helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

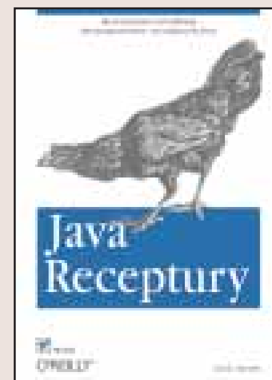
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Java. Receptury



Autor: Ian F. Darwin
Tłumaczenie: Piotr Rajca
ISBN: 83-7197-902-9
Tytuł oryginału: Java Cookbook
Format: B5, stron: 886
Przykłady na ftp: 1157 kB

Książka „Java. Receptury” zawiera kolekcję rozwiązań setek problemów, z którymi programiści używający języka Java spotykają się bardzo często. Receptury znajdują zastosowanie w szerokim spektrum zagadnień: od całkiem prostych, takich jak określanie zmiennej środowiskowej CLASSPATH, aż do całkiem złożonych programów pokazujących jak obsługiwać dokumenty XML lub wzbogacić swą aplikację o mechanizmy obsługi poczty elektronicznej.

Niezależnie od tego, jak planujesz wykorzystać tę książkę — czy jako źródło pomysłów i inspiracji, czy też jako sposób poszerzenia swej wiedzy o języku Java — zawsze będzie ona stanowić ważną część Twojej biblioteki. Mało która książka prezentuje tak wiele możliwości Javy oraz nauczy Cię praktycznego wykorzystania omawianych zagadnień.

W książce zostały omówione następujące zagadnienia:

- Kompilacja, uruchamianie oraz testowanie programów napisanych w Javie
- Interakcja ze środowiskiem
- Łańcuchy znaków oraz dopasowywanie wzorców
- Tablice oraz inne kolekcje danych
- Programowa obsługa portów szeregowych i równoległych
- Pliki, katalogi i system plików
- Tworzenie programów sieciowych pełniących funkcje klientów oraz serwerów
- Aplikacje internetowe, w tym także aplety
- Serwlety oraz dokumenty JSP
- Poczta elektroniczna
- Obsługa baz danych
- Wykorzystanie XML
- Programowanie rozproszone
- Introspekcja
- Tworzenie programów wielojęzycznych
- Wykorzystanie grafiki oraz dźwięku
- Tworzenie graficznego interfejsu użytkownika

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp ... 15

Rozdział 1. *Rozpoczynanie pracy: kompilacja, uruchamianie i testowanie* ... 31

1.0. Wprowadzenie ... 31
1.1. Kompilacja i uruchamianie programów napisanych w Javie — JDK ... 32
1.2. Edycja i kompilacja programów przy użyciu edytorów wyposażonych w kolorowanie syntaktyczne ... 37
1.3. Kompilacja, uruchamianie i testowanie programów przy użyciu IDE ... 39
1.4. Wykorzystanie klas przedstawionych w niniejszej książce ... 44
1.5. Automatyzacja kompilacji przy wykorzystaniu skryptu jr ... 45
1.6. Automatyzacja procesu kompilacji przy użyciu programu make ... 46
1.7. Automatyzacja kompilacji przy użyciu programu Ant ... 48
1.8. Uruchamianie apletów ... 51
1.9. Komunikaty o odrzuconych metodach ... 53
1.10. Testowanie warunkowe bez użycia dyrektywy #ifdef ... 55
1.11. Komunikaty testowe ... 57
1.12. Wykorzystanie programu uruchomieniowego ... 58
1.13. Testowanie jednostkowe — jak uniknąć konieczności stosowania programów uruchomieniowych? ... 60
1.14. Dekompilacja plików klasowych Javy ... 63
1.15. Zapobieganie możliwości dekompilacji plików klasowych Javy ... 65
1.16. Uzyskiwanie czytelnych komunikatów o wyjątkach ... 66
1.17. Poszukiwanie przykładowych kodów źródłowych ... 68
1.18. Program Debug ... 70

