektórych przypadkach zrobienie tego może spowodować zwiększenie jej szybkości działania. Na przykład metoda `addAll` z klasy `TreeSet` może być bardziej efektywna, jeśli dodamy elementy z innego zbioru, który używa tego samego komparatora.

Biorąc pod uwagę liczby przedstawione w tabeli 13.3, można się zastanawiać, czy nie lepiej byłoby zawsze używać zbiorów TreeSet zamiast HashSet. Nie da się ukryć, że wstawianie elementów do tego pierwszego nie zajmuje wiele więcej czasu, a przy okazji obiekty są automatycznie sortowane. Konkretna decyzja zależy od danych, które mają być przechowywane. Jeśli nie muszą one być uporządkowane, nie ma sensu marnować czasu na ich sortowanie. Co więcej, w niektórych przypadkach łatwiej jest napisać funkcję mieszącą niż sortującą. Funkcja mieszącą musi tylko dobrze mieszać obiekty, natomiast funkcja porównująca musi je precyjnie rozróżniać.

Przeanalizujmy na przykład przypadek zbioru prostokątów. Jeśli użyjemy zbioru TreeSet, musimy dostarczyć obiekt Comparator<Rectangle>. Powstaje pytanie, jakie kryteria zastosować do porównywania tych figur. Można na przykład porównywać pola powierzchni, ale to nie jest dobre rozwiązań, ponieważ dwa inaczej wyglądające prostokąty o innych współrzędnych mogą mieć takie samo pole. Zbiór TreeSet musi być zorganizowany w **porządku liniowym**. Musi istnieć możliwość porównania dowolnych dwóch elementów zbioru, a wymikiem porównywania, jeśli obiekty są równe, musi być zero. Istnieje porządek, który nadaje się do zastosowania z prostokątami (porządek leksykograficzny dla współrzędnych), ale jest on nienaturalny i sprawia trudności obliczeniowe. Funkcja miesząca jest już natomiast zdefiniowana w klasie Rectangle. Jej działanie polega na mieszaniu współrzędnych.



Od Java SE 6.0 klasa TreeSet implementuje interfejs NavigableSet. Znajduje się w nim kilka bardzo przydatnych metod służących do lokalizowania elementów i przemierzania zbiorów wstecz. Szczegółowe informacje na ten temat znajdują się w wyciągach z API.

Program przedstawiony na listingu 13.3 tworzy dwa zbiory TreeSet zapelnione obiektami Item. Pierwszy z nich jest sortowany według numerów części, czyli w domyślny sposób. Drugi natomiast posortowano według opisów za pomocą niestandardowego komparatora. Na listingu 13.4 przedstawiona jest klasa Item.

Listing 13.3. treeSet/TreeSetTest.java

```
package treeSet;

/**
 * @version 1.12 2012-01-26
 * @author Cay Horstmann
 */

import java.util.*;

/**
 * Program sortujący zbiór elementów poprzez porównanie ich opisów
 */
public class TreeSetTest
{
    public static void main(String[] args)
    {
        SortedSet<Item> parts = new TreeSet<>();
        parts.add(new Item("Toster", 1234));
        parts.add(new Item("Widget", 4562));
    }
}
```

```
parts.add(new Item("Modem", 9912));
System.out.println(parts);

SortedSet<Item> sortByDescription = new TreeSet<>(new
    Comparator<Item>()
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
});

sortByDescription.addAll(parts);
System.out.println(sortByDescription);
}
}
```

Listing 13.4. *treeSet/Item.java*

```
package treeSet;

import java.util.*;

/**
 * Element z opisem i numerem części
 */
public class Item implements Comparable<Item>
{
    private String description;
    private int partNumber;

    /**
     * Tworzy element
     *
     * @param aDescription
     *      opis elementu
     * @param aPartNumber
     *      numer części elementu
     */
    public Item(String aDescription, int aPartNumber)
    {
        description = aDescription;
        partNumber = aPartNumber;
    }

    /**
     * Pobiera opis elementu
     *
     * @return opis
     */
    public String getDescription()
    {
        return description;
    }
}
```

```

public String toString()
{
    return "[description=" + description + ". partNumber=" + partNumber + "]";
}

public boolean equals(Object otherObject)
{
    if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    Item other = (Item) otherObject;
    return Objects.equals(description, other.description) && partNumber ==
    ↪other.partNumber;
}

public int hashCode()
{
    return Objects.hash(description, partNumber);
}

public int compareTo(Item other)
{
    return Integer.compare(partNumber, other.partNumber);
}
}

```

java.lang.Comparable<T> 1.2

- int compareTo(T other)

Porównuje dwa obiekty i zwraca wartość ujemną, jeśli pierwszy z nich znajduje się przed drugim, zero, jeśli są identyczne, lub wartość dodatnią, jeśli pierwszy z nich występuje za drugim.

java.util.Comparator<T> 1.2

- int compare(T a, T b)

Porównuje dwa obiekty i zwraca wartość ujemną, jeśli a występuje przed b, zero, jeśli obiekty są identyczne, lub wartość dodatnią, jeśli a występuje za b.

java.util.SortedSet<E> 1.2

- Comparator<? super E> comparator()

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

- E first()
- E last()

Zwraca najmniejszy lub największy element posortowanego zbioru.

java.util.NavigableSet<E> 6

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

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

- E ceiling(E value)
- E floor (E value)

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

- E pollFirst()
- E pollLast()

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

- Iterator<E> descendingIterator()

Zwraca iterator, który przemierza zbiór w kolejności malejącej.

java.util.TreeSet<E> 1.2

- TreeSet()

Tworzy zbiór TreeSet do przechowywania obiektów implementujących interfejs Comparable.

- TreeSet(Comparator<? super E> c)

Tworzy zbiór TreeSet i sortuje jego elementy przy użyciu określonego komparatora.

- TreeSet(SortedSet<? extends E> elements)

Tworzy zbiór TreeSet, wstawia do niego wszystkie elementy z posortowanego zbioru oraz wykorzystuje ten sam komparator co określony zbiór uporządkowany.

13.2.6. Kolejki Queue i Deque

Z wcześniejszych podrozdziałów wiemy już, że kolejka jest strukturą danych pozwalającą na szybkie dodawanie elementów na końcu i usuwanie elementów z czoła. Kolejka dwukierunkowa (**deque**) pozwala na szybkie dodawanie i usuwanie elementów po obu stronach. Nie można natomiast dodawać elementów w środku kolejki. W Java SE 6 zaprezentowano nowy interfejs o nazwie Deque. Implementują go klasy ArrayDeque i LinkedList. Służą one do tworzenia kolejek dwukierunkowych, które mogą zmieniać swój rozmiar w zależności od zapotrzebowania. W rozdziale 14. piszemy jeszcze o kolejkach Queue i Deque z ograniczeniami.

java.util.Queue<E> 5.0

- boolean add(E element)
- boolean offer(E element)

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

- E remove()
- E poll()

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

- E element()
- E peek()

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

java.util.Deque<E> 6

- void addFirst(E element)
- void addLast(E element)
- boolean offerFirst(E element)
- boolean offerLast(E element)

Wstawia element na czoło lub ogon kolejki dwukierunkowej. Jeśli kolejka jest pełna, dwie pierwsze metody zgłaszą wyjątek IllegalStateException, a pozostałe dwie zwracają wartość false.

- E removeFirst()
- E removeLast()
- E pollFirst()
- E pollLast()

Usuwa i zwraca element znajdujący się na czole kolejki, jeśli nie jest ona pusta. Jeśli kolejka jest pusta, pierwsze dwie z powyższych metod zgłaszą wyjątek NoSuchElementException, a pozostałe dwie zwracają wartość null.

- E getFirst()
- E getLast()
- E peekFirst()

- `E peekLast()`

Zwraca element znajdujący się na czole kolejki (ale go nie usuwa), pod warunkiem że kolejka nie jest pusta. Jeśli kolejka jest pusta, pierwsze dwie z powyższych metod zgłaszały wyjątek `NoSuchElementException`, a pozostałe zwracają wartość `null`.

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

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

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

13.2.7. Kolejki priorytetowe

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

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

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

Listing 13.5 demonstruje działanie kolejki priorytetowej. W przeciwieństwie do zbioru `TreeSet`, tutaj iterator nie odwiedza elementów w uporządkowanej kolejności. Natomiast usuwany jest zawsze najmniejszy dostępny element.

Listing 13.5. `priorityQueue/PriorityQueueTest.java`

```
package priorityQueue;

import java.util.*;

/**
 * Program demonstrujący zastosowanie kolejki priorytetowej
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
```

```

public class PriorityQueueTest
{
    public static void main(String[] args)
    {
        PriorityQueue<GregorianCalendar> pq = new PriorityQueue<>();
        pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
        pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
        pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
        pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse

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

```

java.util.PriorityQueue **5.0**

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`
Tworzy kolejkę priorytetową do przechowywania obiektów implementujących interfejs Comparable.
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`
Tworzy kolejkę priorytetową i wykorzystuje do jej sortowania określony komparator.

13.2.8. Mapy

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

W bibliotece Javy znajdują się dwie implementacje map ogólnego przeznaczenia: `HashMap` i `TreeMap`. Obie te klasy implementują interfejs `Map`.

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

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

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

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

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

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

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

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

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

Do usuwania elementów z określonym kluczem z mapy służy metoda `remove`. Natomiast metoda `size` zwraca liczbę elementów znajdujących się w mapie.

W architekturze kolekcji mapa nie jest uznawana za kolekcję (inne architektury struktur danych traktują mapę jako kolekcję **par** lub kolekcję wartości indeksowaną za pomocą kluczów). Istnieje jednak możliwość utworzenia **widoku** mapy w postaci obiektów implementujących interfejs `Collection` lub jeden z jego podinterfejsów.

Istnieją trzy rodzaje widoków: zbiór kluczów, kolekcja wartości (która nie jest zbiorem) i zbiór par klucz – wartość. Klucze i pary klucz – wartość tworzą zbiory, ponieważ mapa może zawierać tylko jeden egzemplarz każdego klucza. Poniższe metody zwracają wymienione widoki (elementy zbioru `entrySet` są obiektami statycznej klasy wewnętrznej `Map.Entry`):

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

Należy zauważyć, że `keySet` nie jest zbiorem typu `HashSet` ani `TreeSet`, tylko obiektem jeszcze innej klasy implementującej interfejs `Set`. Interfejs `Set` rozszerza interfejs `Collection`, dzięki czemu zbioru `keySet` można używać w taki sam sposób jak każdej innej kolekcji.

Można na przykład sporządzić listę wszystkich kluczów znajdujących się w mapie:

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



Chcąc uzyskać widok kluczy i wartości, można uniknąć wyszukiwania wartości poprzez sporządzenie **listy pozycji** (ang. *entry*). Można wykorzystać poniższy kod ramowy:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String key = entry.getKey();
    Employee value = entry.getValue();
    działania key i value
}
```

Metoda `remove` iteratora usuwa z mapy klucz i **skojarzoną z nim wartość**. Nie można natomiast **dodać** elementu do widoku `keySet`, ponieważ dodanie klucza bez wartości jest bezcelowe. Próba wywołania metody `add` zakończy się zgłoszeniem wyjątku `UnsupportedOperationException`. Widok `entrySet` jest ograniczony w podobny sposób, mimo że dodanie nowej pary klucz – wartość z teoretycznego punktu widzenia nie byłoby pozbaione sensu.

Listing 13.6 demonstruje praktyczne zastosowanie mapy. Na początku dodawane są pary klucz – wartość do mapy. Następnie program usuwa jeden klucz z mapy, co pociąga za sobą usunięcie skojarzonej z nią wartości. Dalej zostaje zmodyfikowana wartość skojarzona z pewnym kluczem i następuje wywołanie metody `get` wyszukującej wartość. Na zakończenie program iteruje przez zbiór `entrySet`.

Listing 13.6. map/MapTest.java

```
package map;

import java.util.*;

/**
 * Program demonstrujący użycie mapy z kluczami typu String i wartościami typu Employee
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class MapTest
{
    public static void main(String[] args)
    {
        Map<String, Employee> staff = new HashMap<>();
        staff.put("144-25-5464", new Employee("Anna Kowalska"));
        staff.put("567-24-2546", new Employee("Henryk Kwiatek"));
        staff.put("157-62-7935", new Employee("Marcin Nowak"));
        staff.put("456-62-5527", new Employee("Franciszek Frankowski"));

        // wydruk wszystkich pozycji
        System.out.println(staff);

        // usunięcie wartości
        staff.remove("567-24-2546");

        // podmienienie pozycji
        staff.put("456-62-5527", new Employee("Weronika Kowalska"));
    }
}
```

```

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

// iteracja przez wszystkie pozycje

for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String key = entry.getKey();
    Employee value = entry.getValue();
    System.out.println("key=" + key + ", value=" + value);
}
}
}

```

java.util.Map<K, V> 1.2

■ `V get(K key)`

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

■ `V put(K key, V value)`

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

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

Wstawia do mapy wszystkie pozycje z określonej mapy.

■ `boolean containsKey(Object key)`

Zwraca wartość true, jeśli klucz znajduje się w mapie.

■ `boolean containsValue(Object value)`

Zwraca wartość true, jeśli wartość znajduje się w mapie.

■ `Set<Map.Entry<K, V>> entrySet()`

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

■ `Set<K> keySet()`

Zwraca widok zbiorowy wszystkich kluczy znajdujących się w mapie. Jeśli jakiś klucz z tego zbioru zostanie usunięty, z mapy zostanie usunięty ten sam klucz i skojarzona z nim wartość. Nie można nic dodawać.

■ `Collection<V> values()`

Zwraca widok kolekcyjny wszystkich wartości znajdujących się w mapie. Jeśli jakaś wartość z tego zbioru zostanie usunięta, z mapy zostanie usunięta ta sama wartość i skojarzony z nią klucz. Nie można nic dodawać.

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

- K getKey()
- V getValue()

Zwraca klucz lub wartość określonej pozycji.

- V setValue(V newValue)

Zamienia wartość na nową wartość i zwraca starą wartość.

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

- HashMap()
- HashMap(int initialCapacity)
- HashMap(int initialCapacity, float loadFactor)

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

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

- TreeMap(Comparator<? super K> c)
- Tworzy strukturę danych TreeMap i sortuje jej klucze za pomocą określonego komparatora.
- TreeMap(Map<? extends K, ? extends V> entries)
- Tworzy strukturę danych TreeMap i dodaje do niej wszystkie elementy z innej mapy.
- TreeMap(SortedMap<? extends K, ? extends V> entries)
- Tworzy strukturę danych TreeMap, dodaje do niej wszystkie pozycje z posortowanej mapy oraz wykorzystuje ten sam komparator co określona posortowana mapa.

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

- Comparator<? super K> comparator()

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

- K firstKey()
- K lastKey()

Zwraca najmniejszy lub największy klucz dostępny w mapie.

13.2.9. Specjalne klasy Set i Map

W bibliotece kolekcji znajduje się kilka klas map, które mają specjalne przeznaczenie. Omawiamy je krótko w kilku kolejnych podrozdziałach.

13.2.9.1. Klasa WeakHashMap

Klasa WeakHashMap rozwiązuje pewien bardzo interesujący problem. Spróbujmy sobie wyobrazić, co się dzieje z wartością, której klucz nie jest już używany nigdzie w programie. Założymy, że ostatnia referencja do tego klucza została utracona. W takiej sytuacji nie ma już żadnego sposobu dostania się do skojarzonego z tym kluczem obiektu. Ponieważ klucza już nigdzie nie ma, nie można usunąć z mapy pary klucz – wartość. Nasuwa się myśl, że powinien się zająć tym system zbierania nieużytków.

Niestety nie jest to takie proste. System ten przechowuje informacje o aktywnych obiektach. Dopóki mapa jest aktywna, wszystkie jej komórki również są aktywne, przez co nie można ich usunąć. Dlatego należy pamiętać, aby usuwać z programu nieużywane wartości z dugo wykorzystywanych map. Można też wybrać inne rozwiązanie i użyć klasy WeakHashMap. Struktura ta współpracuje z systemem zbierania nieużytków w zakresie usuwania par klucz – wartość, jeśli jedynie odwołanie do klucza pochodzi z pozycji w tablicy.

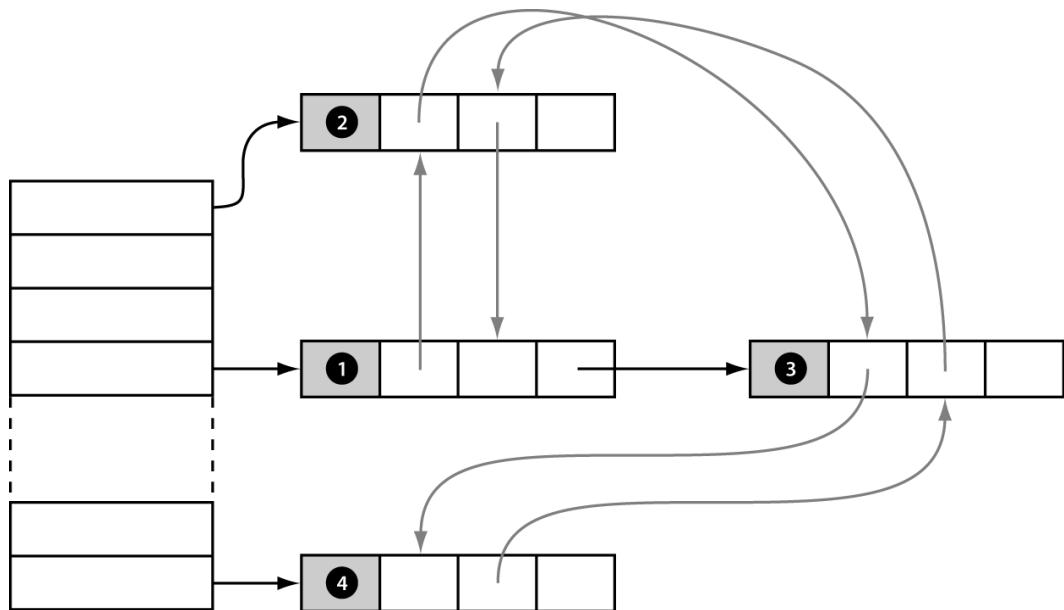
Oto zasada działania tego mechanizmu. Klucze w WeakHashMap są przechowywane w tak zwanych **słabych referencjach** (ang. *weak reference*). Obiekt typu WeakReference przechowuje referencję do innego obiektu, w tym przypadku do klucza w tablicy. Obiekty tego typu są traktowane przez system usuwania nieużytków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

13.2.9.2. Klasa LinkedHashSet i LinkedHashMap

W Java SE 1.5 wprowadzono klasy LinkedHashSet i LinkedHashMap, które pamiętają kolejność wstawiania elementów. Tym samym rozwiązują problem losowej kolejności przechowywanych elementów. Elementy wstawiane do tablicy są łączone w listy dwustronne (zobacz rysunek 13.9).

Przeanalizujmy na przykład poniższe instrukcje wstawiające elementy do mapy (pochodzą one z listingu 13.6):

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



Rysunek 13.9. Powiązana tablica mieszająca

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

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

Natomiast instrukcja `staff.values().iterator()` tworzy następującą listę odpowiadających im wartości:

Anna Kowalska
 Henryk Kwiątek
 Marcin Nowak
 Franciszek Frankowski

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

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

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

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

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

Dzięki temu dodanie nowego elementu spowoduje usunięcie najstarszego elementu (`eldest`), pod warunkiem że wywołana metoda zwróci wartość `true`. Poniższa przykładowa pamięć podręczna przechowuje maksymalnie sto elementów:

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

Istnieje też możliwość podjęcia na jakiejś podstawie decyzji o usunięciu najstarszego elementu. Można na przykład sprawdzić jego znacznik czasu.

13.2.9.3. Klasa `EnumSet` i `EnumMap`

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

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

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

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

`EnumMap` to mapa, w której klucze są typu wyliczeniowego. Jest ona prostą i wydajną tablicą wartości. Typ klucza należy określić w konstruktorze:

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



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

13.2.9.4. Klasa IdentityHashMap

W Java SE 1.4 wprowadzono także klasę `IdentityHashMap`, która również jest przeznaczona do specjalnych celów. Wartości haszowe kluczy nie powinny być w niej obliczane przez metodę `hashCode`, ale przez `System.identityHashCode`. Metody tej używa metoda `Object.hashCode` do obliczania kodu mieszącego z adresu obiektu w pamięci. Ponadto klasa ta porównuje obiekty za pomocą operatora `==` zamiast metody `equals`.

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

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

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

Tworzy pustą mapę `WeakHashMap` o określonej pojemności i określonym współczynniku zapełnienia.

`java.util.LinkedHashSet<E>` **1.4**

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

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

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

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

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

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

Tę metodę należy przedefiniować, aby zwracała wartość `true`, jeśli najstarsza pozycja ma być usuwana. Parametr `eldest` określa pozycję, której usunięcie jest rozważane. Metoda ta jest wywoływana po dodaniu pozycji do mapy. Domyślana implementacja zwraca wartość `false` — stare elementy nie są domyślnie usuwane.

Można jednak ją przedefiniować, aby selektywnie zwracała wartość true, na przykład gdy najstarsza pozycja spełnia określone warunki lub mapa przekracza określony rozmiar.

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

- static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)
Zwraca zbiór zapełniony wartościami z określonego typu wyliczeniowego.
- static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)
Zwraca pusty zbiór pozwalający przechowywać wartości określonego typu wyliczeniowego.
- static <E extends Enum<E>> EnumSet<E> range(E from, E to)
Zwraca zbiór zapełniony wartościami z zakresu od from do to (włącznie).
- static <E extends Enum<E>> EnumSet<E> of(E value)
- static <E extends Enum<E>> EnumSet<E> of(E value, E... values)
Zwraca zbiór zawierający określone wartości.

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

- EnumMap(Class<K> keyType)
Tworzy mapę, której klucze są określonego typu.

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

- IdentityHashMap()
- IdentityHashMap(int expectedMaxSize)
Tworzy pustą mapę IdentityHashMap o pojemności równej najmniejszej potędze cyfry 2 większej niż $1.5 * \text{expectedMaxSize}$ (domyślna wartość parametru expectedMaxSize to 21).

java.lang.System **1.0**

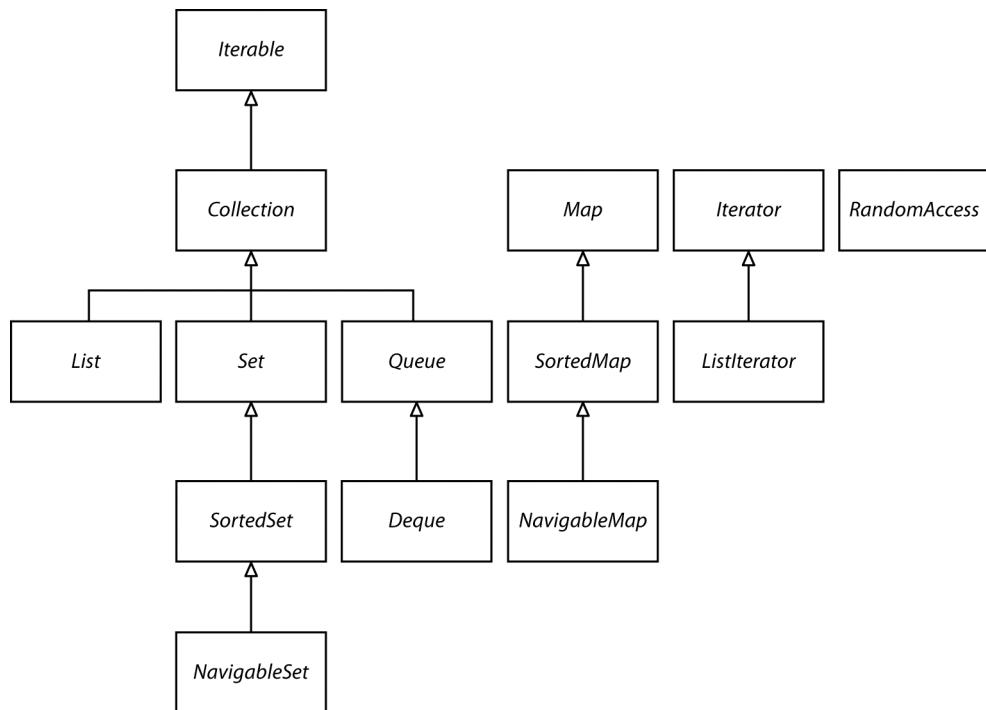
- static int identityHashCode(Object obj) **1.1**
Zwraca ten sam kod mieszający (obliczony na podstawie adresu w pamięci obiektu) co metoda Object.hashCode, nawet jeśli w klasie, do której należy obiekt obj, przedefiniowano metodę hashCode.

13.3. Architektura kolekcji

Architektura (ang. *framework*) to zbiór klas stanowiących podstawę do budowy bardziej zaawansowanych funkcji. Architektura zawiera nadklasy charakteryzujące się określona funkcjonalnością, kierujące pewnymi zasadami oraz udostępniające różne przydatne mechanizmy.

Użytkownik architektury rozszerza jej klasy, dzięki czemu nie musi opracowywać niektórych podstawowych mechanizmów od nowa. Na przykład Swing jest architekturą interfejsów użytkownika.

Biblioteka kolekcji w Javie stanowi architekturę klas kolekcyjnych. Zawiera definicje kilku interfejsów i klas abstrakcyjnych przeznaczonych do użytku przez twórców kolekcji (zobacz rysunek 13.10) oraz udostępnia pewne mechanizmy, jak na przykład protokół iteracyjny. Aby używać klas kolekcyjnych, nie trzeba posiadać wiedzy na temat ich architektury — udowodniliśmy to w poprzednich podrozdziałach. Aby jednak tworzyć algorytmy generyczne działające na wielu typach kolekcji lub całkiem nowe typy kolekcji, trzeba znać budowę tej architektury.



Rysunek 13.10. Interfejsy architektury kolekcji

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

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

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

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

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

```
V get(K key)
```

Lista to **kolekcja uporządkowana**. Elementy są dodawane w określonych miejscach zbiornika. Obiekt można wstawić w odpowiednie miejsce na dwa sposoby: według indeksu całkowito-liczbowego lub przez iterator listy. W interfejsie `List` znajdują się metody dające dostęp do dowolnego elementu kolekcji:

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

Jak było już wcześniej wspominane, interfejs `List` udostępnia te metody bez względu na to, czy są one wydajne w określonej implementacji, czy nie. Aby pozwolić na uniknięcie powolnych operacji dostępu swobodnego, w Java SE 1.4 wprowadzono interfejs o nazwie `RandomAccess`. Nie posiada on żadnych metod, ale można za jego pomocą sprawdzić, czy określona kolekcja umożliwia szybki dostęp swobodny do elementów:

```
if (c instanceof RandomAccess)
{
    algorytm dostępu losowego
}
else
{
    algorytm dostępu liniowego
}
```

Interfejs `RandomAccess` implementują klasy `ArrayList` i `Vector`.



Teoretycznie można by było utworzyć osobny interfejs o nazwie `Array`, który rozszerzałby interfejs `List` i deklarował metody dostępu swobodnego. Gdyby istniał taki osobny interfejs, algorytmy wymagające dostępu swobodnego używałyby parametrów typu `Array` i nie można by było zastosować ich do kolekcji o powolnym dostępie swobodnym. Jednak projektanci architektury kolekcji zdecydowali się nie definiować osobnego interfejsu, ponieważ chcieli, aby liczba interfejsów była jak najmniejsza. Nie chciano też traktować protekcyjnie programistów. Programista może przekazać listę powiązaną do algorytmu implementującego dostęp swobodny — musi tylko mieć świadomość kosztów wydajnościowych tego działania.

W interfejsie `ListIterator` znajduje się metoda wstawiająca element przed miejscem, w którym znajduje się iterator:

```
void add(E element)
```

Do pobierania i usuwania elementów znajdujących się w określonych miejscach służą metody `next` i `remove` z interfejsu `Iterator`.

Interfejs `Set` jest identyczny z interfejsem `Collection`, ale jego metody są nieco bardziej restrykcyjne. Metoda `add` zbioru nie powinna dodawać duplikatów. Metoda `equals` powinna być zdefiniowana w taki sposób, aby dwa zbiory były identyczne, jeśli mają takie same elementy, ale niekoniecznie w takiej samej kolejności. Metoda `hashCode` dla dwóch zbiorów zawierających takie same elementy powinna zwracać ten sam kod mieszający.

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

Interfejsy `SortedSet` i `SortedMap` udostępniają obiekt komparatora użyty do sortowania oraz definiują metody tworzące widoki podzbiorów kolekcji. Widoki te opisujemy w kolejnym podrozdziale.

Wreszcie, w Java SE 6 wprowadzono interfejsy `NavigableSet` i `NavigableMap`, które zawierają dodatkowe metody służące do przemierzania i przeszukiwania uporządkowanych zbiorów i map (w idealnej sytuacji metody te powinny się znajdować w interfejsach `SortedSet` i `SortedMap`). Interfejsy te są implementowane przez klasy `TreeSet` i `TreeMap`.

Kolej na klasy implementujące wymienione interfejsy. Wiemy już, że niektóre metody interfejsów kolekcyjnych można z łatwością zaimplementować na bazie bardziej podstawowych metod. Wiele z tych implementacji znajduje się w klasach abstrakcyjnych:

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

Klasy te można rozszerzać przy tworzeniu własnych klas kolekcyjnych, dzięki czemu dziedziczy się po nich wiele rutynowych procedur.

Konkretnie klasy dostępne w bibliotece Javy to:

```
LinkedList
ArrayList
ArrayDeque
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

Rysunek 13.11 przedstawia relacje zachodzące między tymi klasami.

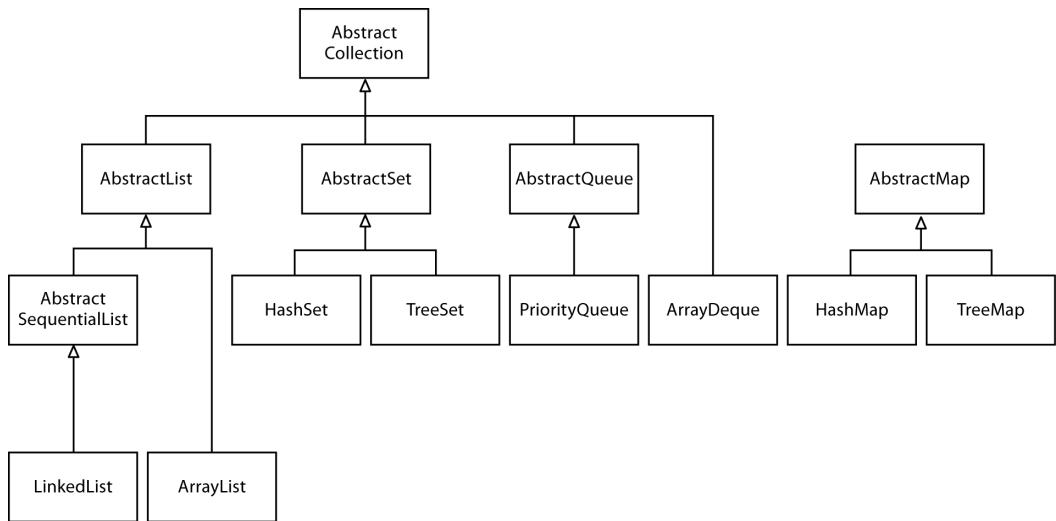
Na koniec należy jeszcze wymienić kilka starszych klas kontenerowych, które są dostępne w Javie od początku, zanim jeszcze powstała architektura kolekcji:

```
Vector
Stack
Hashtable
Properties
```

Zostały one wcielone do kolekcji — rysunek 13.12. Omawiamy te klasy nieco dalej.

13.3.1. Widoki i obiekty opakowujące

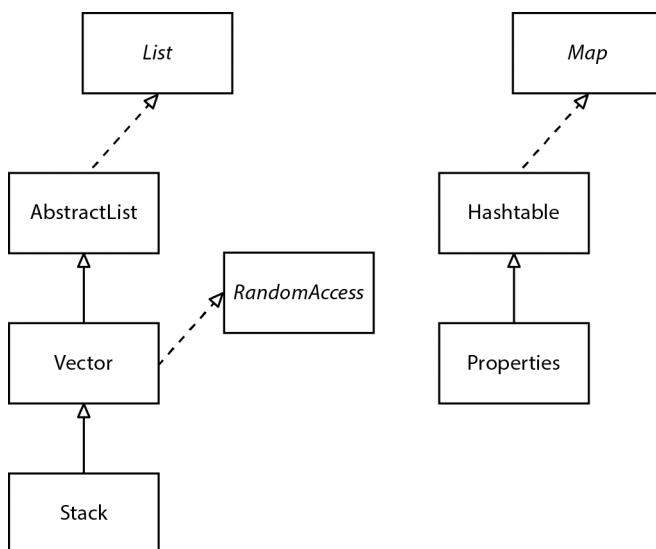
Z rysunków 13.10 i 13.11 można wyciągnąć wniosek, że projektanci Javy wpadli w lekką przesadę, tworząc tak wiele interfejsów i klas abstrakcyjnych do implementacji raczej umiarowanej liczby konkretnych klas kolekcyjnych. Jednak rysunki te nie pokazują wszystkiego. interfejsy `Collection` i `Map`. Przykład takiego działania zaprezentowaliśmy, kiedy użyliśmy



Rysunek 13.11. Klasy w architekturze kolekcji

Rysunek 13.12.

Starsze klasy
w architekturze
kolekcji



Za pomocą tak zwanych **widoków** (ang. *view*) można tworzyć inne obiekty implementujące metody `keySet` mapy. Na pierwszy rzut oka wydaje się, że metoda ta tworzy nowy zbiór, zapełnia go wszystkimi kluczami z mapy i zwraca go. Nie jest to jednak prawda. Metoda `keySet` zwraca obiekt klasy implementującej interfejs `Set`, którego metody operują na oryginalnej mapie. Taka kolekcja nazywana jest **widokiem**.

Widoki mają kilka ważnych zastosowań w architekturze kolekcji. Opisujemy je w poniższych podrozdziałach.

13.3.1.1. Lekkie obiekty opakowujące kolekcje

Metoda `asList` z klasy `Arrays` zwraca obiekt osłonowy typu `List` opakowujący czystą tablicę. Umożliwia ona przekazywanie tablic do metod, które na wejściu przyjmują tylko listy i kolekcje. Na przykład:

```
Card[] cardDeck = new Card[52];
List<Card> cardList = Arrays.asList(cardDeck);
```

Zwrócony obiekt **nie** jest typu `ArrayList`. Jest to obiekt widokowy udostępniający metody `get` i `set`, które operują na leżącej u podłożu tablicy. Wszystkie metody, które mogą zmienić rozmiar tej tablicy (na przykład metody `add` i `remove` dołączonego iteratatora), zgłaszają wyjątek `UnsupportedOperationException`.

Od Java SE 5.0 metoda `asList` może przyjmować różne zestawy argumentów. Zamiast tablicy można jej przekazać poszczególne elementy. Na przykład:

```
List<String> names = Arrays.asList("Ania", "Bartek", "Karol");
```

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

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

Na przykład poniższa instrukcja tworzy listę zawierającą 100 łańcuchów `DEFAULT`:

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

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



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

Metoda wywołana poniżej zwraca obiekt widokowy implementujący interfejs `Set` (w odróżnieniu od metody `nCopies`, która tworzy listę). Zwrócony obiekt implementuje niemodyfikowalny jednoelementowy zbiór pozbawiony narzutu struktury danych. Metody `singletonList` i `singletonMap` mają podobne działanie.

```
Collections.singleton(anObject)
```

13.3.1.2. Widoki przedziałowe

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

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

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

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

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

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

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

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

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

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

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

Wprowadzony w Java SE 6 interfejs NavigableSet daje większe możliwości kontroli działań na przedziałach. Można określić, czy wartości graniczne mają być wliczane (ang. *inclusive*):

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

13.3.1.3. Widoki niemodyfikowalne

Klasa Collections udostępnia metody tworzące **niemodyfikowalne widoki** kolekcji. Widoki te sprawdzają, czy istniejąca kolekcja nie jest modyfikowalna. Jeśli wykryją próbę modyfikacji kolekcji, wyrzucają wyjątek, a kolekcja pozostaje nietknięta.

Widoki niemodyfikowalne tworzy sześć metod:

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

Każda z tych metod działa w kooperacji z jakimś interfejsem. Na przykład metoda `Collections.unmodifiableList` współpracuje z klasami `ArrayList`, `LinkedList` i wszystkimi innymi, które implementują interfejs `List`.

Wyobraźmy sobie, że chcemy zezwolić pewnej procedurze pobrać zawartość jakiejś kolekcji, ale nie chcemy, by ją modyfikowała. Oto, co można w takiej sytuacji zrobić:

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

Metoda `Collections.unmodifiableList` zwraca obiekt klasy implementującej interfejs `List`. Jego metody dostępowe pobierają wartości z kolekcji `staff`. Oczywiście metoda `lookAt` może wywoływać wszystkie metody interfejsu `List`, nie tylko metody dostępowe. Jednak wszystkie metody modyfikujące (`set`) zostały przedefiniowane w taki sposób, aby zamiast przekazywać wywołania do kolekcji, zgłaszały wyjątek `UnsupportedOperationException`.

Niemodyfikowalny widok nie czyni samej kolekcji niemodyfikowaną. Nadal można ją modyfikować przy użyciu jej pierwotnej referencji (w naszym przypadku `staff`). Można też wywoływać metody modyfikujące na rzecz jej elementów.

Ponieważ widoki opakowują interfejs, a nie prawdziwy obiekt kolekcyjny, dostępne są tylko te metody, które zostały zdefiniowane w tym interfejsie. Na przykład klasa `LinkedList` zawiera metody `addFirst` i `addLast`, które nie należą do interfejsu `List`. Dlatego nie można ich używać w widoku niemodyfikowalnym.



Metoda `unmodifiableCollection` (a także `synchronizedCollection` i `checkedCollection`) zwraca kolekcję, której metoda `equals` nie wywołuje metody `equals` oryginalnej kolekcji. Zamiast tego dziedziczy metodę `equals` po klasie `Object`, która tylko sprawdza, czy obiekty są identyczne. Jeśli zamienisz zbiór lub listę w kolekcję, pozbawisz się możliwości porównywania zawartości. Widoki działają w ten sposób, ponieważ na tym poziomie hierarchii porównywanie nie jest dobrze zdefiniowane. W ten sam sposób traktowana jest metoda `hashCode`.

Jednak metody `unmodifiableSet` i `unmodifiableList` używają metod `equals` i `hashCode` oryginalnej kolekcji.

13.3.1.4. Widoki synchronizowane

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

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

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

Teraz można dostarczać się do obiektu `map` z różnych wątków. Metody takie jak `get` i `put` są synchronizowane, to znaczy każde wywołanie metody musi zostać w pełni ukończone, zanim inny wątek będzie mógł wywołać kolejną metodę. Kwestię synchronizowanego dostępu do struktur danych szczegółowo omawiamy w rozdziale 14.

13.3.1.5. Widoki kontrolowane

W Java SE 5.0 wprowadzono zestaw widoków kontrolowanych mających na celu wspieranie programisty w procesie usuwania błędów związanych z typami uogólnionymi. Wiemy już z rozdziału 12., że do uogólnionej kolekcji da się przemycić elementy złego typu. Na przykład:

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

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

Widok kontrolowany wykryje taki błąd. Zdefiniujmy następującą bezpieczną listę:

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

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

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



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

13.3.1.6. Uwagi dotyczące operacji opcjonalnych

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

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

Czy powinno się rozszerzać technikę opcjonalnych operacji na własne projekty? Naszym zdaniem nie. Mimo iż kolekcje są używane często, styl programowania związany z ich implementacją nie jest typowy dla innych dziedzin. Projektanci biblioteki kolekcyjnej stoją przed

niezwykle trudnym zadaniem zaspokojenia sprzecznych wymagań. Z jednej strony użytkownicy wymagają, aby biblioteka była łatwa do nauki, wygodna w użyciu, w pełni uogólniona i niepodatna na błędy, a z drugiej strony ma być tak samo wydajna jak ręcznie optymalizowane algorytmy. Spełnienie tych wszystkich żądań naraz (czy choćby zbliżenie się do nich) jest najzwyczajniej niemożliwe. Jednak we własnej praktyce programistycznej nie będziesz często rozwiązywać tak ekstremalnych problemów. Powinno być możliwe znalezienie rozwiązań, które nie polegają na wykorzystaniu ostatecznego środka w postaci opcjonalnych operacji interfejsu.

java.util.Collections 1.2

- static <E> Collection unmodifiableCollection(Collection<E> c)
- static <E> List unmodifiableList(List<E> c)
- static <E> Set unmodifiableSet(Set<E> c)
- static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)
- static <K, V> Map unmodifiableMap(Map<K, V> c)
- static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)

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

- static <E> Collection<E> synchronizedCollection(Collection<E> c)
- static <E> List synchronizedList(List<E> c)
- static <E> Set synchronizedSet(Set<E> c)
- static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
- static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
- static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)

Tworzy widoki kolekcji, których metody są zsynchronizowane.

- static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)
- static <E> List checkedList(List<E> c, Class<E> elementType)
- static <E> Set checkedSet(Set<E> c, Class<E> elementType)
- static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)
- static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)
- static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)

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

- static <E> List<E> nCopies(int n, E value)

- static <E> Set<E> singleton(E value)

Tworzy widok obiektu będący niemodyfikowalną listą zawierającą n identycznych elementów lub zbiór zawierający jeden element.

java.util.Arrays 1.2

- static <E> List<E> asList(E... array)

Zwraca widok listowy elementów tablicy, który można modyfikować, ale nie można zmieniać jego rozmiaru.

java.util.List<E> 1.2

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

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

java.util.SortedSet<E> 1.2

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

Zwraca widok elementów z określonego przedziału.

API java.util.NavigableSet<E> 6

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

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

java.util.SortedMap<K, V> 1.2

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

Zwraca widok mapy pozycji, których klucze mieścią się w określonym przedziale.

java.util.NavigableMap<K, V> 6

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

■ `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`

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

13.3.2. Operacje zbiorcze

W większości prezentowanych do tej pory przykładów kodu przemierzaliśmy kolekcje po jednym elemencie za pomocą iteratatora. Iteracji można jednak uniknąć, stosując w zamian jedną z operacji zbiorczych dostępnych w bibliotece.

Chcemy znaleźć część wspólną dwóch zbiorów. Zaczniemy od utworzenia nowego zbioru do przechowywania znalezionego wyniku.

```
Set<String> result = new HashSet<String>(a);
```

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

Teraz użyjemy metody `retainAll`:

```
result.retainAll(b);
```

Zwraca ona wszystkie elementy, które znajdują się także w zbiorze b. W ten sposób znaleźliśmy część wspólną dwóch zbiorów bez tworzenia pętli.

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

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

Wystarczy utworzyć zbiór kluczów i usunąć wszystkie identyfikatory zwolnionych pracowników.

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

Ponieważ zbiór kluczów jest widokiem mapy, klucze i skojarzeni z nimi pracownicy są automatycznie usuwani z tej mapy.

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

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

Na podprzedziale tym można także wykonywać operacje modyfikujące.

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

13.3.3. Konwersja pomiędzy kolekcjami a tablicami

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

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

```
String[] values = . . .;
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

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

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

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

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

Metoda `toArray` zwraca tablicę typu `Object[]`, którego nie można zmienić. W zamian należy użyć alternatywnej wersji tej metody. Przekazujemy do niej jako parametr tablicę o długości 0 takiego typu, jakiego potrzebujemy. Zwrócona tablica będzie wtedy miała **taki właśnie typ**:

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

W razie potrzeby można utworzyć tablicę o odpowiednim rozmiarze:

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

W tym przypadku nie jest tworzona żadna nowa tablica.



Możesz się zastanawiać, czemu do metody `toArray` nie można po prostu przekazać obiektu typu `Class` (jak na przykład `String.class`). Metoda ta wykonuje podwójną pracę — zapiełnia istniejącą tablicę (pod warunkiem że jest wystarczająco dłużą) i tworzy nową tablicę.

13.4. Algorytmy

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

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

Oczywiście w przypadku listy tablicowej kod wyglądałby nieco inaczej:

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

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

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

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

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

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

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

Teraz dysponujemy jedną metodą, która znajduje największy element w listach powiązanych, listach tablicowych i tablicach.

Technika ta daje bardzo duże możliwości. W bibliotece standardowej C++ znajduje się mnóstwo przydatnych algorytmów, z których każdy operuje na kolekcjach uogólnionych. Biblioteka Javy nie jest tak bogata, ale posiada podstawowe funkcje: sortowanie, wyszukiwanie binarne i kilka algorytmów użytkowych.

13.4.1. Sortowanie i tasowanie

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

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

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

Ta metoda zakłada, że elementy listy implementują interfejs `Comparable`. Aby posortować tę listę w inny sposób, można jako drugi parametr przekazać obiekt `Comparator` (komparatory omawialiśmy w podrozdziale 13.2.5, „Porównywanie obiektów”). Oto sposób sortowania listy elementów:

```
Comparator<Item> itemComparator = new
    Comparator<Item>()
{
    public int compare(Item a, Item b)
    {
        return a.partNumber - b.partNumber;
    }
};
Collections.sort(items, itemComparator);
```

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

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

Podobnie poniższa instrukcja odwraca kolejność metody `itemComparator`:

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

W tym miejscu może się rodzić pytanie, jak metoda `sort` sortuje listy. Typowe algorytmy sortujące prezentowane w książkach o algorytmach działają na tablicach i korzystają z dostępu swobodnego. W listach ten typ dostępu może być bardzo wolny. Można je jednak szybko sortować przy użyciu algorytmu sortowania przez scalanie (zobacz *Algorithms in C++* autorstwa Roberta Sedgewicka, Addison-Wesley, 1998, s. 336 – 369). W Javie jednak nie wykorzystano tej metody. W zamian wszystkie elementy listy są wrzucane do tablicy, tam sortowane przy użyciu innej wersji algorytmu sortowania przez scalanie, a następnie kopowane w uporządkowanej kolejności z powrotem do listy.

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

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

Właściwości list można przedstawić następująco:

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

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

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

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

Program przedstawiony na listingu 13.7 wstawia do listy tablicowej 49 obiektów typu `Integer` zawierających liczby od 1 do 49. Następnie tasuje zawartość tej listy i wybiera sześć pierwszych elementów. Na końcu sortuje pobrane wartości i drukuje je.

Listing 13.7. `shuffle/_shuffleTest.java`

```
package shuffle;

import java.util.*;

/**
 * Program demonstrujący algorytmy tasowania i sortowania
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class ShuffleTest
{
    public static void main(String[] args)
    {
        List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 49; i++)
            numbers.add(i);
```

```

        Collections.shuffle(numbers);
        List<Integer> winningCombination = numbers.subList(0, 6);
        Collections.sort(winningCombination);
        System.out.println(winningCombination);
    }
}

```

java.util.Collections 1.2

- static <T extends Comparable<? super T>> void sort(List<T> elements)
- static <T> void sort(List<T> elements, Comparator<? super T> c)
 Sortuje elementy listy przy użyciu stabilnego algorytmu sortującego. Długość czasu działania tego algorytmu to $O(n \log n)$, gdzie n oznacza długość listy.
- static void shuffle(List<?> elements)
- static void shuffle(List<?> elements, Random r)
 Tasuje elementy listy. Długość czasu działania tego algorytmu to $O(n \alpha(n))$, gdzie n to długość listy, a $\alpha(n)$ to średni czas dostępu do elementu.
- static <T> Comparator<T> reverseOrder()
 Zwraca komparator sortujący element w odwróconej kolejności w stosunku do porządku określonego przez metodę compareTo z interfejsu Comparable.
- static <T> Comparator<T> reverseOrder(Comparator<T> comp)
 Zwraca komparator sortujący element w odwróconej kolejności w stosunku do porządku określonego przez parametr comp.

13.4.2. Wyszukiwanie binarne

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

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

```

i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);

```

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

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

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

```
if (i < 0)
    c.add(-i - 1, element);
```

Aby było warte uwagi, wyszukiwanie binarne musi obsługiwać dostęp swobodny. Gdyby trzeba było w poszukiwaniu środkowego elementu przemierzyć połowę listy powiązanej, to stracilibyśmy wszystkie jego zalety. Dlatego algorytm `binarySearch` w przypadku list powiązanych przestawia się na przeszukiwanie liniowe.



W Java SE 1.3 nie istniał osobny interfejs dla uporządkowanych kolekcji z szybkim dostępem swobodnym, a metoda `binarySearch` implementowała bardzo prymitywny mechanizm sprawdzający, czy lista podana na wejściu rozszerza klasę `AbstractSequentialList`. Poprawiono to w Java SE 1.4. Obecnie metoda ta sprawdza, czy podana jako parametr lista implementuje interfejs `RandomAccess`. Jeśli tak, przeprowadzane jest wyszukiwanie binarne, w przeciwnym przypadku wyszukiwanie liniowe.

java.util.Collections 1.2

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

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

13.4.3. Proste algorytmy

W klasie `Collections` znajduje się jeszcze kilka prostych, ale przydatnych algorytmów. Wśród nich można znaleźć prezentowany na początku tego rozdziału algorytm wyszukiwania największej wartości w kolekcji. Z pozostałych można wymienić algorytm kopiący elementy z jednej listy do innej, zapełniający kontener stałą wartością czy odwracający listę. Po co w standardowej bibliotece umieszczać takie proste algorytmy? Przecież każdy programista mógłby je zaimplementować za pomocą pojedynczych pętli. Lubimy te algorytmy, ponieważ ułatwiają życie programistom czytającym **cudzy** kod. Czytając kod napisany przez kogoś

innego, zawsze musimy rozszyfrować intencje innego programisty. Kiedy widzimy wywołanie metody typu `Collections.max`, od razu wiemy, jakie jest przeznaczenie danego fragmentu kodu.

Poniższy wyciąg z API opisuje proste algorytmy dostępne w klasie `Collections`.

`java.util.Collections 1.2`

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

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

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

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

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

Wstawia na wszystkich pozycjach w liście tę samą wartość.

- `static <T> boolean addAll(Collection<? super T> c, T... values) 5.0`

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

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

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

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

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

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

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

Zamienia miejscami elementy znajdujące się w określonych miejscach.

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

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

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

Przeprowadza rotację elementów listy, przenosząc obiekt z indeksem `i` do miejsca $(i + d) \% l.size()$. Na przykład rotacja listy `[t, a, r]` o dwa pola daje w wyniku

listę [a, r, t]. Długość czasu działania tej metody to $O(n)$, gdzie n określa długość listy

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

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

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

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

13.4.4. Pisanie własnych algorytmów

Pisząc własny algorytm (a raczej każdą metodę przyjmującą jako parametr kolekcję), należy, kiedy to tylko możliwe, wykorzystywać **interfejsy**, a nie konkretne implementacje. Wyobraźmy sobie na przykład, że chcemy zapełnić obiekt JMenu zbiorem elementów menu. Metodę taką można by było zaimplementować w tradycyjny sposób następująco:

```
void fillMenu(JMenu menu, ArrayList< JMenuItem > items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

Jednak w ten sposób ograniczamy zakres ruchu wywołującego metodę — opcje wyboru muszą być podane w postaci obiektu ArrayList. Jeśli zdarzy się, że opcje te będą w jakimś innym kontenerze, konieczne będzie ich przepakowanie. Znacznie lepiej byłoby przyjąć bardziej ogólną kolekcję.

Należy sobie zadać następujące pytanie: jaka jest najbardziej ogólna metoda, która przyda się w tym zastosowaniu? W tym przypadku konieczne jest tylko odwiedzenie wszystkich elementów, a do tego nadaje się podstawowy interfejs Collection. Poniżej znajduje się poprawiona wersja metody fillMenu przyjmująca kolekcję każdego rodzaju:

```
void fillMenu(JMenu menu, Collection< JMenuItem > items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

Teraz metodę tę można wywołać przy użyciu struktury ArrayList, LinkedList bądź tablicy opakowanej przez metodę opakowującą Arrays.asList.



Skoro używanie interfejsów kolekcjnych jako parametrów metod jest takim dobrym rozwiązaniem, czemu nie jest ono stosowane częściej w bibliotece Java? Na przykład klasa JComboBox ma dwa konstruktory:

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

Powodem jest po prostu czas. Biblioteka Swing została utworzona przed powstaniem biblioteki kolekcji.

Jeśli piszemy metodę **zwracającą** kolekcję, możemy zdecydować się na zwrócenie także interfejsu zamiast klasy, ponieważ później możemy zmienić zdanie i zaimplementować naszą metodę ponownie przy użyciu innej kolekcji.

Napiszmy na przykład metodę o nazwie getAllItems zwracającą wszystkie elementy menu.

```
List<MenuItem> getAllItems(JMenu menu)
{
    ArrayList<MenuItem> items = new ArrayList<MenuItem>()
    for (int i = 0; i < menu.getItemCount(); i++)
        items.add(menu.getItem(i));
    return items;
}
```

Później możemy dojść do wniosku, że nie chcemy kopiować elementów, a tylko utworzyć ich widok. Możemy tego dokonać, zwracając anonimową podkласę klasy `AbstractList`.

```
List<MenuItem> getAllItems(final JMenu menu)
{
    return new
        AbstractList<MenuItem>()
    {
        public MenuItem get(int i)
        {
            return item.getItem(i);
        }
        public int size()
        {
            return item.getItemCount();
        }
    };
}
```

Jest to oczywiście zaawansowana technika. Stosując ją, należy skrupulatnie dokumentować, które opcjonalne operacje są obsługiwane. W tym przypadku należy poinformować wywołującego, że zwracany obiekt jest niemodyfikowaną listą.

13.5. Stare kolekcje

W tym podrozdziale opisujemy klasy kolekcyjne, które są dostępne w Javie od samego początku. Są to klasa `Hashtable` i jej przydatna podklasa `Properties`, podklasa `Stack` klasy `Vector` oraz klasa `BitSet`.

13.5.1. Klasa `Hashtable`

Standardowa metoda `Hashtable` pełni takie samo zadanie co `HashMap` i ma zasadniczo taki sam jak ona interfejs. Metody tej klasy, podobnie jak klasy `Vector`, są synchronizowane. Jeśli nie potrzebujesz synchronizacji lub zgodności ze starym kodem, używaj klasy `HashMap`.



Nazwa klasy to `Hashtable`, z małą literą „t” w środku. W systemie Windows można otrzymać dziwne komunikaty o błędach, jeśli użyje się nazwy `HashTable`, ponieważ system plików systemu Windows nie rozróżnia małych i wielkich liter, ale kompilator Javy tak.

13.5.2. Wyliczenia

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

Na przykład metoda `elements` z klasy `Hashtable` tworzy wyliczenie z wartościami znajdującymi się w tablicy:

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
    Employee e = e.nextElement();
    ...
}
```

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

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



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

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

■ `boolean hasMoreElements()`

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

■ `E nextElement()`

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

java.util.Hashtable<K, V> **1.0**

■ `Enumeration<K> keys()`

Zwraca obiekt wyliczeniowy, który przemierza klucze tablicy `Hashtable`.

- `Enumeration<V> elements()`

Zwraca obiekt wyliczeniowy, który przemierza elementy tablicy `Hashtable`.

`java.util.Vector<E> 1.0`

- `Enumeration<E> elements()`

Zwraca obiekt wyliczeniowy, który przegląda elementy wektora.

13.5.3. Mapy własności

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

- Klucze i wartości są łańcuchami.
- Tablicę tę można zapisać w pliku i załadować z pliku.
- Posiada dodatkową tablicę z wartościami domyślnymi.

Klasa Javy implementująca mapę własności nosi nazwę `Properties`.

Mapy te są powszechnie wykorzystywane do określania opcji konfiguracyjnych programów — zobacz rozdział 10.

`java.util.Properties 1.0`

- `Properties()`

Tworzy pustą mapę własności.

- `Properties(Properties defaults)`

Tworzy pustą mapę własności z zestawem wartości domyślnych.

- `String getProperty(String key)`

Pobiera wartość przypisaną do własności. Zwraca łańcuch przypisany do określonego klucza. Jeśli nie ma go w tablicy głównej, szuka w tablicy wartości domyślnych.

- `String getProperty(String key, String defaultValue)`

Zwraca własność z domyślną wartością, jeśli klucz nie zostanie znaleziony. Zwraca łańcuch przypisany do klucza lub łańcuch domyślny, jeśli łańcuch nie zostanie znaleziony w mapie.

- `void load(InputStream in)`

Ładuje mapę własności ze strumienia wejściowego.

- `void store(OutputStream out, String commentString)`

Zapisuje zawartość mapy własności w strumieniu wyjściowym.

13.5.4. Stosy

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

`java.util.Stack<E> 1.0`

■ `E push(E item)`

Wstawia element na stos i go zwraca.

■ `E pop()`

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

■ `E peek()`

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

13.5.5. Zbiory bitów

Klasa `BitSet` umożliwia przechowywanie szeregów bitowych (nie są to **zbiory** z matematycznego punktu widzenia — lepszymi nazwami byłyby **tablica** bitów lub `BitVector`). Znajduje ona zastosowanie w sytuacjach, w których potrzebna jest szybko działająca struktura do przechowywania szeregów bitów (na przykład flag). Ponieważ obiekt `BitSet` pakuje bity w bajty, struktura ta jest znacznie szersza niż obiekt `ArrayList` wypełniony obiektami typu `Boolean`.

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

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

`bucketOfBits.get(i)`

Podobnie poniższa instrukcja włącza *i*-ty bit.

`bucketOfBits.set(i)`

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

`bucketOfBits.clear(i)`



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

java.util.BitSet 1.0■ **BitSet(int initialCapacity)**

Tworzy zbiór bitów.

■ **int length()**

Zwraca „logiczną długość” zbioru bitów — 1 plus indeks najwyższego ustawionego bitu.

■ **boolean get(int bit)**

Pobiera bit.

■ **void set(int bit)**

Ustawia bit.

■ **void clear(int bit)**

Usuwa bit.

■ **void and(BitSet set)**

Tworzy sumę logiczną dwóch zbiorów bitów.

■ **void or(BitSet set)**

Tworzy alternatywę logiczną dwóch zbiorów.

■ **void xor(BitSet set)**

Wykonuje działanie logiczne XOR (lub wykluczające) na dwóch zbiorach bitów.

■ **void andNot(BitSet set)**

Usuwa wszystkie bity ze zbioru bitów, które są ustawione w innym zbiorze bitów.

13.5.5.1. Test wydajności za pomocą sita Eratostenesa

Zastosowanie zbiorów bitów zademonstrujemy za pomocą algorytmu do znajdowania liczb pierwszych, czyli sita Eratostenesa (liczba pierwsza to taka, która dzieli się bez reszty tylko przez siebie samą i jeden, a sito Eratostenesa jest pierwszą odkrytą metodą obliczania tych liczb). Nie jest to najlepszy algorytm do znajdowania liczb pierwszych, ale z niewyjaśnionych powodów zyskał dużą popularność jako test wydajności kompilatorów (nie jest to też dobry test wydajności, ponieważ testuje przede wszystkim operacje bitowe).

Jest to jednak ukłon w stronę tradycji — przedstawiamy implementację tego algorytmu. Prezentowany program zlicza wszystkie liczby pierwsze w zakresie od 2 do 2 000 000 (jest ich 148 933, a więc pewnie lepiej ich wszystkich nie drukować).

Nie zagłębiając się zbytnio w szczegóły implementacyjne tego programu, jego zadaniem jest przejście przez zbiór bitów zawierający dwa miliony elementów. Najpierw włączamy wszystkie bity. Następnie wyłączamy bity będące wielokrotnościami liczb, które wiadomo, że są pierwsze. Liczby odpowiadające bitom pozostałym po tym procesie są pierwsze. Listing 13.8 przedstawia implementację tego programu w języku Java, a listing 13.9 — w języku C++.

Listing 13.8. sieve/Sieve.java

```

package sieve;

import java.util.*;

/**
 * Program wykonujący test porównawczy na bazie algorytmu sita Eratostenesa. Oblicza wszystkie
 * liczby pierwsze do 2 000 000.
 * @version 1.21 2004-08-03
 * @author Cay Horstmann
 */
public class Sieve
{
    public static void main(String[] s)
    {
        int n = 2000000;
        long start = System.currentTimeMillis();
        BitSet b = new BitSet(n + 1);
        int count = 0;
        int i;
        for (i = 2; i <= n; i++)
            b.set(i);
        i = 2;
        while (i * i <= n)
        {
            if (b.get(i))
            {
                count++;
                int k = 2 * i;
                while (k <= n)
                {
                    b.clear(k);
                    k += i;
                }
            }
            i++;
        }
        while (i <= n)
        {
            if (b.get(i)) count++;
            i++;
        }
        long end = System.currentTimeMillis();
        System.out.println(count + " liczb pierwszych");
        System.out.println((end - start) + " milisekund");
    }
}

```

Listing 13.9. Sieve.cpp

```

/**
 * @version 1.21 2004-08-03
 * @author Cay Horstmann
 */

```



Mimo że sito nie jest dobrym testem wydajności, nie mogliśmy się oprzeć pokusie zmierzenia czasu obu implementacji algorytmu. Oto wyniki uzyskane na notebooku ThinkPad z dwurdzeniowym procesorem 2,4 GHz, 4 GB pamięci RAM i systemem operacyjnym Linux Ubuntu 7.04:

- C++ (g++ 4.6.3): 160 milisekund,
- Java (Java SE 7): 84 milisekundy.

Test ten przeprowadziliśmy we wszystkich dziewięciu wydaniach tej książki i w ostatnich pięciu Java bije C++ na głowę. Trzeba jednak ujawnić, że jeśli zoptymalizujemy kompilator C++, pobije on Javę, uzyskując czas 20 milisekund. Taki wynik w Javie można by było uzyskać tylko wtedy, gdyby program działał na tyle długo, aby włączył się kompilator JIT Hotspot.

```
#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    const int N = 2000000;
    clock_t cstart = clock();

    bitset<N + 1> b;
    int count = 0;
    int i;
    for (i = 2; i <= N; i++)
        b.set(i);
    i = 2;
    while (i * i <= N)
    {
        if (b.test(i))
        {
            count++;
            int k = 2 * i;
            while (k <= N)
            {
                b.reset(k);
                k += i;
            }
        }
        i++;
    }
    while (i <= N)
    {
        if (b.test(i))
            count++;
        i++;
    }
}

clock_t cend = clock();
double millis = 1000.0 * (cend - cstart) / CLOCKS_PER_SEC;
```

```
cout << count << " liczb pierwszych\n" << millis << " milisekund\n";  
return 0;  
}
```

Na tym kończy się nasza podróż po architekturze kolekcji w języku Java. Jak widać, biblioteka w tym języku udostępnia bogaty zestaw klas kolekcyjnych, mających na celu zaspokajanie potrzeb programisty. W ostatnim rozdziale książki zajmiemy się bardzo ważnym tematem współprzeźności.

14

Wielowątkowość

W tym rozdziale:

- Czym są wątki
- Przerywanie wątków
- Stany wątków
- Własności wątków
- Synchronizacja
- Kolejki blokujące
- Kolekcje bezpieczne wątkowo
- Interfejsy Callable i Future
- Klasa Executors
- Synchronizatory
- Wątki a biblioteka Swing

Większość użytkowników systemów operacyjnych zna pojęcie **wielozadaniowości**, czyli zdolności systemu do uruchamiania więcej niż jednego programu pozornie jednocześnie. Można na przykład pisać lub wysyłać e-mail i w tym samym czasie drukować jakiś dokument. W dzisiejszych czasach coraz częściej spotyka się komputery wyposażone w więcej niż jeden procesor, choć liczba procesów działających jednocześnie nie jest ograniczona liczbą procesorów. System operacyjny stwarza pozory równoległego wykonywania zadań, każdemu procesowi przydzielając odpowiedni czas pracy procesora.

Programy wielowątkowe przenoszą koncepcję wielozadaniowości o jeden poziom niżej, gdzie poszczególne programy sprawiają złudzenie wykonywania wielu zadań naraz. Każde z tych zadań jest zwyczajowo nazywane **wątkiem** (ang. *thread*), a pełna nazwa to wątek sterowania (ang. *thread of control*). Programy potrafiące działać w więcej niż jednym wątku nazywają się programami **wielowątkowymi** (ang. *multithreaded*).

Jaka jest zatem różnica pomiędzy wieloma **procesami** a wieloma **wątkami**? Przede wszystkim należy zauważyc, że każdy proces posiada pełen zestaw własnych zmiennych, podczas gdy wątki współdzielą dane z innymi wątkami. Brzmi to dosyć ryzykownie i rzeczywiście może czasami sprawiać problemy, o czym przekonasz się za chwilę. Z drugiej strony dzięki współdzieleniu zmiennych komunikacja pomiędzy wątkami zachodzi sprawniej i jest łatwiejsza do zaprogramowania niż komunikacja międzyprocesowa. Ponadto wątki w niektórych systemach operacyjnych są lżejsze od procesów, to znaczy utworzenie i zniszczenie pojedynczego wątku zajmuje mniej czasu niż uruchomienie nowego procesu.

Wielowątkowość jest niezwykle praktycznym narzędziem. Wiadomo na przykład, że przeglądarka powinna mieć możliwość pobierania kilku obrazów jednocześnie, a serwer sieciowy musi obsługiwać wiele żądań w tym samym czasie. Programy z graficznym interfejsem użytkownika dysponują osobnym wątkiem do zbierania zdarzeń z interfejsu pochodzących od środowiska operacyjnego. Ten rozdział dotyczy pisania aplikacji wielowątkowych w Javie.

Ostrzeżenie: programy wielowątkowe bywają bardzo skomplikowane. My opisujemy wszystkie narzędzia, których może potrzebować programista. Jednak w poszukiwaniu opisów bardziej zaawansowanych technik programowania systemowego odsyłamy do innych źródeł, na przykład książki *Java. Współbieżność dla praktyków*, której autorami są Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea (Helion, Gliwice 2007).

14.1. Czym są wątki

Zaczniemy od programu jednowątkowego, w którym wykonywanie kilku czynności naraz jest utrudnione. Po jego szczegółowym przeanalizowaniu pokażemy, jak łatwo można go przerobić, aby działał w kilku wątkach. Program ten jest animacją odbijającej się piłki, która pozostaje w ciągłym ruchu. Program sprawdza, czy piłka nie uderza o ścianę, i rysuje ją ponownie (rysunek 14.1).

Rysunek 14.1.

Jednowątkowa animacja odbijającej się piłki



Po uruchomieniu programu i naciśnięciu przycisku *Start* piłka zaczyna się odbijać, wychodząc z lewego górnego rogu okna. Obiekt obsługujący przycisk *Start* wywołuje metodę `addBall`, która zawiera pętlę powtarzającą się przez 1000 ruchów. Metoda `move` przesuwa nieco piłkę, koryguje kierunek w przypadku odbicia od ściany i ponownie rysuje panel.

```

Ball ball = new Ball();
panel.add(ball);
for (int i = 1; i <= STEPS; i++)
{
    ball.move(panel.getBounds());
    panel.paint(panel.getGraphics());
    Thread.sleep(DELAY);
}

```

Wywołanie metody `Thread.sleep` nie powoduje utworzenia nowego wątku, ponieważ metoda ta wstrzymuje działanie bieżącego wątku na określoną liczbę milisekund.

Metoda `sleep` może zgłosić wyjątek `InterruptedException`. Zajmiemy się nim dalej. Na razie jeśli wystąpi ten wyjątek, po prostu kończymy odbijanie piłki.

Po uruchomieniu programu piłka bardzo ładnie odbija się od ścian, ale niestety całkowicie pochłania aplikację. Jeśli zechcemy zatrzymać odbijanie przed wykonaniem przez piłkę tysiąca ruchów i naciśniemy w tym celu przycisk *Zamknij*, nic się nie stanie. Piłka będzie dalej się odbijać, ponieważ nie można robić w programie nic innego, dopóki nie zakończy się jedno zadanie.



Na końcu kodu prezentowanego w tym podrozdziale znajduje się poniższa instrukcja:

```
comp.paint(comp.getGraphics())
```

Mieści się ona w metodzie `addBall` klasy `BounceFrame`. Jest to bardzo dziwna technika programowania — normalnie wywołano by metodę `repaint`, a resztą zajęłaby się biblioteka AWT. Jeśli jednak wywołamy metodę `repaint` w tym programie, panel nie zostanie nigdy odświeżony, ponieważ metoda `addBall` całkowicie zajęła wszystkie procesy. Dodatkowo warto zauważyć, że komponent piłki dziedziczy po klasie `JPanel`, dzięki czemu łatwiej jest wyczyścić tło. W następnym programie, w którym położenie piłki jest obliczane w osobnym wątku, wróćmy do znanej nam metody `repaint` z klasy `JComponent`.

Oczywiście funkcjonalność tego programu nie jest zbyt bogata. Z pewnością nie chcielibyśmy, aby programy wykonujące czasochłonne zadania działały w ten sposób. Na przykład podczas pobierania danych z sieci bardzo często zdarza się nam utknąć w zadaniu, które chcielibyśmy przerwać. Wyobraźmy sobie na przykład, że pobieramy duży obraz i po zobaczeniu jego fragmentu wiemy już, że nie chcemy oglądać reszty. W takiej sytuacji przydaje się przycisk *Stop* lub *Wstecz*, który przerywa proces ładowania. W kolejnym podrozdziale nauczysz się pozostawiać kontrolę w rękach użytkownika dzięki uruchamianiu najważniejszych części kodu w osobnym **wątku**.

Kod programu jest przedstawiony na listingach 14.1 – 14.3.

Listing 14.1. `bounce/Bounce.java`

```

package bounce;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
/*
 * Wyświetla animowaną piłkę
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class Bounce
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new BounceFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca komponent piłki i przyciski
 */
class BounceFrame extends JFrame
{
    private BallComponent comp;
    public static final int STEPS = 1000;
    public static final int DELAY = 3;

    /**
     * Tworzy ramkę z komponentem zawierającym odbijającą się piłkę oraz przyciskami
     * Start i Zamknij.
     */
    public BounceFrame()
    {
        setTitle("Piłka");

        comp = new BallComponent();
        add(comp, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "Start", new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                addBall();
            }
        });

        addButton(buttonPanel, "Zamknij", new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });
        add(buttonPanel, BorderLayout.SOUTH);
    }
}
```

```

        pack();
    }

    /**
     * Dodaje przycisk do kontenera.
     * @param c kontener
     * @param title tytuł przycisku
     * @param listener słuchacz akcji przycisku
     */
    public void addButton(Container c, String title, ActionListener listener)
    {
        JButton button = new JButton(title);
        c.add(button);
        button.addActionListener(listener);
    }

    /**
     * Dodaje odbijającą się piłkę do panelu i odbija ją 1000 razy
     */
    public void addBall()
    {
        try
        {
            Ball ball = new Ball();
            comp.add(ball);

            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(comp.getBounds());
                comp.paint(comp.getGraphics());
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException e)
        {
        }
    }
}

```

Listing 14.2. bounce/Ball.java

```

package bounce;

import java.awt.geom.*;

/**
 * Piłka, która porusza się i odbija od krawędzi prostokąta
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class Ball
{
    private static final int XSIZE = 15;
    private static final int YSIZE = 15;
    private double x = 0;
    private double y = 0;
    private double dx = 1;

```

```
private double dy = 1;

/**
 * Przesuwa piłkę do następnego położenia, odwracając kierunek, jeśli piłka uderzy w krawędź
 */
public void move(Rectangle2D bounds)
{
    x += dx;
    y += dy;
    if (x < bounds.getMinX())
    {
        x = bounds.getMinX();
        dx = -dx;
    }
    if (x + XSIZE >= bounds.getMaxX())
    {
        x = bounds.getMaxX() - XSIZE;
        dx = -dx;
    }
    if (y < bounds.getMinY())
    {
        y = bounds.getMinY();
        dy = -dy;
    }
    if (y + YSIZE >= bounds.getMaxY())
    {
        y = bounds.getMaxY() - YSIZE;
        dy = -dy;
    }
}

/**
 * Ustawia piłkę w jej aktualnym położeniu
 */
public Ellipse2D getShape()
{
    return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
}
```

Listing 14.3. bounce/BallComponent.java

```
package bounce;

import java.awt.*;
import java.util.*;
import javax.swing.*;

/**
 * Komponent rysujący piłki
 * @version 1.34 2012-01-26
 * @author Cay Horstmann
 */
public class BallComponent extends JPanel
{
    private static final int DEFAULT_WIDTH = 450;
    private static final int DEFAULT_HEIGHT = 350;
```

```

private java.util.List<Ball> balls = new ArrayList<>();

/**
 * Dodaje piłkę do komponentu
 * @param b piłka, która ma zostać dodana
 */
public void add(Ball b)
{
    balls.add(b);
}

public void paintComponent(Graphics g)
{
    super.paintComponent(g); // Czyszczenie tła
    Graphics2D g2 = (Graphics2D) g;
    for (Ball b : balls)
    {
        g2.fill(b.getShape());
    }
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
    DEFAULT_HEIGHT); }
}

```

java.lang.Thread **1.0**

■ static void sleep(long millis)

Zatrzymuje wykonywanie na określoną liczbę milisekund.

Parametr: millis Liczba milisekund

14.1.1. Wykonywanie zadań w osobnych wątkach

Usprawnimy nasz program, uruchamiając kod poruszający piłką w osobnym wątku. Będzie można nawet wypuścić kilka piłek naraz, a każda z nich będzie poruszana przez oddzielny wątek. Dodatkowo równolegle z nimi będzie działał **wątek dystrybucji zdarzeń**, który zajmuje się zdarzeniami interfejsu użytkownika. Dzięki temu, że każdy wątek może zostać uruchomiony, wątek dystrybucji zdarzeń może odnotować naciśnięcie przycisku *Zakończ* w czasie, gdy piłki są w ruchu, i wykonać akcję zamknięcia aplikacji.

Program z piłką ma za zadanie wykazać, że współbieżność jest niezbędna. Ogólnie rzecz biorąc, zawsze trzeba uważać na długotrwałe operacje. Wchodzą one w skład jakiegoś większego szkieletu, jak na przykład GUI albo interfejs sieciowy. Użytkownik zawsze oczekuje, że dostanie szybką odpowiedź. Dlatego wszystkie czasochłonne operacje należy wykonywać w osobnych wątkach.

Poniżej przedstawiamy prosty zestaw czynności, które należy wykonać, aby uruchomić zadanie w osobnym wątku:

1. Kod zadania umieść w metodzie run klasy implementującej interfejs Runnable.
Ten bardzo prosty interfejs posiada tylko jedną metodę:

```
public interface Runnable
{
    void run();
}
```

Implementacja klasy jest prosta i wygląda następująco:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        kod zadania
    }
}
```

- 2.** Utwórz obiekt powstałej klasy:

```
Runnable r = new MyRunnable();
```

- 3.** Utwórz obiekt typu Thread z obiektu Runnable:

```
Thread t = new Thread(r);
```

- 4.** Uruchom wątek:

```
t.start();
```

Umieszczenie programu odbijającej się piłki w osobnym wątku wymaga tylko zaimplementowania klasy BallRunnable i umieszczenia kodu animacji w metodzie run:

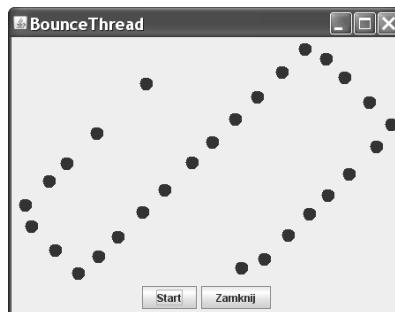
```
class BallRunnable implements Runnable
{
    ...
    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
        {
        }
    }
    ...
}
```

Nie można zapomnieć o przechwyceniu wyjątku InterruptedException, którym straszy metoda sleep. Wyjątek ten opisujemy dokładnie w kolejnym podrozdziale. Z reguły oznacza on konieczność przerwania wątku. Dlatego nasza metoda run kończy działanie w chwili jego wystąpienia.

Każde kliknięcie przycisku *Start* powoduje, że metoda addBall uruchamia nowy wątek (zobacz rysunek 14.2):

Rysunek 14.2.

Uruchomienie kilku wątków



```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```

To wszystko, co trzeba zrobić, aby uruchomić zadania w osobnych wątkach! Pozostała część tego rozdziału jest poświęcona komunikacji pomiędzy wątkami.

Pełny kod programu przedstawia listing 14.4.

Listing 14.4. bounceThread/BounceThread.java

```
package bounceThread;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Wyświetla animowane piłki
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class BounceThread
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new BounceFrame();
                frame.setTitle("BounceThread");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Klasa implementująca interfejs Runnable i tworząca animację piłki
 */
```

```
class BallRunnable implements Runnable
{
    private Ball ball;
    private Component component;
    public static final int STEPS = 1000;
    public static final int DELAY = 5;

    /**
     * Tworzy obiekt Runnable
     * @param aBall piłka
     * @param aComponent komponent, w którym odbija się piłka
     */
    public BallRunnable(Ball aBall, Component aComponent)
    {
        ball = aBall;
        component = aComponent;
    }

    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException e)
        {
        }
    }
}

/**
 * Ramka z panelem i przyciskami
 */
class BounceFrame extends JFrame
{
    private BallComponent comp;

    /**
     * Tworzy ramkę z komponentem zawierającym piłkę i przyciski Start oraz Zamknij
     */
    public BounceFrame()
    {
        comp = new BallComponent();
        add(comp, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "Start", new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                addBall();
            }
        });
    }
}
```

```

 addButton(buttonPanel, "Zamknij", new ActionListener()
 {
     public void actionPerformed(ActionEvent event)
     {
         System.exit(0);
     }
 });
 add(buttonPanel, BorderLayout.SOUTH);
 pack();
}

/*
 * Dodaje przycisk do kontenera
 * @param c kontener
 * @param title tytuł przycisku
 * @param listener słuchacz akcji przycisku
 */

public void addButton(Container c, String title, ActionListener listener)
{
    JButton button = new JButton(title);
    c.add(button);
    button.addActionListener(listener);
}

/*
 * Dodaje piłkę do obszaru roboczego i uruchamia wątek wykonujący kod odpowiedzialny za jej odbijanie
 */

public void addBall()
{
    Ball b = new Ball();
    comp.add(b);
    Runnable r = new BallRunnable(b, comp);
    Thread t = new Thread(r);
    t.start();
}
}
}

```



Wątek można także zdefiniować, tworząc podklasę klasy Thread:

```

class MyThread extends Thread
{
    public void run()
    {
        kod zadania
    }
}

```

Wtedy należy utworzyć obiekt powstałej podklasy i wywołać na jego rzecz metodę start. Podejście to jest jednak obecnie odradzane. **Zadanie**, które ma być uruchamiane, powinno być oddzielone od **mechanizmu** je uruchamiającego. W przypadku dużej liczby zadań tworzenie osobnego wątku dla każdego z nich może być zbyt kosztowne. W takiej sytuacji należy użyć puli wątków — zobacz podrozdział 14.9, „Klasa Executors”.



Nie wywołuj metody `run` z klasy `Thread` lub obiektu typu `Runnable`. Bezpośrednie wywołanie tej metody powoduje jedynie wykonanie zadania w **tym samym** wątku — żaden nowy wątek nie jest uruchamiany. W zamian należy wywoływać metodę `Thread.start`. Tworzy ona nowy wątek, który uruchamia metodę `run`.

java.lang.Thread 1.0

- `Thread(Runnable target)`

Tworzy nowy wątek, który wywołuje metodę `run()` określonego obiektu `target`.

- `void start()`

Uruchamia wątek, wywołując metodę `run()`. Ta metoda zwraca wartość natychmiast. Nowy wątek działa równolegle.

- `void run()`

Wywołuje metodę `run` obiektu implementującego interfejs `Runnable`.

java.lang.Runnable 1.0

- `void()`

Metodę tę należy przedefiniować, wstawiając do niej instrukcje zadania, które ma zostać wykonane.

14.2. Przerywanie wątków

Wątek kończy działanie w chwili zwrócenia przez jego metodę `run` wartości w wyniku wywołania instrukcji `return`, po wykonaniu ostatniej instrukcji w ciele metody `run` lub jeśli wystąpi nieprzechwycony w tej metodzie wyjątek. W pierwszej wersji Javy istniała jeszcze metoda `stop`, którą mógł wywołać jeden wątek w celu zamknięcia innego wątku. Jest ona jednak obecnie odradzana. Powody tego stanu rzeczy opisujemy w podrozdziale 14.5.15, „Dlaczego metody `stop` i `suspend` są wycofywane”.

Istnieje sposób na **zmuszenie** wątku do zamknięcia. Służy do tego metoda `interrupt`, która wysyła **żądanie** zamknięcia wątku.

Wywołanie metody `interrupt` na rzecz wątku powoduje ustawienie jego **statusu przerwania** (ang. *interrupted state*). Jest to zmienna logiczna obecna w każdym wątku. Każdy wątek powinien co jakiś czas sprawdzać, czy nie został przerwany.

Aby sprawdzić, czy status przerwania został ustawiony, należy najpierw pobrać bieżący wątek za pomocą metody `Thread.currentThread`, a następnie wywołać metodę `isInterrupted`:

```
while (!Thread.currentThread().isInterrupted() && więcej instrukcji)
{
    dodatkowe działania
}
```

Statusu przerwania nie można jednak sprawdzić, jeśli wątek jest zablokowany. W takiej sytuacji do gry wchodzi wyjątek `InterruptedException`. Kiedy metoda `interrupt` jest wywoływana na rzecz wątku, który blokują metody takie jak `sleep` czy `wait`, wywołania blokujące zostają zakończone przez wyjątek `InterruptedException` (istnieją metody blokujące wejścia-wyjścia, których nie można przerwać; w takiej sytuacji należy rozważyć użycie ich przerywalnych zamienników — szczegółowe informacje na ten temat znajdują się w rozdziałach 1. i 3. drugiego tomu).

Nie istnieje żaden wymóg formalny, że wątek, który został przerwany, musi zostać zamknięty. Przerwanie wątku powoduje jedynie zwrócenie jego uwagi, a decyzja, jak na to zareagować, należy do niego samego. Niektóre bardzo ważne wątki powinny obsługiwać wyjątek i kontynuować pracę. Często jednak przerwanie jest przez wątek interpretowane jako żądanie zamknięcia. Metoda `run` takiego wątku wygląda następująco:

```
public void run()
{
    try
    {

        while (!Thread.currentThread().isInterrupted() && dodatkowe instrukcje)
        {
            instrukcje
        }
    }
    catch(InterruptedException e)
    {
        // Wątek został przerwany, będąc w stanie uśpienia bądź oczekiwania.
    }
    finally
    {
        czyszczenie w razie potrzeby
    }
    // Wyjście z metody run powoduje zakończenie wątku.
}
```

Wywoływanie metody `isInterrupted` staje się bezcelowe, jeśli po każdej iteracji wywoływana jest metoda `sleep` (lub inna pozwalająca na przerwanie). Jeśli metoda `sleep` zostanie wywołana na rzecz wątku z ustawionym statusem przerwania, wątek ten nie zostanie uśpiony. Zamiast tego jego status zostanie wyzerowany oraz zostanie zgłoszony wyjątek `InterruptedException`. Dlatego jeśli pętla zawiera wywołanie metody `sleep`, nie należy w niej sprawdzać statusu przerwania. W zamian należy przechwycić wyjątek `InterruptedException`:

```
public void run()
{
    try
    {

        while (instrukcje)
        {
            instrukcje
            Thread.sleep(delay);
        }
    }
    catch(InterruptedException e)
```

```

    {
        // Wątek został przerwany w czasie uśpienia.
    }
    finally
    {
        Czyszczenie w razie potrzeby
    }
    // Wyjście z metody run powoduje zamknięcie wątku.
}

```



Istnieją dwie bardzo podobne do siebie metody: `interrupted` oraz `isInterrupted`. Pierwsza z nich jest statyczna i sprawdza, czy **bieżący** wątek nie został przerwany. Dodatkowo jej wywołanie powoduje **wyzerowanie** statusu przerwania wątku. Druga natomiast jest metodą obiektową, za pomocą której można sprawdzić status przerwania dowolnego wątku, bez jego zmiany.

Często spotyka się fragmenty kodu, w których wyjątek `InterruptedException` jest tłumiony w następujący sposób:

```

void mySubTask()
{
    .
    .
    try { sleep(delay); }
    catch (InterruptedException e) {} // Nie ignoruj!
    .
}

```

Nie należy tego robić! Jeśli nie masz żadnego dobrego pomysłu na klauzulę `catch`, masz jeszcze dwa inne dobre wyjścia:

- W klauzuli `catch` umieść instrukcję `Thread.currentThread().interrupt()`, która ustawi status przerwania. Dzięki temu wywołujący będzie mógł sprawdzić wyjątek.

```

void mySubTask()
{
    .
    .
    try { sleep(delay); }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    .
}

```

- Lepszym wyjściem jest dodanie do metody instrukcji `throws InterruptedException` i pominięcie bloku `try`. Wtedy wywołujący (lub ostatecznie metoda `run`) może przechwycić ten wyjątek.

```

void mySubTask() throws InterruptedException
{
    .
    .
    sleep(delay);
    .
}

```

java.lang.Thread **1.0**

■ void interrupt()

Wysyła żądanie przerwania do wątku. Status przerwania zostaje ustawiony na true. Jeśli wątek jest aktualnie zablokowany przez metodę sleep, zgłoszony jest wyjątek InterruptedException.

■ static boolean interrupted()

Sprawdza, czy **bieżący** wątek (to znaczy ten, który wykonuje tę instrukcję) nie został przerwany. Należy zauważyć, że jest to metoda statyczna. Jej wywołanie ma jeden efekt uboczny — zeruje status przerwania bieżącego wątku na wartość false.

■ boolean isInterrupted()

Sprawdza, czy wątek nie został przerwany. W przeciwieństwie do statycznej metody interrupted, ta nie zmienia statusu przerwania wątku.

■ static Thread currentThread()

Zwraca obiekt Thread reprezentujący aktualnie wykonywany wątek.

14.3. Stany wątków

Wątek może być w jednym z sześciu stanów:

- NEW (nowy),
- RUNNABLE (wykonywalny),
- BLOCKED (zablokowany),
- WAITING (oczekujący),
- TIMED WAITING (oczekujący określoną ilość czasu),
- TERMINATED (zakończony).

Znaczenie tych wszystkich stanów zostało opisane poniżej.

Do sprawdzania aktualnego stanu wątku służy metoda getState.