Rozdział 2. *Interakcja ze środowiskiem* ... 71

2.0. Wprowadzenie ... 71
2.1. Pobieranie wartości zmiennych środowiskowych ... 71

2.2. Właściwości systemowe ... 73
2.3. Tworzenie kodu zależnego od używanej wersji JDK ... 75
2.4. Tworzenie kodu zależnego od używanego systemu operacyjnego ... 77
2.5. Efektywne wykorzystanie zmiennej CLASSPATH ... 79
2.6. Stosowanie rozszerzających interfejsów programistycznych lub API zapisanych w plikach JAR ... 82
2.7. Analiza argumentów podanych w wierszu wywołania programu ... 83

Rozdział 3. *Łańcuchy znaków i przetwarzanie tekstów* ... 89

3.0. Wprowadzenie ... 89
3.1. Odczytywanie fragmentów łańcucha ... 92
3.2. Dzielenie łańcuchów znaków za pomocą obiektu klasy StringTokenizer ... 93
3.3. Łączenie łańcuchów znaków przy użyciu operatora + i klasy StringBuffer ... 96
3.4. Przetwarzanie łańcucha znaków po jednej literze ... 97
3.5. Wyrównywanie łańcuchów znaków ... 98
3.6. Konwersja pomiędzy znakami Unicode a łańcuchami znaków ... 101
3.7. Odwracanie kolejności słów lub znaków w łańcuchu ... 102
3.8. Rozwijanie i kompresja znaków tabulacji ... 104
3.9. Kontrola wielkości liter ... 108
3.10. Wcinanie zawartości dokumentów tekstowych ... 109
3.11. Wprowadzanie znaków niedrukowalnych ... 111
3.12. Usuwanie odstępów z końca łańcucha ... 112
3.13. Przetwarzanie danych rozdzielonych przecinkami ... 113
3.14. Program — proste narzędzie do formatowania tekstów ... 118
3.15. Program — fonetyczne porównywanie nazwisk ... 120

Rozdział 4. *Dopasowywanie wzorców i wyrażenia regularne* ... 125

4.0. Wprowadzenie ... 125
4.1. Składnia wyrażeń regularnych ... 128
4.2. Jak wyrażenia regularne działają w praktyce? ... 130
4.3. Wykorzystanie wyrażeń regularnych w języku Java ... 132
4.4. Interaktywne testowanie wyrażeń regularnych ... 134
4.5. Określanie tekstu pasującego do wzorca ... 135
4.6. Zastępowanie określonego tekstu ... 136
4.7. Wyświetlanie wszystkich wystąpień wzorca ... 137
4.8. Wyświetlanie wierszy zawierających fragment pasujący do wzorca ... 139
4.9. Kontrola wielkości znaków w metodach match() i subst() ... 141
4.10. Prekompilacja wyrażeń regularnych ... 141

4.11. Odnajdywanie znaków nowego wiersza ... 143
4.12. Program — analizowanie danych ... 144
4.13. Program — pełna wersja programu grep ... 146

Rozdział 5. ***Liczby*** ... ***151***

5.0. Wprowadzenie ... 151
5.1. Sprawdzanie, czy łańcuch znaków stanowi poprawną liczbę ... 154
5.2. Zapisywanie dużych wartości w zmiennych „mniejszych” typów ... 155
5.3. Pobieranie ułamka z liczby całkowitej bez konwertowania go do postaci zmiennoprzecinkowej ... 156
5.4. Wymuszanie zachowania dokładności liczb zmiennoprzecinkowych ... 158
5.5. Porównywanie liczb zmiennoprzecinkowych ... 160
5.6. Zaokrąglanie wartości zmiennoprzecinkowych ... 161
5.7. Formatowanie liczb ... 162
5.8. Konwersje pomiędzy różnymi systemami liczbowymi — dwójkowym, ósemkowym, dziesiętnym i szesnastkowym ... 165
5.9. Operacje na grupie liczb całkowitych ... 166
5.10. Posługiwanie się cyframi rzymskimi ... 167
5.11. Formatowanie z zachowaniem odpowiedniej postaci liczby mnogiej ... 172
5.12. Generowanie liczb losowych ... 173
5.13. Generowanie lepszych liczb losowych ... 174
5.14. Obliczanie funkcji trygonometrycznych ... 176
5.15. Obliczanie logarytmów ... 176
5.16. Mnożenie macierzy ... 177
5.17. Operacje na liczbach zespolonych ... 179
5.18. Obsługa liczb o bardzo dużych wartościach ... 181
5.19. Program TempConverter ... 183
5.20. Program — generowanie liczbowych palindromów ... 186