14.3.1. Wątki NEW

Wątek utworzony za pomocą operatora new — na przykład new Thread(r) — nie jest od razu uruchamiany. Oznacza to, że pozostaje on w stanie NEW. Jeśli wątek znajduje się w stanie NEW, program nie zaczął jeszcze wykonywać znajdującego się w nim kodu. Przed uruchomieniem wątku trzeba wykonać jeszcze kilka dodatkowych czynności.

14.3.2. Wątki RUNNABLE

Po wywołaniu metody `start` wątek przechodzi w stan RUNNABLE. Wątek taki może, ale nie musi być uruchomiony. Przydział czasu dla wątku leży w gestii systemu operacyjnego (w Javie dla stanu działania wątku nie wprowadzono osobnej nazwy, dlatego uruchomiony wątek nadal pozostaje w stanie RUNNABLE).

Po uruchomieniu wątek nie musi działać cały czas. Zaleca się nawet wstrzymywanie działających wątków co jakiś czas, aby dać szansę na działanie innym wątkom. Szczegółowa kontrola harmonogramu wykonywania wątków zależy od usług udostępnianych przez system operacyjny. Systemy planowania wywolaszczającego wątków przydzielają każdemu wykonywalnemu wątkowi określoną ilość czasu na wykonanie zadania. Kiedy czas mija, system operacyjny **wywłaszcza** wątek i przydziela czas innemu wątkowi (zobacz rysunek 14.4). Przy wybieraniu kolejnego wątku system operacyjny kieruje się **priorytetami** wątków — zobacz podrozdział 14.4.1, „Priorytety wątków”.

Wszystkie nowoczesne systemy operacyjne — zarówno serwerowe, jak i przeznaczone na komputery osobiste — stosują planowanie wywolaszczające. Mniejsze urządzenia, jak telefony komórkowe, mogą wykorzystywać planowanie kooperacyjne. W takim urządzeniu wątek traci sterowanie, jeśli wywoła metodę `yield` lub zostanie zablokowany czy przestawiony w stan oczekiwania.

W komputerach z kilkoma procesorami każdy procesor może wykonywać osobny wątek, dzięki czemu wiele wątków może działać równolegle. Oczywiście jeśli wątków jest więcej niż procesorów, system planujący i tak musi się zająć przydzielaniem czasu.

Należy zawsze pamiętać, że wykonywalny wątek może w danej chwili być uruchomiony lub nie (dlatego stan ten nazwano RUNNABLE, co oznacza „wykonywalny”, zamiast na przykład RUNNING, co znaczyłoby „wykonywany”).

14.3.3. Wątki BLOCKED i WAITING

Kiedy wątek znajduje się w stanie BLOCKED (zablokowany) lub WAITING (oczekujący), jest okresowo nieaktywny. Nie wykonuje żadnego kodu i zużywa minimalne ilości zasobów. Decyzja o jego reaktywacji należy do algorytmu planującego, który uzależnia ją od sposobu, w jaki wątek wszedł w stan nieaktywności:

- Kiedy wątek usiłuje założyć blokadę wewnętrzną na obiekt (ale nie utworzyć obiekt klasy `Lock` z biblioteki `java.util.concurrent`), który jest aktualnie w dyspozycji innego wątku, zostaje **zablokowany** (blokady `java.util.concurrent` opisujemy w podrozdziale 14.5.3, „Obiekty klasy Lock”, a blokady wewnętrzne obiektów w podrozdziale 14.5.5, „Słowo kluczowe synchronized”). Wątek zostaje odblokowany, gdy wszystkie pozostałe wątki zwolnią blokadę, a algorytm planujący zezwoli mu na przejęcie tego obiektu.
- Kiedy wątek czeka, aż inny wątek powiadomi algorytm planujący o warunku, wchodzi w stan WAITING (oczekujący). Warunki opisujemy w podrozdziale 14.5.4, „Warunki”. Stan ten jest wyzwalany przez wywołanie metody `Object.wait`

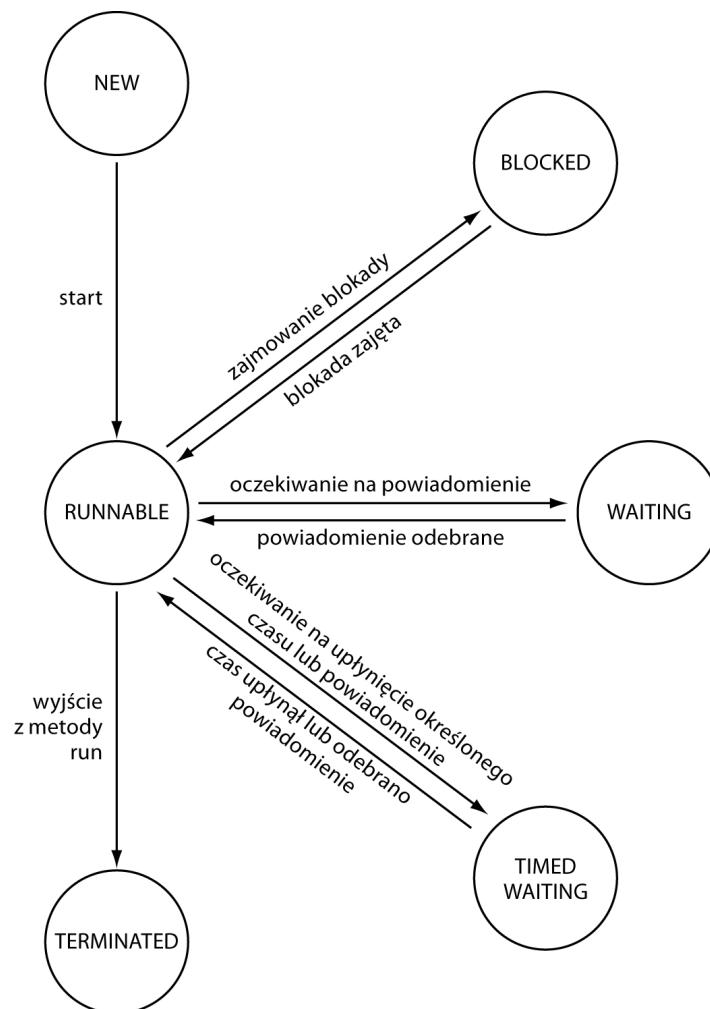
lub `Thread.join` bądź oczekiwanie na obiekt klasy `Lock` lub `Condition` z biblioteki `java.util.concurrent`. W praktyce różnica pomiędzy stanem zablokowania a oczekiwania nie jest jasna.

- Niektóre metody posiadają parametr określający długość czasu wykonywania. Ich wywołanie powoduje wejście wątku w stan `TIMED_WAITING`. Stan ten utrzymuje się, aż upłynie określony czas lub wątek odbierze odpowiednie powiadomienie. Do tego typu metod zalicza się metodę `Thread.sleep` oraz czasowe wersje metod `Object.wait`, `Thread.join`, `Lock.tryLock` i `Condition.wait`.

Rysunek 14.3 przedstawia stany, w których może się znaleźć wątek, oraz możliwe przejścia pomiędzy poszczególnymi stanami. Kiedy wątek zostanie zablokowany lub przejdzie w stan oczekiwania (bądź zostanie zakończony), planowane jest uruchomienie kolejnego wątku. Kiedy wątek jest reaktywowany (ponieważ skończył się jego czas oczekiwania lub zajął blokadę), algorytm planujący sprawdza, czy ma on wyższy priorytet niż aktualnie działające wątki. Jeśli tak, wypłaszcza jeden z uruchomionych wątków i uruchamia nowy wątek.

Rysunek 14.3.

Stany wątków



14.3.4. Zamknięcie wątków

Wątek może zostać zamknięty na jeden z dwóch sposobów:

- naturalnie wraz z zakończeniem metody run;
- niespodziewanie z powodu nieprzechwyconego wyjątku, który zakończył metodę run.

Do zamykania wątków służy metoda stop. Wyrzuca ona błąd ThreadDeath, który zabija wątek. Metoda ta jest jednak odradzana, dlatego nie powinno się jej już używać.

java.lang.Thread **1.0**

- void join()

Oczekuje na zamknięcie określonego wątku.
- void join(long millis)

Czeka na zamknięcie określonego wątku albo na upływ określonej liczby milisekund.
- Thread.State getState() **5.0**

Zwraca stan wątku: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING lub TERMINATED.
- void stop()

Zatrzymuje wątek. Metoda ta jest odradzana.
- void suspend()

Zawiesza wykonywanie wątku. Metoda odradzana.
- void resume()

Ponownie uaktywnia wątek. Metoda ta może działać tylko po wywołaniu metody suspend(). Metoda odradzana.

14.4. Właściwości wątków

W tym podrozdziale opisujemy różne właściwości wątków: priorytety, demony, grupy oraz procedury obsługi nieprzechwyconych wyjątków.

14.4.1. Priorytety wątków

Każdy wątek ma swój **priorytet** (ang. *priority*). Domyślnie wątek dziedziczy priorytet po wątku, w którym został utworzony. Aby zmniejszyć lub zwiększyć priorytet wątku, należy użyć metody setPriority. Priorytet wątku może mieć dowolną wartość z przedziału od MIN_PRIORITY (1 w klasie Thread) do MAX_PRIORITY (10 w klasie Thread). Priorytet normalny (NORM_PRIORITY) ma wartość 5.

Kiedy algorytm planujący musi wybrać nowy wątek, decyduje się na ten o najwyższym priorytecie. Należy jednak pamiętać, że priorytety wątków są w dużym stopniu **uzależnione od systemu**. Jeśli maszyna wirtualna korzysta z implementacji wątków leżącej u podłożu platformy, priorytety Javy są odwzorowywane na poziomy platformy, która może dysponować większą lub mniejszą liczbą poziomów priorytetu.

Na przykład system Windows wyróżnia siedem poziomów priorytetu, a więc niektóre z priorytetów Javy zostaną odwzorowane na tym samym poziomie priorytetów systemowych. W maszynie wirtualnej Javy firmy Sun dla systemu Linux priorytety wątków są całkiem ignorowane — wszystkie wątki mają taki sam priorytet.

Początkujący programiści miewają tendencję do nadużywania priorytetów wątków. Należy jednak pamiętać, że powodów do manipulowania nimi jest niewiele. W szczególności nie należy projektować programu w taki sposób, aby jego poprawne funkcjonowanie było zależne od wysokości priorytetów.



Każdy, kto zdecyduje się na użycie priorytetów, powinien wystrzegać się powszechnego błędu popełnianego przez początkujących. Jeśli istnieje kilka wątków o wysokim priorytecie, które nigdy nie są dezaktywowane, wątki o niższych priorytetach mogą **nigdy** nie dojść do głosu. Algorytm planujący, wybierając wątek do uruchomienia, zawsze wybierze jeden spośród tych o najwyższym priorytecie, nawet jeśli ma to oznaczać, że wątki o niższych priorytetach nie będą w ogóle wykonywane.

java.lang.Thread 1.0

■ `void setPriority(int newPriority)`

Ustawia priorytet wątku. Wartość priorytetu musi się mieścić w przedziale od `Thread.MIN_PRIORITY` do `Thread.MAX_PRIORITY`. Normalny priorytet określa wartość `Thread.NORM_PRIORITY`.

■ `static int MIN_PRIORITY`

Najmniejszy priorytet, jaki może mieć wątek. Wartość minimalnego priorytetu to 1.

■ `static int NORM_PRIORITY`

Domyślny priorytet — domyślnie jest to wartość 5.

■ `static int MAX_PRIORITY`

Najwyższy priorytet, jaki może mieć wątek. Maksymalna wartość priorytetu to 10.

■ `static void yield()`

Powoduje ustąpienie aktualnie wykonywanego wątku. Jeśli są jakieś inne wykonywalne wątki o priorytetach nie niższych od tego, zostaną one uruchomione w następnej kolejności. Warto zwrócić uwagę, że jest to metoda statyczna.

14.4.2. Wątki demony

Aby zamienić zwykły wątek w **demona**, należy użyć poniższej instrukcji:

```
t.setDaemon(true);
```

Nie ma w nim jednak nic demonicznego. Demon to taki wątek, którego istnienie polega na służeniu innym wątkom. Należą do nich wątki zegarowe, które wysyłają w równych odstępach czasu tyknięcie zegara do innych wątków, lub takie, które usuwają przestarzałe obiekty z pamięci podręcznej. Jeśli w programie pozostaną same demony, maszyna wirtualna zostaje zamknięta, ponieważ nie ma sensu kontynuować działania programu, w którym nie ma nic poza demonami.

Niektórzy poczynając programiści popełniają błąd używania demonów w celu uniknięcia obsługi zamykania zasobów. Metoda ta może być jednak niebezpieczna. Demon nie powinien nigdy mieć dostępu do zasobów stałych, jak plik czy baza danych, ponieważ może zakończyć działanie w każdej chwili, nawet w środku operacji.

`java.lang.Thread 1.0`

■ `void setDaemon(boolean isDaemon)`

Określa, czy wątek jest demonem, czy zwykłym wątkiem. Wywołanie tej metody musi nastąpić przed uruchomieniem wątku.

14.4.3. Procedury obsługi nieprzechwyconych wyjątków

Metoda `run` wątku nie może zgłaszać wyjątków kontrolowanych, ale jej działanie może zakończyć wyjątek niekontrolowany. W takiej sytuacji wątek zostaje zamknięty.

Nie ma jednak klauzuli `catch`, do której wyjątek taki można było przesłać. Zamiast tego bezpośrednio przed zamknięciem wątku wyjątek jest przekazywany do procedury obsługi nieprzechwyconych wyjątków.

Obiekt ten musi należeć do klasy implementującej interfejs `Thread.UncaughtExceptionHandler`. Interfejs ten posiada jedną metodę:

`void uncaughtException(Thread t, Throwable e)`

Procedurę obsługi w wątku można zainstalować za pomocą metody `setUncaughtExceptionHandler`. Można także dodać procedurę domyślną dla wszystkich wątków za pomocą statycznej metody `setDefaultUncaughtExceptionHandler` z klasy `Thread`. Taka zapasowa procedura mogłaby za pośrednictwem API rejestracyjnego wysyłać raporty o nieprzechwyconych wyjątkach do dziennika.

Jeśli domyślna procedura obsługi nie zostanie zainstalowana, będzie ona `null`. Jeśli jednak zabraknie procedury obsługi dla konkretnego wątku, będzie nią jego obiekt typu `ThreadGroup`.



Grupa to kolekcja wątków, którymi można zarządzać razem. Domyślnie wszystkie tworzone wątki należą do tej samej grupy, ale można założyć inne zgrupowania. Ponieważ teraz są lepsze narzędzia do operowania kolekcjami wątków, nie należy w programach używać grup.

Klasa `ThreadGroup` implementuje interfejs `Thread.UncaughtExceptionHandler`. Znajdująca się w nim metoda `uncaughtException` wykonuje następujące działania:

- Jeśli grupa wątków posiada rodzica, wywoływana jest metoda uncaughtException grupy nadzędnej.
- W przeciwnym razie, jeśli metoda Thread.getDefaultExceptionHandler zwraca procedurę obsługi niebędącą null, metoda uncaughtException zostaje wywołana.
- W przeciwnym razie, jeśli Throwable jest egzemplarzem klasy ThreadDeath, nic się nie dzieje.
- W przeciwnym razie nazwa wątku i dane ze śledzenia stosu zostają wydrukowane w strumieniu System.err.

Dane ze śledzenia stosu z pewnością każdy widział już wiele razy w swoich programach.

java.lang.Thread **1.0**

- static void setDefaultUncaughtExceptionHandler(Thread
↳UncaughtExceptionHandler handler) **5.0**
 - static Thread.UncaughtExceptionHandler
↳getDefaultUncaughtExceptionHandler() **5.0**
- Ustawia lub zwraca domyślną procedurę obsługi dla nieprzechwyconych wyjątków.
- void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) **5.0**
 - Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() **5.0**
- Ustawia lub zwraca procedurę obsługi nieprzechwyconych wyjątków. Jeśli nie ma zainstalowanej procedury, jej funkcję pełni obiekt grupy wątków.

java.lang.Thread.UncaughtExceptionHandler **5.0**

- void uncaughtException(Thread t, Throwable e)

Zapisuje w dzienniku raport, gdy wątek zostanie zamknięty z powodu nieprzechwyconego wyjątku.

Parametry: t Wątek, który został zamknięty z powodu nieprzechwyconego wyjątku
 e Obiekt nieprzechwyconego wyjątku

java.lang.ThreadGroup **1.0**

- void uncaughtException(Thread t, Throwable e)

Wywołuje metodę nadzędnej grupy wątków, jeśli taka istnieje, lub domyślną procedurę obsługi klasy Thread, jeśli istnieje domyślana procedura, lub w przeciwnym przypadku drukuje dane ze śledzenia stosu w standardowym strumieniu błędów (jeśli e jest obiektem typu ThreadDeath, dane ze śledzenia stosu są tłumione; obiekty ThreadDeath są generowane przez odradzaną metodę stop).

14.5. Synchronizacja

W większości aplikacji wielowątkowych znajdujących praktyczne zastosowanie jeden zestaw danych jest współdzielony przez co najmniej dwa wątki. Co się stanie, jeśli dwa wątki mające dostęp do tego samego obiektu wywołają metodę zmieniającą jego stan? Jak pewnie się domyślasz, wątki mogą sobie wzajemnie przeszkadzać. Zależnie od kolejności dostępu do danych, w opisanych wyżej sytuacjach mogą powstawać uszkodzone obiekty. Sytuacje te nazywa się **wyścigami** (ang. *race condition*).

14.5.1. Przykład sytuacji powodującej wyścig

Aby uniknąć uszkodzenia współdzielonych przez wątki danych, trzeba umieć **synchronizować dostęp** do nich. W tym podrozdziale zobaczymy, co się dzieje, jeśli zabraknie synchronizacji. W kolejnym natomiast nauczysz się synchronizować operacje dostępu do danych.

Kolejny przykładowy program jest symulacją banku, w którym założono kilka kont. Przeprowadzamy różne losowe transakcje mające na celu przemieszczenie pieniędzy pomiędzy tymi kontami. Każde konto dysponuje własnym wątkiem. Każda transakcja przelewa losowo określoną ilość pieniędzy z jednego konta na inne losowo wybrane konto.

Kod symulacji jest bardzo prosty. Zawiera klasę Bank z metodą transfer, która przelewa środki pieniężne pomiędzy kontami (na razie nie przejmujemy się tym, że saldo na koncie może stać się ujemne). Oto kod metody transfer z klasy Bank:

```
public void transfer(int from, int to, double amount)
    // Ostrzeżenie: metoda niebezpieczna, jeśli wywoływana w kilku wątkach.
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f z %d na %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
}
```

Poniżej znajduje się kod klasy TransferRunnable. Jej metoda run przelewa pieniądze z określonego konta bankowego. W każdej iteracji metoda ta losowo wybiera jedno konto docelowe i sumę pieniędzy do przelania, wywołuje metodę transfer na rzecz obiektu banku i przechodzi w stan uśpienia.

```
class TransferRunnable implements Runnable
{
    public void run()
    {
        try
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
    }
}
```

```
        }  
    catch(InterruptedException e) {}  
}  
}
```

W żadnym momencie działania symulacji nie wiadomo, ile jest pieniędzy na każdym z kont. Wiadomo natomiast, że ogólna suma nie powinna się zmieniać, ponieważ program tylko przelewa środki pomiędzy różnymi kontami.

Na końcu każdej transakcji metoda transfer oblicza sumę pieniędzy dostępnych na wszystkich kontach i drukuje wynik.

Program ten nigdy się nie kończy. Aby go zamknąć, należy nacisnąć kombinację klawiszy **Ctrl+C**.

Oto typowy wydruk z programu:

```
Thread[Thread-11,5.main] 588.48 z 11 na 44 Saldo ogólne: 100000.00
Thread[Thread-12,5.main] 976.11 z 12 na 22 Saldo ogólne: 100000.00
Thread[Thread-14,5.main] 521.51 z 14 na 22 Saldo ogólne: 100000.00
Thread[Thread-13,5.main] 359.89 z 13 na 81 Saldo ogólne: 100000.00
.
.
.
Thread[Thread-36,5.main] 401.71 z 36 na 73 Saldo ogólne: 99291.06
Thread[Thread-35,5.main] 691.46 z 35 na 77 Saldo ogólne: 99291.06
Thread[Thread-37,5.main] 78.64 z 37 na 3 Saldo ogólne: 99291.06
Thread[Thread-34,5.main] 197.11 z 34 na 69 Saldo ogólne: 99291.06
Thread[Thread-36,5.main] 85.96 z 36 na 4 Saldo ogólne: 99291.06
.
.
.
Thread[Thread-4,5.main] Thread[Thread-33,5.main] 7.31 z 31 na 32 Saldo ogólne
99979.24
627.50 z 4 na 5 Saldo ogólne: 99979.24
```

Jak widać, program zawiera poważny błąd. Przez kilka transakcji saldo łączne wszystkich rachunków wynosi 100 000 dolarów, co jest prawidłową kwotą, zważywszy, że jest 100 kont po 1000 dolarów. Jednak po jakimś czasie saldo ulega nieznacznej zmianie. Błędy w obliczeniach mogą się pojawić na krótko po uruchomieniu programu lub dopiero po dłuższym czasie. Taka sytuacja nie napawa optymizmem i z pewnością nikt nie chciałby złożyć w tym banku swoich ciężko zarobionych pieniędzy.

Listingi od 14.5 do 14.7 przedstawiają kompletny kod źródłowy omawianego programu. Spróbuj odszukać błąd w kodzie, a rozwiązanie zagadki znajdziesz w kolejnym podrozdziale.

Listing 14.5. unsynch/UnsynchBankTest.java

```
package unsynch;

/**
 * Program demonstrujący zniszczenie danych spowodowane dostępem kilku wątków do struktury
 * danych
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class UnsynchBankTest
```

```
{  
    public static final int NACCOUNTS = 100;  
    public static final double INITIAL_BALANCE = 1000;  
  
    public static void main(String[] args)  
    {  
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);  
        int i;  
        for (i = 0; i < NACCOUNTS; i++)  
        {  
            TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);  
            Thread t = new Thread(r);  
            t.start();  
        }  
    }  
}
```

Listing 14.6. unsynch/Bank.java

```
package unsynch;  
  
/**  
 * Bank z kilkoma kontami  
 * @version 1.30 2004-08-01  
 * @author Cay Horstmann  
 */  
public class Bank  
{  
    private final double[] accounts;  
  
    /**  
     * Tworzy bank.  
     * @param n liczba kont  
     * @param initialBalance saldo początkowe na każdym koncie  
     */  
    public Bank(int n, double initialBalance)  
    {  
        accounts = new double[n];  
        for (int i = 0; i < accounts.length; i++)  
            accounts[i] = initialBalance;  
    }  
  
    /**  
     * Przelewa pieniądze pomiędzy kontami.  
     * @param from konto, z którego ma nastąpić przelew  
     * @param to konto, na które mają zostać przelane środki  
     * @param amount kwota do przelania  
     */  
    public void transfer(int from, int to, double amount)  
    {  
        if (accounts[from] < amount) return;  
        System.out.print(Thread.currentThread());  
        accounts[from] -= amount;  
        System.out.printf(" %10.2f z %d na %d", amount, from, to);  
        accounts[to] += amount;  
        System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());  
    }  
}
```

```

    /**
 * Zwraca sumę sald wszystkich kont.
 * @return saldo ogólne
 */
public double getTotalBalance()
{
    double sum = 0;

    for (double a : accounts)
        sum += a;

    return sum;
}

    /**
 * Zwraca liczbę kont w banku.
 * @return liczba kont
 */
public int size()
{
    return accounts.length;
}
}

```

Listing 14.7. unsynch/TransferRunnable.java

```

package unsynch;

/**
 * Obiekt Runnable przelewający pieniądze z jednego konta bankowego na inne
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class TransferRunnable implements Runnable
{
    private Bank bank;
    private int fromAccount;
    private double maxAmount;
    private int DELAY = 10;

    /**
     * Tworzy obiekt Runnable do przelewania środków
     * @param b bank, na którego kontach wykonywany jest przelew
     * @param from konto, z którego mają być przelane pieniądze
     * @param max maksymalna suma, jaka może zostać przelana za każdym razem
     */
    public TransferRunnable(Bank b, int from, double max)
    {
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }

    public void run()
    {
        try
        {

```

```
        while (true)
    {
        int toAccount = (int) (bank.size() * Math.random());
        double amount = maxAmount * Math.random();
        bank.transfer(fromAccount, toAccount, amount);
        Thread.sleep((int) (DELAY * Math.random()));
    }
}
catch (InterruptedException e)
{
}
}
```

14.5.2. Wyścigi

W poprzednim podrozdziale napisaliśmy program, w którym kilka wątków aktualizowało salda na kontach bankowych. Po jakimś czasie wkradał się błąd, który powodował pojawienie się lub zniknięcie pewnej kwoty pieniędzy. Problem ten występował w sytuacjach, w których dwa wątki równocześnie próbowały zaktualizować jedno konto. Wyobraźmy sobie, że dwa wątki w tej samej chwili wykonują poniższą instrukcję:

```
accounts[to] += amount;
```

Problem polega na tym, że nie są to operacje **niepodzielne**. Instrukcja ta może zostać wykonana w następujący sposób:

- 1 Załadowanie accounts[to] do rejestru.
- 2 Dodanie wartości amount.
3. Zapisanie wyniku z powrotem w accounts[to].

Wyobraźmy sobie teraz, że pierwszy z wątków wykonuje dwa pierwsze kroki i zostaje wywłączony. Następnie budzi się drugi wątek, który aktualizuje tę samą pozycję w tablicy accounts. Potem budzi się pierwszy wątek i kończy działanie, wykonując krok trzeci.

Ta czynność wymazuje zmiany dokonane przez drugi wątek, w wyniku czego zmienia się ogólna suma (zobacz rysunek 14.4).

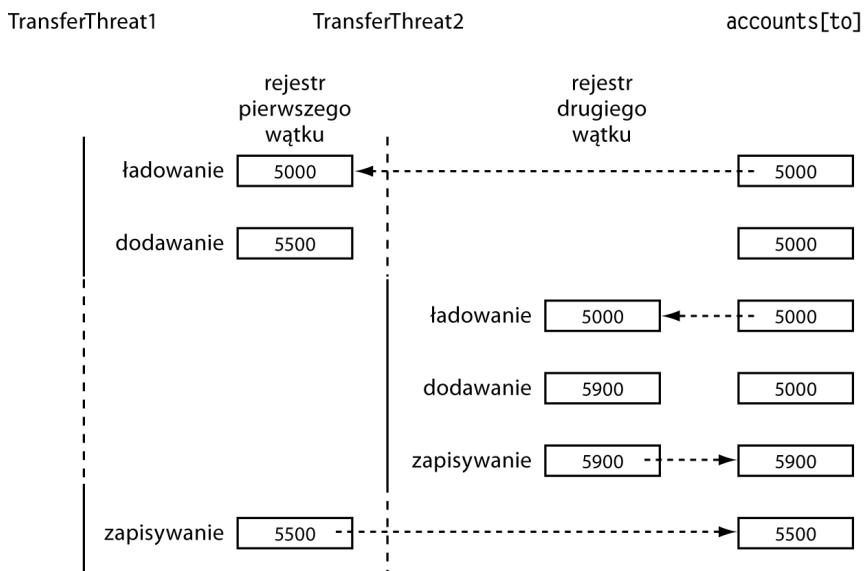
Nasz program testowy wykrywa ten błąd (oczywiście istnieje niewielkie ryzyko fałszywego alarmu, który może nastąpić w sytuacji, gdy zostanie zakłócona praca wątku przeprowadzającego test).

Jakie jest ryzyko wystąpienia tego błędu? Zwiększyliśmy je, przeplatając instrukcje drukowania z instrukcjami aktualizującymi saldo.

Jeśli usuniemy instrukcje drukowania, ryzyko znacznie się zmniejszy, ponieważ każdy wątek przed zaśnięciem będzie wykonywał bardzo mało pracy, a poza tym jest mało prawdopodobne, aby algorytm planujący wywalczył wątek w trakcie wykonywania obliczeń. Nie znaczy to jednak, że ryzyka wystąpienia błędu nie ma już w ogóle. Jeśli na poważnie obciążonej maszynie

Rysunek 14.4.

Jednoczesny
dostęp
przez dwa wątki



Istnieje możliwość podejrzenia kodu bajtowego maszyny wirtualnej wykonującego każdą z instrukcji klasy. W tym celu należy za pomocą poniższego polecenia zdekomplilować plik *Bank.class*:

```
javap -c -v Bank
```

Na przykład kod bajtowy odpowiadający instrukcji `accounts[to] += amount` jest następujący:

```

aload_0
getfield #2;  //Field accounts:/D
iload_2
dup2
daloa
dload_3
dadd
dastore

```

Nieważne, co oznaczają te wszystkie instrukcje. Problem polega na tym, że instrukcja zwiększenia wartości została rozbita na kilka mniejszych instrukcji, a praca wykonującej ją wątku może zostać przerwana w każdym momencie.

uruchomimy bardzo dużo wątków, program nadal będzie robił błędy i nie pomoże usunięcie instrukcji drukujących. Na wystąpienie błędów może przyjść nam czekać kilka minut, godzin, a nawet dni. Szczerze mówiąc, w życiu programisty jest niewiele gorszych rzeczy od błędu, który daje o sobie znać tylko raz na kilka dni.

Istotą problemu jest to, że działanie metody `transfer` może zostać przerwane w środku operacji. Gdybyśmy zapewnili ukończenie metody przed utratą przez wątek kontroli, stan obiektu konta bankowego byłby niezagrożony przez błędy.

14.5.3. Obiekty klasy Lock

Do dyspozycji programistów są dwa mechanizmy służące do ochrony bloków kodu przed jednocześnie dostępnem kilku wątków. Służy do tego słowo kluczowe `synchronized`, a w Java SE 5.0 wprowadzono klasę `ReentrantLock`. Słowo kluczowe `synchronized` automatycznie zakłada blokadę oraz tworzy odpowiadający jej warunek, dzięki czemu jest bardzo pozytycznym i wygodnym w użyciu narzędziem wykorzystywany w większości sytuacji, w których potrzebna jest jawną blokadą. Wydaje nam się jednak, że działanie tego słowa kluczowego łatwiej zrozumieć po zapoznaniu się z blokadami i warunkami osobno. Klasy implementujące te podstawowe funkcje znajdują się w pakiecie `java.util.concurrent`. Opisujemy je poniżej i w podrozdziale 14.5.4, „Warunki”. Po zapoznaniu się z tymi podstawowymi elementami przejdziemy do sekcji 14.5.5, „Słowo kluczowe `synchronized`”.

Szkielet konstrukcji chroniącej blok kodu przy użyciu klasy `ReentrantLock` ma następującą postać:

```
myLock.lock();           // Obiekt klasy ReentrantLock.
try
{
    sekcja krytyczna
}
finally
{
    myLock.unlock();    // Zapewnienie, że blokada zostanie zdjęta, nawet jeśli wystąpi wyjątek.
}
```

Dostęp do sekcji krytycznej powyższej instrukcji w jednym czasie może mieć tylko jeden wątek. Kiedy jeden wątek zablokuje obiekt blokady, żaden inny wątek nie będzie mógł przejść przez instrukcję `lock`. Jeśli jakiś inny wątek wywoła metodę `lock`, zostanie dezaktywowany do czasu, aż poprzedni wątek odblokuje obiekt blokady.



Metoda `unlock` musi się bezwzględnie znajdująć w bloku `finally`. Jeśli kod w sekcji krytycznej spowoduje wyjątek, blokada musi zostać zdjęta. W przeciwnym przypadku reszta wątków pozostanie zablokowana na zawsze.



Z blokadami nie można używać instrukcji `try z zasobami`. Przede wszystkim metoda zdejmująca blokadę nie nazywa się `close`. Jednak nawet gdyby zmieniono jej nazwę, to instrukcja `try z zasobami` i tak by nie działała. W jej nagłówku powinna się znaleźć deklaracja nowej zmiennej, a w blokadzie używa się jednej zmiennej, z której korzystają różne wątki.

Spróbujmy za pomocą blokady ochronić metodę `transfer` z klasy `Bank`.

```
public class Bank
{
    private Lock bankLock = new ReentrantLock(); // Klasa ReentrantLock implementuje
                                                // interfejs Lock.

    public void transfer(int from, int to, int amount)
```

```

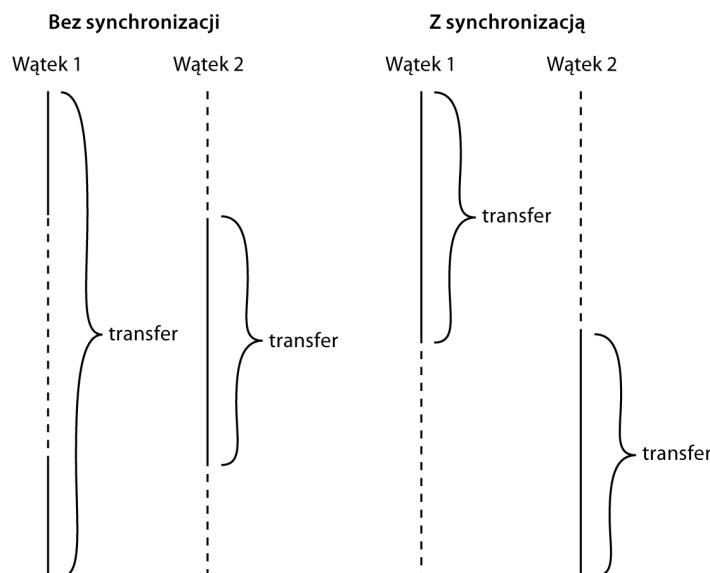
{
    bankLock.lock();
    try
    {
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f z %d na %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
    }
    finally
    {
        bankLock.unlock();
    }
}
}

```

Założymy, że jakiś wątek wywołuje metodę transfer, ale zostaje wywolany przed jej ukończeniem. Następnie inny wątek również wywołuje tę metodę. Nie może on jednak założyć blokady i zostaje zablokowany wywołaniem metody lock. Jest dezaktywowany i musi poczekać, aż pierwszy wątek skończy wykonywanie metody transfer. Kiedy ten zdejmie blokadę, drugi wątek może kontynuować (zobacz rysunek 14.5).

Rysunek 14.5.

Porównanie wątków synchronizowanych i niesynchronizowanych



Wypróbuj, czy to działa. Dodaj kod blokujący do metody transfer i ponownie uruchom program, a przekonasz się, że saldo bankowe nie zmieni się bez względu na długość czasu działania programu.

Należy zauważyć, że każdy obiekt klasy Bank posiada własny obiekt klasy ReentrantLock. Jeśli dwa wątki próbują uzyskać dostęp do tego samego obiektu Bank, blokada ustawia je w kolejce. Jeśli natomiast każdy z wątków dobiera się do innego obiektu Bank, zakładają one osobne blokady i żaden z nich nie jest blokowany. Jest to jak najbardziej prawidłowe działanie, ponieważ wątki działające na różnych obiektach nie mogą sobie przeszkadzać.

Blokada ta jest **wielowejściowa** (ang. *reentrant*), ponieważ wątek może wielokrotnie zakładać blokadę, którą już posiada. Blokada posiada **licznik pamiętający** liczbę zagnieźdzonych wywołań metody `lock`. Dlatego, aby blokada została zwolniona, wątek musi wywołać tyle razy metodę `unlock`, ile razy wywołał metodę `lock`. Dzięki temu kod chroniony przez blokadę może wywołać inną metodę, która wykorzystuje te same blokady.

Na przykład metoda `transfer` wywołuje metodę `getTotalBalance`, która blokuje obiekt `bankLock` mający obecnie licznik o wartości 2. Kiedy metoda `getTotalBalance` kończy działanie, wartość licznika spada do 1. Zakończenie metody `transfer` zmniejsza go do 0 i wątek zwalnia blokadę.

Z reguły ochroną obejmuje się bloki kodu, które aktualizują lub badają współdzielone obiekty. Daje to pewność, że operacja zostanie zakończona, zanim inny wątek będzie mógł użyć tego samego obiektu.



Trzeba uważać, aby procedury zawarte w sekcji krytycznej nie zostały pominięte z powodu wystąpienia wyjątku. Jeśli wyjątek wystąpi przed końcem sekcji, klauzula `finally` zwolni blokadę, ale obiekt może pozostać w naruszonym stanie.

java.util.concurrent.Locks.Lock 5.0

- `void lock()`

Zakłada blokadę. Zostaje zablokowana, jeśli blokada ta jest aktualnie w posiadaniu innego wątku.

- `void unlock()`

Zwalnia blokadę.

java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock()`

Tworzy wielowejściową blokadę, za pomocą której można chronić sekcję krytyczną.

- `ReentrantLock(boolean fair)`

Tworzy obiekt blokady z określona zasadą uczciwości. Uczciwa blokada ustawia na pierwszym miejscu wątek, który czekał najdłużej. Jednak zasada ta może powodować duże straty szybkości. Dlatego domyślnie blokady nie muszą być uczciwe.



Opcja uczciwości wydaje się lepszym rozwiązaniem, ale uczciwe blokady są **znacznie wolniejsze** od zwykłych. Uczciwe blokady należy stosować wyłącznie w sytuacjach, w których takie zachowanie jest całkowicie niezbędne. Stosując uczciwą blokadę, nie ma gwarancji, że algorytm odpowiedzialny za harmonogram uruchamiania wątków również jest uczciwy. Jeśli algorytm ten dyskryminuje wątek, który oczekiwany przez długie czas na blokadę, blokada nie ma szansy potraktować go lepiej.

14.5.4. Warunki

Często zdarza się tak, że po wejściu do sekcji krytycznej wątek dowiaduje się, iż nie może kontynuować, dopóki nie zostanie spełniony warunek. Do zarządzania wątkami, które używały blokadę, ale nie mogą robić nic pozytywnego, służą **obiekty warunków**. W tym rozdziale opisujemy implementację warunków w bibliotece Javy (ze względu na przeszłość obiekty warunków są czasami nazywane **zmiennymi warunkowymi**).

Ulepszymy naszą symulację banku. Nie chcemy, aby pieniędze były przelewane z kont, na których nie ma wystarczających środków. Zauważ, że nie możemy użyć instrukcji jak poniżej:

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount);
```

Jest całkiem możliwe, że aktualny wątek zostanie dezaktywowany pomiędzy pomyślnym wynikiem testu a wywołaniem metody transfer.

```
if (bank.getBalance(from) >= amount)
    // W tym miejscu wątek może być nieaktywny.
    bank.transfer(from, to, amount);
```

Zanim wątek zostanie ponownie uruchomiony, saldo na koncie może spaść poniżej minimalnej potrzebnej kwoty. Należy przypilnować, aby żaden wątek nie zmodyfikował salda pomiędzy testem a wykonaniem przelewu. Dlatego zarówno test, jak i operację przelewu chronimy przy użyciu blokady:

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // czekanie
            .
            .
        }
        // przelew środków
        .
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Kolejna podjęcie decyzji, co zrobić, jeśli na koncie będzie za mało pieniędzy. W takiej sytuacji czekamy, aż jakiś inny wątek zwiększy jego saldo. Pamiętamy jednak, że pierwszy wątek całkowicie zablokował dostęp do obiektu bankLock, przez co żaden inny wątek nie może dokonać depozytu. W takim przypadku do gry wchodzą obiekty warunków.

Z obiektem blokady może być związanych nawet kilka warunków. Obiekty warunków tworzy się za pomocą metody newCondition. Istnieje zwyczaj nadawania obiektom warunków takich

nazw, które w jakiś sposób przypominają reprezentowane warunki. Na przykład w poniższym fragmencie programu tworzymy obiekt warunku reprezentujący warunek wystarczających środków.

```
class Bank
{
    private Condition wystSrodki;
    ...
    public Bank()
    {
        ...
        wystSrodki = bankLock.newCondition();
    }
}
```

Jeśli metoda `transfer` odkryje, że na koncie nie ma dostępnych wystarczających środków, wykonuje instrukcję `wystSrodki.await()`:

Dzięki temu aktualny wątek zostaje dezaktywowany i następuje zdjęcie blokady. To umożliwia działanie kolejnemu wątkowi, który, mamy nadzieję, zwiększy saldo konta.

Pomiędzy wątkiem, który oczekuje na blokadę, a wątkiem, który wywołał metodę `await`, istnieje zasadnicza różnica. Ten drugi zostaje umieszczony w kolejce **wątków oczekujących** (ang. *wait set*) warunku. Wątek ten **nie** przechodzi w stan wykonywalności, dopóki inny wątek nie wywoła metody `signalAll` na rzecz tego samego warunku.

Inny wątek przelewający pieniądze powinien wykonać instrukcję `wystSrodki.signalAll()`:

Powoduje ona reaktywację wszystkich wątków oczekujących na warunek. Wątki usunięte z kolejki oczekujących są z powrotem wykonywalne, a algorytm odpowiedzialny za harmonogram w końcu ponownie je uaktywni. Wtedy spróbują one ponownie wejść do obiektu. Jak tylko będzie dostępna blokada, jeden z wątków ją założy i będzie kontynuował pracę **od momentu, w którym ją przerwał**, wracając z wywołania metody `await`.

W tym momencie wątek powinien ponownie sprawdzić warunek. Nie ma gwarancji, że teraz zostanie on spełniony. Metoda `signalAll` tylko sygnalizuje wątkom, że tym razem warunek **może** zostać spełniony i dlatego dobrze było to sprawdzić.



Ogólnie rzecz biorąc, metoda `await` powinna być wywoływana w pętli o następującej formie:

```
while (!można kontynuować)
    condition.await();
```

Ważne jest, aby metoda `signalAll` była wywoływana także przez jakiś inny wątek, ponieważ wątek wywołujący metodę `await` nie ma możliwości reaktywowania samego siebie. Musi on liczyć na inne wątki. Jeśli żaden z nich go nie reaktywuje, nie zostanie on nigdy więcej uruchomiony. To może prowadzić do nieprzyjemnych **zakleszczeń** (ang. *deadlock*). Jeśli prawie wszystkie wątki zostaną zablokowane, a ostatni aktywny wątek wywoła metodę `await`, nie odblokowując reszty, nie będzie komu zdjąć blokady i program zawiesi się.

Kiedy powinno się wywoływać metodę `signalAll`? Główna reguła nakazuje zrobienie tego zawsze wtedy, gdy stan obiektu zmieni się w taki sposób, który może być korzystny dla wątków oczekujących. Na przykład wątki powinny mieć możliwość sprawdzenia salda na koncie za każdym razem, gdy ulegnie ono zmianie. W naszym przykładowym programie metodę `signalAll` wywołujemy po zakończeniu przelewu pieniędzy.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            wystSrodki.await();
        //przelew środków
        ...
        wystSrodki.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Należy pamiętać, że wywołanie metody `signalAll` nie powoduje natychmiastowej aktywacji oczekującego wątku. Ona tylko odblokowuje oczekujące wątki, aby mogły konkurować o wejście do obiektu po wyjściu przez aktualny wątek z synchronizowanej metody.

Istnieje także metoda `signal`, która odblokowuje tylko jeden losowo wybrany wątek. Jest to mniej obciążająca czynność niż odblokowywanie wszystkich wątków, ale wiąże się z nią pewne ryzyko. Jeśli losowo wybrany wątek „dojdzie do wniosku”, że nadal nie może nic zrobić, zostanie z powrotem zablokowany. Jeśli żaden inny wątek nie wywoła metody `signal` jeszcze jeden raz, system ulegnie zakleszczeniu.



Wątek pozostający w posiadaniu blokady warunku może na jego rzecz wywołać tylko metodę `await`, `signalAll` lub `signal`.

Po uruchomieniu programu przedstawionego na listingu 14.8 widać, że nie ma żadnych błędów w obliczeniach. Saldo ogólne cały czas wynosi 1000 dolarów. Saldo żadnego z kont nigdy nie jest ujemne (przypominamy, że aby zakończyć program, trzeba wcisnąć kombinację klawiszy *Ctrl+C*). Da się również zauważyć, że program działa nieco wolniej — jest to cena, jaką płacimy za synchronizację.

Listing 14.8. `synch/Bank.java`

```
package synch;

import java.util.concurrent.locks.*;

/**
 * Bank z kilkoma kontami, kontrolujący dostęp za pomocą blokad
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
```

```
public class Bank
{
    private final double[] accounts;
    private Lock bankLock;
    private Condition sufficientFunds;

    /**
     * Tworzy bank
     * @param n liczba kont
     * @param initialBalance saldo początkowe na każdym koncie
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        for (int i = 0; i < accounts.length; i++)
            accounts[i] = initialBalance;
        bankLock = new ReentrantLock();
        sufficientFunds = bankLock.newCondition();
    }

    /**
     * Przelewa pieniądze pomiędzy kontami.
     * @param from konto, z którego ma nastąpić przelew
     * @param to konto, na które mają zostać przelane środki
     * @param amount kwota do przelania
     */
    public void transfer(int from, int to, double amount) throws InterruptedException
    {
        bankLock.lock();
        try
        {
            while (accounts[from] < amount)
                sufficientFunds.await();
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f z %d na %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
            sufficientFunds.signalAll();
        }
        finally
        {
            bankLock.unlock();
        }
    }

    /**
     * Zwraca sumę sald wszystkich kont.
     * @return saldo ogólne
     */
    public double getTotalBalance()
    {
        bankLock.lock();
        try
        {
            double sum = 0;
```

```

        for (double a : accounts)
            sum += a;

        return sum;
    }
    finally
    {
        bankLock.unlock();
    }
}

/**
 * Zwraca liczbę kont w banku.
 * @return liczba kont
 */
public int size()
{
    return accounts.length;
}
}

```

Poprawne zastosowanie warunków w praktyce może być sporym wyzwaniem. Przed podjęciem próby zaimplementowania własnych obiektów warunków dobrze by było najpierw wziąć pod uwagę jedną z konstrukcji opisanych w podrozdziale 14.10, „Synchronizatory”.

java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`

Zwraca obiekt warunku związanego z blokadą.

java.util.concurrent.locks.Condition 5.0

- `void await()`

Umieszcza wątek w kolejce oczekujących do warunku.

- `void signalAll()`

Odblokowuje wszystkie wątki znajdujące się w kolejce oczekujących do warunku.

- `void signal()`

Odblokowuje losowo wybrany wątek znajdujący się w kolejce oczekujących do warunku.

14.5.5. Słowo kluczowe `synchronized`

W poprzednich podrozdziałach nauczyliśmy się używać obiektów typu `Lock` i `Condition`. Zanim przejdziemy dalej, zrobimy podsumowanie najważniejszych wiadomości na temat blokad i warunków:

- Blokada chroni blok kodu, pozwalając wykonywać go tylko jednemu wątkowi w danym czasie.

- Blokada zarządza wątkami, które próbują wejść do chronionego segmentu kodu.
- Z blokadą może być związany jeden lub więcej obiektów warunkowych.
- Każdy obiekt warunkowy zarządza wątkami, które weszły do sekcji kodu chronionego, ale które nie mogą kontynuować działania.

Interfejsy Lock i Condition umożliwiają programistom zyskanie większej kontroli nad blokadami. W większości sytuacji kontrola ta jest jednak zbędna, ponieważ można wykorzystać mechanizm wbudowany w język. Od wersji 1.0 każdy obiekt w Javie posiada **blokadę wewnętrzną**. Jeśli w deklaracji metody zostanie użyte słowo kluczowe synchronized, blokada obiektu chroni całą tę metodę. To znaczy, że aby ją wywołać, wątek musi założyć wewnętrzną blokadę obiektu.

Innymi słowy, poniższy kod:

```
public synchronized void method()
{
    ciało metody
}
```

jest równoważny z tym:

```
public void method()
{
    this.intrinsicLock.lock();
    try
    {
        ciało metody
    }
    finally { this.intrinsicLock.unlock(); }
}
```

Na przykład zamiast stosować blokadę jawną, wystarczy zadeklarować metodę transfer z klasy Bank jako synchronizowaną (synchronized).

Z wewnętrzną blokadą obiektu jest związany jeden warunek. Metoda wait dodaje wątek do kolejki oczekujących, a metody notifyAll i notify odblokowują oczekujące wątki. Innymi słowy, wywołanie metody wait lub notifyAll jest równoznaczne z poniższym:

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```



Metody wait, notifyAll i notify są metodami finalnymi klasy Object. Aby uniknąć konfliktów nazw, odpowiadające im metody w interfejsie Condition zostały nazwane await, signalAll i signal.

Na przykład implementacja klasy Bank może wyglądać następująco:

```
class Bank
{
    private double[] accounts;

    public synchronized void transfer(int from, int to, int amount) throws
        →InterruptedException
```

```

{
    while (accounts[from] < amount)
        wait();           // Oczekiwanie na warunek wewnętrznej blokady obiektu.
    accounts[from] -= amount;
    accounts[to] += amount;
    notifyAll();    // Powiadomienie wszystkich wątków oczekujących na warunek.
}
public synchronized double getTotalBalance() { . . . }
}

```

Jak widać, słowo kluczowe `synchronized` pozwala na pisanie znacznie bardziej zwięzłego kodu. Oczywiście, aby go zrozumieć, trzeba wiedzieć, że każdy obiekt posiada wewnętrzną blokadę, która z kolei posiada wewnętrzny warunek. Blokada zarządza wątkami, które próbują wejść do metody synchronizowanej. Warunek zajmuje się wątkami, które wywołyły metodę `wait`.



Metody synchronizowane są względnie proste. Jednak początkujący często szarpią się z warunkami. Przed przejściem do używania metod `wait` i `notifyAll` lepiej zastanowić się nad użyciem jednej z konstrukcji opisanych w podrozdziale 14.10, „Synchronizatory”.

Synchronizowane mogą być także metody statyczne. Jeśli taka metoda zostanie wywołana, uzyskuje dostęp do blokady wewnętrznej obiektu związanego z nią klasy. Jeśli na przykład klasa `Bank` zawierałaby statyczną metodę synchronizowaną, blokada obiektu `Bank.class` byłaby blokowana w chwili wywołania tej metody. W wyniku tego żaden inny wątek nie mógłby wywołać tej ani żadnej innej statycznej metody synchronizowanej tej klasy.

Wewnętrzne blokady i warunki mają pewne ograniczenia. Oto niektóre z nich:

- Nie można przerwać wątku, który próbuje założyć blokadę.
- Nie można określić maksymalnego czasu próby dostępu do blokady.
- Sytuacja, w której na jedną blokadę przypada jeden warunek, może nie być najlepsza pod kątem wydajności.

Czego najlepiej używać — obiektów `Lock` i `Condition` czy metod synchronizowanych? Oto nasze zalecenia w tej kwestii:

- Najlepiej nie używać interfejsów `Lock` i `Condition` ani słowa kluczowego `synchronized`. W wielu sytuacjach można poradzić sobie przy użyciu jednego z mechanizmów z pakietu `java.util.concurrent`, które zajmują się działaniami związanymi z blokowaniem. Na przykład w podrozdziale 14.6, „Kolejki blokujące”, opisujemy sposób synchronizacji wątków pracujących nad wspólnym zadaniem za pomocą blokowania kolejek.
- Jeśli słowo kluczowe `synchronized` sprawdza się w określonej sytuacji, należy go użyć. Technika ta pozwala na zmniejszenie liczby wierszy kodu i pozostawia mniej okazji do popełnienia błędu. Listing 14.9 przedstawia program symulujący bank zaimplementowany przy użyciu metod synchronizowanych.
- Interfejsów `Lock` i `Condition` należy używać, gdy nie można się obyć bez dodatkowych funkcji, które one udostępniają.

Listing 14.9. synch2/Bank.java

```
package synch2;

/**
 * Bank z kilkoma kontami, wykorzystujący synchronizację
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
    private final double[] accounts;

    /**
     * Tworzy bank.
     * @param n liczba kont
     * @param initialBalance saldo początkowe na każdym koncie
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        for (int i = 0; i < accounts.length; i++)
            accounts[i] = initialBalance;
    }

    /**
     * Przelewa pieniądze pomiędzy kontami.
     * @param from konto, z którego ma nastąpić przelew
     * @param to konto, na które mają zostać przelane środki
     * @param amount kwota do przelania
     */
    public synchronized void transfer(int from, int to, double amount) throws
        InterruptedException
    {
        while (accounts[from] < amount)
            wait();
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f z %d na %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
        notifyAll();
    }

    /**
     * Zwraca sumę sald wszystkich kont.
     * @return saldo ogólne
     */
    public synchronized double getTotalBalance()
    {
        double sum = 0;

        for (double a : accounts)
            sum += a;
```

```

        return sum;
    }

    /**
     * Zwraca liczbę kont w banku.
     * @return liczba kont
     */

    public int size()
    {
        return accounts.length;
    }
}

```

java.lang.Object 1.0

■ void notifyAll()

Odblokowuje wszystkie wątki, które wywołały metodę `wait` na rzecz obiektu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

■ void notify()

Odblokowuje jeden losowo wybrany wątek spośród tych, które wywołały metodę `wait` na rzecz obiektu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

■ void wait()

Przestawia wątek w stan oczekiwania, aż nadjejdzie odpowiednie powiadomienie. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

■ void wait(long millis)

■ void wait(long millis, int nanos)

Przestawia wątek w stan oczekiwania, aż nadjejdzie odpowiednie powiadomienie lub upłynie określona ilość czasu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

Parametry:	millis	Liczba milisekund
	nanos	Liczba nanosekund; musi być mniejsza od 1 000 000

14.5.6. Bloki synchronizowane

Jak już wiemy, każdy obiekt w Javie posiada blokadę. Wątek może ją przejąć za pomocą metody synchronizowanej. Istnieje jeszcze jeden sposób na pozyskiwanie blokad, który polega na wejściu do **bloku synchronizowanego**. Wątek, który wejdzie do bloku podobnego do tego poniżej, stanie się właścicielem blokady dla obiektu obj.

```
synchronized (obj)          // składnia bloku synchronizowanego
{
    sekcja krytyczna
}
```

Czasami można spotkać blokady tworzone ad hoc, na przykład:

```
public class Bank
{
    private double[] accounts;
    private Object lock = new Object();

    public void transfer(int from, int to, int amount)
    {
        synchronized (lock) // blokada utworzona ad hoc
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
        System.out.println(. . .);
    }
}
```

W tym przypadku obiekt lock został utworzony tylko po to, aby można było użyć blokady, którą posiada każdy obiekt w Javie.

Czasami programiści wykorzystują blokady obiektów do implementacji dodatkowych niepodzielnych operacji. Technika ta nazywa się **blokowaniem po stronie klienta** (ang. *client-side locking*). Weźmy na przykład klasę Vector implementującą listy, których metody są synchronizowane. Wyobraźmy sobie, że salda kont w naszym banku zapisaliśmy w liście Vector<Double>. Oto naiwna implementacja metody transfer:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) // błąd
{
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
    System.out.println(. . .);
}
```

Mimo że metody get i set klasy Vector są synchronizowane, nic nam to nie daje. Istnieje możliwość, że wątek zostanie wywieszony w metodzie transfer po zakończeniu pierwszego wywołania metody get. Wtedy inny wątek może w tym samym miejscu zapisać całkiem inną wartość. Można jednak przejąć blokadę:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
    }
}
```

```

        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println(. . .);
}

```

Technika ta zdaje egzamin, ale jest w pełni uzależniona od tego, że wszystkie metody modyfikujące w klasie `Vector` mają wewnętrzne blokady. Czy tak jest jednak naprawdę? W dokumentacji tej klasy nic takiego nie napisano. Trzeba bardzo uważnie przestudiować jej kod źródłowy i mieć nadzieję, że w przyszłości nie zostaną wprowadzone mutatory niesynchronizowane. Stanowi to dowód na to, że blokowanie po stronie klienta jest bardzo niepewną techniką, i dlatego ogólnie nie polecamy jej stosowania.

14.5.7. Monitor

Blokady i warunki stwarzają bardzo duże możliwości, jeśli chodzi o synchronizację wątków, ale są mało obiektowe. Badacze przez wiele lat poszukiwali sposobów na uczynienie wielowątkowości bezpieczną techniką, nie zmuszając jednocześnie programistów do zajmowania się jawnymi blokadami. Jednym z najlepszych rozwiązań w tej dziedzinie są **monitory**, opracowane w latach 70. ubiegłego wieku przez Pera Brincha Hansena i Tony'ego Hoare'a. W Javie monitor ma następujące własności:

- Monitorem jest klasa posiadająca same pola prywatne.
- Z każdym obiektem tej klasy związana jest blokada.
- Wszystkie metody są blokowane przez tę blokadę. Innymi słowy, jeśli klient wykona instrukcję `obj.method()`, to blokada obiektu `obj` zostanie automatycznie założona na początku wywołania metody i zwolniona w chwili zwrócenia przez tę metodę wartości. Ponieważ wszystkie pola są prywatne, mamy pewność, że podczas gdy jeden wątek wykonuje działania na nich, żaden inny wątek nie ma do nich dostępu.
- Z blokadą może być skojarzona dowolna liczba warunków.

Poprzednie wersje monitorów dysponowały tylko jednym warunkiem o dosyć eleganckiej składni. Można było użyć wywołania typu `await accounts[from] >= balance` bez stosowania jawniej zmiennej warunkowej. Badania wykazały jednak, że masowe powtarzanie testów warunków może być bardzo mało wydajne. Problem ten rozwiązują jawnie zmienne warunkowe, z których każda zarządza osobnym zestawem wątków.

Projektanci Javy luźno potraktowali adaptację koncepcji monitorów. **Każdy obiekt** w Javie posiada wewnętrzną blokadę i wewnętrzny warunek. Jeśli w deklaracji metody znajduje się słowo kluczowe `synchronized`, działa ona jak metoda monitorowa. Dostęp do zmiennej warunkowej uzyskuje się za pośrednictwem metod `wait`, `notifyAll` i `notify`.

Pomiędzy zwykłym obiektem a monitorem są trzy istotne różnice mające wpływ na bezpieczeństwo wątków:

- Pola nie muszą być prywatne.
- Metody nie muszą być synchronizowane.
- Blokada wewnętrzna jest dostępna dla klientów.

Ten brak poszanowania dla zabezpieczeń rozwścieczył Pera Brincha Hansena. W zjadliwej recenzji na temat wielowątkowości w Javie napisał: „Zdumiewa fakt, że pozbawiony zabezpieczeń parallelizm w Javie jest poważnie traktowany przez programistów, zwłaszcza że od wynalezienia monitorów i powstania języka Concurrent Pascal upłynęło już pół wieku. To nie ma sensu” (*Java's Insecure Parallelism*, „ACM SIGPLAN Notices” 1999, nr 34, s. 38 – 45).

14.5.8. Pola ulotne

Czasami wydaje się, że obciążenie powodowane przez synchronizację jest zbyt duże, jeśli chcemy tylko odczytać lub zapisać jedno czy dwa pola. Co w takiej sytuacji może się nie udać? Niestety nowoczesne procesory i kompilatory stwarzają mnóstwo sytuacji, w których może zostać popełniony błąd.

- Komputery wieloprocesorowe mogą tymczasowo przechowywać wartości w rejestrach lub lokalnych pamięciach podręcznych. W wyniku tego wątki działające na różnych procesorach mogą w tej samej lokalizacji widzieć inne dane!
- Kompilatory mogą zmieniać kolejność instrukcji w celu zoptymalizowania programu. Kompilator nie zmieni kolejności w taki sposób, aby zmienił się sposób działania kodu, ale wyjdzie z założenia, że wartości w pamięci ulegają zmianom tylko w wyniku konkretnych instrukcji w kodzie. Jednak wartość może zostać zmodyfikowana przez inny wątek!

Problemy te nie występują po zastosowaniu blokad do ochrony kodu, do którego mogą mieć dostęp różne wątki. Kompilatory muszą respektować blokady poprzez opróżnianie w razie potrzeby lokalnych pamięci podręcznych i nieprzestawianie instrukcji w nieodpowiedni sposób. Szczegółowe informacje na ten temat można znaleźć w dokumencie *Java Memory Model and Thread Specification* opracowanym przez grupę JSR 133 (<http://www.jcp.org/en/jsr/detail?id=133>). Znaczną część tej specyfikacji jest bardzo skomplikowana i ma czysto techniczny charakter, ale jest kilka jaśniejszych fragmentów. Bardziej przystępny artykuł na ten temat napisany przez Briana Goetza znajduje się pod adresem <http://www-106.ibm.com/developerworks/java/library/j-jtp02244.html>.



Brian Goetz jest autorem „motta synchronizacyjnego”: jeśli zapisujesz zmienną, która może następnie zostać odczytana przez inny wątek, albo odczytujesz zmienną, która mogła zostać zapisana przez inny wątek, musisz skorzystać z synchronizacji.

Słowo kluczowe *volatile* (ulotny) umożliwia synchronizację dostępu do pól egzemplarza bez użycia blokad. Jeśli w deklaracji pola znajduje się to słowo kluczowe, kompilator i maszyna wirtualna wiedzą, że może ono być współbieżnie modyfikowane przez inny wątek.

Wyobraźmy sobie na przykład, że pewien obiekt ma znacznik logiczny *done*, który jest ustawiany przez jeden wątek, a sprawdzany przez inny. Zgodnie z wcześniejszymi informacjami wiemy, że można w tej sytuacji użyć blokady:

```
private boolean done;
```

```
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

Użycie wewnętrznej blokady wydaje się niezbyt dobrym pomysłem. Metody `isDone` i `setDone` mogą zostać zablokowane, jeśli inny wątek zablokuje obiekt. Jeśli istnieje taka obawa, można zastosować osobną blokadę tylko dla tej zmiennej. To jednak zaczyna robić się coraz bardziej kłopotliwe.

Rozsądny wyjściem z tej sytuacji jest zadeklarowanie pola jako `volatile`:

```
private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```



Zmienne ulotne nie zapewniają niepodzielności. Na przykład nie ma gwarancji, że poniższa metoda zmieni wartość pola na przeciwną.

```
public void flipDone() { done = !done; } //podzielna
```

14.5.9. Zmienne finalne

Jak wiesz z poprzedniego podrozdziału, nie można bezpiecznie odczytać pola w wielu wątkach, jeśli nie użyje się blokad lub modyfikatora `volatile`.

Jest jeszcze jedna sytuacja, w której można bezpiecznie uzyskać dostęp do wspólnego pola — gdy pole to jest `finalne`:

```
final Map<String, Double> accounts = new HashMap<>();
```

Inne wątki zobaczą zmienną `accounts`, gdy konstruktor zakończy działanie.

Bez modyfikatora `final` nie byłoby gwarancji, że wątki zobaczą zaktualizowaną wartość zmiennej `accounts` — mogłyby widzieć `null` zamiast utworzonej struktury `HashMap`.

Oczywiście działania na mapie nie są bezpieczne wątkowo. Jeśli ma ją modyfikować i odczytywać kilka wątków, to nadal konieczna jest synchronizacja.

14.5.10. Zmienne atomowe

Wspólne zmienne można deklarować jako ulotne, pod warunkiem że jedyną wykonywaną operacją będzie przypisanie.

W pakiecie `java.util.concurrent.atomic` znajduje się kilka klas, które zapewniają atomowość innych operacji dzięki wykorzystaniu wydajnych instrukcji na poziomie maszynowym. Przykładowo klasa `AtomicInteger` zawiera metody `incrementAndGet` i `decrementAndGet`, które automatycznie inkrementują i dekrementują liczby całkowite. Można bezpiecznie używać obiektów klasy `AtomicInteger` jako wspólnych liczników bez stosowania synchronizacji.

Dostępne są też klasy `AtomicBoolean`, `AtomicLong` i `AtomicReference` oraz atomowe tablice wartości logicznych, liczb całkowitych, wartości długich i referencji. Ich użycie należy pozostawić programistom systemów, którzy programują narzędzia współbieżności. Programiści aplikacji nie powinni ich używać.

14.5.11. Zakleszczenia

Blokady i warunki nie wystarczą do rozwiązania wszystkich problemów związanych z wielowątkowością. Rozważmy następującą sytuację:

Konto 1: 200 dol.

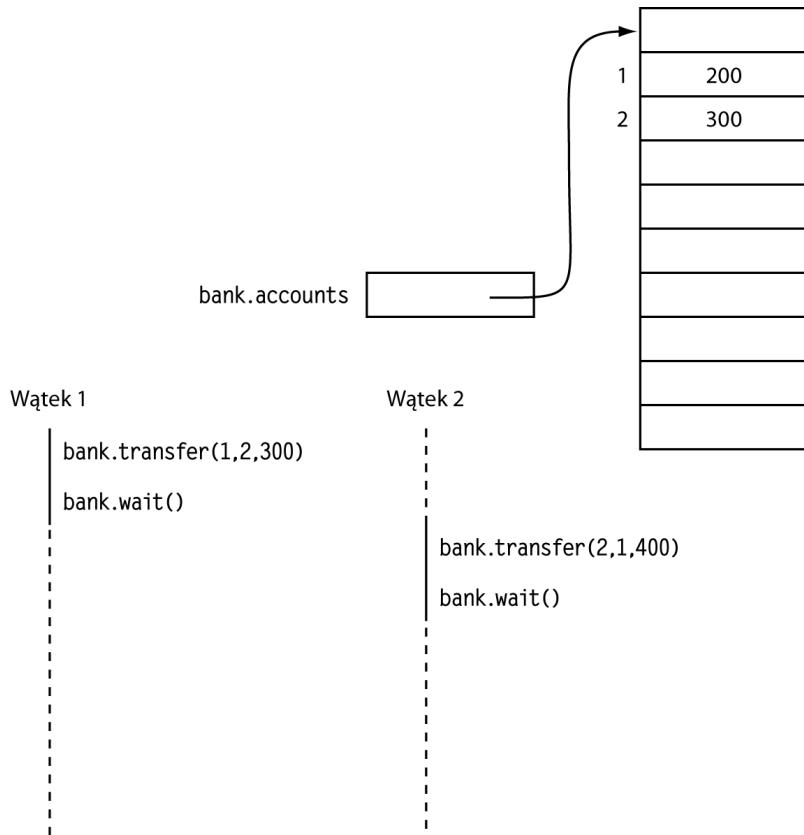
Konto 2: 300 dol.

Wątek 1: przelew 300 dol. z konta 1 na konto 2

Wątek 2: przelew 400 dol. z konta 2 na konto 1

Jak widać na rysunku 14.6, wątki 1 i 2 są zablokowane. Żaden z nich nie może kontynuować działania, ponieważ salda na obu kontach są zbyt niskie.

Rysunek 14.6.
Zakleszczenie



Sytuacja, w której wszystkie wątki są zablokowane, w tym przypadku (ponieważ każdy czeka na więcej pieniędzy) nazywa się **zakleszczeniem** (ang. *deadlock*).

W naszym programie zakleszczenie nie może wystąpić z prostego powodu. Kwota przelewu nie może przekraczać 1000 dolarów. Ponieważ jest sto kont, na których w sumie znajduje się 100 000 dolarów, przynajmniej jedno z nich zawsze musi zawierać więcej niż 1000 dolarów. Dzięki temu wątek przelewający pieniądze z tego konta może kontynuować działanie.

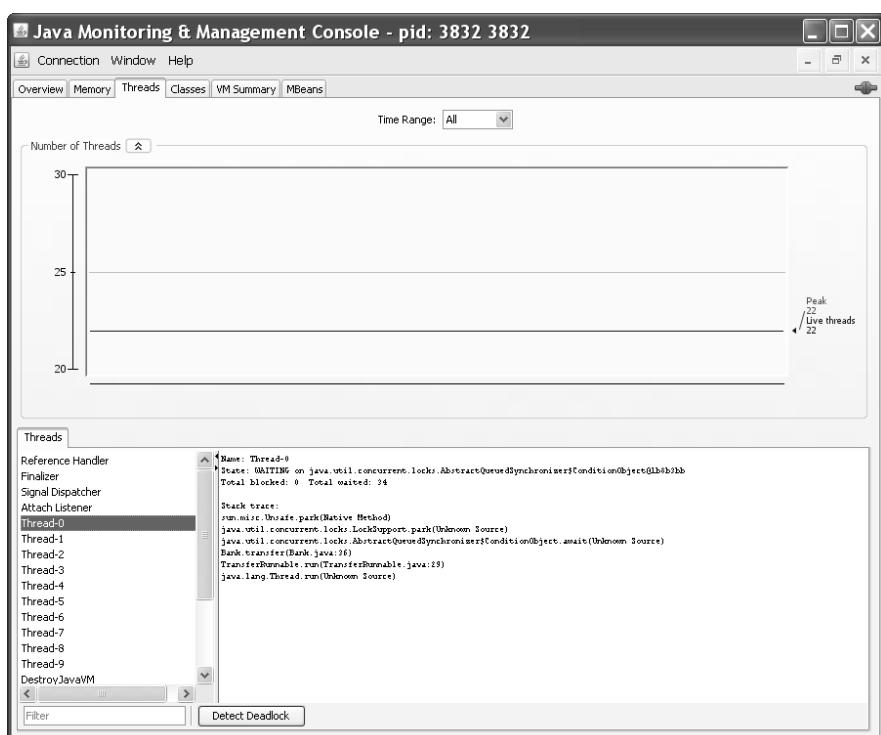
Jeśli jednak z metody `run` usuniemy ograniczenie wysokości transakcji do 1000 dolarów, zakleszczenia mogą nastąpić bardzo szybko. Można to sprawdzić. Ustaw wartość stałej `NACCOUNTS` na 10, a parametr `max` konstruktora klasy `TransferRunnable` na `2 * INITIAL_BALANCE` i uruchom program. Podzieli on przez jakiś czas i zawiesi się.



Kiedy program zawiesi się, naciśnij kombinację klawiszy `Ctrl+Shift+Esc`, aby otrzymać listę wszystkich wątków. Każdemu wątkowi towarzyszy informacja ze stosu, dzięki czemu wiadomo, w którym miejscu jest on aktualnie zablokowany. Można też użyć narzędzia `jconsole`, które zostało opisane w rozdziale 11., i sprawdzić dane na karcie `Threads` (rysunek 14.7).

Rysunek 14.7.

Karta Threads w jconsole



Innym sposobem na spowodowanie zakleszczenia jest uczynienie i -tego wątku odpowiedzialnym za umieszczenie pieniędzy na i -tym koncie zamiast za pobranie ich z i -tego konta. Istnieje wtedy szansa, że wszystkie wątki zbiegną się nad jednym kontem i będą próbować usunąć z niego więcej pieniędzy, niż się na nim znajduje. Aby to wypróbować, należy w programie

SynchBankTest znaćć metodę run w klasie TransferRunnable. W wywołaniu metody transfer zamień miejscami parametry fromAccount i toAccount. Uruchom program i przekonaj się, że prawie natychmiast nastąpi zakleszczenie.

Oto jeszcze jedna sytuacja, w której może łatwo dojść do zakleszczenia: w programie SynchBankTest zamień metodę signalAll na signal. Po pewnym czasie program w końcu zawiesza się (tym razem także lepiej ustawić wartość NACCOUNTS na 10, aby efekt był szybciejauważalny). W przeciwnieństwie do metody signalAll, która wysyła powiadomienie do wszystkich wątków oczekujących na zwiększenie funduszy, metoda signal odblokowuje tylko jeden wątek. Jeśli wątek ten nie może kontynuować, wszystkie wątki mogą zostać zablokowane. Przeanalizujmy scenariusz prezentujący, jak może dojść do zakleszczenia.

Konto 1: 1990 dol.

Pozostałe konta: po 990 dol.

Wątek 1: przelew 995 dol. z konta 1 na konto 2

Pozostałe wątki: przelew 995 dol. ze swoich kont na inne konto

Bez wątpienia wszystkie wątki poza pierwszym są zablokowane, ponieważ na ich kontach nie ma wystarczająco dużo pieniędzy.

Wątek 1 kontynuuje działanie. Dochodzimy do następującej sytuacji:

Konto 1: 995 dol.

Konto 2: 1985 dol.

Pozostałe konta: po 990 dol.

Wtedy wątek 1 wywołuje metodę signal, która odblokowuje jeden losowo wybrany wątek. Założymy, że padło na wątek 3. Zostaje on obudzony, stwierdza, że na jego koncie nie ma wystarczająco dużo środków, i ponownie wywołuje metodę wait. Jednak wątek 1 cały czas działa. Generuje nową losową transakcję, na przykład taką jak poniżej:

Wątek 1: przelew 997 dol. z konta 1 na konto 2

Tym razem także wątek 1 wywołuje metodę wait i **wszystkie** wątki są zablokowane. System ulega zakleszczeniu.

Źródłem problemów jest tutaj metoda signal odblokowująca tylko jeden wątek, który może nie mieć znaczenia dla utrzymania postępu programu (w naszym scenariuszu wątek 2 musi kontynuować pracę, aby pobrać pieniądze z drugiego konta).

Niestety język Java nie udostępnia żadnego mechanizmu pozwalającego uniknąć takich zakleszczeń lub je przerwać. Program musi być tak zaprojektowany, aby sytuacje tego typu nie miały szans się wydarzyć.

14.5.12. Zmienne lokalne wątków

W poprzednich podrozdziałach zostały opisane problemy, jakie można napotkać, używając w wątkach wspólnych zmiennych. Czasami można uniknąć współdzielienia, każdemu wątkowi dając egzemplarz na własność. Używa się do tego celu klasy pomocniczej `ThreadLocal`. Przykładowo klasa `SimpleDateFormat` nie jest bezpieczna wątkowo. Przypuśćmy, że mamy stałą zmienną:

```
public static final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
```

Jeśli dwa wątki wykonają taką operację jak poniższa:

```
String dateStamp = dateFormat.format(new Date());
```

to wynik może być niepoprawny, ponieważ wewnętrzne struktury danych używane przez `dateFormat` mogą zostać uszkodzone przez wspólnie dostępny dostęp. Jako rozwiązanie można zastosować synchronizację, która jest kosztowna, albo tworzyć lokalny obiekt `SimpleDateFormat` za każdym razem, gdy jest potrzebny. To jednak również oznacza marnotrawstwo zasobów.

Aby utworzyć po jednym egzemplarzu na wątek, należy użyć poniższego kodu:

```
public static final ThreadLocal<SimpleDateFormat> dateFormat =
    new ThreadLocal<SimpleDateFormat>()
{
    protected SimpleDateFormat initialValue()
    {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

Aby uzyskać dostęp do formatera, należy zastosować poniższe wywołanie:

```
String dateStamp = dateFormat.get().format(new Date());
```

Przy pierwszym wywołaniu metody `get` w wątku następuje wywołanie metody `initialValue`. Od tej pory metoda `get` zwraca egzemplarz należący do bieżącego wątku.

Podobny problem przedstawia generowanie liczb losowych w wielu wątkach. Klasa `java.util.Random` jest bezpieczna wątkowo, ale mimo to nie działa wydajnie, gdy wiele wątków musi czekać na jeden wspólny generator.

Można by było użyć klasy pomocniczej `ThreadLocal`, aby każdemu wątkowi dać osobny generator, ale w Java SE 7 udostępniono specjalną klasę, dzięki której praca ta jest wygodniejsza. Wystarczy wykonać poniższe wywołanie:

```
int random = ThreadLocalRandom.current().nextInt(upperBound);
```

Metoda `ThreadLocalRandom.current()` zwraca egzemplarz klasy `Random` dostępny tylko dla bieżącego wątku.

java.lang.ThreadLocal<T> 1.2

- `T get()`
Pobiera bieżącą wartość wątku. Jeśli metoda `get` jest wywoływana po raz pierwszy, wartość otrzymywana jest poprzez wywołanie metody `initialize`.
- `protected initialize()`
Metodę tę należy przesłonić, aby dostarczała wartość domyślną. Domyślnie zwraca `null`.
- `void set(T t)`
Ustawia nową wartość wątku.
- `void remove()`
Usuwa wartość wątku.

java.util.concurrent.ThreadLocalRandom 7

- `static ThreadLocalRandom current()`
Zwraca egzemplarz klasy `Random` należący do bieżącego wątku.

14.5.13. Testowanie blokad i odmierzanie czasu

Jeśli wątek wywoła metodę `lock` w celu uzyskania blokady będącej w posiadaniu innego wątku, zostaje zablokowany na nieokreśloną ilość czasu. Przy zakładaniu blokady można zachować większą ostrożność. Metoda `tryLock` próbuje założyć blokadę i jeśli operacja ta zakończy się powodzeniem, zwraca wartość `true`. W przeciwnym przypadku zwraca wartość `false`, a wątek może wykonywać jakieś inne działania.

```
if (myLock.tryLock())
    // Wątek jest w posiadaniu blokady.
    try { . . . }
    finally { myLock.unlock(); }
else
    // Wątek przechodzi do innych działań.
```

Metodę `tryLock` można wywołać z parametrem czasowym:

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

`TimeUnit` to wyliczenie zawierające wartości `SECONDS`, `MILLISECONDS`, `MICROSECONDS` i `NANOSECONDS`.

Metody `lock` nie można przerwać. Jeśli wątek oczekujący na blokadę zostanie przerwany, pozostaje on zablokowany, dopóki metoda `lock` nie będzie dostępna. Jeśli wystąpi zakleszczenie, metoda `lock` może nigdy nie zakończyć działania.

Jeśli natomiast metoda `tryLock` zostanie wywołana z parametrem czasowym, przerwanie oczekującego wątku spowoduje wygenerowanie wyjątku `InterruptedException`. Jest to niewątpliwie pozytywna funkcja, ponieważ pozwala przerywać zakleszczenia.

Można także wywołać metodę `lockInterruptibly`. Działa ona tak samo jak `tryLock`, tylko bez ograniczenia czasowego.

Ograniczyć czasowo można także oczekiwanie na warunek:

```
myCondition.await(100, TimeUnit.MILLISECONDS)
```

Metoda `await` zwraca kontrolę, jeśli inny wątek aktywuje ten wątek za pomocą metody `signalAll` lub `signal`, jeśli minie określony czas bądź gdy nastąpi przerwanie tego wątku.

Jeśli oczekujący wątek zostanie przerwany, metoda `await` zgłasza wyjątek `InterruptedException`. W (mało prawdopodobnej) sytuacji, w której lepiej kontynuować czekanie, należy w zamian użyć metody `awaitInterruptibly`.

java.util.concurrent.locks.Lock 5.0

- `boolean tryLock()`

Próbuje założyć blokadę, nie blokując wątku. Jeśli operacja zakończy się powodzeniem, zwraca wartość `true`. Metoda ta przejmuje blokadę, jeśli jest ona dostępna, bez względu na zasadę uczciwości i inne oczekujące wątki.

- `boolean tryLock(long time, TimeUnit unit)`

Próbuje założyć blokadę, blokując wątek przez czas nie dłuższy od określonego. W razie powodzenia zwraca wartość `true`.

- `void lockInterruptibly()`

Zajmuje blokadę i blokuje wątek na nieokreślony czas. Jeśli wątek zostanie przerwany, zgłasza wyjątek `InterruptedException`.

java.util.concurrent.locks.Condition 5.0

- `boolean await(long time, TimeUnit unit)`

Wchodzi do kolejki oczekujących na warunek, blokując wątek do czasu, aż zostanie on usunięty z kolejki lub minie określona ilość czasu. Zwraca wartość `false`, jeśli metoda zakończy działanie z powodu upłynięcia określonego czasu, lub `true` w przeciwnym przypadku.

- `void awaitUninterruptibly()`

Wchodzi do kolejki oczekujących na warunek, blokując wątek do czasu, aż zostanie on usunięty z kolejki. Jeśli wątek zostanie przerwany, metoda ta nie zgłasza wyjątku `InterruptedException`.

14.5.14. Blokady odczytu-zapisu

W pakiecie `java.util.concurrent.locks` znajdują się dwie klasy blokad: `ReentrantLock`, którą właśnie skończyliśmy opisywać, oraz `ReentrantReadWriteLock`. Druga z wymienionych znajduje zastosowanie w sytuacjach, gdy wiele wątków odczytuje dane ze struktury danych, a mniej ją modyfikuje. Stanowi to podstawę do zezwolenia procedurom odczytującym na dostęp współdzielony. Oczywiście algorytm zapisujący musi mieć dostęp na wyłączność.

Oto lista czynności związanych z użyciem blokady odczytu-zapisu:

1. Utwórz obiekt ReentrantReadWriteLock:

```
private ReentrantReadWriteLock rw1 = new ReentrantReadWriteLock();
```

2. Pozyskaj blokady odczytu i zapisu:

```
private Lock readLock = rw1.readLock();
private Lock writeLock = rw1.writeLock();
```

3. Użyj blokady odczytu we wszystkich metodach dostępowych:

```
public double getTotalBalance()
{
    readLock.lock();
    try { . . . }
    finally { readLock.unlock(); }
}
```

4. Użyj blokady zapisu we wszystkich metodach modyfikujących:

```
public void transfer(. . .)
{
    writeLock.lock();
    try { . . . }
    finally { writeLock.unlock(); }
}
```

java.util.concurrent.locks.ReentrantReadWriteLock **5.0**

- Lock readLock()

Zwraca blokadę odczytu, którą może zakładać wiele algorytmów odczytujących, ale żaden zapisujący.

- Lock writeLock()

Zwraca blokadę zapisu, która wyklucza wszystkie pozostałe algorytmy odczytujące i zapisujące.

14.5.15. Dlaczego metody stop i suspend są wycofywane

Początkowo w Javie dostępna była metoda `stop`, która kończyła wątek, i metoda `suspend`, która blokowała wątek do czasu, gdy inny wątek wywołał metodę `resume`. Dwie pierwsze z wymienionych metod coś łączy: obie próbują kontrolować działanie wątku, nie współpracując z nim.

Obie te metody są wycofywane. Metoda `stop` jest z gruntu niebezpieczna, a jeśli chodzi o `suspend`, to z doświadczenia wiadomo, że często prowadzi do zakleszczeń. W tym podrozdziale wyjaśniamy, dlaczego metody te sprawiają problemy i co można zrobić, aby ich uniknąć.

Zacznijmy od metody `stop`. Zamyka ona wszystkie oczekujące metody, włącznie z metodą `run`. Jeśli zostanie zastosowana na rzecz wątku, natychmiast zdejmuję on wszystkie blokady, które założył. To może prowadzić do uszkodzenia obiektów. Wyobraźmy sobie na przykład,

że wątek TransferRunnable został zatrzymany w trakcie przelewania pieniędzy z jednego konta na inne — zdążył pobrać pieniądze, ale nie zdążył ich zapisać na drugim koncie. W tej sytuacji obiekt banku zostaje **zniszczony**. Ponieważ blokada została zdjęta, zniszczenie jest widoczne także dla innych wątków, które jeszcze nie zostały zatrzymane.

Wątek chcący zatrzymać inny wątek nie ma sposobu na sprawdzenie, kiedy wywołanie metody stop jest bezpieczne, a kiedy doprowadzi do zniszczenia obiektu. Dlatego odradza się używania tej metody. Aby zatrzymać wątek, należy go przerwać. Przerwany wątek może się zatrzymać wtedy, gdy jest to bezpieczne.



Niektórzy twierdzą, że metoda stop jest odradzana, ponieważ poprzez zatrzymywanie wątków może blokować obiekty na stałe. To jednak nieprawda. Zatrzymany wątek wychodzi z wszystkich metod synchronizowanych, które wywołał, zgłoszając wyjątek ThreadDeath. W rezultacie zwalniane są wszystkie wewnętrzne blokady obiektów, które wątek ten założył.

Kolejna metodą jest suspend. W przeciwieństwie do metody stop nie powoduje ona uszkodzenia obiektów. Jeśli jednak zawieszony zostanie wątek posiadający blokadę będzie ona niedostępna aż do odwieszenia tego wątku. Jeśli wątek, który wywołał tę metodę suspend, próbuje założyć tę samą blokadę, program zostaje zakleszczony — zawieszony wątek czeka na odwieszenie, a wątek, który go zawiesił, czeka na blokadę.

Sytuacje tego typu często zdarzają się w graficznych interfejsach użytkownika. Założymy, że mamy graficzną symulację naszego banku. Przycisk z etykietą *Wstrzymaj* zawiesza wątki dokonujące przelewów, a przycisk z etykietą *Wznów* odwiesza je.

```
pauseButton.addActionListener(new
    ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        for (int i = 0; i < threads.length; i++)
            threads[i].suspend();           // Nie rób tego.
    }
});
resumeButton.addActionListener(. . .); // Wywołuje metodę resume na rzecz wszystkich
                                         // wątków przelewowych.
```

Metoda paintComponent będzie rysować wykres każdego konta. W tym celu utworzy tablicę sald kont za pomocą metody getBalances.

Jak przekonasz się w podrozdziale 14.11, „Wątki a biblioteka Swing”, zarówno akcje przycisków, jak i ponowne rysowanie odbywają się w tym samym wątku — **wątku dystrybucji zdarzeń** (ang. *event dispatch thread*). Przeanalizujmy następujący scenariusz:

- 1 Jeden z wątków przelewowych zakłada blokadę obiektu bank.
- 2 Użytkownik kliką przycisk *Wstrzymaj*.
- 3 Wszystkie wątki przelewowe zostają zawieszone. Jeden z nich cały czas trzyma blokadę na obiekcie bank.

4. Z jakiegoś powodu konieczne jest ponowne narysowanie wykresu konta.
5. Metoda paintComponent wywołuje metodę getBalances.
6. Metoda ta próbuje założyć blokadę obiektu bank.

Program zostaje zamrożony.

Wątek dystrybucji zdarzeń nie może kontynuować, ponieważ blokada znajduje się w posiadaniu jednego z zawieszonych wątków. Dlatego użytkownik nie może kliknąć przycisku *Wznów* i wątki nigdy nie zostaną odwieszone.

Aby bezpiecznie zawieszać wątki, należy utworzyć zmienną suspendRequested i testować ją w bezpiecznym miejscu metody run — w takim miejscu, w którym wątek nie blokuje obiektów potrzebnych innym wątkom. Kiedy wątek odkryje, że zmienna suspendRequested została ustawiona, powinien czekać, aż będzie ona ponownie dostępna.

Implementacja opisywanej techniki znajduje się poniżej:

```
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();

public void run()
{
    while (...)
    {
        ...
        if (suspendRequested)
        {
            suspendLock.lock();
            try { while (suspendRequested) suspendCondition.await(); }
            finally { suspendLock.unlock(); }
        }
    }
}
public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
    suspendRequested = false;
    suspendLock.lock();
    try { suspendCondition.signalAll(); }
    finally { suspendLock.unlock(); }
}
```

14.6. Kolejki blokujące

Zapoznaliśmy się z niskopoziomowymi mechanizmami kładącymi podwaliny pod współbieżność w Javie. Jednak w codziennej pracy programistycznej lepiej jest trzymać się od nich jak najdalej. O wiele łatwiej i bezpieczniej jest używać struktur wyższego poziomu, które zostały zaimplementowane przez ekspertów od współbieżności.