Rozdział 6. ***Daty i godziny*** ... ***191***

6.0. Wprowadzenie ... 191
6.1. Określanie bieżącej daty ... 193
6.2. Wyświetlanie daty i czasu w zadanym formacie ... 194
6.3. Przedstawianie dat w innych systemach ... 196
6.4. Konwersja liczb określających datę i czas na obiekt Calendar lub ilość sekund ... 197
6.5. Analiza łańcuchów znaków i ich zamiana na daty ... 198
6.6. Konwersja dat wyrażonych w formie sekund na obiekt Date ... 200
6.7. Dodawanie i odejmowanie dat przy wykorzystaniu klas Date oraz Calendar ... 201

6.8. Obliczanie różnic pomiędzy dwiema datami ... 202
6.9. Porównywanie dat ... 203
6.10. Określanie dnia tygodnia, miesiąca lub roku oraz numeru tygodnia ... 205
6.11. Wyświetlanie kalendarza ... 207
6.12. Czasomierze o dużej dokładności ... 209
6.13. Wstrzymywanie wykonywania programu ... 211
6.14. Program — usługa przypominająca ... 212

Rozdział 7. *Strukturalizacja danych w języku Java* ... 215

7.0. Wprowadzenie ... 215
7.1. Strukturalizacja danych przy użyciu tablic ... 216
7.2. Modyfikacja wielkości tablic ... 218
7.3. Klasa podobna do tablicy, lecz bardziej dynamiczna ... 219
7.4. Iteratory — dostęp do danych w sposób niezależny od ich typów ... 221
7.5. Strukturalizacja danych przy wykorzystaniu list połączonych ... 223
7.6. Odwzorowywanie przy wykorzystaniu klas Hashtable oraz HashMap ... 225
7.7. Zapisywanie łańcuchów znaków w obiektach Properties i Preferences ... 226
7.8. Sortowanie kolekcji ... 230
7.9. Sortowanie w języku Java 1.1 ... 234
7.10. Unikanie konieczności sortowania danych ... 235
7.11. Zbiory ... 237
7.12. Odnajdywanie obiektu w kolekcji ... 238
7.13. Zamiana kolekcji na tablicę ... 240
7.14. Tworzenie własnego iteratora ... 241
7.15. Stos ... 243
7.16. Struktury wielowymiarowe ... 244
7.17. Kolekcje ... 246
7.18. Program — porównanie szybkości działania ... 246

Rozdział 8. *Techniki obiektowe* ... 251

8.0. Wprowadzenie ... 251
8.1. Wyświetlanie obiektów — formatowanie obiektów przy użyciu metody toString() ... 253
8.2. Przesłanianie metody equals ... 255
8.3. Przesłanianie metody hashCode ... 257
8.4. Metoda clone ... 259
8.5. Metoda finalize ... 261
8.6. Wykorzystanie klas wewnętrznych ... 263
8.7. Tworzenie metod zwrotnych przy wykorzystaniu interfejsów ... 264

8.8. Polimorfizm i metody abstrakcyjne 268
8.9. Przekazywanie wartości 270
8.10. Zgłaszanie własnych wyjątków 273
8.11. Program Plotter 274

Rozdział 9. *Wejście i wyjście* 277
9.0. Wprowadzenie 277
9.1. Odczytywanie informacji ze standardowego strumienia wejściowego 282
9.2. Zapis danych w standardowym strumieniu wyjściowym 285
9.3. Otwieranie pliku o podanej nazwie 287
9.4. Kopiowanie plików 287
9.5. Odczytywanie zawartości pliku i zapisywanie jej w obiekcie String 291
9.6. Zmiana skojarzeń strumieni standardowych 292
9.7. Powielanie strumienia podczas realizacji operacji zapisu 293
9.8. Odczyt i zapis danych zakodowanych w innym zbiorze znaków 296
9.9. Te kłopotliwe znaki końca wiersza 297
9.10. Kod operujący na plikach w sposób zależny od systemu operacyjnego 298
9.11. Odczytywanie „podzielonych" wierszy tekstu 299
9.12. Analiza zawartości pliku 304
9.13. Dane binarne 309
9.14. Przeszukiwanie 310
9.15. Zapisywanie danych w strumieniu przy wykorzystaniu języka C 311
9.16. Zapisywanie i odczytywanie serializowanych obiektów 313
9.17. Unikanie wyjątków ClassCastException spowodowanych nieprawidłowymi wartościami SerialVersionUID 315
9.18. Odczytywanie i zapisywanie danych w archiwach JAR oraz Zip 317
9.19. Odczytywanie i zapisywanie plików skompresowanych 320
9.20. Program — zamiana tekstu do postaci PostScript 322
9.21. Program TarList (konwerter plików) 324