Wiele problemów z wątkami można zgrabnie i bezpiecznie sformułować za pomocą jednej lub większej liczby kolejek. Wątki producenta umieszczają elementy w kolejce, a wątki konsumenta pobierają je stamtąd. Kolejka umożliwia bezpieczną wymianę danych pomiędzy wątkami. Weźmy na przykład nasz program symulujący bank. Wątki przelewające — zamiast bezpośrednio operować na obiekcie banku — wstawiają obiekty instrukcji przelewu do kolejki. Inny wątek usuwa te instrukcje i wykonuje przelewy. Tylko ten wątek ma dostęp do wnętrza obiektu banku. Nie jest potrzebna synchronizacja (oczywiście projektanci klas kolejek bezpiecznych dla wątków musieli zająć się blokadami i warunkami, ale to był ich problem, a nie nasz).

Kolejka blokująca (ang. *blocking queue*) powoduje zablokowanie wątku podczas próby dodania elementu, jeśli jest pełna, lub podczas próby usunięcia elementu, jeśli jest pusta. Kolejki tego typu znajdują zastosowanie w koordynacji działań wielu wątków. Niektóre wątki robocze mogą co jakiś czas odkładać pośrednio wyniki w kolejce blokującej, a pozostałe mogą je stamtąd usuwać i poddawać dalszej obróbce. Kolejka automatycznie kontroluje przebieg pracy. Jeśli jeden zestaw wątków działa wolniej niż drugi, ten drugi musi poczekać na wyniki pierwszego. Jeśli pierwszy zestaw działa szybciej od drugiego, kolejka zapewnia się, dopóki drugi zestaw wątków nadąży z odbieraniem. Tabela 14.1 zawiera zestawienie metod blokujących kolejek.

Metody kolejek blokujących można podzielić na trzy kategorie w zależności od działania, kiedy kolejka jest pełna lub pusta. Jeśli kolejka jest wykorzystywana jako narzędzie do zarządzania wątkami, należy używać metod `put` i `take`. Metody `add`, `remove` i `element` zgłoszą wyjątek, kiedy element jest dodawany do pełnej kolejki lub pobierany z pustej. Oczywiście w programie wielowątkowym kolejka może się zapłnić i zrobić pusta w każdej chwili. Dlatego w takich sytuacjach należy używać metod `offer`, `poll` i `peek`. Metody te nie zgłoszą wyjątku, tylko zwracają wartość oznaczającą niepowodzenie operacji, jeśli zakończą się niepowodzeniem.



Metody `poll` i `peek` informują o niepowodzeniu za pomocą wartości zwrotnej `null`. Dlatego do tego typu kolejek nie można wstawić referencji `null`.

Istnieją także wersje czasowe metod `offer` i `poll`. Na przykład poniższa instrukcja:

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

przez sto milisekund próbuje wstawić element do ogona kolejki. Jeśli się jej powiedzie, zwróci wartość `true`, w przeciwnym przypadku, jeśli nie wykona operacji w wyznaczonym czasie, zwróci `false`. Podobnie instrukcja:

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

przez sto milisekund próbuje usunąć element z czoła kolejki. Jeśli się jej powiedzie, zwróci ten element, w przeciwnym przypadku, jeśli nie wykona operacji w wyznaczonym czasie, zwróci `false`.

Metoda `put` włącza blokadę, jeśli kolejka jest pełna, a metoda `take` robi to samo, gdy kolejka jest pusta. Metody te są odpowiednikami metod `offer` i `poll` bez ograniczenia czasowego.

W pakiecie `java.util.concurrent` znajduje się kilka wersji kolejek blokujących. Kolejka `LinkedBlockingQueue` nie posiada domyślnej górnej granicy pojemności, ale można ją określić.

Tabela 14.1. Metody kolejek blokujących

Metoda	Normalne działanie	Działanie w specjalnych warunkach
add	Dodaje element.	Zgłasza wyjątek IllegalStateException, jeśli kolejka jest pełna.
element	Zwraca element z czoła.	Zgłasza wyjątek NoSuchElementException, jeśli kolejka jest pusta.
offer	Dodaje element i zwraca wartość true.	Zwraca wartość false, jeśli kolejka jest pełna.
peek	Zwraca element z czoła.	Zwraca wartość null, jeśli kolejka jest pusta.
poll	Usuwa i zwraca element z czoła.	Zwraca wartość null, jeśli kolejka jest pusta.
put	Dodaje element.	Blokuje, jeśli kolejka jest pełna.
remove	Usuwa i zwraca element z czoła.	Zgłasza wyjątek NoSuchElementException, jeśli kolejka jest pusta.
take	Usuwa i zwraca element z czoła.	Blokuje, jeśli kolejka jest pusta.

Jej dwustronna wersja to `LinkedBlockingDeque`. Kolejka `ArrayBlockingQueue` ma określoną pojemność i opcjonalny parametr włączający wymóg uczciwości. Jeśli kolejka jest uczciwa, preferencyjnie traktowane są te wątki, które czekają najdłużej. Należy jednak pamiętać, że uczciwość zawsze powoduje straty szybkości, przez co opcję tę powinno się stosować wyłącznie wtedy, gdy jest to całkowicie niezbędne.

Kolejka `PriorityBlockingQueue` jest kolejką priorytetową, nie typu „pierwszy wszedł, pierwszy wyszedł”. Elementy są usuwane zgodnie z ich priorytetami. Kolejka ta ma nieograniczoną pojemność, ale pobieranie elementów z pustej konstrukcji powoduje blokadę (więcej informacji na temat kolejek priorytetowych znajduje się w rozdziale 13.).

W końcu kolejka `DelayQueue` przechowuje obiekty, które implementują interfejs `Delayed`:

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit);
}
```

Metoda `getDelay` zwraca ilość pozostałego czasu opóźnienia obiektu. Wartość ujemna oznacza, że czas ten upłynął. Elementy z tej kolejki mogą zostać usunięte dopiero wtedy, gdy upłynie określony czas opóźnienia. Konieczna jest także implementacja metody `compareTo`. Kolejka `DelayQueue` używa tej metody do sortowania elementów.

W Java SE 7 dodano interfejs `TransferQueue` pozwalający wątkowi producenta poczekać, aż konsument będzie gotowy do przyjęcia elementu. Gdy producent wywołuje poniższą metodę:

```
q.transfer(item);
```

wywołanie to zostaje zablokowane do czasu, aż blokadę usunie inny wątek. Opisywany interfejs jest zaimplementowany w klasie `LinkedTransferQueue`.

Program przedstawiony na listingu 14.10 demonstruje sposób kontroli zestawu wątków za pomocą kolejki blokującej. Przeszukuje on wszystkie pliki znajdujące się w katalogu i jego podkatalogach oraz drukuje linijki, które zawierają dane słowo kluczowe.

Listing 14.10. blockingQueue/BlockingQueueTest.java

```

package blockingQueue;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class BlockingQueueTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Podaj katalog bazowy (np. /usr/local/jdk1.6.0/src): ");
        String directory = in.nextLine();
        System.out.print("Podaj słowo kluczowe (np. volatile): ");
        String keyword = in.nextLine();

        final int FILE_QUEUE_SIZE = 10;
        final int SEARCH_THREADS = 100;

        BlockingQueue<File> queue = new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);

        FileEnumerationTask enumerator = new FileEnumerationTask(queue, new
            ↪File(directory));
        new Thread(enumerator).start();
        for (int i = 1; i <= SEARCH_THREADS; i++)
            new Thread(new SearchTask(queue, keyword)).start();
    }
}

/**
 * Zadanie tworzące wyliczenie wszystkich plików w katalogu i jego podkatalogach
 */
class FileEnumerationTask implements Runnable
{
    public static File DUMMY = new File("");
    private BlockingQueue<File> queue;
    private File startingDirectory;

    /**
     * Tworzy obiekt klasy FileEnumerationTask.
     * @param queue kolejka blokująca, do której dodawane są pliki
     * @param startingDirectory katalog, od którego ma się zacząć zbieranie plików
     */
    public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
    {
        this.queue = queue;
        this.startingDirectory = startingDirectory;
    }

    public void run()
    {
        try
        {

```

```
        enumerate(startingDirectory);
        queue.put(DUMMY);
    }
    catch (InterruptedException e)
    {
    }
}

/**
* Rekursywna enumeracja wszystkich plików znajdujących się w danym katalogu i jego podkatalogach
* @param directory katalog początkowy
*/
public void enumerate(File directory) throws InterruptedException
{
    File[] files = directory.listFiles();
    for (File file : files)
    {
        if (file.isDirectory()) enumerate(file);
        else queue.put(file);
    }
}

/**
* Zadanie przeszukujące pliki w celu znalezienia określonego słowa kluczowego
*/
class SearchTask implements Runnable
{
    private BlockingQueue<File> queue;
    private String keyword;

    /**
     * Tworzy obiekt klasy SearchTask.
     * @param queue kolejka, z której mają być pobierane pliki
     * @param keyword słowo kluczowe, które ma zostać znalezione
     */
    public SearchTask(BlockingQueue<File> queue, String keyword)
    {
        this.queue = queue;
        this.keyword = keyword;
    }

    public void run()
    {
        try
        {
            boolean done = false;
            while (!done)
            {
                File file = queue.take();
                if (file == FileEnumerationTask.DUMMY)
                {
                    queue.put(file);
                    done = true;
                }
                else search(file);
            }
        }
    }
}
```

```

        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e)
        {
        }
    }

    /**
     * Przeszukuje plik w celu znalezienia określonego słowa kluczowego i drukuje wszystkie zawierające je linijki
     * @param file plik do przeszukania
    */
}

public void search(File file) throws IOException
{
    try (Scanner in = new Scanner(file))
    {
        int lineNumber = 0;
        while (in.hasNextLine())
        {
            lineNumber++;
            String line = in.nextLine();
            if (line.contains(keyword))
                System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber, line);
        }
    }
}
}

```

Wątek producenta (producent) tworzy wyliczenie wszystkich plików znalezionych we wszystkich podkatalogach i wstawia je do kolejki blokującej. Operacja ta jest bardzo szybka i gdyby nie ograniczenie pojemności, kolejka w szybkim tempie zapełniłaby się wszystkimi plikami znajdującymi się w systemie plików.

Uruchamiamy także dużą liczbę wątków przeszukujących. Każdy taki wątek pobiera plik z kolejki, otwiera go, drukuje wszystkie linijki zawierające dane słowo kluczowe i pobiera następny plik. Do zakończenia aplikacji, kiedy dalsza jej praca jest już zbędna, wykorzystaliśmy pewną sztuczkę. Wątek wyliczeniowy sygnalizuje ukończenie pracy, umieszczając w kolejce atrapę obiektu (przypomina to umieszczanie walizki z etykietą „Ostatnia torba” na końcu taśmy z walizkami na lotnisku). Kiedy wątek przeszukujący pobierze taki obiekt, odkłada go z powrotem i kończy działanie.

Zwróć uwagę, że nie trzeba bezpośrednio stosować synchronizacji. W tej aplikacji do synchronizacji używamy kolejki.

java.util.concurrent.ArrayBlockingQueue<E> **5.0**

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

Tworzy kolejkę blokującą o określonej pojemności i z ustawioną zasadą uczciwości. Kolejka ta jest zaimplementowana jako tablica cykliczna.

java.util.concurrent.LinkedBlockingQueue<E> **5.0**
 java.util.concurrent.LinkedBlockingDeque<E> **6**

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`

Tworzy nieograniczoną kolejkę blokującą jedno- lub dwustronną, zaimplementowaną jako lista powiązana.

- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

Tworzy ograniczoną kolejkę jedno- lub dwustronną blokującą o określonej pojemności, zaimplementowaną jako lista powiązana.

java.util.concurrent.DelayQueue<E extends Delayed> **5.0**

- `DelayQueue()`

Tworzy nieograniczoną kolejkę elementów typu `Delayed`. Z kolejki tej można usuwać tylko te elementy, których czas opóźnienia upłynął.

java.util.concurrent.Delayed **5.0**

- `long getDelay(TimeUnit unit)`

Zwraca opóźnienie obiektu mierzone w określonej jednostce czasu.

java.util.concurrent.PriorityBlockingQueue<E> **5.0**

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

Tworzy nieograniczoną priorytetową kolejkę blokującą zaimplementowaną jako sterta.

Parametry:	<code>initialCapacity</code>	Początkowa pojemność kolejki priorytetowej. Domyślna wartość to 11.
	<code>comparator</code>	Komparator używany do porównywania elementów. Jeśli nie zostanie podany, elementy muszą implementować interfejs <code>Comparable</code> .

java.util.concurrent.BlockingQueue<E> **5.0**

- `void put(E element)`

Dodaje element i w razie konieczności włącza blokowanie.

■ E take()

Usuwa i zwraca element z czoła, w razie konieczności włącza blokowanie.

■ boolean offer(E element, long time, TimeUnit unit)

Dodaje określony element i zwraca wartość true, jeśli operacja zakończy się powodzeniem. W razie konieczności włącza blokowanie, aż element zostanie dodany lub upłynie określony czas.

■ E poll(long time, TimeUnit unit)

Usuwa i zwraca element z czoła. W razie konieczności włącza blokowanie, aż element będzie dostępny lub upłynie określony czas. W razie niepowodzenia zwraca wartość null.

java.util.concurrent.BlockingDeque<E> 6

■ void putFirst(E element)

■ void putLast(E element)

Dodaje element i w razie potrzeby włącza blokadę.

■ E takeFirst()

■ E takeLast()

Usuwa i zwraca element z czoła lub ogona i w razie potrzeby włącza blokadę.

■ boolean offerFirst(E element, long time, TimeUnit unit)

■ boolean offerLast(E element, long time, TimeUnit unit)

Dodaje określony element i zwraca wartość true, jeśli operacja zakończy się powodzeniem. W razie potrzeby włącza blokadę, aż element zostanie dodany albo upłynie wyznaczony czas.

■ E pollFirst(long time, TimeUnit unit)

■ E pollLast(long time, TimeUnit unit)

Usuwa i zwraca element z czoła lub ogona. W razie potrzeby włącza blokadę, aż element będzie dostępny lub upłynie wyznaczony czas. W przypadku niepowodzenia zwraca wartość null.

java.util.concurrent.TransferQueue<E> 7

■ void transfer(E element)

■ boolean tryTransfer(E element, long time, TimeUnit unit)

Przesyła wartość albo próbuje ją przesłać w określonym czasie, zakładając blokadę do czasu, aż inny wątek usunie element. Druga metoda zwraca true w razie powodzenia.

14.7. Kolekcje bezpieczne wątkowo

Jeśli kilka wątków równocześnie modyfikuje jakąś strukturę danych, na przykład tablicę mieszającą, to może ona łatwo ulec uszkodzeniu (więcej informacji na temat tablic mieszających znajduje się w rozdziale 13.). Na przykład jeden wątek może zacząć operację wstawiania nowego elementu. Założymy, że zostaje on wywolany w trakcie przekierowywania łączy pomiędzy komórkami tablicy. Jeśli w tym czasie inny wątek zacznie przemierzać tę strukturę danych, może się natknąć na złe połączenia i spowodować kompletny bałagan, przy okazji zgłaszając wyjątek bądź wpadając w nieskończoną pętlę.

Współdzieloną strukturę danych można ochronić za pomocą blokady, ale często łatwiejszym rozwiązaniem jest użycie implementacji bezpiecznej wątkowo. Kolejki blokujące omówione w poprzednim podrozdziale są przykładem takich bezpiecznych wątkowo kolekcji. Poniższe podrozdziały opisują inne kolekcje znajdujące się w bibliotece Javy, które są bezpieczne ze względu na wątki.

14.7.1. Szybkie mapy, zbiory i kolejki

W pakiecie `java.util.concurrent` znajdują się następujące szybkie implementacje map, zbiórów uporządkowanych i kolejek: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` oraz `ConcurrentLinkedQueue`.

W kolekcjach tych zastosowano zaawansowane algorytmy minimalizujące rywalizację wątków poprzez umożliwianie równoległego dostępu do różnych części struktury danych.

W przeciwieństwie do większości kolekcji, w tych metoda `size` niekoniecznie działa w stałym czasie. Określenie aktualnego rozmiaru tych kolekcji zazwyczaj wymaga ich przemierzenia.

Kolekcje te zwracają tak zwane **slabo spójne iteratory** (ang. *weakly consistent iterators*). Oznacza to, że mogą one (choć nie muszą) odzwierciedlać wszystkie modyfikacje dokonane po ich skonstruowaniu. Nie zwracają one jednak dwukrotnie wartości i nie zgłaszają wyjątku `ConcurrentModificationException`.



W przeciwieństwie do opisywanych iteratorów, iteratory kolekcji z pakietu `java.util.concurrent` zgłaszą wyjątek `ConcurrentModificationException`, jeśli kolekcja zostanie zmodyfikowana po ich utworzeniu.

Struktura `ConcurrentHashMap` jest zdolna szybko obsłużyć dużą liczbę czytników i ustaloną liczbę algorytmów zapisujących. Domyślnie założono, że może być do 16 algorytmów zapisujących **działających jednocześnie**. Może być ich więcej, ale jeśli więcej niż 16 z nich zapisuje w tym samym czasie, reszta pozostaje tymczasowo zablokowana. Można podać większą liczbę w konstruktorze, ale istnieje niewielkie prawdopodobieństwo, że będzie to potrzebne.

Klasy `ConcurrentHashMap` i `ConcurrentSkipListMap` udostępniają metody służące do wstawiania i usuwania par klucz – wartość za pomocą niepodzielnych operacji. Metoda `putIfAbsent` dodaje nową parę klucz – wartość, pod warunkiem że nie było jej wcześniej. Metoda ta jest

przydatna w pamięciach podręcznych, do których dostęp ma wiele wątków, ponieważ daje pewność, że dana para zostanie wstawiona tylko przez jeden wątek:

```
cache.putIfAbsent(key, value);
```

Operację przeciwną wykonuje metoda remove (której nazwa powinna chyba brzmieć remove ↪IfPresent). Poniższe wywołanie usuwa za pomocą niepodzielnej operacji klucz i jego wartość, jeśli znajdują się one w mapie.

```
cache.remove(key, value)
```

W końcu poniższe wywołanie zamienia za pomocą niepodzielnej operacji starą wartość (oldValue) na nową (newValue), pod warunkiem że stara wartość jest skojarzona z określonym kluczem.

```
cache.replace(key, oldValue, newValue)
```

java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- **ConcurrentLinkedQueue<E>()**

Tworzy nieograniczoną kolejkę nieblokującą, którą można bezpiecznie przetwarzanie w wielu wątkach.

java.util.concurrent.ConcurrentSkipListSet<E> 6

- **ConcurrentSkipListSet<E>()**
- **ConcurrentSkipListSet<E>(Comparator<? super E> comp)**

Tworzy zbiór uporządkowany, do którego można bezpiecznie uzyskać dostęp w wielu wątkach. Pierwszy z konstruktorów wymaga, aby elementy implementowały interfejs Comparable.

java.util.concurrent.ConcurrentHashMap<K, V> 5.0

java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- **ConcurrentHashMap<K, V>()**
- **ConcurrentHashMap<K, V>(int initialCapacity)**
- **ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)**

Tworzy mapę haszową, do której można bezpiecznie uzyskać dostęp w wielu wątkach.

Parametry: `initialCapacity` Początkowa pojemność kolekcji, domyślana wartość to 16.

`loadFactor` Kontroluje zmianę rozmiaru: jeśli średnie zapełnienie komórki przekracza ten współczynnik, rozmiar tablicy jest zmieniany. Domyślna wartość to 0,75.

`concurrencyLevel` Przewidywana liczba współbieżnych wątków zapisujących.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

Tworzy uporządkowaną mapę, do której można uzyskać bezpieczny dostęp w wielu wątkach. Pierwszy z konstruktorów wymaga, aby klucze implementowały interfejs Comparable.

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

Jeśli klucza nie ma jeszcze w mapie, zostaje on skojarzony z podaną wartością i metoda zwraca wartość null. W przeciwnym przypadku zwraca istniejącą wartość skojarzoną z tym kluczem.

- `boolean remove(K key, V value)`

Jeśli podany klucz jest aktualnie skojarzony z podaną wartością, metoda ta usuwa je i zwraca wartość true. W przeciwnym przypadku zwraca wartość false.

- `boolean replace(K key, V oldValue, V newValue)`

Jeśli podany klucz jest aktualnie skojarzony ze starą wartością (oldValue), zostaje skojarzony z nową wartością (newValue). W przeciwnym przypadku zwraca wartość false.

14.7.2. Tablice kopowane przy zapisie

`CopyOnWriteArrayList` i `CopyOnWriteArraySet` to bezpieczne wątkowo kolekcje, których mutatory tworzą kopie tablic. Taki sposób działania sprawdza się w sytuacjach, w których liczba wątków iterujących po kolekcji znacznie przewyższa liczbę wątków ją modyfikujących. Utworzony iterator zawiera referencję do aktualnej tablicy. Jeśli tablica ta zostanie później zmodyfikowana, iterator ten nadal będzie miał starą tablicę, mimo że tablica kolekcji jest zamieniona. Dzięki temu starszy iterator dysponuje spójnym (choć potencjalnie przestarzałym) widokiem, do którego ma dostęp nieobciążony żadnym dodatkowym narzutem synchronizacji.

14.7.3. Starsze kolekcje bezpieczne wątkowo

Od samego początku istnienia Javy klasy `Vector` i `Hashtable` udostępniały bezpieczne wątkowo implementacje tablicy dynamicznej i mieszającej. Klasy te są już uważane za przestarzałe i zastąpiono je klasami `ArrayList` i `HashMap`. Klasy te nie są bezpieczne wątkowo. W zamian w bibliotece kolekcji zaproponowano inną technikę. Każdą klasę kolekcji można uczynić bezpieczną wątkowo za pomocą **synchronizacyjnych obiektów opakowujących**:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

Metody tak powstałych kolekcji są chronione przez blokadę, co umożliwia bezpieczny wątkowo dostęp.

Należy zapewnić, że żaden wątek nie będzie miał dostępu do struktury danych poprzez oryginalne niesynchronizowane metody. Najprostszym sposobem jest niezapisywanie żadnych referencji do oryginalnego obiektu. Po utworzeniu kolekcji od razu należy przekazać ją do opakowania, tak jak zrobiliśmy to w prezentowanych przykładach.

Nadal trzeba stosować blokowanie po stronie klienta, aby móc **iterować** po kolekcji, podczas gdy inny wątek może ją modyfikować:

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) . . .;
}
```

Tego samego kodu musimy użyć, jeśli korzystamy z pętli typu `for each`, ponieważ pętla ta używa iteratatora. Należy pamiętać, że iteratator zgłosi wyjątek `ConcurrentModificationException`, jeśli w trakcie iteracji po kolekcji inny wątek ją zmodyfikuje. Synchronizacja jest nadal wymagana, dzięki czemu można wykryć współbieżne modyfikacje.

Zamiast używać synchronizacyjnych obiektów opakowujących, zazwyczaj lepiej jest skorzystać z kolekcji z pakietu `java.util.concurrent`. Mapa `ConcurrentHashMap` została bardzo starannie zaimplementowana w taki sposób, aby można było uzyskać do niej dostęp w wielu wątkach, nie powodując ich wzajemnego blokowania się, pod warunkiem że działają one na różnych komórkach. Jeden wyjątek stanowi lista tablicowa, która jest często modyfikowana. W takim przypadku synchronizowana lista `ArrayList` może się okazać lepsza od listy `CopyOnWriteArrayList`.

java.util.Collections 1.2

- static <E> Collection<E> synchronizedCollection(Collection<E> c)
- static <E> List synchronizedList(List<E> c)
- static <E> Set synchronizedSet(Set<E> c)
- static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
- static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
- static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)

Tworzy widoki kolekcji, których metody są synchronizowane.

14.8. Interfejsy Callable i Future

Obiekt implementujący interfejs `Runnable` opakowuje zadania, które działają asynchronicznie. Można go traktować jako asynchroniczną metodę bez parametrów i wartości zwrotnej. Obiekt `Callable` jest podobny do `Runnable`, ale posiada wartość zwrotną. Interfejs `Callable` jest typem parametryzowanym zawierającym jedną metodę o nazwie `call`.

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Parametr typowy określa typ zwracanej wartości. Na przykład interfejs `Callable<Integer>` reprezentuje asynchroniczne działania, których wynikiem jest obiekt typu `Integer`.

Obiekt `Future` przechowuje **wynik** asynchronicznych obliczeń. Można rozpoczęć obliczenia, przekazać gdzieś obiekt `Future` i zapomnieć o nim. Właściciel tego obiektu może pobrać wynik, kiedy będzie gotowy.

W interfejsie `Future` znajdują się następujące metody:

```
public interface Future<V>
{
    V get() throws . . .;
    V get(long timeout, TimeUnit unit) throws . . .;
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

Wywołanie pierwszej z metod `get` jest zablokowane do zakończenia obliczeń. Druga wersja tej metody zgłasza wyjątek `TimeoutException`, jeśli obliczenia nie zakończą się przed upływem określonego czasu. Obie te metody zgłoszają wyjątek `InterruptedException`, jeśli wątek prowadzący obliczenia zostanie przerwany. Kiedy obliczenia zakończą się, metoda `get` natychmiast zwraca wartość.

Metoda `isDone` zwraca wartość `false`, jeśli obliczenia są jeszcze w toku, lub `true` w przeciwnym przypadku.

Operację można przerwać za pomocą metody `cancel`. Jeśli jeszcze się nie rozpoczęła, zostanie anulowana i nigdy się nie rozpocznie. Jeśli jest w toku, zostanie przerwana, gdy parametr `mayInterrupt` ma wartość `true`.

Obiekt `FutureTask` jest wygodnym narzędziem do zamianiania obiektów `Callable` zarówno na `Future`, jak i `Runnable`, ponieważ implementuje oba te interfejsy. Na przykład:

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // Runnable
t.start();

Integer result = task.get(); // Future
```

Program przedstawiony na listingu 14.11 demonstruje praktyczne zastosowanie omawianych technik. Jest podobny do poprzedniego programu, który znajdował pliki zawierające dane słowo kluczowe. Tym razem jednak poprzestaniemy tylko na zliczaniu znalezionych plików. W związku z tym mamy jedno długo trwające zadanie, które zwraca liczbę całkowitą — przykład interfejsu `Callable<Integer>`.

```
class MatchCounter implements Callable<Integer>
{
    public MatchCounter(File directory, String keyword) { . . . }
    public Integer call() { . . . } // Zwraca liczbę pasujących plików.
}
```

Listing 14.11. future/FutureTest.java

```
package future;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class FutureTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Podaj katalog bazowy (np. /usr/local/jdk1.6.0/src): ");
        String directory = in.nextLine();
        System.out.print("Podaj słowo kluczowe (np. volatile): ");
        String keyword = in.nextLine();

        MatchCounter counter = new MatchCounter(new File(directory), keyword);
        FutureTask<Integer> task = new FutureTask<>(counter);
        Thread t = new Thread(task);
        t.start();
        try
        {
            System.out.println("Liczba znalezionych plików " + task.get() + ".");
        }
        catch (ExecutionException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e)
        {
        }
    }
}

/**
 * Zadanie liczące pliki znajdujące się w katalogu i jego podkatalogach, zawierające dane słowo kluczowe
 */
class MatchCounter implements Callable<Integer>
{
    private File directory;
    private String keyword;
    private int count;

    /**

```

```
* Tworzy obiekt klasy MatchCounter.
* @param directory katalog, od którego ma się zacząć szukanie
* @param keyword słowo kluczowe do znalezienia
*/
public MatchCounter(File directory, String keyword)
{
    this.directory = directory;
    this.keyword = keyword;
}

public Integer call()
{
    count = 0;
    try
    {
        File[] files = directory.listFiles();
        List<Future<Integer>> results = new ArrayList<>();

        for (File file : files)
            if (file.isDirectory())
            {
                MatchCounter counter = new MatchCounter(file, keyword);
                FutureTask<Integer> task = new FutureTask<>(counter);
                results.add(task);
                Thread t = new Thread(task);
                t.start();
            }
            else
            {
                if (search(file)) count++;
            }

        for (Future<Integer> result : results)
            try
            {
                count += result.get();
            }
            catch (ExecutionException e)
            {
                e.printStackTrace();
            }
        catch (InterruptedException e)
        {
        }
    }
    return count;
}

/**
* Przeszukuje plik w celu znalezienia danego słowa kluczowego.
* @param file plik do przeszukania
* @return wartość true, jeśli plik zawiera dane słowo kluczowe
*/
public boolean search(File file)
{
    try
    {
```

```

try (Scanner in = new Scanner(file))
{
    boolean found = false;
    while (!found && in.hasNextLine())
    {
        String line = in.nextLine();
        if (line.contains(keyword)) found = true;
    }
    return found;
}
catch (IOException e)
{
    return false;
}
}
}

```

Następnie konstruujemy obiekt typu `FutureTask` z obiektu `MatchCounter` i wykorzystujemy go do uruchomienia wątku.

```

FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();

```

Na końcu drukujemy wynik.

```
System.out.println("Liczba znalezionych plików " + task.get() + ".");
```

Oczywiście wywołanie metody `get` powoduje blokadę do chwili, aż wynik jest rzeczywiście dostępny.

W metodzie `call` wykorzystujemy rekursywnie ten sam mechanizm. Dla każdego podkatalogu tworzymy nowy obiekt `MatchCounter` i uruchamiamy dla niego wątek. Ponadto odkładamy obiekty `FutureTask` w tablicy `ArrayList<Future<Integer>>`. Na końcu sumujemy wszystkie wyniki.

```

for (Future<Integer> result : results)
    count += result.get();

```

Każde wywołanie metody `get` powoduje blokadę do chwili, aż zostanie udostępniony wynik. Oczywiście wątki działają równolegle, dzięki czemu jest duża szansa, że wszystkie wyniki będą dostępne mniej więcej w tym samym czasie.

`java.util.concurrent.Callable<V>` **5.0**

■ `V call()`

Uruchamia zadanie, które zwraca wynik.

`java.util.concurrent.Future<V>` **5.0**

■ `V get()`

■ `V get(long time, TimeUnit unit)`

Zwraca wynik, włączając blokadę, dopóki nie jest on dostępny lub nie upłynie określona ilość czasu. Druga wersja zgłasza wyjątek `TimeoutException`, jeśli zakończy się niepowodzeniem.

- `boolean cancel(boolean mayInterrupt)`

Próbuje anulować wykonywanie zadania. Zadanie, które zostało już uruchomione, a ma parametr `mayInterrupt` ustawiony na `true`, zostanie przerwane. Jeśli operacja anulowania zakończy się pomyślnie, metoda ta zwraca wartość `true`.

- `boolean isCancelled()`

Zwraca wartość `true`, jeśli zadanie zostało anulowane przed ukończeniem.

- `boolean isDone()`

Zwraca wartość `true`, jeśli zadanie zostało ukończone w normalny sposób, zostało anulowane lub spowodowało wyjątek.

`java.util.concurrent.FutureTask<V> 5.0`

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`

Tworzy obiekt, który jest zarówno typu `Future<V>`, jak i `Runnable`.

14.9. Klasa Executors

Tworzenie nowego wątku jest nieco czasochłonne, ponieważ wymaga interakcji z systemem operacyjnym. Jeśli w programie tworzona jest duża liczba krótko żyjących wątków, powinno się stosować **pule wątków**. Pula taka zawiera pewną liczbę nieaktywnych wątków, które są gotowe do działania. Przekazanie do niej obiektu `Runnable` powoduje wywołanie przez jeden z wątków metody `run`. Kiedy metoda ta zakończy działanie, wątek nie zostaje zakończony, a przechodzi w stan oczekiwania na kolejne zadanie.

Dodatkowym powodem przemawiającym za stosowaniem puli wątków jest chęć zmniejszenia liczby wątków wykonywanych jednocześnie. Zbyt duża ich liczba może poważnie spowolnić aplikację, a nawet doprowadzić do awarii maszyny wirtualnej. Jeśli w programie jest algorytm tworzący dużą liczbę wątków, powinna się w nim znaleźć także ustalona pula wątków, ograniczająca liczbę wątków działających jednocześnie.

W klasie `Executors` znajduje się kilka statycznych metod fabrycznych służących do tworzenia puli wątków (tabela 14.2).

Tabela 14.2. Metody fabryczne klasy Executors

Metoda	Opis
newCachedThreadPool	W razie potrzeby tworzy nowe wątki. Nieaktywne wątki są przetrzymywane przez 60 sekund.
newFixedThreadPool	Pula zawierająca ustaloną liczbę wątków. Nieaktywne wątki są zachowywane bezterminowo.
newSingleThreadExecutor	Pula składająca się z jednego wątku wykonującego zadania po kolei (podobnie do wątku dystrybucji zdarzeń w Swing).
newScheduledThreadPool	Ustalona harmonogramowana pula wątków. Zastępstwo dla <code>java.util.Timer</code> .
newSingleThreadScheduledExecutor	Harmonogramowana pula składająca się z jednego wątku.

14.9.1. Pule wątków

Przyjrzymy się dokładniej trzem pierwszym metodom z tabeli 14.2. Pozostałe z nich zostały opisane w podrozdziale 14.9.2, „Planowanie wykonywania”. Metoda `newCachedThreadPool` tworzy pulę wątków, która wykonuje zadania natychmiast za pomocą jednego z wątków nieaktywnych, jeśli taki jest, lub tworząc nowy wątek w przeciwnym przypadku. Metoda `newFixedThreadPool` tworzy pulę wątków o ustalonym rozmiarze. Jeśli zadań jest więcej niż wolnych wątków, są one umieszczone w kolejce i wykonywane po zakończeniu wcześniejszych zadań. Metoda `newSingleThreadExecutor` tworzy pulę składającą się z jednego wątku, który wykonuje zadania jedno po drugim. Wszystkie trzy opisane metody zwracają obiekt klasy `ThreadPoolExecutor`, która implementuje interfejs `ExecutorService`.

Obiekt `Runnable` lub `Callable` można przekazać do `ExecutorService` za pomocą jednej z poniższych metod:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

Pula wykona powierzone jej zadanie przy najbliższej sposobności. Metoda `submit` zwraca obiekt typu `Future`, który zawiera informacje o stanie zadania.

Pierwsza z wymienionych metod zwraca dość osobliwie wyglądający typ `Future<?>`. Na rzecz tego obiektu można wywołać metody `isDone`, `cancel` lub `isCancelled`. Natomiast metoda `get` w chwili ukończenia zwraca wartość `null`.

Druga wersja metody `submit` także przesyła obiekt `Runnable`, a metoda `get` interfejsu `Future` zwraca wynik operacji, gdy jest już gotowy.

Trzecia wersja przesyła obiekt `Callable` i zwrócony obiekt `Future` otrzymuje wynik obliczeń, gdy jest gotowy.

Po zakończeniu pracy w puli wątków należy wywołać metodę `shutdown`. Inicjuje ona operację zamkającą pulę. Egzekutor, który jest zamknięty, nie przyjmuje żadnych nowych zadań.

Po zakończeniu wszystkich zadań wątki puli zostają zakończone. Istnieje także metoda `shutdownNow`, która powoduje, że pula anuluje wszystkie jeszcze niezaczęte zadania i próbuje przerwać aktualnie uruchomione.

Oto zestawienie działań, które należy wykonać, aby użyć puli wątków:

- 1 Wywołaj statyczną metodę `newCachedThreadPool` lub `newFixedThreadPool` z klasy `Executors`.
- 2 Prekaż obiekty `Runnable` lub `Callable` za pomocą metody `submit`.
- 3 Wykorzystaj zwrócone obiekty `Future`, jeśli chcesz mieć możliwość anulowania zadań lub jeśli przekażesz obiekty `Callable`.
- 4 Jeśli nie chcesz przekazywać więcej zadań, wywołaj metodę `shutdown`.

Na przykład wcześniej prezentowany program tworzył dużą liczbę krótkotrwałych wątków — po jednym dla każdego katalogu. Program z listingu 14.12 wykonuje zadania pod kontrolą puli wątków.

Listing 14.12. `threadPool/ThreadPoolTest.java`

```
package threadPool;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class ThreadPoolTest
{
    public static void main(String[] args) throws Exception
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Podaj katalog bazowy (np. /usr/local/jdk1.6.0/src): ");
        String directory = in.nextLine();
        System.out.print("Podaj słowo kluczowe (np. volatile): ");
        String keyword = in.nextLine();

        ExecutorService pool = Executors.newCachedThreadPool();

        MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
        Future<Integer> result = pool.submit(counter);

        try
        {
            System.out.println(result.get() + " pasujących plików.");
        }
        catch (ExecutionException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e)
        {
```

```

        }
    pool.shutdown();

    int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
    System.out.println("Największy rozmiar puli=" + largestPoolSize);
}
}

/*
 * Zadanie liczące pliki w katalogu i jego podkatalogach, zawierające dane słowo kluczowe
 */

class MatchCounter implements Callable<Integer>
{
    private File directory;
    private String keyword;
    private ExecutorService pool;
    private int count;

    /*
 * Tworzy obiekt typu MatchCounter.
 * @param directory katalog, od którego ma się zacząć szukanie
 * @param keyword słowo kluczowe do znalezienia
 * @param pool pula wątków, do której wysyłane są zadania
 */

    public MatchCounter(File directory, String keyword, ExecutorService pool)
    {
        this.directory = directory;
        this.keyword = keyword;
        this.pool = pool;
    }

    public Integer call()
    {
        count = 0;
        try
        {
            File[] files = directory.listFiles();
            List<Future<Integer>> results = new ArrayList<>();

            for (File file : files)
                if (file.isDirectory())
                {
                    MatchCounter counter = new MatchCounter(file, keyword, pool);
                    Future<Integer> result = pool.submit(counter);
                    results.add(result);
                }
                else
                {
                    if (search(file)) count++;
                }

            for (Future<Integer> result : results)
                try
                {
                    count += result.get();
                }
                catch (ExecutionException e)
                {
                    e.printStackTrace();
                }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

```
        }
    }
    catch (InterruptedException e)
    {
    }
    return count;
}

/**
* Przeszukuje plik w celu znalezienia danego słowa kluczowego.
* @param file plik do przeszukania
* @return wartość true, jeśli plik zawiera słowo kluczowe
*/
public boolean search(File file)
{
    try
    {
        try (Scanner in = new Scanner(file))
        {
            boolean found = false;
            while (!found && in.hasNextLine())
            {
                String line = in.nextLine();
                if (line.contains(keyword)) found = true;
            }
            return found;
        }
        catch (IOException e)
        {
            return false;
        }
    }
}
```

Dla celów informacyjnych program drukuje rozmiar największej puli. Informacja ta nie jest dostępna za pośrednictwem interfejsu ExecutorService. Z tego powodu musielibyśmy rzutować obiekt puli na klasę ThreadPoolExecutor.

java.util.concurrent.Executors 5.0

■ ExecutorService newCachedThreadPool()

Zwraca pulę wątków, która w razie potrzeby tworzy wątki, i kończy te, które są nieaktywne przez 60 sekund.

■ ExecutorService newFixedThreadPool(int threads)

Zwraca pulę wątków, która wykonuje zadania przy użyciu określonej liczby wątków.

■ ExecutorService newSingleThreadExecutor()

Zwraca egzekutor, który wykonuje zadania kolejno w jednym wątku.

java.util.concurrent.ExecutorService 5.0

- Future<T> submit(Callable<T> task)
- Future<T> submit(Runnable task, T result)
- Future<?> submit(Runnable task)

Przekazuje zadanie do wykonania.

- void shutdown()

Zamyka usługę. Kończy przekazane wcześniej zadania, ale nie przyjmuje nowych.

java.util.concurrent.ThreadPoolExecutor 5.0

- int getLargestPoolSize()

Zwraca największy rozmiar puli wątków w czasie działania egzekutora.

14.9.2. Planowanie wykonywania

Interfejs ScheduledExecutorService zawiera metody służące do planowego i wielokrotnego wykonywania zadań. Tworzenie pul wątków jest możliwe dzięki uogólnieniu klasy `java.util.concurrent.Executor`. Metody `newScheduledThreadPool` i `newSingleThreadScheduledExecutor` klasy `Executors` zwracają obiekty implementujące interfejs `ScheduledExecutorService`.

Zadania `Runnable` i `Callable` można zaplanować do jednorazowego wykonania po uprzednim odłożeniu ich na jakiś czas. Zadanie `Runnable` można także wykonywać w określonych odstępach czasu. Szczegółowe informacje na ten temat znajdują się w wyciągach z API.

java.util.concurrent.Executors 5.0

- ScheduledExecutorService newScheduledThreadPool(int threads)

Zwraca pulę wątków, która wykonuje zadania według harmonogramu w określonej liczbie wątków.

- ScheduledExecutorService newSingleThreadScheduledExecutor()

Zwraca egzekutor, który wykonuje zadania według harmonogramu w jednym wątku.

java.util.concurrent.ScheduledExecutorService 5.0

- ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)
- ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)

Wykonuje dane zadanie po upływie określonej ilości czasu.

- ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)

Ustawia harmonogram uruchamiania danego zadania w równych odstępach czasu, zaczynając po upływie początkowego opóźnienia (`initialDelay`).

- ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)

Ustawia harmonogram uruchamiania danego zadania w określonych odstępach czasu. Długość czasu opóźnienia kolejnych wykonań wynosi tyle, ile upłynęło czasu od ukończenia jednego wywołania do rozpoczęcia kolejnego. Pierwsze wykonanie następuje po upływie initialDelay czasu.

14.9.3. Kontrolowanie grup zadań

Wiemy już, jak wykorzystywać egzekutor w roli puli wątków mającej na celu zwiększenie szybkości wykonywania zadań. Czasami egzekutory są wykorzystywane do bardziej taktycznych celów, na przykład kontrolowania grup spokrewnionych zadań. Na przykład wszystkie zadania w egzekutorze można zakończyć za pomocą jednego wywołania metody shutdownNow.

Metoda invokeAny przekazuje wszystkie obiekty z kolekcji obiektów typu Callable i zwraca wynik ukończonego zadania. Nie wiadomo, które to zadanie — prawdopodobnie to, które zostało ukończone najwcześniej. Metody tej można użyć w algorytmie wyszukującym, który może przyjąć każde rozwiązywanie. Wyobraźmy sobie na przykład, że chcemy rozłożyć na czynniku dużą liczbę całkowitą — operacja wymagana do łamania szyfru RSA. Można przekazać kilka zadań, z których każde próbuje rozkładu przy użyciu liczb z innego zakresu. Kiedy tylko którykolwiek z nich ma odpowiedź, dalsze obliczenia można zatrzymać.

Metoda invokeAll przesyła wszystkie obiekty Callable z kolekcji i zwraca listę obiektów Future, które reprezentują rozwiązania wszystkich zadań. Wyniki te można przetwarzać w następujący sposób:

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
    processFurther(result.get());
```

Wadą tej metody jest to, że można niepotrzebnie czekać, jeśli pierwsze zadanie zajmuje zbyt dużo czasu. Dużo lepiej byłoby pobierać wyniki w takiej kolejności, w jakiej są udostępniane. Można to osiągnąć za pomocą klasy ExecutorCompletionService.

Należy zacząć od utworzenia w normalny sposób egzekutora. Następnie tworzymy obiekt ExecutorCompletionService, do którego przekazujemy zadania. Obiekt ten zarządza kolejką blokującą zawierającą obiekty typu Future, w których zapisywane są wyniki zadań. W związku z tym powyższe obliczenia można wykonać szybciej za pomocą poniższego algorytmu:

```
ExecutorCompletionService<T> service = new ExecutorCompletionService<T>(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
    processFurther(service.take().get());
```

java.util.concurrent.ExecutorService 5.0

- T invokeAny(Collection<Callable<T>> tasks)
- T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)

Wykonuje podane zadania i zwraca wynik jednego z nich. Druga z tych metod zgłasza wyjątek `TimeoutException`, jeśli zostanie przekroczony dozwolony czas.

- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

Wykonuje dane zadania i zwraca wyniki ich wszystkich. Druga z tych metod zgłasza wyjątek `TimeoutException`, jeśli zostanie przekroczony dozwolony czas.

`java.util.concurrent.ExecutorCompletionService 5.0`

- `ExecutorCompletionService(Executor e)`
Tworzy obiekt typu `ExecutorCompletionService`, który przechowuje wyniki zadań określonego egzekutora.
- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
Przekazuje zadanie do egzekutora.
- `Future<T> take()`
Usuwa następny wynik lub włącza blokadę, jeśli nie ma dostępnych żadnych wyników.
- `Future<T> poll()`
- `Future<T> poll(long time, TimeUnit unit)`

Usuwa następny wynik lub zwraca wartość `null`, jeśli nie ma dostępnych żadnych wyników. Druga wersja tej metody oczekuje określona ilość czasu.

14.9.4. Szkielet rozgałęzienie-złączenie

W niektórych aplikacjach używanych jest wiele wątków, z których większość jest nieaktywna. Przykładem jest serwer sieciowy obsługujący każde połączenie w osobnym wątku. Są też aplikacje tworzące po jednym wątku dla każdego rdzenia procesora. Robią tak choćby aplikacje wykonujące wymagające obliczenia, np. przy przetwarzaniu grafiki albo filmów. Szkielet rozgałęzienie-złączenie (ang. *fork-join*), który powstał w Java SE 7, służy do rozwiązywania problemów dotyczących drugiego z wymienionych przypadków. Wyobraźmy sobie, że mamy zadanie przetwarzania, które można naturalnie rozłożyć na podzadania:

```
if (rozmiarProblemu < prog)
    rozwiąż problem bezpośrednio
else
{
    podziel problem na części
    rekurencyjnie wykonaj każdą część
    połącz wyniki
}
```

Jednym z przykładów jest przetwarzanie obrazu. Obraz można poprawić, przekształcając jego górną i dolną połowę. Jeśli ma się do dyspozycji wystarczającą liczbę wolnych procesorów, operacje te można wykonywać równocześnie (trzeba będzie dodatkowo wykonać pracę związaną z połączeniem połówek, ale to mało istotny szczegół).

My przeanalizujemy prostszy przykład. Przypuśćmy, że chcemy się dowiedzieć, ile elementów tablicy spełnia pewien warunek. Dzielimy ją na pół, wykonujemy obliczenia dla każdej z połówek osobno, a następnie sumujemy wyniki.

Aby wykonać nasze rekursywne obliczenia w odpowiedni sposób, utworzymy klasę rozszerzającą klasę `RecursiveTask<T>` (jeśli wynik obliczenia jest typu `T`) lub `RecursiveAction` (jeśli nie ma wyniku). Przesłonimy metodę `compute`, aby generowała i wywoływała części zadania oraz łączyła ich wyniki.

```
class Counter extends RecursiveTask<Integer>
{
    . . .
    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            bezpośrednie rozwiązanie problemu
        }
        else
        {
            int mid = (from + to) / 2;
            Counter first = new Counter(values, from, mid, filter);
            Counter second = new Counter(values, mid, to, filter);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}
```

Metoda `invokeAll` otrzymuje liczbę zadań i wyłącza blokadę, dopóki wszystkie one nie zostaną wykonane. Metoda `join` generuje wynik. Stosujemy ją do wszystkich podzadań, aby otrzymać sumę.



Istnieje też metoda `get` do pobierania aktualnego wyniku, ale jest ona mniej atrakcyjna, ponieważ może zgłaszać kontrolowane wyjątki, których nie możemy ponownie zgłaszać w metodzie `compute`.

Pełny kod źródłowy przykładu jest przedstawiony na listingu 14.13.

Listing 14.13. forkJoin/forkJoinTest.java

```
package forkJoin;

import java.util.concurrent.*;

/**
 * Program demonstrujący szkielet rozgałęzienie-złączenie
 * @version 1.00 2012-05-20

```

```

* @author Cay Horstmann
*/
public class ForkJoinTest
{
    public static void main(String[] args)
    {
        final int SIZE = 10000000;
        double[] numbers = new double[SIZE];
        for (int i = 0; i < SIZE; i++) numbers[i] = Math.random();
        Counter counter = new Counter(numbers, 0, numbers.length,
            new Filter()
            {
                public boolean accept(double x) { return x > 0.5; }
            });
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(counter);
        System.out.println(counter.join());
    }
}

interface Filter
{
    boolean accept(double t);
}

class Counter extends RecursiveTask<Integer>
{
    public static final int THRESHOLD = 1000;
    private double[] values;
    private int from;
    private int to;
    private Filter filter;

    public Counter(double[] values, int from, int to, Filter filter)
    {
        this.values = values;
        this.from = from;
        this.to = to;
        this.filter = filter;
    }

    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            int count = 0;
            for (int i = from; i < to; i++)
            {
                if (filter.accept(values[i])) count++;
            }
            return count;
        }
        else
        {
            int mid = (from + to) / 2;
            Counter first = new Counter(values, from, mid, filter);

```

```
        Counter second = new Counter(values, mid, to, filter);
        invokeAll(first, second);
        return first.join() + second.join();
    }
}
```

Szkielet rozgałęzienie-złączenie wykorzystuje efektywny algorytm heurystyczny pozwalający zrównoważyć obciążenie poszczególnych wątków, o nazwie **podkradanie pracy** (ang. *work stealing*). Każdy wątek roboczy ma kolejkę dwukierunkową dla zadań i podzadania umieszcza na jej początku (tylko jeden wątek ma dostęp do tego miejsca, więc nie ma potrzeby stosować blokady). Gdy wątek jest nieaktywny, „podkrada” zadanie z końca innej kolejki dwukierunkowej. Jako że duże podzadania są w ogonie, do podkradania dochodzi rzadko.

14.10. Synchronizatory

W pakiecie `java.util.concurrent` znajduje się kilka klas, które ułatwiają zarządzanie zbiogrami spokrewnionymi ze sobą zadań — zobacz tabela 14.3. Algorytmy te udostępniają gotowe rozwiązania często spotykanych problemów związanych ze współpracą pomiędzy wątkami. Mając zestaw współpracujących ze sobą wątków działających według jednego z wzorców, należy — zamiast we własnym zakresie tworzyć zbiór blokad i warunków — użyć jednej z tych klas.

14.10.1. Semafony

Z założenia semafor służy do zarządzania pewną liczbą **zezwoleń** (ang. *permit*). Aby przejść obok semafora, wątek próbuje uzyskać zezwolenie, wywołując w tym celu metodę `acquire`. Liczba dostępnych zezwoleń jest ograniczona, co pozwala na kontrolę liczby wątków, które mogą przejść dalej. Inne wątki mogą wydawać zezwolenia za pomocą metody `release` (w rzeczywistości nie istnieją żadne obiekty zezwoleń, ich licznik jest po prostu przechowywany w semaforze). Jako że dostępna jest określona liczba zezwoleń, semafor ogranicza liczbę wątków, które mogą przejść. Ponadto zezwolenie nie musi zostać zwolnione przez wątek, który je uzyskał. Każdy wątek może wydać dowolną liczbę zezwoleń, a więc potencjalnie może zwiększyć ich liczbę powyżej początkowego limitu.

Semafony wynalazł w 1968 roku programista o nazwisku Edsger Dijkstra, który potrzebował **mechanizmu synchronizacji**. Wykazał on, że semafony mogą być szybkie i są na tyle wszechstronne, iż mogą służyć do rozwiązania wielu często występujących problemów związanych z synchronizacją. W prawie każdej książce na temat systemów operacyjnych znajduje się opis implementacji kolejek ograniczonych wykorzystujących semafony.

Oczywiście programiści aplikacji nie powinni ponownie wynajdować kolejek ograniczonych. Semafony rzadko odpowiadają typowym sytuacjom programistycznym.

Tabela 14.3. Synchronizatory

Klasa	Działanie	Zastosowanie
CyclicBarrier	Pozwala zbiorowi wątków oczekiwać, aż określona ich liczba osiągnie pewną wspólną barierę, a następnie opcjonalnie wykonuje akcję barierii.	Gdy pewna liczba wątków musi się zakończyć, zanim ich wyniki będą mogły być użyte.
Phaser	Podobny do CyclicBarrier, ale ze zmiennym licznikiem.	Wprowadzony w Java SE 7.
CountDownLatch	Pozwala zbiorowi wątków oczekiwać, aż licznik zostanie zmniejszony do zera.	Kiedy jeden lub więcej wątków musi oczekiwać, aż wystąpi określona liczba zdarzeń.
Exchanger	Umożliwia dwóm wątkom wymieniać się obiektami, jeśli oba są do tej wymiany gotowe.	Kiedy dwa wątki działają na dwóch egzemplarzach tej samej struktury danych — jeden go zapełnia, a drugi opróżnia.
Semaphore	Pozwala zbiorowi wątków oczekiwać, aż będą dostępne pozwolenia na kontynuację.	Do ograniczania liczby wątków mających dostęp do zasobu. Jeśli liczba pozwoleń wynosi jeden, blokada wątków jest zdejmowana, gdy inny wątek wyda pozwolenie.
SynchronousQueue	Pozwala wątkowi na przekazanie obiektu do innego wątku.	Do przesyłania obiektów z jednego wątku do innego, kiedy oba są na to gotowe, bez jawnej synchronizacji.

14.10.2. Klasa CountDownLatch

Obiekt klasy CountDownLatch zmusza wątki do oczekiwania, aż licznik dojdzie do zera. Zatrzaszcen ten jest jednorazowego użytku, to znaczy, że jeśli licznik dojdzie do zera, nie można go zwiększyć.

Przydatnym rodzajem takiego zatrzaszku jest zatrzaszcen z licznikiem o wartości 1. Stanowi on jednorazową bramkę. Wątki są zatrzymywane przed bramką, dopóki inny wątek nie ustawi licznika na zero.

Wyobraźmy sobie na przykład zestaw wątków, które do wykonania swoich zadań potrzebują pewnych danych inicjujących. Wątki są uruchomione i czekają przed bramką. Inny wątek przygotowuje dane. Kiedy jest gotowy, wywołuje metodę `countdown` i wszystkie wątki kontynuują swoje zadania.

Aby sprawdzić, kiedy wszystkie wątki zakończyły swoje działania, można użyć kolejnego zatrzaszku. Należy zainicjować go liczbą wątków. Każdy wątek przed zakończeniem działania odejmuje jeden od licznika zatrzaszku. Inny wątek, który zbiera wyniki tej pracy, czeka na zatrzaszku i przechodzi do działania, gdy wszystkie pozostałe wątki zakończą swoje działanie.

14.10.3. Bariery

Klasa `CyclicBarrier` służy do tworzenia obiektów nazywanych **barierami** (ang. *barrier*). Wyobraźmy sobie kilka wątków, z których każdy wykonuje porcję obliczeń stanowiącą fragment jednej całości. Kiedy wszystkie części są gotowe, ich wyniki trzeba połączyć. Kiedy wątek zakończy swoje zadanie, zatrzymuje się na barierze. Kiedy wszystkie wątki dotrą do bariery, zostaje ona otwarta i wątki mogą kontynuować działanie.

Oto szczegółowa analiza tego problemu. Najpierw tworzymy barierę, przekazując do niej liczbę wątków biorących udział w zadaniu:

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Każdy z wątków wykonuje jakieś działania i po ich zakończeniu wywołuje na rzecz bariery metodę `await`:

```
public void run()
{
    doWork();
    barrier.await();
    ...
}
```

Metoda `await` może przyjmować opcjonalny parametr czasowy:

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

Jeśli któryś z wątków oczekujących na barierę zniknie, bariera zostaje **złamana** (wątek może zniknąć, kiedy wywoła metodę `await` z ograniczeniem czasowym lub zostanie przerwany). W takim przypadku metoda `await` wszystkich pozostałych wątków zgłasza wyjątek `BrokenBarrierException`. Metoda `await` oczekujących wątków zostaje natychmiast zakończona.

Można określić opcjonalną **akcję bariery**, która będzie wykonywana, kiedy wszystkie wątki osiągną tę barierę:

```
Runnable barrierAction = ...;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

Akcja ta może zbierać wyniki poszczególnych wątków.

Bariera jest **cykliczna** (ang. *cyclic*), ponieważ po uwolnieniu wszystkich oczekujących wątków można jej użyć ponownie. Różni się ona pod tym względem od zatrzasku `CountDownLatch`, który może zostać użyty tylko jeden raz.

Klasa `Phaser` pozwala na większą elastyczność, umożliwiając zmienianie liczby wątków biorących udział w fazach.

14.10.4. Klasa Exchanger

Obiekty klasy `Exchanger` znajdują zastosowanie, gdy dwa wątki działają na dwóch egzemplarzach jednego bufora danych. Z reguły jeden z nich zapełnia bufor, a drugi pobiera te dane. Kiedy oba wątki są gotowe, wymieniają się buforami.

14.10.5. Kolejki synchroniczne

Kolejka synchroniczna jest mechanizmem pozwalającym skojarzyć w pary wątki producenckie i konsumentckie. Gdy wątek wywoła metodę `put` na obiekcie `SynchronousQueue`, zostaje on zablokowany do czasu, aż inny wątek wywoła metodę tąż i odwrotnie. W odróżnieniu od obiektów klasy `Exchanger`, dane są przekazywane tylko w jednym kierunku — od producenta do konsumenta.

Mimo że klasa `SynchronousQueue` implementuje interfejs `BlockingQueue`, to w istocie nie jest kolejką — nie zawiera żadnych elementów i jej metoda `size` zawsze zwraca wartość 0.

14.11. Wątki a biblioteka Swing

Na początku tego rozdziału napisaliśmy, że jednym z powodów używania wątków w programach jest usprawnienie możliwości ich interakcji z użytkownikiem. Kiedy program ma wykonywać jakieś czasochłonne obliczenia, powinno się uruchamiać nowy wątek roboczy, zamiast blokować cały interfejs użytkownika.

Należy jednak ściśle kontrolować, jakie zadania wykonuje wątek roboczy, ponieważ, co może być pewnym zaskoczeniem, Swing **nie jest bezpieczny dla wątków**. Manipulacja elementami interfejsu użytkownika w wielu wątkach może doprowadzić do jego awarii.

Aby sprawdzić, na czym polega ten problem, uruchom program z listingu 14.14. Kliknięcie przycisku *Zły* powoduje uruchomienie nowego wątku. Jego metoda `run` „dręczy” pole listy rozwijalnej, dodając do niej i usuwając z niej losowe wartości.

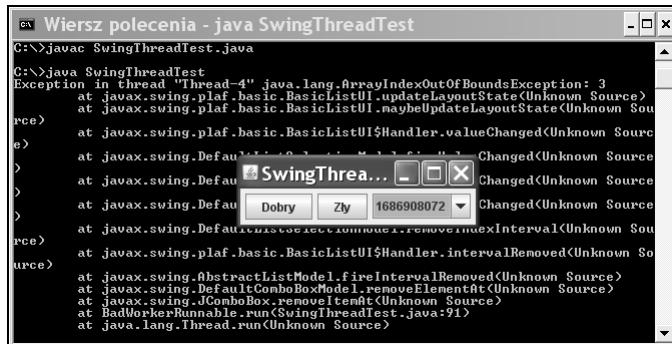
```
public void run()
{
    try
    {
        while (true)
        {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
                combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
                combo.removeItemAt(i % combo getItemCount());
            sleep(1);
        }
        catch (InterruptedException e) {}
    }
}
```

Wypróbujmy ten program. Kliknij przycisk *Zły*. Kliknij kilkakrotnie pole listy. Porusz paskiem przewijania. Przesuń okno. Jeszcze raz kliknij przycisk *Zły*. Poklikaj listę rozwijalną. W końcu powinien się pojawić komunikat o wyjątku (rysunek 14.8).

Co się stało? Kiedy do listy dodawany jest element, uruchamia ona zdarzenie aktualizacji ekranu. Wtedy do akcji wchodzi kod wyświetlający komponenty na ekranie, który wczytuje

Rysunek 14.8.

Komunikaty o wyjątkach w oknie konsoli



aktualny rozmiar pola listy i przygotowuje się do wyświetlenia wartości. Jednak wątek roboczy nie przestaje działać — to od czasu do czasu powoduje zmniejszenie licznika wartości na liście. Kod odpowiedzialny za wyświetlanie komponentów spodziewa się więcej wartości w modelu, niż ich rzeczywiście jest, i prosi o nieistniejące wartości, co powoduje powstanie wyjątku `ArrayIndexOutOfBoundsException`.

Sytuacji tej można było uniknąć, umożliwiając programiście zablokowanie pola listy podczas jego wyświetlania. Jednak projektanci biblioteki Swing postanowili nie robić nic w kierunku bezpieczeństwa dla wątków, i to z dwóch powodów. Synchronizacja jest czasochłonna, a nikt nie chciał, aby Swing był jeszcze wolniejszy. Ponadto zespół pracujący nad Swingiem wziął pod uwagę doświadczenia innych zespołów, które pracowały nad zestawami narzędzi do budowy bezpiecznych dla wątków interfejsów. Programiści wykorzystujący tego typu narzędzia mieli problemy z opanowaniem wymagań związanych z synchronizacją i często tworzyli programy, które łatwo wpadały w zakleszczenia.

14.11.1. Uruchamianie czasochłonnych zadań

Używając wątków w połączeniu z biblioteką Swing, trzeba dostosować się do dwóch prostych zasad:

- Jeśli jakieś działanie jest czasochłonne, należy je wykonać w osobnym wątku roboczym — nigdy w wątku dystrybucji zdarzeń.
 - Nie należy operować na komponentach Swing w żadnym innym wątku niż wątek dystrybucji zdarzeń.

Podstawę do sformułowania pierwszej z wymienionych zasad łatwo zrozumieć. Jeśli w wątku dystrybucji zdarzeń będzie wykonywane czasochłonne działanie, aplikacja będzie wyglądała na zawieszoną, ponieważ nie będzie mogła reagować na żadne inne zdarzenia. Wątek ten nie powinien nigdy wywoływać żadnych metod wejścia-wyjścia, które mogą zablokować się na stałe, ani metody `sleep` (jeśli musisz odczekać określona ilość czasu, użyj zdarzeń zegara).

Druga z wymienionych zasad w programowaniu Swing jest często nazywana **zasadą jednego wątku**. Szerzej na jej temat piszemy w sekcji 14.11.3.

Wydaje się, że zasady te są ze sobą sprzeczne. Wyobraźmy sobie, że uruchamiamy osobny watek do wykonania czasochłonnego zadania. Zazwyczaj podczas pracy wątku chcemy pokazać

postęp za pomocą aktualizacji interfejsu użytkownika. Po ukończeniu zadania aktualizujemy GUI jeszcze jeden raz. Nie możemy jednak operować na komponentach Swing z poziomu wątku. Na przykład aktualizacja paska postępu lub etykiety tekstopowej nie może polegać na zmianie wartości w wątku.

Rozwiązaniem tego problemu są dwie metody użytkowe, za pomocą których można w dowolnym wątku dodać dowolne akcje do kolejki zdarzeń. Wyobraźmy sobie na przykład, że co jakiś czas chcemy w wątku aktualizować etykię, aby pokazać postęp operacji. Nie możemy wywołać metody `label.setText` bezpośrednio w tym wątku.

W zamian należy użyć metod `invokeLater` i `invokeAndWait` z klasy `EventQueue`, aby wywołanie to zostało wykonane w wątku dystrybucji zdarzeń.

Oto wymagane czynności. Kod Swing umieszczamy w metodzie `run` klasy implementującej interfejs `Runnable`. Następnie tworzymy obiekt tej klasy i przekazujemy go do statycznej metody `invokeLater` lub `invokeAndWait`. Poniższa przykładowa procedura aktualizuje tekst etykiety:

```
EventQueue.invokeLater(new
    Runnable()
    {
        public void run()
        {
            label.setText("ukończono " + percentage + "%");
        }
    });
}
```

Metoda `invokeLater` zwraca wartość natychmiast po tym, jak zdarzenie zostanie wysłane do kolejki zdarzeń. Metoda `run` jest wykonywana asynchronicznie. Metoda `invokeAndWait` czeka, aż metoda `run` zostanie wykonana.

W przypadku aktualizacji etykiety postępu bardziej odpowiednia jest metoda `invokeLater`. Użytkownicy raczej przedkładają szybkość działania wątku roboczego nad precyzję wskaźnika postępu.

Obie te metody wykonują metodę `run` w wątku dystrybucji zdarzeń — nie jest tworzony żaden nowy wątek.

Listing 14.14 przedstawia program demonstrujący bezpieczną modyfikację zawartości listy rozwijalnej za pomocą metody `invokeLater`. Kliknięcie przycisku *Dobry* powoduje, że wątek wstawia i usuwa liczby, ale modyfikacje te odbywają się w wątku dystrybucji zdarzeń.

Listing 14.14. swing/SwingThreadTest.java

```
package swing;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

/**
 * Program udowadniający, że wątek działający równolegle z wątkiem dystrybucji zdarzeń może
 * powodować błędy w komponentach Swing

```

```
* @version 1.23 2007-05-17
* @author Cay Horstmann
*/
public class SwingThreadTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new SwingThreadFrame();
                frame.setTitle("SwingThreadTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka mająca dwa przyciski służące do zapełniania listy w osobnym wątku. Przycisk Dobry
 * wykorzystuje kolejkę zdarzeń, a Zły modyfikuje listę bezpośrednio.
*/
class SwingThreadFrame extends JFrame
{
    public SwingThreadFrame()
    {
        final JComboBox<Integer> combo = new JComboBox<>();
        combo.insertItemAt(Integer.MAX_VALUE, 0);
        combo.setPrototypeDisplayValue(combo.getItemAt(0));
        combo.setSelectedIndex(0);

        JPanel panel = new JPanel();

        JButton goodButton = new JButton("Dobry");
        goodButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                new Thread(new GoodWorkerRunnable(combo)).start();
            }
        });
        panel.add(goodButton);
        JButton badButton = new JButton("Zły");
        badButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                new Thread(new BadWorkerRunnable(combo)).start();
            }
        });
        panel.add(badButton);

        panel.add(combo);
        add(panel);
```

```

        pack();
    }
}

/*
 * Klasa modyfikująca listę rozwijaną poprzez dodanie do niej i usunięcie z niej losowych liczb. Może to
 * spowodować błędy, ponieważ metody listy rozwijalnej nie są synchronizowane, przez co wątek roboczy
 * i wątek dystrybucji zdarzeń uzyskują dostęp do tej listy.
 */

class BadWorkerRunnable implements Runnable
{
    private JComboBox<Integer> combo;
    private Random generator;

    public BadWorkerRunnable(JComboBox<Integer> aCombo)
    {
        combo = aCombo;
        generator = new Random();
    }

    public void run()
    {
        try
        {
            while (true)
            {
                int i = Math.abs(generator.nextInt());
                if (i % 2 == 0) combo.insertItemAt(i, 0);
                else if (combo getItemCount() > 0) combo.removeItemAt(i %
                    combo getItemCount());
                Thread.sleep(1);
            }
        }
        catch (InterruptedException e)
        {
        }
    }
}

/*
 * Klasa modyfikująca listę rozwijaną poprzez dodanie do niej i usunięcie z niej losowych liczb.
 * Aby uniknąć uszkodzenia tej listy, operacje edycji są przesyłane do wątku dystrybucji zdarzeń.
 */

class GoodWorkerRunnable implements Runnable
{
    private JComboBox<Integer> combo;
    private Random generator;

    public GoodWorkerRunnable(JComboBox<Integer> aCombo)
    {
        combo = aCombo;
        generator = new Random();
    }

    public void run()
    {

```

```
try
{
    while (true)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                int i = Math.abs(generator.nextInt());
                if (i % 2 == 0) combo.insertItemAt(i, 0);
                else if (combo.getItemCount() > 0) combo.removeItemAt(i
                    % combo getItemCount());
            }
        });
        Thread.sleep(1);
    }
}
catch (InterruptedException e)
{
}
```

java.awt.EventQueue 1.1

■ static void invokeLater(Runnable runnable) **1.2**

Po przetworzeniu oczekujących zdarzeń powoduje wykonanie metody run obiektu klasy implementującej interfejs Runnable w wątku dystrybucji zdarzeń.

■ static void invokeAndWait(Runnable runnable) 1.2

Po przetworzeniu oczekujących zdarzeń powoduje wykonanie metody `run` obiektu klasy implementującej interfejs `Runnable` w wątku dystrybucji zdarzeń. Metoda ta blokuje do czasu, aż metoda `run` zostanie zakończona.

- static boolean isDispatchThread() **1.2**

Zwraca wartość true, jeśli metoda działa w wątku dystrybucji zdarzeń.

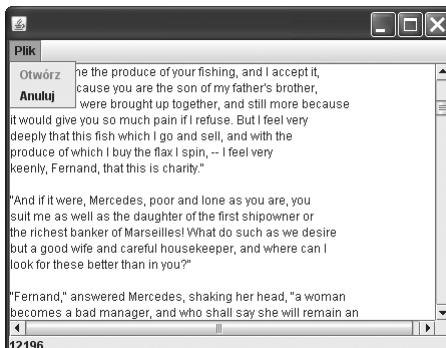
14.11.2. Klasa SwingWorker

Kiedy użytkownik wyda polecenie, którego wykonanie zajmuje dużo czasu, zazwyczaj do jego wykonania uruchamiamy nowy wątek. Jak pamiętamy z poprzedniego podrozdziału, w wątku tym do aktualizacji interfejsu użytkownika powinno się użyć metody `EventQueue.invokeLater`. Klasa `SwingWorker` pozwala zmniejszyć ilość żmudnej pracy związanej z implementacją zadań wykonywanych w tle.

Program przedstawiony na listingu 14.15 posiada polecenia ładowania pliku tekstowego i anulowania tego procesu. Aplikację tę należy testować na pliku o dużych rozmiarach, jak *The Count of Monte Cristo* znajdującym się w katalogu *gutenberg* razem z katalogami z kodem. Plik jest ładowany w osobnym wątku. W trakcie tej operacji polecenie *Otwórz*

w menu *Plik* jest nieaktywne, a *Anuluj* aktywne (rysunek 14.9). Po wczytaniu każdej linijki tekstu aktualizowany jest licznik w pasku stanu. Po ukończeniu ładowania polecenie *Otwórz* staje się z powrotem aktywne, polecenie *Anuluj* nieaktywne, a w pasku stanu wyświetla się napis *Zakończono*.

Rysunek 14.9.
Ładowanie pliku
w osobnym wątku



Listing 14.15. swingWorker/SwingWorkerTest.java

```
package swingWorker;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.List;
import java.util.concurrent.*;

import javax.swing.*;

/**
 * Program demonstrujący wątek roboczy wykonujący potencjalnie czasochłonne zadanie
 * @version 1.1 2007-05-18
 * @author Cay Horstmann
 */
public class SwingWorkerTest
{
    public static void main(String[] args) throws Exception
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new SwingWorkerFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka mająca obszar tekstowy pokazujący zawartość pliku tekstowego, menu pozwalające otworzyć plik
     * i anulować proces otwierania pliku oraz wiersz stanu pokazujący postęp ładowania pliku
     */
}
```

```
class SwingWorkerFrame extends JFrame
{
    private JFileChooser chooser;
    private JTextArea textArea;
    private JLabel statusLine;
    private JMenuItem openItem;
    private JMenuItem cancelItem;
    private SwingWorker<StringBuilder, ProgressData> textReader;
    public static final int TEXT_ROWS = 20;
    public static final int TEXT_COLUMNS = 60;

    public SwingWorkerFrame()
    {
        chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));

        textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
        add(new JScrollPane(textArea));

        statusLine = new JLabel(" ");
        add(statusLine, BorderLayout.SOUTH);

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("Plik");
        menuBar.add(menu);

        openItem = new JMenuItem("Otwórz");
        menu.add(openItem);
        openItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                // Wyświetlenie okna dialogowego wyboru pliku
                int result = chooser.showOpenDialog(null);

                // Jeśli plik został wybrany, zostanie on ustawiony jako ikona etykiety
                if (result == JFileChooser.APPROVE_OPTION)
                {
                    textArea.setText("");
                    openItem.setEnabled(false);
                    textReader = new TextReader(chooser.getSelectedFile());
                    textReader.execute();
                    cancelItem.setEnabled(true);
                }
            }
        });
    }

    cancelItem = new JMenuItem("Anuluj");
    menu.add(cancelItem);
    cancelItem.setEnabled(false);
    cancelItem.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            textReader.cancel(true);
        }
    });
}
```

```

        });
        pack();
    }

    private class ProgressData
    {
        public int number;
        public String line;
    }

    private class TextReader extends SwingWorker<StringBuilder, ProgressData>
    {
        private File file;
        private StringBuilder text = new StringBuilder();

        public TextReader(File file)
        {
            this.file = file;
        }

        // Poniższa metoda jest wykonywana w wątku roboczym — nie operuje na komponentach Swing

        @Override
        public StringBuilder doInBackground() throws IOException, InterruptedException
        {
            int lineNumber = 0;
            try (Scanner in = new Scanner(new FileInputStream(file)))
            {
                while (in.hasNextLine())
                {
                    String line = in.nextLine();
                    lineNumber++;
                    text.append(line);
                    text.append("\n");
                    ProgressData data = new ProgressData();
                    data.number = lineNumber;
                    data.line = line;
                    publish(data);
                    Thread.sleep(1); // Test operacji anulowania, nie ma potrzeby robienia tego w
                                     // swoich programach
                }
            }
            return text;
        }

        // Poniższe metody są wykonywane w wątku dystrybucji zdarzeń

        @Override
        public void process(List<ProgressData> data)
        {
            if (isCancelled()) return;
            StringBuilder b = new StringBuilder();
            statusLine.setText(" " + data.get(data.size() - 1).number);
            for (ProgressData d : data)
            {
                b.append(d.line);
                b.append("\n");
            }
        }
    }
}

```

```
        textArea.append(b.toString());
    }

    @Override
    public void done()
    {
        try
        {
            StringBuilder result = get();
            textArea.setText(result.toString());
            statusLine.setText("Zakończono");
        }
        catch (InterruptedException ex)
        {
        }
        catch (CancellationException ex)
        {
            textArea.setText("");
            statusLine.setText("Anulowano");
        }
        catch (ExecutionException ex)
        {
            statusLine.setText(ex.getCause());
        }
    }

    cancelItem.setEnabled(false);
    openItem.setEnabled(true);
}

}:
```

Program ten demonstruje typowy wygląd interfejsu użytkownika podczas wykonywania zadania w tle:

- Po każdym etapie pracy następuje aktualizacja interfejsu użytkownika w celu pokazania postępu.
- Po zakończeniu pracy w interfejsie dokonywana jest ostateczna zmiana.

Dzięki klasie `SwingWorker` zadanie to jest łatwe do wykonania. Wystarczy przeddefiniować metodę `doInBackground`, aby wykonywała czasochłonne działania, i co jakiś czas wywoływać metodę `publish` mającą na celu pokazanie postępu. Metoda ta jest wykonywana w wątku roboczym. Metoda `publish` powoduje wykonanie metody `process` w wątku dystrybucji zdarzeń. Jej zadaniem jest obsługa danych dotyczących postępu. Po zakończeniu pracy w wątku dystrybucji zdarzeń wywoływana jest metoda `done` pozwalająca zakończyć aktualizację interfejsu użytkownika.

Aby wykonać jakieś działania w wątku roboczym, należy utworzyć obiekt klasy `SwingWorker` (każdy taki obiekt może być użyty tylko jeden raz). Następnie należy wywołać metodę `execute`. Metodę tę z reguły wywołuje się na rzecz wątku dystrybucji zdarzeń, ale nie jest to wymogiem.

Z założenia obiekt klasy `SwingWorker` powinien zwrócić jakiś wynik. Dlatego klasa `SwingWorker<T, V>` implementuje interfejs `Future<T>`. Wynik ten można pobrać za pomocą metody

get tego interfejsu. Ponieważ metoda ta włącza blokadę, dopóki wynik nie jest dostępny, nie należy wywoływać jej bezpośrednio po metodzie `execute`. Dobrym rozwiązańiem jest wywoływanie jej dopiero wówczas, gdy wiadomo, że praca została zakończona. Zazwyczaj metodę `get` wywołuje się w metodzie `done` (wywołanie metody `get` nie jest konieczne — czasami wystarczy przetworzenie danych postępu).

Zarówno pośrednie dane postępu, jak i końcowy wynik mogą być dowolnego typu. Typy te są określone w klasie `SwingWorker` jako parametry typowe. Klasa `SwingWorker<T, V>` tworzy wynik typu `T` i dane postępu typu `V`.

Do anulowania zadania w toku służy metoda `cancel` z interfejsu `Future`. Kiedy zadanie jest anulowane, metoda `get` zgłasza wyjątek `CancellationException`.

Jak już wiemy, wywołanie w wątku roboczym metody `publish` spowoduje wywołanie metody `process` na rzecz wątku dystrybucji zdarzeń. Aby zwiększyć wydajność, wyniki zwrócone przez kilka wywołań metody `publish` można zgrupować w jednym wywołaniu metody `process`. Metoda `process` odbiera obiekt `List<V>` zawierający wszystkie wyniki pośrednie.

Użyjemy tej techniki do wczytywania pliku tekstowego. Okazuje się, że komponent `JTextArea` jest niezbyt szybki. Dodawanie linii tekstu z dużego pliku tekstowego (jak *The Count of Monte Cristo*) zajmuje dużo czasu.

Aby pokazać użytkownikowi, że coś się dzieje, w pasku stanu będziemy wyświetlać liczbę wczytyanych linijek tekstu. Dlatego dane postępu składają się z aktualnej liczby linii tekstu oraz aktualnej linii tekstu. Dane te pakujemy w prostej klasie wewnętrznej:

```
private class ProgressData
{
    public int number;
    public String line;
}
```

Ostateczny wynik stanowi tekst, który został wczytany do obiektu typu `StringBuilder`. W związku z tym klasa, której potrzebujemy, to `SwingWorker<StringBuilder, ProgressData>`.

Metoda `doInBackground` wczytuje dane z pliku wiersz po wierszu. Po każdym wierszu wywołujemy metodę `publish` publikującą numer i zawartość aktualnej linii.

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
    int lineNumber = 0;
    Scanner in = new Scanner(new FileInputStream(file));
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        lineNumber++;
        text.append(line);
        text.append("\n");
        ProgressData data = new ProgressData();
        data.number = lineNumber;
        data.line = line;
        publish(data);
        Thread.sleep(1); // Test operacji anulowania, nie ma potrzeby robienia tego w swoich programach.
    }
}
```

```

    }
    return text;
}

```

Po każdej linii tekstu usypiamy wątek na jedną milisekundę, aby można było spokojnie przetestować anulowanie. Oczywiście w programach przeznaczonych do użytku nie należy tego robić, aby ich nie spowalniać. Jeśli postawimy przed tym wierszem symbol komentarza, zauważymy, że tekst książki wczytuje się dość szybko i jest tylko kilka większych aktualizacji interfejsu użytkownika.



Aby program działał płynniej, pole tekstowe można aktualizować w wątku roboczym. Nie jest to jednak możliwe w przypadku wszystkich komponentów Swing. Prezentujemy ogólną technikę, polegającą na aktualizacji wszystkich komponentów w wątku dystrybucji zdarzeń.

Metoda `process` ignoruje wszystkie linie tekstu poza ostatnią oraz łączy wszystkie linie w jednej aktualizacji obszaru tekstowego.

```

@Override public void process(List<ProgressData> data)
{
    if (isCancelled()) return;
    StringBuilder b = new StringBuilder();
    statusLine.setText(" " + data.get(data.size() - 1).number);
    for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
    textArea.append(b.toString());
}

```

W metodzie `done` obszar tekstowy jest aktualizowany kompletnym tekstem, a polecenie *Anuluj* zostaje wyłączone.

Warto zwrócić uwagę na sposób uruchomienia obiektu klasy `SwingWorker` w słuchaczu zdarzeń elementu menu *Otwórz*.

Ta prosta technika pozwala na wykonywanie czasochłonnych zadań przy zachowaniu wrażliwości interfejsu użytkownika.

`javax.swing.SwingWorker<T, V>` **6**

■ `abstract T doInBackground()`

Tę metodę należy przedefiniować, aby wykonywała zadanie w tle i zwracała wynik swojego działania.

■ `void process(List<V> data)`

Tę metodę należy przedefiniować, aby przetwarzała pośrednie dane przetwarzania w wątku dystrybucji zdarzeń.

■ `void publish(V... data)`

Przesyła pośrednie dane postępu do wątku dystrybucji zdarzeń. Należy ją wywoływać w metodzie `doInBackground`.

- void execute()

Planuje wykonanie obiektu klasy SwingWorker w wątku roboczym.

- SwingWorker.StateValue getState()

Sprawdza stan obiektu SwingWorker — PENDING, STARTED lub DONE.

14.11.3. Zasada jednego wątku

Każda aplikacja w Javie zaczyna się w metodzie `main`, która działa w wątku głównym. W programach opartych na Swingu wątek ten żyje jednak bardzo krótko. Rozplanowuje konstrukcję interfejsu użytkownika w wątku dystrybucji zdarzeń i kończy działanie. Po utworzeniu interfejsu wątek dystrybucji zdarzeń przetwarza powiadomienia o zdarzeniach, takie jak wywołania metod `actionPerformed` czy `paintComponent`. Inne wątki, jak ten, który wysyła zdarzenia do kolejki zdarzeń, działają w tle, ale są niewidoczne dla programisty.

We wcześniejszej części tego rozdziału wprowadziliśmy zasadę jednego wątku, mówiącą, że na obiektach Swing należy operować wyłącznie w wątku dystrybucji zdarzeń. Przeanalizujemy tę zasadę nieco bardziej szczegółowo.

Od zasady jednego wątku jest kilka wyjątków.

- Słuchaczy zdarzeń można bezpiecznie dodawać i usuwać w każdym wątku. Oczywiście metody słuchaczy są wywoływanie w wątku dystrybucji zdarzeń.
- Niektóre metody Swing są bezpieczne wątkowo. W dokumentacji API są one oznaczone specjalnym zdaniem: „This method is thread safe, although most Swing methods are not” (**Metoda ta jest bezpieczna wątkowo, mimo że większość metod biblioteki Swing nie jest**). Najbardziej przydatne metody z tej grupy to:

```
JTextComponent.setText
JTextArea.insert
JTextArea.append
JTextArea.replaceRange
JComponent.repaint
JComponent.revalidate
```



Metody `repaint` używaliśmy już wielokrotnie, natomiast metoda `revalidate` jest znacznie mniej popularna. Jej zadaniem jest wymuszenie ułożenia komponentu po zmianie zawartości. W bibliotece AWT służy do tego metoda `validate` (aby wymusić ułożenie komponentu `JFrame`, konieczne jest wywołanie metody `validate` — `JFrame` jest komponentem, ale nie typu `JComponent`).

W przeszłości zasada jednego wątku była mniej restrykcyjna. Każdy wątek mógł tworzyć komponenty, ustawiać ich własności i dodawać je do kontenerów, jeśli żaden z tych komponentów nie był **realizowany**. Komponent jest realizowany, kiedy może odbierać zdarzenia rysowania lub walidacji. W związku z tym problem zaczynał się w chwili wywołania na rzecz komponentu metody `setVisible(true)` lub `pack` (!) bądź w momencie dodania go do zrealizowanego kontenera.

Tamta wersja zasady jednego wątku była bardzo dogodna. Pozwalała na utworzenie GUI w metodzie `main`, a następnie wywołanie metody `setVisible(true)` na rzecz ramki najwyższego poziomu. Nie było potrzeby kłopotliwego planowania obiektów implementujących interfejs `Runnable` na wątku dystrybucji zdarzeń.

Niestety niektórzy programiści komponentów nie zważyli na misterność pierwotnej zasady jednego wątku. Uruchamiali działania na rzecz wątku dystrybucji zdarzeń bez sprawdzenia, czy komponent został zrealizowany. Na przykład wywołanie metod `setSelectionStart` lub `setSelectionEnd` na rzecz komponentu `JTextComponent` spowoduje, że przesunięcie karetki zostanie wykonane w wątku dystrybucji zdarzeń, chociaż komponent jest niewidoczny.

Problemy te można by było odnaleźć i naprawić, ale projektanci Swinga wybrali łatwiejsze rozwiązanie. Orzekli, że bezpieczny dostęp do komponentów można uzyskać wyłącznie w wątku dystrybucji zdarzeń. Dlatego interfejs użytkownika musi być konstruowany w wątku dystrybucji zdarzeń przy użyciu metody `EventQueue.invokeLater`, którą oglądaliśmy we wszystkich przykładowych programach.

Oczywiście istnieje mnóstwo programów, które wciąż działają zgodnie ze starą wersją zasady jednego wątku, czyli inicjują interfejs użytkownika w głównym wątku. W aplikacjach tych istnieje ryzyko, że inicjacja interfejsu spowoduje działania w wątku dystrybucji zdarzeń będące w konflikcie z działaniami w wątku głównym. Jak pisaliśmy w rozdziale 7., nikt nie chce być tym nieszczęśliwcem, któremu się to przytrafi i który będzie musiał poświęcić mnóstwo czasu na odnalezienie błędu. Dlatego najlepiej ściśle trzymać się zasady jednego wątku.

W tym miejscu kończy się pierwszy tom *Core Java*. Opisano w nim podstawy języka Java oraz niektóre fragmenty jego API, które są potrzebne w większości projektów programistycznych. Mamy nadzieję, że podobała Ci się podróż przez podstawowe zagadnienia związane z Java i że udało Ci się tu znaleźć przydatne wiadomości. Dodatkowe informacje na temat programowania sieciowego, zaawansowanego programowania AWT i Swing, bezpieczeństwa aplikacji czy internacjonalizacji zostały zawarte w drugim tomie.

A

Słowa kluczowe Javy

Słowo kluczowe	Opis	Rozdział
abstract	Abstrakcyjna klasa lub metoda	5.
assert	Lokalizacja wewnętrznych błędów programu	11.
boolean	Typ logiczny	3.
break	Przerywa działanie instrukcji switch lub pętli	3.
byte	Ośmiobitowy typ całkowitoliczbowy	3.
case	Klauzula instrukcji switch	3.
catch	Klauzula bloku try przechwytyująca wyjątek	11.
char	Typ znaku Unicode	3.
class	Definicja klasy	4.
const	Nieużywane	
continue	Przekazuje sterowanie na koniec pętli	3.
default	Domyślna klauzula instrukcji switch	3.
do	Góra część pętli do-while	3.
double	Liczby zmiennoprzecinkowe o podwójnej precyzji	3.
else	Klauzula else instrukcji if	3.
enum	Wyliczenie	3.
extends	Definicja klasy nadzędnej innej klasy	4.
final	Stała, klasa lub metoda, której nie można przesłonić	5.
finally	Zawsze wykonywana część bloku try	11.
float	Zmiennoprzecinkowa liczba o pojedynczej precyzji	3.

Słowo kluczowe	Opis	Rozdział
for	Rodzaj pętli	3.
goto	Nieużywane	
if	Instrukcja warunkowa	3.
implements	Definicja interfejsu (lub interfejsów) implementowanego przez klasę	6.
import	Import pakietu	4.
instanceof	Sprawdzanie, czy obiekt jest egzemplarzem danej klasy	5.
int	32-bitowa liczba całkowita	3.
interface	Typ abstrakcyjny zawierający metody, które klasa może zaimplementować	6.
long	64-bitowa liczba całkowita	3.
native	Metoda zaimplementowana przez hosta	11. (tom II)
new	Przydzielenie pamięci dla nowego obiektu lub nowej tablicy	3.
null	Referencja null	3.
package	Pakiet, do którego należy klasa	4.
private	Cecha dostępna tylko dla metod określonej klasy	4.
protected	Cecha dostępna tylko dla metod określonej klasy, jej potomków i innych klas z tego samego pakietu	5.
public	Cecha dostępna dla wszystkich metod z wszystkich klas	4.
return	Zwraca wartość z metody	3.
short	16-bitowa liczba całkowita	3.
static	Cecha właściwa tylko swojej klasie, nie jej obiektom	3.
strictfp	Stosowanie ścisłych reguł dotyczących obliczeń na liczbach zmiennoprzecinkowych	2.
super	Obiekt lub konstruktor nadklasy	5.
switch	Instrukcja wyboru	3.
synchronized	Metoda lub blok kodu, który jest niepodzielny dla wątku	14.
this	Niejawnny argument metody lub konstruktor tej klasy	4.
throw	Zgłoszenie wyjątku	11.
throws	Wyjątki, które metoda może zgłosić	11.
transient	Oznaczanie danych, które nie powinny być trwale	1. (tom II)
try	Blok kodu przechwytyjącego wyjątki	11.
void	Oznaczenie metody niezwracającej żadnej wartości	3.
volatile	Zapewnienie spójnego dostępu do pola przez wiele wątków	14.
while	Pętla	3.

Skorowidz

A

- abstrakcja, 214
- ActiveX, 26, 35, 540
- adnotacja, 111, 640
 - @SafeVarargs, 643
 - @SuppressWarnings, 643, 648
- adres URL, 527, 546
- agregacja, 135
- akceleratory, 439
- akcesorium podglądu, 499
- akcesory, 142
- akcje, 355, 373
- aktualizacje, updates, 39
- aktualizowanie preferencji, 558
- aktywność komponentu, 375
- algorytm, 132, 718
 - binarySearch, 723
 - obliczania kodu mieszącego, 226
 - QuickSort, 121, 721
 - znajdujący największy element, 718
- algorytmy
 - sortujące, 720
 - w klasie Collections, 724
- allokacja listy tablicowej, 235
- analiza
 - funkcjonalności klasy, 252
 - MVC, 397
 - obiektów w czasie działania programu, 257
- animacja piłki, 736–742
- animowane gify, 546
- anonimowe klasy wewnętrzne, 289, 300, 363
- API
 - Javy, 33
 - JNLP, 525
 - Logging, 591
- Preferences, 555, 595
- String, 83
- applet, 28, 34, 53, 54
- applet, 511, 533
 - Jmol, 29
 - WelcomeApplet, 53
- aplety
 - implementacja, 533
 - komunikacja, 547
 - konwersja programów, 536
 - obrazy, 546
 - pliki audio, 546
 - środowisko działania, 547
 - uruchamianie, 535
- aplikacja
 - ImageViewer, 51
 - Java Web Start, 519
 - WebStartCalculator, 528
- aplikacje
 - graficzne, 50
 - serwerowe, 29
- architektura, framework, 706
 - kolekcji, 706, 710
 - model-widok-kontroler, 396
- argument, 61
- ASCII, 65
- asercje, 587
 - wyłączanie, 588
 - zastosowania, 589, 590
- asocjacja, 135
- atak, 25
- atrybut
 - classid, 540
 - codebase, 541
 - codetype, 540

atrybuty
 pozycjonujące, 540
 znacznika applet, 537–540
 znacznika param, 541, 542
 autoboxing, 32
 automatyczna konwersja typów, 32
 automatyczne opakowywanie, 241
 AWT, Abstract Window Toolkit, 314

B

bariera cykliczna, 814
 bariry, 814
 bazowy katalog drzewa pakietu, 187
 bezpieczeństwo, 25, 35
 typów, 639, 649
 wątków, 775
 biała księga Javy, 22
 biblioteka, 33
 AWT, 314, 387
 fdlibm, 74
 IFC, 314
 Java2D, 332, 333
 JFC, 314
 kolekcji, 666, 707
 refleksyjna, 247
 STL, 666
 Swing, 315, 815
 biblioteki
 struktur danych, 666
 zabezpieczeń, 25
 bit, 729
 blok, 98
 inicjujący, 175
 try-catch, 250, 578, 585
 try-finally, 578
 blokada, 762, 769
 jawna, 770
 odczytu, 783
 uczciwa, 764
 wewnętrzna, 770
 zapisu, 784
 bloki synchronizowane, 774
 blokowanie po stronie klienta, 774, 775
 błąd
 AssertionError, 587
 pomyłki o jeden, 719
 ThreadDeath, 752
 typu, 650
 typu cannot read, 46
 błędy
 danych wejściowych, 564
 komplikacji, 49, 153, 181, 629, 646
 programisty, 568

przydzielania pamięci, 25
 urządzeń, 564
 w Eclipse, 49
 w kodzie, 565
 wejścia-wyjścia, 565
 wewnętrzne, 568
 wykonawcze, 644
 zabezpieczeń, 25
 zaokrąglania, 64, 333

C

catch, 250
 cechy
 języka, 22, 33
 komponentu, 393
 certyfikat
 bezpieczeństwa, 524
 niebezpieczny, 525
 własny, 524
 chwytanie typu wieloznacznego, 655
 ciało metody, 60
 czasochłonne zadania, 816
 czerwonki, 343
 bezszeryfowe, 346
 nazwy, 343
 nazwy logiczne, 344
 PostScript type 1, 345
 styl, 345
 TrueType, 345
 wysokość, 346

D

dane, 195
 binarne, 26
 wejściowe, 89
 wyjściowe, 91
 debugger, 28, 621
 debugger JSwat, 623
 debugowanie, 609
 debugowanie aplikacji z GUI, 614
 definiowanie
 klasy, 148
 klasy ogólnej, 630
 kolorów, 340
 słuchacza, 356
 stałej klasowej, 69
 wątku, 745
 wyjątków kontrolowanych, 567
 zmiennej, 66, 68
 zmiennej obiektowej, 138
 zmiennej tablicowej, 117

- dekompilowanie pliku, 761
 dekrementacja, 71
 delegacja, 265
 delegacja zdarzeń, 356
 demony, 753
 dezaktywacja elementów menu, 441
 diagram
 dziedziczenia klasy, 215, 534
 dziedziczenia zdarzeń AWT, 387
 hierarchii wyjątków, 565
 klas, 136
 przepływy sterowania, 100–106, 110
 dodawanie
 akcji do menu, 375
 elementu do listy powiązanej, 678
 elementu do mapy, 702
 ikony, 435
 klasy do pakietu, 182
 klauzuli throws, 96
 komponentów, 401
 dokumentacja, 194
 API, 85–87
 JSR, 627
 założeń, 590
 dopasowywanie typów, 659
 dopełnienie, 459
 wewnętrzne, 452
 zewnętrzne, 452
 dostęp
 chroniony, 219
 do apletu, 539
 do elementów kolekcji, 672, 708
 do elementów listy tablicowej, 236
 do elementu tablicy, 117
 do formatera, 781
 do komponentów, 473
 do pakietów, 518
 do plików lokalizacyjnych, 516
 do pliku, 96
 do pliku JNLP, 521
 do pól, 155
 do pól generycznych, 637
 do prywatnych pól nadklasy, 202
 do sekcji krytycznej, 762
 do stanu obiektu, 289
 do usługi, 526
 do wartości, 243
 do węzła drzewa, 555
 do zasobów lokalnych, 525
 do zmiennej warunkowej, 775
 do zmiennych finalnych, 297
 jednoczesny wątków, 761
 swobodny, 719, 723
 wątków do struktury danych, 757
 drukowanie, 31
 drukowanie informacji o klasie, 252
 drzewo katalogów, 42
 duże liczby, big numbers, 62
 dymki, tooltips, 446
 dyrektywa
 #include, 182
 import, 90
 działanie
 kontrolera, 395
 metody transfer, 761
 dziedziczenia klasy Applet, 534
 dziedziczenie, inheritance, 133, 199, 268
 hierarchia, 206
 klasy abstrakcyjne, 214
 klasy finalne, 211
 metody finalne, 211
 ochrona dostępu, 219
 polimorfizm, 207
 pomiędzy klasami par, 649
 rzutowanie, 212
 typów ogólnych, 649
 wiązanie dynamiczne, 209
 wielokrotne, 279
 dzielenie
 całkowitoliczbowe, 69
 modulo, 69
 zmiennoprzecinkowe, 69
 dzienniki, 591
 dzienniki rotacyjne, 599

E

- Eclipse, 44, 47
 edycja
 kodu źródłowego, 48
 ścieżki dostępu, 39
 edytor tekstowy
 Emacs, 44
 JEdit, 44
 TextPad, 44
 edytowalna lista rozwijalna, 423
 EE, Enterprise Edition, 38
 egzemplarz klasy, 132
 elementy menu, 432
 aktywowanie, 440
 dezaktywowanie, 440
 ikony, 435
 pole wyboru, 436
 przelączniki, 436
 elementy tablicy, 116
 eliminacja wywołań funkcji, 27
 elipsa, 335

etykiety, 459
 HTML, 409
 komponentów, 408
 ewolucja Javy, 32

F

figury
 2D, 332
 geometryczne, 333
 filtr
 plików, 496, 497, 505
 rekordów, 600
 firma
 Oracle, 32, 34
 Sun Fellow, 30
 Sun Microsystems, 25, 34
 format

binarny liczby, 63
 JNLP, 519
 Unicode, 26
 XML, 556
 formatory, 600
 formatowanie
 danych wyjściowych, 91
 daty, 95
 funkcja unexpected, 569
 funkcje
 czysto wirtualne, 216
 matematyczne, 73
 sieciowe, 24
 składowe, 60

G

GC, Garbage Collector, 702
 generowanie
 dokumentacji, 194
 obiektów klas ogólnych, 646
 generyczne
 klasy, 629
 listy tablicowe, 233
 generyczny kod tablicowy, 261
 graficzny interfejs użytkownika, GUI, 28, 57, 313, 391, 614
 grafika, 313
 grupa, 754
 przycisków radiowych, 416
 wątków, 755
 zadań, 808
 GTK, 316

H

harmonogram
 wykonywania wątków, 750
 zadań, 696
 hasło, 90, 410
 hermetyzacja, 133, 155
 hierarchia
 dziedziczenia, 206
 dziedziczenia interfejsu Type, 660
 dziedziczenia klasy Component, 400
 dziedziczenia klasy JFrame, 321
 interfejsów, 278
 wyjątków, 566, 586
 zdarzeń, 387
 historia Javy, 30
 HTML, 33

I

IDE, 39, 47
 identyfikacja klas, 134
 IFC, Internet Foundation Classes, 314
 ikony, 435
 ikony komunikatów, 475
 implementacja
 apletów, 533
 ArrayList, 644
 interfejsu, 271, 273
 klasy Bank, 770
 klasy ogólnej, 629
 kolejki, 667
 import
 klas, 180
 statyczny, 182
 indeks, 123
 indeks argumentu, 95
 informacje
 o klasie, 252
 o typach, 28
 o typach czasu wykonywania, 248
 o typach generycznych, 659
 o typach obiektów, 248
 o uruchomionym programie, 613
 o zdarzeniach, 602
 inicjalizacja
 pól, 172
 pól statycznych, 176
 pól wartościami domyślnymi, 171
 tablic, 118
 z podwójną klamrą, 302
 zmiennej obiektowej, 138
 zmiennych, 68

- inkrementacja, 71
- instalacja
 - bibliotek, 41
 - dokumentacji, 41
 - filtru, 600
 - JDK, 38, 39
 - programów, 42
- instrukcja
 - break, 111–114
 - break z etykietą, 112
 - case, 111
 - continue, 113
 - do-while, 104
 - for, 77
 - goto, 111
 - if, 100, 113
 - if-else, 100
 - if-else if, 102
 - import, 180, 182
 - lock, 762
 - return, 578
 - switch, 109–111
 - try, 580, 581
 - while, 103
- instrukcje
 - sterujące, 98
 - warunkowe, 98, 109, 113
 - złożone, 99
- interfejs
 - Action, 373, 379, 433
 - ActionListener, 286, 300, 357, 364, 373, 389
 - AdjustmentListener, 389
 - AppletContext, 547
 - AutoCloseable, 580
 - BasicService, 527
 - BlockingDeque<E>, 793
 - BlockingQueue<E>, 792
 - ButtonModel, 397, 398, 417
 - Callable, 797
 - Callable<V>, 801
 - Cloneable, 282
 - Collection, 668–672, 679, 707, 711
 - Collection<E>, 672
 - Comparable, 272, 279, 308, 634, 654, 689
 - Comparator<T>, 690, 693
 - Condition, 769–771
 - Delayed, 792
 - Deque<E>, 695
 - Enumeration, 670, 727
 - ExecutorService, 803, 808
 - FileFilter, 497
 - Filter, 600
 - FocusListener, 389
 - Formattable, 92
 - Future, 798
 - Future<V>, 801
 - GenericArrayType, 660, 664
 - InvocationHandler, 307, 311
 - ItemListener, 389
 - Iterable, 117, 669
 - Iterator, 669, 677, 680
 - Iterator<E>, 674
 - klasy, 146
 - LayoutManager, 469
 - LayoutManager2, 469
 - LinkedList<E>, 683
 - List, 684, 708, 711
 - List<E>, 682
 - ListIterator, 677, 708
 - ListIterator<E>, 683
 - Lock, 764, 769, 771, 783
 - KeyListener, 389
 - Map, 707
 - Map<K, V>, 700
 - MenuListener, 441
 - MouseListener, 381, 383, 389
 - MouseMotionListener, 381, 383, 389
 - MouseWheelListener, 389
 - nasłuchu, 356
 - NavigableMap, 709
 - NavigableMap<K, V>, 716
 - NavigableSet, 709, 712
 - NavigableSet<E>, 694, 716
 - ParameterizedType, 660, 664
 - PersistenceService, 527
 - PersistentService, 528
 - Powered, 278
 - Queue, 666, 667
 - Queue<E>, 695
 - RandomAccess, 708, 721
 - Runnable, 797
 - ScheduledExecutorService, 807
 - Set, 708
 - Shape, 333
 - SortedMap, 709
 - SortedMap<K, V>, 716
 - SortedSet, 709, 712
 - SortedSet<E>, 693, 716
 - SwingConstants, 278, 408
 - Thread.UncaughtExceptionHandler, 754
 - TransferQueue, 788
 - TransferQueue<E>, 793
 - Type, 660
 - TypeVariable, 660, 664
 - WildcardType, 660, 664
 - WindowFocusListener, 389
 - WindowListener, 369, 372, 389
 - WindowStateListener, 372, 389

interfejsu, 219, 265

hierarchia, 278

implementacja, 271, 273

metody, 278

sprzężenie zwrotne, 286

zmienne, 277

interfejsy

architektury kolekcji, 707

kolekcyjne, 665, 707, 719

nasłuchowe, 389

nasłuchujące AWT, 389

przenośne, 27

użytkownika, 34, 391

znacznikowe, 282

interlinia, 346

interpreter, 27, 184

iterator jako parametr, 727

iteratory, 669, 670

J

JAR, Java Archive, 512

Java look and feel, 315

Java Micro Edition, 23

Java Plug-in, 540

Java Runtime System, 26

Java Web Start, 511, 519

JavaBeans, 247

JavaFX, 317

jawna inicjalizacja pól, 172

JDK, Java Development Kit, 37

jednostki kodowe, 66, 81

język

Algol, 165

C, 25

C#, 28, 34

C++, 23, 24

HTML, 33

J#, 28

J++, 28

Java, 22

JavaScript, 35

UML, 136

Visual Basic, 23, 137

języki

interpretowane, 34

obiektowe, 24

proceduralne, 24

JFC, Java Foundation Classes, 314

JIT, just-in-time compiler, 27

JNLP, Java Network Launch Protocol, 519

JRE, Java Runtime Environment, 38

JSR, Java Specification Requests, 627

K

kalendarz, 139

kalkulator, 403, 521

karta

HSB, 506

RGB, 506

Swatches, 506

katalog

bazowy drzева pakietu, 187

bin, 39

com, 183

gutenberg, 820

src, 43

klas

coupling, 135

diagramy, 136

dziedziczenie, 133, 200

identyfikacja, 134

implementacja interfejsu, 271

komentarze, 191

konstruktory, 137, 152

metody, 133

metody prywatne, 157

metody statyczne, 160

nadklasy, 200

plik źródłowy, 189

podklasy, 200

pola statyczne, 159

predefiniowanie, 137

projektowanie, 195

relacje, 135

rozszerzanie, 133

pola stałe, 158

ścieżka, 187

klasa, 58, 132

ExampleFileView, 499

AbstractAction, 374, 377

AbstractButton, 419, 434–436, 440

AbstractCollection, 672

AbstractList, 726

AbstractQueue, 668

AbstractSequentialList, 723

AbstractSet, 223

AccessibleObject, 257, 261

ActionMap, 376

AnonymousInnerClassTest, 301

Applet, 533, 537, 545, 549

AppletContext, 540, 548, 549

Array, 262

ArrayAlg, 663

ArrayBlockingQueue<E>, 791

ArrayDeque, 694

ArrayDeque<E>, 696

ArrayDeque, 668
 ArrayList, 234–236, 240, 628, 642, 674, 684
 ArrayList<T>, 650
 ArrayListTest, 238
 Arrays, 118, 121, 123
 AtomicInteger, 777
 AWTEvent, 387
 BallRunnable, 742
 BasicButtonUI, 398
 BasicService, 528, 531
 bazowa Object, 133, 220
 BigDecimal, 64, 114, 116
 BigInteger, 114, 115
 BigIntegerTest, 115
 BitSet, 726, 729
 BlockingQueueTest, 789
 BorderFactory, 419, 421
 BorderLayout, 401, 402
 BounceFrame, 737
 BuggyButtonTest, 622
 ButtonFrame, 360
 ButtonGroup, 417, 418
 ButtonModel, 417, 418
 ButtonUIListener, 398
 Calendar, 140
 CalendarTest, 145
 CheckBoxTest, 415
 CircularArrayQueue, 668
 Class, 233, 248–251, 255, 261, 658
 Class<T>, 658, 663
 CloneTest, 284
 Collections, 679, 711, 715, 724
 Color, 340, 342, 343
 ColorAction, 360, 362
 Component, 322, 325, 332, 343, 399, 408
 ConcurrentHashMap, 794
 ConcurrentHashMap<K, V>, 5.0, 795
 ConcurrentLinkedQueue<E>, 795
 ConcurrentSkipListMap, 794
 ConcurrentSkipListSet<E>, 795
 Console, 90, 91
 ConsoleHandler, 600, 607
 Constructor, 250–252, 256, 658
 ConstructorTest, 177
 Container, 362, 399
 CopyOfTest, 263
 CountDownLatch, 813
 Cursor, 381
 CyclicBarrier, 814
 Date, 139, 140
 DateFormatSymbols, 144, 148
 DateInterval, 638
 Dimension, 330
 Double, 276
 DrawTest, 337
 Ellipse2D, 335
 Ellipse2D.Double, 340
 Employee, 148, 151, 163, 184, 639
 EmployeeSortTest, 275
 EmployeeTest, 149
 Enum, 246
 EnumMap, 704
 EnumMap<K extends Enum<K>, V>, 706
 EnumSet, 704
 EnumSet<E extends Enum<E>>, 706
 EnumTest, 246
 EOFException, 570
 EqualsTest, 230
 Error, 565
 EventHandler, 364, 365
 EventObject, 363, 387
 EventTracer, 616
 Exception, 251, 565, 583
 Exchanger, 814
 Executors, 802
 ExtendedService, 527
 Field, 252, 256, 258, 261, 265
 File, 497
 FileContents, 531
 FileFilter, 504
 FileHandler, 600, 607
 FileInputStream, 567
 FileNameExtensionFilter, 505
 FileOpenService, 526, 532
 FileSaveService, 532
 FileView, 497, 505
 Filter, 609
 FlowLayout, 400
 Font, 345, 349
 FontMetrics, 351
 FontParamApplet, 541
 FontRenderContext, 346
 FontTest, 348
 ForkJoinTest, 811
 Formatter, 600, 609
 Frame, 318, 326
 FutureTask<V>, 802
 FutureTest, 799
 GenericReflectionTest, 661
 Graphics, 332, 343, 350, 353
 Graphics2D, 332, 333, 343, 351
 GraphicsDevice, 325, 617
 GraphicsEnvironment, 344, 620
 GregorianCalendar, 139–143, 146, 654
 GridBagConstraints, 454, 458
 GridLayout, 399, 405
 GroupLayout, 459, 463, 466
 Handler, 598, 607

klasa

HashMap<K, V>, 701
HashSet, 684, 686
HashSet<E>, 687
Hashtable, 709, 726
IdentityHashMap, 705
IdentityHashMap<K, V>, 706
ImageIcon, 327, 351
ImagePreviewer, 499
ImageTest, 352
InnerClassTest, 292
InputEvent, 386
InputMap, 376
Integer, 242, 276, 308
Iterator, 290
JApplet, 533
JButton, 361, 397
JCheckBox, 415
JCheckBoxMenuItem, 436
JColorChooser, 505, 509
JComboBox, 425, 725
JComponent, 328, 347, 351, 379, 408, 419, 438
JDialog, 484, 488
JEditorPane, 412
JFileChooser, 495, 503
JFrame, 318, 321, 331, 434, 484
klasa JLabel, 408, 499
klasa JList, 425
JMenu, 433
JMenuBar, 432
JMenuItem, 434, 440
JOptionPane, 288, 474, 481, 484
 JPanel, 330
JPasswordField, 406, 410
JPopupMenu, 437
JRadioButton, 418
JRadioButtonMenuItem, 436
JScrollPane, 413
JSlider, 426, 431, 640
JTextArea, 406, 410, 412
JTextComponent, 406
JTextField, 406, 408
JToolBar, 445, 447
KeyStroke, 375, 379
LayoutManager, 472
Line2D.Double, 340
LineBorder, 422
LineMetrics, 347, 350
LinkedBlockingQueue<E>, 792
LinkedHashMap, 702, 704
LinkedHashMap<K, V>, 705
LinkedHashSet, 702
LinkedHashSet<E>, 705
LinkedList, 290, 668, 675, 684, 694, 713

LinkedListQueue, 668
LinkedListTest, 681
LinkedTransferQueue, 788
ListIterator, 679
Lock, 762
Logger, 605
LogManager, 595
LogRecord, 609
LookAndFeelInfo, 368
Manager, 232
ManagerTest, 205
MapTest, 699
Math, 73, 74
MenuListener, 441
Method, 252, 265, 663
MethodTableTest, 266
Modifier, 252, 256
MouseEvent, 380, 386
MouseHandler, 383
MouseMotionHandler, 383
MouseMotionListener, 381
Object, 220
ObjectAnalyzer, 258
ObjectAnalyzerTest, 259
PackageTest, 183, 184
Pair, 303, 637, 641, 648, 655
Pair<T>, 659
PairTest1, 631
PairTest2, 635
PairTest3, 656
ParallelGroup, 468
ParamTest, 169
PasswordChooser, 490
Paths, 97
PersistenceService, 532
PersonTest, 217
PlafFrame, 368
Point2D, 335
Point2D.Double, 340
Preferences, 556, 560
PreferencesFrame, 558
PrintWriter, 97
PriorityBlockingQueue<E>, 792
PriorityQueue, 697
PriorityQueueTest, 697
Properties, 550–555, 709, 728
PropertiesTest, 551
Proxy, 307, 311
ProxyTest, 309
Rectangle2D, 334, 335
Rectangle2D.Double, 334, 339
Rectangle2D.Float, 334, 339
RectangularShape, 335, 339
RecursiveTask<T>, 810

- ReentrantLock, 762–764, 783
 ReentrantReadWriteLock, 783
 ReflectionTest, 253
 ResourceBundle, 596
 ResourceTest, 517
 Robot, 617, 618
 Runnable, 746
 RuntimeException, 566–568, 583
 Scanner, 89, 90, 97
 SequentialGroup, 468
 ServiceManager, 531
 SetTest, 686
 ShuffleTest, 721
 SimpleDateFormat, 781
 SimpleFrame, 319
 SimpleFrameTest, 318
 Singleton, 645
 SizedFrameTest, 324
 SoftBevelBorder, 421, 422
 SortedMap<K, V>, 701
 SQLException, 576
 Stack, 709, 729
 StackTraceElement, 581, 583
 StackTraceTest, 582
 StaticInnerClassTest, 305
 StaticTest, 162
 StreamHandler, 598
 StrictMath, 74
 String, 77, 83, 86, 211
 StringBuilder, 86, 88
 SwingThreadTest, 818
 SwingUtilities, 494
 SwingWorker, 820, 824, 826
 SwingWorkerTest, 821
 SynchronousQueue, 815
 System, 43, 554
 SystemColor, 341
 TalkingClock, 289, 295–297
 Thread, 647, 746, 749–753
 ThreadGroup, 754, 755
 ThreadLocal, 781
 ThreadLocal<T>, 782
 ThreadPoolTest, 804
 Throwable, 251, 565, 570, 581, 583
 TimePrinter, 291–295
 Timer, 286–288
 TimerTest, 287
 ToolBarTest, 447
 Toolkit, 288, 323, 327, 386
 TraceHandler, 307
 TransferRunnable, 780
 TreeMap<K, V>, 701
 TreeSet, 688, 691
 TreeSet<E>, 688, 694
 TreeSetTest, 691
 UIManager, 368
 Vector, 234, 684, 709, 726
 WeakHashMap, 702
 WeakHashMap<K, V>, 705
 Window, 322, 326
 WindowAdapter, 370
 WindowEvent, 372
 klasy
 abstrakcyjne, 214, 216, 279
 adaptacyjne, adapter class, 369
 anonimowe, 300
 bazowe, 200, 279
 blokad, 783
 finalne, 211
 generyczne, 234
 graniczne, 636
 kolekcyjne, 627, 672, 674, 709, 726
 kontenerowe, 709
 macierzyste, 200
 modelowe, 397
 niezmienne, 158
 ogólne, generic class, 629
 osłonowe, 241
 pochodne, 200
 podpisywane cyfrowo, 25
 pomocnicze, 453, 781
 potomne, 200
 proxy, 306, 311
 publiczne, 180
 specjalne, 702
 statyczne, 303
 surowe, 237
 szablonowe, 632
 w architekturze kolekcji, 710
 wewnętrzne, 271, 289
 anonimowe, 300
 bezpieczeństwo, 296
 dostęp do stanu obiektu, 289
 dostęp do zmiennych finalnych, 297
 lokalne, 296
 obsługa zdarzeń, 362
 prawa dostępu, 296
 referencja do klasy zewnętrznej, 293
 referencja do obiektu zewnętrznego, 291
 reguły składniowe, 293
 składnia, 289
 statyczne, 303
 wyjątków, 570
 wyliczeniowe, 245
 zagnieździone, 290
 zdarzeniowe AWT, 387
 klasyfikacja wyjątków, 565

- klauzula
 - catch, 572, 748
 - finally, 576, 578
 - throws, 96, 567
 - try, 572
- klawiatura, 375
- klawisze specjalne, 380
- klonowanie obiektów, 280, 285
- klucz URL, 528
- klucze, 698
- klucze należące do węzła, 560
- kod
 - bajtowy, 26
 - błędu, 570
 - kod generyczny, 630
 - kod maszynowy, 26
 - kod mieszący, hash code, 225, 684, 687
 - kod ogólny, 635
 - kod wyjścia, exit code, 60
- kodowanie
 - Unicode, 65, 67
 - UTF-16, 66, 82
- kolejka, queue, 666
 - ArrayBlockingQueue, 788
 - DelayQueue, 788
 - Deque, 694
 - LinkedBlockingQueue, 787
 - PriorityBlockingQueue, 788
 - Queue, 694
- kolejki
 - blokujące, 786
 - dostępu, 472
 - priorytetowe, 696
 - synchroniczne, 815
- kolejność
 - dostępu, 703
 - dostępu do komponentów, 473
 - ograniczeń, 637
- kolekcja, 117, 665
 - par, 698
 - wątków, 754
- kolekcje
 - bezpieczne wątkowo, 794, 796
 - ograniczone, 668
 - uporządkowane, 677, 708
 - w bibliotece, 675
- kolizja nazw, 180
- kolizje, 685
- kolor, 340
- kolor tła, 341, 507
- komentarze, 61
 - do klas, 191
 - do metod, 191
 - do pakietów, 194
 - do pól, 192
- dokumentacyjne, 190
- ogólne, 192
- komparator, 690
- kompilacja
 - programu, 44
 - w czasie rzeczywistym, 26
- kompilator, 25, 45, 776
 - czasu rzeczywistego, 34
 - javac, 188
 - JIT, 27, 212
- komponenty
 - Swing, 391–510
 - tekstowe, 411
- kompresja ZIP, 512
- komunikacja
 - między apletami, 540, 547
 - międzyprocesowa, 736
- komunikat, 592
 - o błędzie, 46, 50, 569
 - o wyjątkach, 816
- konektor UML, 136
- konfiguracja
 - komponentów, 319
 - menedżera dzienników, 598
 - projektu, 48
- konflikt metod, 648
- konkatenacja, 78
- konsola, 44
- konstruktor, 137, 152
 - bezargumentowy, 172
 - domyślny, 172
 - kopiujący, 140
 - przeciążony, 172
 - wirtualny, 250
- konstruktorysty
 - klasy FileHandler, 607
 - klasy HashSet<E>, 687
 - klasy TreeMap<K, V>, 701
 - klasy TreeSet<E>, 694
- kontekst
 - graficzny, 328
 - urządzenia, 328
- kontener, 330, 399
- kontrola
 - dostępu, 290
 - nazw, 290
 - typów, 627
- kontroler, controller, 394
- konwersja
 - łańcucha na liczbę, 242
 - pomiędzy kolekcjami a tablicami, 718
 - programów na aplety, 536
 - tablic, 718
 - typów, 650
 - typów numerycznych, 74

kończenie działania programu, 60
 kopie
 łańcucha, 80
 obrazu, 352
 kopiowanie
 głębokie, 281
 obiektów, 280
 pływkie, 281
 tablicy, 119
 zmiennej tablicowej, 119
 koszty uzyskania certyfikatu, 524
 kowariantne typy zwrotne, 209, 639
 kubelek, bucket, 685
 kury, 382
 kwalifikator `this`, 294

L

licencja GPL, 34
 liczba
 kliknięć, 381
 parametrów, 244
 liczby
 całkowite, 62
 zmiennoprzecinkowe, 64
 linia
 bazowa, 346, 347
 dolna pisma, 346
 górska pisma, 346
 lista
 ArrayList, 117
 kluczy, 556, 698
 modyfikowalna, 721
 rozwijalna, combo box, 423
 wątków, 779
 listy
 cykliczne, 666, 668
 dwukierunkowe, 25, 238, 674
 powiązane, 668, 674, 680
 tablicowe, 234, 236, 684
 surowe, 239
 z typem, 240
 lokalizacja, 143, 596
 komunikatów, 596
 pliku, 546
 lokalne klasy wewnętrzne, 289, 296

Ł

ładowanie
 klas, 307
 pliku w osobnym wątku, 821
 zasobów, 516

łańcuch
 blank, 548
 dziedziczenia, 206
 null, 81
 prompt, 91
 pusty, 81
 testowy, 407
 wyjątków, 575
 łańcuchy, 77
 łączenie, 78
 modyfikowanie, 79
 porównywanie, 79
 składanie, 86
 współdzielenie, 79
 zmiennalne, mutable, 80
 łączenie narastające, 27

M

makro assert, 588
 manifest, 512
 mapa, 697
 akcji, 376
 HashMap, 698
 haszowa, 795
 wejścia, 376
 własności, property map, 550, 553, 728
 mapy klawiaturowe, 376
 maska bitowa, 380
 maszyna wirtualna, JVM, 26, 514
 opcja -verbose, 612
 opcja -Xlint, 612
 opcja -Xprof, 614
 ME, Micro Edition, 38
 menedżer
 dzienników, 595
 zabezpieczeń, 522
 menu, 432
 menu podrzędne, pop-up menu, 437
 metadane, 32
 metoda, 133
 accept, 504
 acquire, 812
 actionPerformed, 286, 291, 357, 368, 414, 433
 add, 142, 237, 399, 433, 629, 677
 addActionListener, 364
 addAll, 629, 672, 682
 addBall, 737, 742
 addChangeListener, 426
 addChoosableFileFilter, 504
 addComponent, 467
 addContainerGap, 468
 addFirst, 683
 addGap, 467

metoda
 addGroup, 467
 addItem, 423, 426
 addLast, 683
 addPropertyChangeListener, 373
 addSeparator, 434, 447
 addSuppressed, 583
 addWindowListener, 370
 akcesora get, 141
 and, 730
 andNot, 730
 append, 87, 413
 appendCodePoint, 88
 Arrays.hashCode, 227
 Arrays.toString, 230
 asList, 711
 await, 766, 769, 783, 814
 awaitUninterruptibly, 783
 beep, 289
 binarySearch, 123, 308, 722
 BorderFactory, 421
 brighter, 341
 call, 801
 cancel, 802
 canRead, 532
 canWrite, 532
 cast, 658
 ceiling, 694
 charAt, 83
 checkedCollection, 713
 clear, 672, 730
 clone, 280–284
 close, 179, 580, 600, 607, 762
 codePointAt, 83
 codePointCount, 84
 compare, 276, 690, 693
 compareTo, 83, 246, 272–274, 299, 634, 653,
 693, 720
 Component.show, 320
 config, 605
 console, 91
 contains, 672, 680, 686
 containsAll, 672, 673
 containsKey, 700
 containsValue, 700
 copy, 724
 copyArea, 352, 354
 copyOf, 123
 countdown, 813
 create, 365
 createCompoundBorder, 422
 createCustomCursor, 382, 386
 createEmptyBorder, 421
 createEtchedBorder, 421
 createFont, 345
 createLineBorder, 421
 createLoweredBevelBorder, 421
 createMatteBorder, 421
 createParallelGroup, 467
 createRaisedBevelBorder, 421
 createScreenCapture, 618
 createTitledBorder, 422
 darker, 341
 decrementAndGet, 777
 metoda delay, 621
 metoda delete, 88
 deriveFont, 345, 350
 destroy, 537
 disjoint, 725
 divide, 115
 doInBackground, 824–826
 draw, 333, 340
 drawImage, 353
 drawString, 329, 341, 347, 351
 elements, 728
 endsWith, 83
 ensureCapacity, 236
 entering, 600, 605
 entrySet, 700
 equals, 79, 83, 124, 221, 242, 648, 705
 equalsIgnoreCase, 83
 execute, 827
 exiting, 600, 605
 fill, 340, 343, 724
 fillMenu, 725
 finalize, 179
 fine, 605
 finer, 605
 finest, 605
 firstKey, 701
 floor, 694
 flush, 600, 607
 Font.createFont, 345
 format, 609
 formatMessage, 609
 formatTo, 92
 forName, 248, 251
 frame.setUndecorated, 320
 frequency, 725
 get, 142, 236, 261, 264, 628, 682
 getActionCommand, 363, 388, 417
 getActionMap, 379
 getActualTypeArguments, 664
 getAllItems, 726
 getAncestorOfClass, 494
 getApplet, 540
 getAppletContext, 547, 549
 getAppletInfo, 545

getApplets, 548, 549
getAudioClip, 547
getAutoCreateContainerGaps, 467
getAutoCreateGaps, 467
getAvailableFontFamilyNames, 344
getBackground, 343
getBounds, 664
getCause, 583
getCenterX, 339
getClass, 222, 248, 641
getClassName, 368, 584
getClickCount, 380, 386
getCodeBase, 528, 531
getColor, 343, 510
getColumns, 408
getComponentPopupMenu, 438
getConstructor, 658, 659
getConstructors, 252, 255
getContentPane, 327, 331
getDeclaredFields, 261
getDeclaredConstructor, 658
getDeclaredConstructors, 252, 256
getDeclaredField, 261
getDeclaredFields, 252, 255, 258
getDeclaredMethods, 252, 255
getDeclaringClass, 256
getDefaultValue, 621
getDefToolkit, 288, 323, 327
getDelay, 788, 792
getDescent, 350
getDescription, 505
getDocumentBase, 546
getDouble, 258, 610
getEnumConstants, 658
getExceptionTypes, 256
getExtendedState, 326
getFamily, 349
getField, 261
getFields, 252, 255, 261
getFileName, 583
getFilter, 607
getFirst, 637, 652, 683
getFirstDayOfWeek, 143
getFont, 350, 408
getFontMetrics, 347, 351
getFontName, 349
getFontRenderContext, 346, 351
getForeground, 343
getFormatter, 607
getGenericComponentType, 664
getGenericInterfaces, 663
getGenericParameterTypes, 663
getGenericReturnType, 663
getGenericSuperclass, 663
getHandlers, 606
getHead, 609
getHeight, 347
getHonorsVisibility, 467
getIcon, 409, 505
getIconImage, 326
getImage, 327, 547
getInheritsPopupMenu, 438
getInputMap, 376, 379
getInputStream, 526, 531
getInstalledLookAndFeel, 368
getKey, 701
getKeyStroke, 375
getLast, 683
getLeading, 350
getLength, 262, 264
getLevel, 606, 607
getLineMetrics, 347, 350
getLineNumber, 584
getLocalGraphicsEnvironment, 344
getLogger, 605
getLoggerName, 608
getLowerBounds, 664
getMaxX, 339
getMessage, 571, 608
getMethodName, 584
getMethods, 255
getMethods, 252
getMillis, 608
getMinX, 339
getModifiers, 252, 256
getModifiersEx, 381, 386
getModifiersExText, 386
getMonths, 148
getName, 150, 249, 349, 505, 664
getNewState, 372
getOldState, 372
getOutputStream, 526, 532
getOwnerType, 664
getPaint, 343
getParameter, 541, 545
getParameterInfo, 546
getParameters, 608
getParameterTypes, 256
getParent, 606
getPassword, 410
getPoint, 386
getPredefinedCursor, 381
getPreferredSize, 330
getProperties, 550, 554
getProperty, 553, 728
getProxyClass, 311
getRawType, 664
getResource, 516

metoda
 getResourceBundle, 608
 getResourceBundleName, 608
 getReturnType, 256
 getRootPane, 494
 getSalary, 214, 265
 getScreenSize, 323, 327
 getSelectedFile, 504
 getSelectedItem, 423–426
 getSelectedObjects, 417
 getSelection, 417, 418
 getSequenceNumber, 609
 getServiceNames, 531
 getShortMonths, 148
 getShortWeekdays, 144, 148
 getSource, 388, 424
 getSourceClassName, 608
 getSourceMethodName, 608
 getStackTrace, 581, 583
 getState, 752
 getStringBounds, 346
 getSuperclass, 659
 getSuppressed, 581
 getTail, 609
 getText, 406, 409
 getThreadID, 609
 getThrown, 608
 getTitle, 322, 326
 getTotalBalance, 764
 getType, 252
 getTypeDescription, 505
 getTypeParameters, 663
 getUpperBounds, 664
 getUseParentHandlers, 607
 getValue, 373, 379, 701
 getWeekdays, 148
 getWidth, 334, 335, 339, 346
 getX, 339, 386
 getY, 386
 hashCode, 225, 684, 706
 hasMoreElements, 670, 727
 hasNext, 91, 669, 674, 677
 hasNextDouble, 91
 hasNextInt, 91
 headMap, 712, 716
 headSet, 712, 716
 higher, 694
 IconImage, 522
 in.close, 580
 incrementAndGet, 777
 indexOf, 84
 indexOfSubList, 724
 info, 605
 init, 536
 initCause, 583
 initialize, 782
 initialValue, 781
 insert, 88, 434
 insertItemAt, 426
 InsertItemAt, 424
 insertSeparator, 434
 interrupt, 746, 749
 interrupted, 748, 749
 intValue, 243
 invoke, 265–267, 311
 invokeAll, 808, 810
 invokeAndWait, 820
 invokeAny, 808
 invokeLater, 817, 820
 isAbstract, 256
 isAccessible, 261
 isDispatchThread, 820
 isDone, 777
 isEditable, 406, 425
 isEmpty, 672, 673
 isEnabled, 373, 379
 isFinal, 252, 256
 isInterface, 256
 isInterrupted, 746–749
 isJavaIdentifierPart, 67
 isJavaIdentifierStart, 67
 isLocationByPlatform, 323, 326
 isLoggable, 600, 609
 isNative, 256
 isNativeMethod, 584
 isPopupTrigger, 438
 isPrivate, 252, 256
 isProtected, 256
 isProxyClass, 311, 312
 isPublic, 252, 256
 isResizable, 326
 isSelected, 415, 436
 isStatic, 256
 isStrict, 257
 isSynchronized, 257
 isTraversable, 498, 505
 isUndecorated, 326
 isVisible, 325
 isVolatile, 257
 isWebBrowserSupported, 531
 itemComparator, 720
 iterator, 668, 672
 join, 752, 810
 JTextField, 406
 keyPress, 621
 keyRelease, 621
 keySet, 700
 KeyStroke, 379

lastIndexOf, 84
 lastIndexOfSubList, 724
 lastKey, 701
 layoutContainer, 472
 length, 84, 730
 linkSize, 466
 listFiles, 497
 listIterator, 677, 682
 load, 554, 728
 lock, 762, 764, 782
 lockInterruptibly, 783
 log, 593, 606
 logp, 606
 logrb, 606
 lookup, 531
 lower, 694
 main, 59, 148, 162, 249
 makeButton, 362
 makePair, 644
 Math.random, 122
 Math.round, 75
 menuCanceled, 441
 menuDeselected, 441
 menuSelected, 441
 minimumLayoutSize, 472
 mod, 115
 modifiers, 256
 mouseClicked, 380, 381
 mouseDragged, 382
 mouseEntered, 383
 mouseExited, 383
 mouseMove, 621
 mouseMoved, 381, 382
 mousePress, 621
 mousePressed, 380, 381
 mouseRelease, 621
 mouseReleased, 380
 move, 736
 multiply, 115, 116
 mutatora set, 142
 newCondition, 765, 769
 newFixedThreadPool, 803
 newInstance, 249–251, 262, 658
 newProxyInstance, 307, 311
 newScheduledThreadPool, 807
 newSingleThreadScheduledExecutor, 807
 next, 669, 679
 nextDouble, 89, 91, 610
 nextElement, 670, 727
 nextInt, 89, 91
 nextLine, 89, 91
 node, 560
 notify, 773
 notifyAll, 773

Object.clone, 282
 Objects.hashCode, 227
 offer, 787, 793
 offerFirst, 793
 offerLast, 793
 offsetByCodePoints, 83
 openFileDialog, 526, 532
 openMultiFileDialog, 532
 or, 730
 ordinal, 247
 pack, 330, 332
 paintComponent, 328, 347, 475, 569
 parse, 244
 parseInt, 242, 243
 peek, 729, 787
 play, 546
 poll, 787, 793, 809
 pollFirst, 694, 793
 pollLast, 694, 793
 pop, 729
 preferredLayoutSize, 472
 previous, 677–680
 previousIndex, 680
 print, 96
 printBuddies, 651
 printf, 92, 244
 println, 60, 96, 229, 310
 printStack, 251
 printStackTrace, 251, 581, 611
 process, 826
 publish, 599, 607, 825
 push, 729
 put, 556, 787, 792
 putFirst, 793
 putIfAbsent, 794
 putLast, 793
 putValue, 373, 379
 raiseSalary, 284
 readConfiguration, 595
 readLine, 91
 readLock, 784
 readPassword, 91
 release, 812
 remove, 434, 669–673, 678, 682
 removeAll, 672, 673, 681
 removeAllItems, 426
 removeEldestEntry, 705
 removeFirst, 683
 removeHandler, 607
 removeItem, 424, 426
 removeItemAt, 424, 426
 removeLast, 683
 removeLayoutComponent, 472
 removePropertyChangeListener, 373

metoda
repaint, 329, 332, 827
replace, 84, 796
replaceAll, 724
res.close, 580
resetChoosableFileFilters, 504
resize, 537
resume, 752
retainAll, 672, 717
revalidate, 407, 408
reverse, 724
reverseOrder, 722
rotate, 724
run, 647, 754
Runtime.addShutdownHook, 179
saveAsFileDialog, 532
saveFileDialog, 526
schedule, 807
scheduleAtFixedRate, 807
scheduleWithFixedDelay, 808
ServiceManager, 526
set, 142, 236, 261, 679, 682
setAccelerator, 440
setAccessible, 257, 261
setAccessory, 504
setAction, 434
setActionCommand, 363, 417, 419
setAutoCreateContainerGaps, 467
setAutoCreateGaps, 467
setBackground, 341, 343
setBorder, 419, 423
setBounds, 321, 322, 325, 468
setCharAt, 88
setColor, 343, 510, 625
setColumns, 408, 410, 413
setComponentPopupMenu, 438
setCursor, 386
setDaemon, 754
setDebugGraphicsOptions, 615
setDefaultButton, 494
setDefaultCloseOperation, 320, 536
setDefaultUncaughtExceptionHandler, 611
setDone, 777
setEditable, 406, 423, 425
setEnabled, 373, 379, 441
setExtendedState, 325, 326
setFileFilter, 497, 504
setFileSelectionMode, 503
setFileView, 499, 504
setFilter, 600, 607
setFirst, 652
setFont, 350, 408
setForeground, 341, 343
setFormatter, 601, 607
setFrameFromCenter, 337
setFrameFromDiagonal, 336
setHonorsVisibility, 467
setHorizontalGroup, 466
setHorizontalTextPosition, 435
setIcon, 409, 435
setIconImage, 321, 326
setInheritsPopupMenu, 438
setJMenuBar, 432, 434
setLabelTable, 428, 431, 640
setLayout, 399
setLevel, 606, 607
setLineWrap, 411, 413
setLocation, 321, 325
setLocationByPlatform, 322, 326
setLookAndFeel, 368
setMajorTickSpacing, 431
setMinorTickSpacing, 431
setMnemonic, 440
setModel, 424
setMultiSelectionEnabled, 503
setPaint, 340, 341, 343
setPaintLabels, 428, 431
setPaintTicks, 428, 431
setPaintTrack, 432
setParent, 606
setPriority, 753
setRect, 335
setResizable, 321, 326
setRows, 410, 413
setSecond, 638
setSelected, 414, 436
setSelectedFiles, 503
setSize, 325
setSnapToTicks, 432
setSource, 363
setText, 406, 407, 409
setTitle, 321, 326, 537
setToolTip, 446
setToolTipText, 447
setUncaughtExceptionHandler, 754
setUndecorated, 326
setUseParentHandlers, 607
setValue, 701
setVerticalGroup, 466
setVisible, 320, 325, 489, 537, 827
setWrapStyleWord, 413
severe, 605
show, 320, 437
showConfirmDialog, 474, 476, 482
showDialog, 490
showDocument, 531, 548, 549
showInputDialog, 475, 476, 483
showInternalConfirmDialog, 482

showInternalInputDialog, 484
 showInternalMessageDialog, 482
 showMessageDialog, 288, 474, 481, 530
 showOptionDialog, 474, 476
 showSaveDialog, 495
 showStatus, 548, 549
 shuffle, 721, 722
 shutdown, 803
 shutdownNow, 804
 signal, 769, 780
 signalAll, 766–769
 size, 236, 672, 673
 sleep, 737, 742
 sort, 121, 274, 720
 start, 288
 startsWith, 84
 stateChanged, 426
 stop, 288, 752, 784
 store, 550, 554, 728
 subList, 716
 subMap, 712, 716
 submit, 803, 809
 subSet, 712, 716
 substring, 78, 84, 711
 subtract, 115, 116
 super.clone, 282
 super.paintComponent, 330
 suspend, 752, 785
 swap, 168, 724
 swapHelper, 656
 SwingUtilities.updateComponentTreeUI, 366
 synchronizedCollection, 713, 797
 synchronizedList, 797
 synchronizedMap, 713, 797
 synchronizedSet, 797
 synchronizedSortedMap, 797
 synchronizedSortedSet, 797
 System.exit, 60, 320
 System.out.println, 89
 System.runFinalizersOnExit, 179
 systemNodeForPackage, 560
 systemRoot, 560
 tailMap, 712, 716
 tailSet, 712, 716
 take, 793
 takeFirst, 793
 takeLast, 793
 text, 91
 Thread.getAllStackTraces, 581
 ThreadLocalRandom.current, 781
 throwing, 594, 606
 toArray, 237, 645, 672, 718
 toBack, 326
 toFront, 322, 326
 toString, 88, 118, 228, 259, 310, 424, 584, 610
 toUpperCase, 84
 transfer, 756, 761, 770, 793
 trim, 85, 407
 trimToSize, 235, 236
 tryLock, 782, 783
 tryTransfer, 793
 UIManager.setLookAndFeel, 366
 uncaughtException, 754, 755
 unlock, 762, 764
 unmodifiableCollection, 713
 unmodifiableList, 713
 unmodifiableSet, 713
 update, 240
 userNodeForPackage, 560
 userRoot, 560
 validate, 407, 408
 valueOf, 114, 243, 246
 wait, 773
 warning, 605
 windowActivated, 369, 372
 windowClosed, 369, 372
 windowClosing, 369, 370, 372
 windowDeactivated, 369, 372
 windowDeiconified, 369, 372
 windowIconified, 369, 372
 windowOpened, 369, 372
 windowStateChanged, 372
 writeLock, 784
 xor, 730
 metody
 abstrakcyjne, 215
 akcesora, 155
 fabryczne, 161, 704, 803
 finalne, 211, 770
 graficzne, 332
 interfejsu
 Action, 373
 BlockingDeque<E>, 793
 BlockingQueue<E>, 792
 Collection, 672
 Collection<E>, 672
 Deque<E>, 695
 Future<V>, 801
 GenericArrayType, 664
 Iterator<E>, 674
 LinkedList<E>, 683
 List<E>, 682
 ListIterator<E>, 683
 Map<K, V>, 700
 NavigableSet<E>, 694
 ParameterizedType, 664
 Queue<E>, 695
 SortedSet, 712

metody
interfejsu
 TypeVariable, 664
 WildcardType, 664
 WindowListener, 369, 372

klas wewnętrznych, 289

klasy
 AccessibleObject, 261
 Applet, 537, 545, 546
 Array, 264
 Arrays, 123
 BigDecimal, 116
 BigInteger, 115
 BitSet, 730
 BorderFactory, 421
 Class, 252, 255
 Class<T>, 658
 Collections, 712, 715, 722, 724
 Component, 325
 Console, 91
 Constructor, 256
 Date, 141
 Employee, 151
 Executors, 803
 FileView, 498, 505
 Font, 349
 Frame, 326
 Graphics, 350
 Graphics2D, 351
 GregorianCalendar, 147
 GroupLayout, 466
 Integer, 243
 JComboBox, 425
 JFileChooser, 503
 JFrame, 321
 JMenu, 433
 JOptionPane, 481
 JSlider, 431
 JTextArea, 412
 JTextComponent, 406
 LayoutManager, 472
 LineMetrics, 350
 Logger, 605
 Modifier, 256
 Object, 773
 Preferences, 560
 Properties, 553
 RectangularShape, 335, 339
 Robot, 621
 Scanner, 90
 String, 83, 87
 StringBuilder, 88
 Thread, 749–755

ThreadLocal<T>, 782
Throwable, 583
Timer, 288
Toolkit, 327
Window, 326

kollekji blokujących, 787, 788
komentarze, 191
monitorowe, 775
o zmiennej liczbie parametrów, 244
odradzane, 141
ogólne, 632
pomostowe, 638, 649
prywatne, 157
przeciążanie, 171
przesłaniające, 201
publiczne, 59
rejestrujące, 594
rodzime, 160
statyczne, 73, 160, 645
statyczne ze zmiennymi typowymi, 645
sygnatura, 171, 209
synchronizowane, 771
tworzące niemodyfikowalne widoki, 712
udostępniające, 141
uogólnione, 671
wstawiane, 154
z parametrami typowymi, 632
 zmieniające wartość elementu, 141

mieszanie współrzędnych, 691

mnemoniki, 438

modalność, 485

model, 394, 395
 pole tekstowego, 394
 wskaźnikowy, 24

moduł ładujący klasy, 588

modyfikacja
 parametru obiektowego, 167
 zbioru EnumSet, 704

modyfikator
 dostępu, 58, 220
 dostępu private, 186
 dostępu public, 185
 final, 158, 211, 777
 volatile, 777
 zdarzenia, 386

modyfikowanie elementów zbioru, 687

monitor, 775

motyw Ocean, 316

mutatory, 142

MVC, Model-View-Controller, 392

mysz, 380

N

nadklasa, superclass, 200
 nadtypy, 652, 654
 NaN, 64, 70
 narzędzi
 graficzne, 317
 wiersza poleceń, 44
 narzędzi
 appletviewer, 53, 535
 ButtonTest, 618
 ImageViewer, 51, 500
 jar, 512, 514
 Jar Bundler, 515
 javac, 45
 javadoc, 190–195
 javap, 294
 jconsole, 595, 613, 779
 jmap, 614
 Matisse, 460, 461
 OptionDialogTest, 477
 ReflectionTest, 295, 296
 Swing graphics debugger, 614
 natywna biblioteka interfejsowa, 33
 nawiasy
 klamrowe, 59, 300
 kwadratowe, 116
 ostre, 632
 puste, 234
 nazwa
 akejii, 377
 klasy, 45, 58, 134, 197
 konstruktora, 137
 parametru, 174
 pliku, 45
 pliku dziennika, 598
 rejestratora, 595
 zasobu, 516
 zmiennej, 67
 zmiennej typowej, 630
 nazwy
 klas komponentów Swing, 318
 klas proxy, 311
 logiczne czcionek, 344
 metod, 197
 rodziny czcionek, 343
 NetBeans, 38, 44, 459
 niezależność od architektury, 26
 niezawodność, 24
 niezmienialność łańcuchów, 79
 niszczenie obiektów, 179
 notacja wielbłędzia, 58

O

obiekt, 24, 132
 Action, 446
 ActionEvent, 357
 AssertionException, 587
 BasicButtonUI, 398
 builder, 86
 ButtonGroup, 415
 Callable, 803
 ColorAction, 360
 Comparator, 693
 Console, 90
 Date, 138
 DefaultButtonModel, 398
 ExecutorCompletionService, 808
 FutureTask, 798
 Graphics, 328, 341
 GridLayout, 406
 Handler, 595, 598, 608
 Iterator, 672
 JButton, 397
 JPanel, 402
 JRadioButton, 415
 PaintEvent, 388
 Path, 97
 PrintWriter, 96, 97
 Properties, 553
 Runnable, 759, 803
 Scanner, 96
 System.out, 60
 WeakReference, 702
 obiektów, 133
 autoboxing, 241
 hermetyzacja, 133
 klonowanie, 280
 kopiowanie, 280
 metody, 133
 metody prywatne, 157
 niszczenie, 179
 polimorfizm, 204
 porównywanie, 221
 składowe, 133, 155
 stan, 133, 134
 tożsamość, 134
 właściwości, 133
 zachowanie, 133
 obiekty
 blokady, 762
 funkcyjne, 690
 klasy Class, 249
 klasy Lock, 762
 klasy ogólnej, 646
 nasłuchujące zdarzeń, listener objects, 287, 371

obiekty
obsługujące wywołanie, 307
opakowujące kolekcje, 711
proxy, 308
typu wyliczeniowego, 548
warunków, 765
zdarzeń, event objects, 356
obliczanie kodu mieszącego, 226
obramowanie, 419
obrazy, 351
obsługa
appletów, 533
błędów, 564
kliknięcia przycisku, 357
nieprzechwyconych wyjątków, 754
przycisku OK, 486
ramek, 325
wyjątków, 249, 564, 646
zdarzeń, 329, 355
zdarzeń AWT, 389, 390
zdarzeń myszy, 380, 383
obszar
surogatów, surrogates area, 66
tekstowy, text area, 406, 410
ochrona bloku kodu, 762
odczyt plików, 96
odmierzanie czasu, 782
odpakowywanie, 242
odradzane metody, 141
odwołanie, 193
odwzorowanie nazw czcionek, 345
ograniczenia
blokad wewnętrznych, 771
nadtypów, 652, 653
typów generycznych, 240
widoczności metody, 219
widoków, 714
zmiennych typowych, 633
okna dialogowe
modalne, 474
niemodalne, 474
okno, 369, 372
dialogowe
opcji, 474, 483
potwierdzenia, 482
przyjmowania danych, 484
typu O programie, 485
wyboru kolorów, 505
wyboru plików, 495
z komunikatem, 482
z polem hasła, 489
dokumentacji API, 85
konsoli, 44
przeglądarki appletów, 53
wyboru plików, 498, 558

OOP, Object Oriented Programming, 132
opakowywanie, autoboxing, 241
opakowywanie wyjątków, 648
opcja
Step Into, 623
Step Over, 623
Terminate, 625
opcje
debugera, 623
narzędzia jar, 513
operacje
niepodzielne, 760
opcjonalne, 714
zbiorcze, 717
operator
::, 202
[], 120
==, 705
dekrementacji, 71
inkrementacji, 71
instanceof, 213, 222, 278
new, 86, 137, 153, 276
przecinka, 77
operatorzy
arytmetyczne, 69
bitowe, 72
logiczne, 71
relacyjne, 71
opis
danych, 33
elementu, 692
klasy, 86, 192
metod, 87
struktury stron, 33
zmiennej, 192
optymalizacja, 27
osłona obiektów, 241
ostrzeżenie, 643
otwieranie menu, 438
overloading resolution, 171
oznaczenia relacji, 136

P

pakiet
com.horstmann.corejava, 183
com.mycompany.mylib, 588
com.mycompany.util, 518
com.sun.java, 366
domyślny, 183
java.awt, 186
java.awt.event, 286, 388
java.lang, 83, 90
java.lang.reflect, 252, 660
java.math, 114

- java.sql, 181
- java.util, 90, 181, 387
- java.util.concurrent, 762, 771, 797
- java.util.concurrent.atomic, 777
- java.util.concurrent.locks, 783
- javax.swing, 286, 319
- javax.swing.event, 390
- JDK, 38
- org.omg.CORBA, 243
- Swing, 314
- pakiet, packages, 180
 - dodawanie klasy, 182
 - komentarze, 194
 - lokalizacyjne, 596
 - zasięg, 185
- panel
 - JPanel, 398
 - przewijany, scroll pane, 411, 413
 - z przyciskami, 359, 398
- para klucz – wartość, 556, 561, 697
- parametr, 61
 - anchor, 452
 - fill, 452
 - gridheight, 451, 452
 - gridwidth, 451, 452
 - gridx, 451, 452
 - gridy, 451, 452
- parametry
 - jawne, 153
 - konfiguracyjne obiektu Handler, 598
 - łańcuchowe, 96
 - metod, 164
 - niejawne, 153, 174
 - obiektowe, 167
 - określające cel, 549
 - typowe, 628, 641
 - wiersza poleceń, 120
- pasek
 - menu, 432
 - narzędzi, toolbar, 444
- petla
 - do-while, 102
 - for, 98, 106–108
 - for each, 32, 98, 117, 126
 - w stylu for each, 669
 - while, 101, 102
- piaskownica, sandbox, 519, 522
- pieczętowanie pakietów, 186, 518
- plik
 - AboutDialog.java, 487
 - ActionFrame.java, 377
 - AnonymousInnerClassTest.java, 301
 - ArrayListTest.java, 238
 - Ball.java, 739
 - BallComponent.java, 740
 - Bank.class, 761
 - Bank.java, 758, 767, 772
 - BigIntegerTest.java, 115
 - BlockingQueueTest.java, 789
 - BorderFrame.java, 420
 - Bounce.java, 737
 - BounceThread.java, 743
 - BuggyButtonTest.java, 622
 - ButtonFrame.java, 360
 - ButtonPanel.java, 481
 - Calculator.jnlp, 519
 - CalculatorFrame.java, 528
 - CalculatorPanel.java, 403
 - CalendarTest.java, 144
 - Chart.java, 543
 - CheckBoxTest.java, 414
 - CircleLayout.java, 469
 - CircleLayoutFrame.java, 472
 - CloneTest.java, 284
 - ColorChooserPanel.java, 508
 - ComboBoxFrame.java, 424
 - CompoundInterest.java, 125
 - ConstructorTest.java, 177
 - CopyOfTest.java, 263
 - DataExchangeFrame.java, 491
 - deskryptora, 519
 - DialogFrame.java, 486
 - doc-files, 191
 - DrawTest.java, 337
 - Employee.java, 151, 184, 205, 231, 275, 284
 - EmployeeSortTest.java, 275
 - EmployeeTest.java, 149, 151
 - en.properties, 596
 - EnumTest.java, 246
 - EqualsTest.java, 230
 - EventTracer.java, 615
 - FileIconView.java, 503
 - fontconfig.properties, 345
 - FontFrame.java, 454, 463
 - FontTest.java, 347
 - forkJoinTest.java, 810
 - FutureTest.java, 799
 - GBC.java, 456
 - GenericReflectionTest.java, 661
 - ImagePreviewer.java, 502
 - ImageTest.java, 352
 - ImageViewer.java, 51
 - ImageViewerFrame.java, 500
 - InnerClassTest.java, 292
 - InputTest.java, 89
 - Item.java, 692
 - javan.log, 597
 - javaws.jar, 526

plik
jogging.properties, 594
LinkedListTest.java, 681
LoggingImageViewer.java, 602
LotteryArray.java, 128
LotteryDrawing.java, 121
LotteryOdds.java, 108
Manager.java, 206, 232
ManagerTest.java, 204
MANIFEST.MF, 512
manifestu
 klasa główna, 514
 sekcja główna, 512
 wstawianie sekcji, 518
 zmienianie zawartości, 513
MapTest.java, 699
MenuFrame.java, 442
MethodTableTest.java, 266
MouseComponent.java, 383
MouseFrame.java, 383
NotHelloWorld.java, 330, 534
ObjectAnalyzerTest.java, 259
OptionDialogFrame.java, 477
overview.html, 194
PackageTest.java, 184
PairTest1.java, 631
PairTest2.java, 634
PairTest3.java, 656
ParamTest.java, 169
PasswordChooser.java, 492
Person.java, 217
PersonTest.java, 217
PlafFrame.java, 367
PreferencesTest.java, 557
PriorityQueueTest.java, 696
program.properties, 551
PropertiesTest.java, 551
ProxyTest.java, 309
RadioButtonFrame.java, 417
ReflectionTest.java, 253
ResourceTest.java, 517
Retirement.java, 103
Retirement2.java, 105
RobotTest.java, 618
rt.jar, 187
SetTest.java, 686
ShuffleTest.java, 721
Sieve.cpp, 731
Sieve.java, 731
SimpleFrameTest.java, 318
SizedFrameTest.java, 324
SliderFrame.java, 428
src.zip, 41, 42
StackTraceTest.java, 582
StaticInnerClassTest.java, 305
StaticTest.java, 162
Student.java, 218
swing.properties, 366
SwingThreadTest.java, 817
SwingWorkerTest.java, 821
System.java, 43
TalkingClock\$TimePrinter.class, 294
TextComponentFrame.java, 411
ThreadPooITest.java, 804
TimerTest.java, 287
ToolBarTest.java, 446
TransferRunnable.java, 759
TreeSetTest.java, 691
UnsynchBankTest.java, 757
Welcome.java, 45
WelcomeApplet.class, 53
WelcomeApplet.html, 53
WelcomeApplet.java, 55
pliki
 .class, 514
 .exe, 514
 .java, 58
 .jnlp, 519
 .audio, 546
 .cookie, 527
 .graficzne, 546
 .JAR, 187, 512, 514
 obrazów, 515
 podpisane cyfrowo, 523
 tekstowe, 515
 .XML, 600
 z danymi binarnymi, 515
 zasobów, 516
 źródłowe, 151, 189
pobieranie
 hasła, 90
 właściwości systemowych, 554
podklasa, subclass, 200
 Properties, 726
 Stack, 726
 podklasa, subclass, 200
podkradanie pracy, 812
podłańcuchy, 78
podmenu, 432
podpakiety, 180
podpis cyfrowy, 26, 523
podpisywanie kodu, 523
podtyp typu granicznego, 634
podziałka, 427
pola
 finalne, 211
 haseł, 406, 410
 klasowe, 159

- niestatyczne, 159
- prywatne, 155
- publiczne, 155
- statyczne, 159, 176
- tekstowe, 394, 406
- ulotne, 776
- weight, 451
- wyboru, 413
- pole value, 243
- polecenie
 - cmd, 41
 - dir, 46
 - jcontrol, 536
- polimorfizm, 204, 207, 269
- połączenie na poziomie gniazd, 24
- położenie przycisków, 402
- porównywanie
 - elementów tablic, 225
 - łańcuchów, 79, 81
 - obiektów, 221, 689
 - obiektów osłonowych, 242
 - w pętli, 107
 - w podklasach, 277
- powiadamianie o zdarzeniach, 358
- powiązana tablica mieszająca, 703
- poziom
 - FINE, 595, 597, 601
 - rejestracji obiektu Handler, 597
- poziomy
 - rejestracji, 595
 - ważności komunikatów, 592
- pozycjonowanie
 - bez względne, 468
 - ramki, 321
- preferencje
 - użytkownika, 549, 555
 - węzła, 561
- priorytet
 - informacji, 592
 - wątków, 750
 - wątku, 752, 753
 - operatorów, 76
- procedura
 - obsługi błędów, 565
 - obsługi zdarzeń, 355
- proces, 736
- proces inliningu, 212
- program, Patrz narzędzie
- programowanie
 - interfejsów graficznych, 391
 - interfejsu użytkownika, 317
 - obiektowe, 24, 132
 - ogólne, generic programming, 627
 - pasków narzędzi, 445
- proceduralne, 133
- sieciowe, 33
- wielowątkowe, 28
- programy
 - jednowątkowe, 736
 - refleksyjne, 247
 - wielowątkowe, 735
- projektant formy, 398
- projektowanie klas, 195
- prostokąt, 334
- protokолy sieciowe, 24
- prywatne pola, 155
- przechwytywanie
 - strumienia błędów, 611
 - wielu typów wyjątków, 574
 - wyjątków, 250, 571, 747
- przeciąganie paska narzędzi, 445
- przeciążanie
 - konstruktorów, 172
 - metod, 171
- przedział, 712
- przeglądarka
 - appletów, 535
 - HotJava, 31
 - pamięci podręcznej, 521
- przejmowanie blokady, 774
- przekazywanie
 - obiektu, 138
 - przez wartość, 167
 - wyjątków, 586
- przełącznik, radio button, 413
- przełączniki, 415
- przełączniki w elementach menu, 436
- przenośność, 26, 70
- przepływ sterowania, 98, 111, 298
- przerywanie
 - działania pętli, 111
 - procesu ładowania, 737
 - wątków, 746
- przesłanianie metod, 202, 225, 647
- przestrzenie numeracyjne, 66
- przesuwanie iteratora, 671
- przeszukiwanie liniowe, 723
- przetwarzanie XML, 33
- przycisk, 358, 397
 - domyślny, 490
 - JButton, 398
 - OK, 486
 - położenie, 402
 - rozmiar, 402
 - w rozkładzie brzegowym, 401
- przyciski radiowe, 415
- prywileje klasowe, 157

publiczne metody

akcesora, 155
mutatora, 155

publiczne pola, 155

pule wątków, 802–804

punkt wstrzymania, 623, 624

pusta mapa własności, 553

R

ramka, frame, 318

nadrzędna, 485, 490

wyświetlająca tekst, 327

ramki

pozycjonowanie, 321

rozmiar, 323

warstwy, 327

własności, 322

wyświetlanie, 320

reaktywacja wątków, 766

referencja

do elementów typu Object, 628

do klasy zewnętrznej, 291, 293

do obiektu, 138

do parametru niejawnego, 203

null, 250, 565

refleksja, reflection, 199, 247, 270, 658

analiza

funkcjonalności klasy, 252

obiektów w czasie działania programu, 257

generyczny kod tablicowy, 261

rejestr, 556

rejestr zdarzeń, 615

rejestratory, logger, 591, 597

rejestrujący obiekt pośredni, 610

rekordy dziennika, 597

relacje

agregacja, 135

dziedziczenie, 136

zależność, 135

reorganizacja tablicy mieszącej, 686

repozytorium Preferences, 555

robot, 618

rodzaje

ataków, 25

modalności, 485

obramowań, 419

suwaków, 429

rozkład

brzegowy, 400, 402, 448

ciągły, 448

GridLayout, 448–453

grupowy, 459, 461

komponentów, 407

siatkowy, 402, 448

sprzęzowy, 449

SpringLayout, 449

rozmiar

apletu, 537

ekranu, 323

ikon, 522

interpretatora, 23

pola tekstowego, 407

przycisków, 402

ramki, 323

tablicy, 117

rozmiieszczanie komponentów, 398

rozstrzyganie przeciążania, 171, 209

rozszerzanie

klasy, 133, 216

klasy Throwable, 646

programów, 211

stylu, 317

rysowanie, 314

figur, 333, 337

na komponencie, 328

obrazu, 354

wykresu, 543

rzutowanie, casting, 75, 212, 637, 644, 718

S

sandbox, 519, 522

scalanie list, 681

SDK, Software Development Kit, 38

SE, Standard Edition, 38

semafora, 812

separatator, 445

serializacja, 539

serwer pochodzenia, 523

serwer Tomcat, 519, 520

siatka, 450

sito Eratostenesa, 730

skaner, 89

skład tekstów, 346

składanie łańcuchów, 86

składnia

diamentowa, diamond syntax, 234

Javy, 23

klas wewnętrznych, 289

wewnętrznych klas anonimowych, 371

składowe, 133, 155

składowe chronione, 220

skróty klawiszowe, 439

słabe referencje, 702

słowo kluczowe, 829

abstract, 215

assert, 587

catch, 250

class, 58

- extends, 200, 634
- final, 68, 158, 211, 299
- implements, 273, 634
- import, 180
- instanceof, 213
- interface, 272
- package, 183, 186
- private, 152, 158, 220
- protected, 190, 219
- public, 58, 152, 220
- static, 69, 160, 304
- strictfp, 70
- super, 202
- synchronized, 762, 769, 775
- this, 154, 160, 174
- throws, 569
- try, 250
- void, 60
- volatile, 776
- słuchacz
 - akcji, 414, 416
 - przycisku, 359
 - z źródłami zdarzeń, 373
 - zdarzeń, event listener, 356, 364, 388
- sortowanie, 275, 299, 720
 - kluczy, 701
 - listy elementów, 720
 - tablicy, 121
- specyfikacja, 59
- specyfikacja wyjątku, 568
- specyfikator
 - dostępu, 150
 - formatu, 92, 96
 - throws, 283, 569, 573
- sprawdzanie
 - parametrów, 589
 - pól obiektu, 265
 - typów, 641
 - typów pól, 257
 - zakresu, 120
- sprzężenie zwrotne, 286
- stała, 68
 - ACCELERATOR_KEY, 374
 - ACTION_COMMAND_KEY, 374
 - BorderLayout.SOUTH, 401
 - DEFAULT, 374
 - Double.NaN, 64
 - Double.NEGATIVE_INFINITY, 64
 - Double.POSITIVE_INFINITY, 64
 - LONG_DESCRIPTION, 374
 - MNEMONIC_KEY, 374
 - NAME, 374
 - SHORT_DESCRIPTION, 374
 - SMALL_ICON, 374
- stałe
 - interfejsu Action, 374
 - interfejsu SwingConstants, 409
 - klasowe, 69
 - klasy BorderLayout, 401
 - łańcuchowe, 80
 - matematyczne, 73
 - statyczne, 159
- stan
 - obiektu, 133, 134
 - okna, 372
 - wątków, 749, 751
- standard
 - ECMA-262, 35
 - IEEE 754, 64
 - ISO/ANSI, 80
 - wyrażenia czasu, 139
- status przerwania wątku, 746
- statyczne
 - funkcje składowe, 60
 - klasy wewnętrzne, 303
 - sterta, heap, 120, 140, 696
- STL, Standard Template Library, 666
- stopień powiązań między klasami, 135
- stopy oprocentowania, 126
- stos, stack, 120, 729
- stos wywołań, 251, 582
- stosowanie
 - blokady, 765
 - dziedziczenia, 269
 - refleksji, 270
 - warunków, 769
 - wyjątków, 584
- struktura
 - katalogów, 43
 - ramki JFrame, 328
- struktury danych, 665
- strumień
 - ByteArrayInputStream, 526
 - ByteArrayOutputStream, 526
 - InputStream, 526
 - PrintStream, 526
 - wejściowy, 89
- styl
 - GTK, 316
 - Metal, 315, 366
 - Nimbus, 317
 - Synth, 317
- style
 - obramowań, 419
 - projektowania, 196
- suwak, slider, 413, 426, 430
- suwak z podziałką, 427
- SWT, 317

sygnatura metody, 171, 209
symbole zastępcze, 65
symulacja banku, 756, 759, 768
synchronizacja, 756
synchronizatory, 812, 813
synchronizowane wątki, 763
system kolorów, 342
szablon
 bitset, 729
 vector, 235
szerokość kolumny, 407
szkielet rozgałęzienie-złączenie, 809, 812

Ś

ścieżka
 dostępu, 39
 klas, 187, 189
ścieżki
 bezwzględne, 97
 względne, 97
ścisła kontrola typów, 62, 274
śledzenie
 przepływu wykonywania, 593
 stosu, 251, 581, 755
środowisko działania apletu, 547
środowisko programistyczne
 Eclipse, 44, 47
 NetBeans, 44

T

tabela metod, 210
tablic, 116
 inicjowanie, 118
 kopiowanie, 119
 numerowanie, 116
 przeglądanie, 117
 sortowanie, 121
tablica
 accounts, 760
 args, 120
 arrayToFill, 673
 par klucz – wartość, 556
 referencji, 628
 result, 123
 trójkątna, 129
tablice
 anonimowe, 118
 generyczne, 644
 kopiwane przy zapisie, 796
 mieszające, 226, 428, 684, 703
 postrzepione, 127

typów ogólnych, 642
typów wieloznacznych, 642
wielowymiarowe, 124
tasowanie, 720
tasowanie elementów listy, 721
technologia Flash, 29
tekst, 406, 410
terminal, 35
test wydajności, 730
testowanie
 apletu, 53
 blokad, 782
 mechanizmu własności, 551
Tomcat, 519
tożsamość obiektu, 134
translacja
 metod ogólnych, 637
 poprzez wymazywanie typów, 648
 typów ogólnych, 639
 wyrażeń generycznych, 637
tryb pełnoekranowy, 325
tworzenie
 akceleratora, 439
 apletów, 29, 53
 dziennika, 601
 egzemplarza klasy, 132
 etykiety, 409
 klas wyjątków, 570
 konstruktorów, 152
 listy cyklicznej, 668
 listy powiązanej, 668
 menu, 432
 obiektów, 96, 132, 137, 171
 obiektu proxy, 307
 obiektu typu Class, 249
 obramowań, 421
 ogólnych tablic, 643
 okien dialogowych, 484
 okna komunikatu, 477
 osobnego wątku, 741, 745
 plików JAR, 512
 pół wyboru, 413
 przycisku, 358, 362
 puli wątków, 802
 ramki, 318
 robota, 617
 słuchacza akcji, 363
 tablic, 129, 203
 postrzepionych, 128
 generycznych, 644
 trójkątnych, 129
 z kolekcji, 718
 widoku mapy, 698

typ

- boolean, 66
- byte, 62
- char, 65
- Date, 92
- double, 64
- float, 63
- graniczny, 634
- int, 26, 62
- Integer, 243
- long, 63
- MIME, 519, 520
- osłony, 265
- parametryzowany, 650
- short, 62
- short int, 26
- surowy, 636, 641, 650
- wbudowany, 137
- wieloznaczny, 629, 634
- wieloznaczny z ograniczeniem nadtypów, 654
- wyliczeniowy, 77
- zwrotny, 639

typy

- całkowite, 62
- interfejsowe, 667
- kolekcji, 675
- komunikatów, 475
- konwersji, 74
- kurSORów, 382
- listowe, 651
- ogólne, 627, 658
- parametryzowane, 627
- podstawowe, 221
- sparametryzowane, generic types, 32
- surowe, 234
- tablicowe, 221
- wieloznaczne, 650, 653
 - bez ograniczeń, 655
 - mechanizm chwytania, 656
 - wyliczeniowe, 245, 704
- zmiennoprzecinkowe, 63
- zwrotne kowariantne, 639

U

- ukrywanie danych, 133
- UML, Unified Modeling Language, 136
- Unicode, 65
- uruchamianie
 - apletu, 53, 535
 - aplikacji graficznej, 50
 - aplikacji Java Web Start, 519, 520
 - osobnego wątku, 741

programów w konsoli, 45

- programu, 44, 49
- uruchomienie kilku wątków, 743

usługi

- API, 526
- JNLP, 526

ustawianie

- preferencji, 557
- ścieżki klas, 189

usuwanie

- elementów, 670
- elementów z kolekcji, 673
- elementu z listy powiązanej, 676
- elementu z tablicy, 675
- nieużytków, garbage collecting, 22, 702
- przedziału, 712

UTC, Coordinated Universal Time, 139

UTF-16, 66

utrata wyjątku, 579

V

varargs, 244

W

warstwa

- implementacji, 666
- interfejsów, 666
- ramki, 327

wartości

- graniczne przedziału, 712
- zwrotne okna potwierdzenia, 476

wartość

- NaN, 64, 70
- null, 77, 81, 139, 172
- skrótu, hash value, 227

warunek wstępny, 590

warunki, 765, 769

wątek, thread, 735, 815

- dystrybucji zdarzeń, 319, 741, 785, 819, 828
- roboczy, 824

sterowania, thread of control, 735

TransferRunnable, 785

wyliczeniowy, 791

zablokowany, 747

zamknięty, 747

wątki

kończenie działania, 746

oczekujące, 766

niesynchronizowane, 763

priorytet, 752

stan

wątki
 stan
 BLOCKED, 749
 NEW, 749
 RUNNABLE, 749
 TERMINATED, 749
 TIMED WAITING, 749
 WAITING, 749
 synchronizowane, 763
 usunięte z kolejki, 766
 wywieszczanie, 760
 zamienianie w demona, 753
 zamknięcie, 752
 wczytywanie zasobów, 516
 wejście, 89
 wejście System.in, 96
 wersje Javy, 32
 węzeł, 556, 560
 wiązanie
 dynamiczne, 204, 209–211
 statyczne, 209
 widoczność metod, 211
 widok, view, 394, 710
 kolekcji, 715
 listowy elementów, 716
 mapy, 698
 podprzedziału, 717
 pola tekstowego, 394
 widoki
 kontrolowane, 714
 niemodyfikowalne, 712
 przedziałowe, 711
 synchronizowane, 713
 wielkie liczby, 114
 wielkość liter, 58
 wielowątkowość, 28, 33, 735
 wielozadaniowość, 735
 wiersz poleceń, 44, 120
 wizualne budowanie interfejsów, 51
 własne typy wyjątków, 571
 własności
 czcionki, 450, 454
 interfejsów, 276
 interfejsu ButtonModel, 397
 klas proxy, 311
 metody equals, 222
 monitorów, 775
 ramek, 322, 551
 wątków, 752
 własność, property, 322
 właściwości
 list, 721
 systemowe, 554

włączanie
 asercji, 588
 zegara, 293
 wnioskowanie o typie, 632
 wprowadzanie tekstu, 406
 wskaźniki, 25
 do funkcji, 286
 do metod, 264
 do obiektów, 140
 współczynnik zapelnienia tablicy, 686
 współrzędne
 figur, 333–336
 kodowe znaków, 66, 81
 siatki, 451
 wstawianie komentarzy, 190
 wybór
 kolorów, 505
 plików, 495
 wyciszanie wyjątków, 586
 wydajność, 27
 wydłużenie
 dolne, 346
 górne, 346
 wyjątek
 ArrayListIndexOutOfBoundsException, 117
 ArrayListIndexOutOfBoundsExceptionException, 566, 816
 ArrayStoreException, 642, 650
 BadCastException, 658
 Class.forName, 250
 ClassCastException, 213, 262, 650, 714
 CloneNotSupportedException, 283
 ConcurrentModificationException, 679, 797
 EmptyStackException, 584, 586
 FileNotFoundException, 97, 528, 569, 648
 IllegalAccessException, 257
 IllegalMonitorStateException, 773
 IllegalStateException, 670, 674, 683
 InterruptedException, 737, 742, 748
 IOException, 569, 572
 NoSuchElementException, 674, 695
 NullPointerException, 566, 586, 590
 RuntimeException, 565, 566
 ServletException, 575
 ThreadDeath, 785
 typu RuntimeException, 583
 UnavailableServiceException, 526
 UnsupportedOperationException, 711–715
 wyjątki
 kontrolowane, 249, 567, 646
 konwersji, 629
 niekontrolowane, 250, 567, 647
 typu IOException, 569
 typu SQLException, 576
 zabezpieczeń, 554
 wykonawcze, 565

wyjątków, 563
 deklarowanie, 567
 powtórne generowanie, 575
 przechwytywanie, 250, 571
 przechwytywanie wielu typów, 574
 przekazywanie, 586
 własne typy, 571
 zgłaszanie, 569
 wyjście, 89
 wyjście System.out, 96
 wykres, 543
 wykres słupkowy, 542
 wyliczenia, 727
 wyłączanie
 asercji, 588
 dziedziczenia, 211
 sprawdzania wyjątków, 646
 wymazywanie typów, 637–648
 wymiana danych, 489
 wymuszanie rysowania, 329
 wypełnianie figur, 340
 wyrażenia generyczne, 637
 wyrównywanie etykiet i pól, 462
 WYSIWYG, 395
 wysyłanie
 rekordów, 597
 zdarzeń, 319
 wyszukiwanie binarne, 722
 wyścig, 756, 760
 wyświetlanie
 elementów w przeglądarce, 548
 informacji, 327
 komponentów, 614
 obrazów, 52, 351
 ramki, 320
 rekordów dziennika, 604
 tekstu, 327, 329
 zasobu, 515
 wyświetlanie wątku, 750, 760
 wywołanie
 dowolnych metod, 264
 innego konstruktora, 174
 przez nazwę, 165
 przez referencję, 164
 przez wartość, 164
 setFirst(null), 655
 wzorce
 nazw plików dziennika, 599
 projektowe, 392
 wzorzec
 Composite, 393
 Decorator, 393
 MVC, 392–396
 Strategy, 393

XML, 33

X

zachowanie obiektu, 133
 zagnieździanie
 bloków instrukcji, 98
 pętli, 113
 zakleszczenie, deadlock, 766, 778
 zależność, 135
 zamiana parametrów obiektowych, 168
 zamykanie
 aplikacji, 320
 ramki, 320
 wątków, 752
 zapelnianie tablicy, 237
 zapis
 błędów w pliku, 611
 danych w repozytorium, 556
 do dziennika, 592–594
 plików, 96
 preferencji użytkownika, 549
 zarządcą rozkładu, layout manager, 391, 400, 448
 CircleLayout, 469
 brzegowego, 400
 ciąglego, 398
 FlowLayout, 402
 grupowego, 449
 niestandardowy, 469
 siatkowego, 402
 zasada
 jednego wątku, 816, 827
 uczciwości, 764
 zamienialności, 207
 zasięg
 blokowy, 98
 pakietów, 185
 zmiennych, 98
 zasoby, resources, 515
 zastosowanie
 asercji, 589, 590
 klas abstrakcyjnych, 279
 klas wewnętrznych, 294
 kolejek priorytetowych, 696
 parametrów Class<T>, 659
 refleksji, 252
 typów wyliczeniowych, 246
 zawartość pól danych, 257
 zawijanie wierszy, 411
 zbiór, set, 686, 697
 HashSet, 684
 TreeSet, 688–691
 uporządkowany, 688

Z

- zdarzenia, 355, 389
 interfejs nasłuchu, 356
 myszy, 380, 438
 niskiego poziomu, 388
okna, 370
 semantyczne, 388
słuchacz, 356
źródło, 356
zdarzenie, 287
 FocusEvent, 388
 KeyEvent, 388
 MouseEvent, 388
 MouseWheelEvent, 388
 WindowEvent, 369, 388
zdjęcie blokady, 766
zegar, 286, 293
zezwolenie, permit, 812
zgłaszanie wyjątków, 569
zintegrowane środowisko programistyczne, IDE, 39, 47
zmiana
 koloru, 374
 koloru tła, 507
 rozmiaru tablicy, 117
 stanu okna, 372, 388
 stylu, 366, 368
 typu wyjątków, 647
 własności czcionek, 454
zmienna, 66
zmienna środowiskowa
 CLASSPATH, 46, 189
 Path, 40, 41
zmienne
 atomowe, 777
 finalne, 299, 777
 finalne puste, 299
 interfejsowe, 277
 lokalne, 153
 lokalne wątków, 781
 obiektowe, 137–140, 216
 parametryczne, 174
 polimorficzne, 207
 statyczne, 159
 tablicowe, 116, 119
 typowe, 630, 633, 643, 645
 warunkowe, 765
- znacznik
 @author, 192
 @deprecated, 193
 @link, 194
 @Override, 225
 @param, 192
 @see, 193
 @since, 193
 @version, 192
 append, 599
 applet, 54, 537
 object, 540
 param, 541, 542
- znaczniki
 dokumentacyjne, 190
 polecenia printf, 93
- znak
 \$, 67
 /, 516, 560
 ampersand, 634
 konwersji, 92
 końca pliku, 565
 końca wiersza, 514
 powrotu karetki, 60
 równości, 68
 trzykropka, 244
- znaki
 dodatkowe, 66
 echa, 410
 konwersji Date i Time, 94, 95
 konwersji polecenia printf, 93
 nowego wiersza, 477
 specjalne, 65
- zniszczenie danych, 757
zoptymalizowane wywoływanie metod, 212
zwalnianie blokady, 764
zwracanie referencji, 157

ż

- źródła zdarzeń biblioteki AWT, 389
źródło zdarzeń, event sources, 356, 389

ż

- żądanie zamknięcia wątku, 746, 747

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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 WYDAWNICZA
Helion SA