Rozdział 10. *Operacje na katalogach i systemie plików* 337
10.0. Wprowadzenie 337
10.1. Pobieranie informacji o pliku 338
10.2. Tworzenie pliku 341
10.3. Zmiana nazwy pliku 342
10.4. Usuwanie plików 342
10.5. Tworzenie plików tymczasowych 344
10.6. Zmiana atrybutów pliku 346

10.7. Tworzenie listy zawartości katalogu 347
10.8. Pobieranie katalogów głównych 349
10.9. Tworzenie nowych katalogów 351
10.10. Program Find 352

Rozdział 11. *Programowa obsługa portu szeregowego i równoległego* 355

11.0. Wprowadzenie 355
11.1. Wybieranie portu 358
11.2. Otwieranie portu szeregowego 361
11.3. Otwieranie portu równoległego 365
11.4. Rozwiązywanie konfliktów portów 368
11.5. Odczyt i zapis — metoda krokowa 372
11.6. Odczyt i zapis — przetwarzanie sterowane zdarzeniami 374
11.7. Odczyt i zapis — wątki 378
11.8. Program — obsługa plotera Penman 380

Rozdział 12. *Grafika i dźwięk* 385

12.0. Wprowadzenie 385
12.1. Rysowanie przy użyciu obiektu Graphics 386
12.2. Testowanie komponentów graficznych 387
12.3. Wyświetlane tekstu 389
12.4. Wyświetlanie wyśrodkowanego tekstu w komponencie 389
12.5. Rysowanie cienia 391
12.6. Wyświetlanie obrazu 393
12.7. Odtwarzanie pliku dźwiękowego 398
12.8. Prezentacja ruchomego obrazu 399
12.9. Wyświetlanie tekstu przy użyciu biblioteki grafiki dwuwymiarowej 402
12.10. Drukowanie — JDK 1.1 405
12.11. Drukowanie — Java 2 408
12.12. Program Ploter AWT 410
12.13. Program Grapher 412

Rozdział 13. *Graficzny interfejs użytkownika* 417

13.0. Wprowadzenie 417
13.1. Wyświetlanie komponentów graficznego interfejsu użytkownika 419
13.2. Projektowanie układu okna 420
13.3. Zakładki — nowe spojrzenie na świat 423
13.4. Obsługa czynności — tworzenie działających przycisków 424
13.5. Obsługa czynności przy wykorzystaniu anonimowych klas wewnętrznych 427

13.6. Kończenie programu przy użyciu przycisku „Zamknij” 429
13.7. Okna dialogowe — tego nie można zrobić później 434
13.8. Wyświetlanie wyników wykonania programu w oknie 436
13.9. Wybieranie plików przy użyciu klasy JFileChooser 439
13.10. Wybieranie koloru 443
13.11. Wyświetlanie okna głównego pośrodku ekranu 445
13.12. Zmiana sposobów prezentacji programów pisanych z wykorzystaniem pakietu Swing 447
13.13. Program — własne narzędzie do wybierania czcionek 451
13.14. Program — własny menedżer układu 456

Rozdział 14. ***Tworzenie programów wielojęzycznych oraz lokalizacja* *463***

14.0. Wprowadzenie 463
14.1. Tworzenie przycisku w różnych wersjach językowych 464
14.2. Tworzenie listy dostępnych ustawień lokalnych 466
14.3. Tworzenie menu przy wykorzystaniu zasobów wielojęzycznych 467
14.4. Tworzenie metod pomocniczych, przydatnych przy tworzeniu programów wielojęzycznych 468
14.5. Tworzenie okien dialogowych przy wykorzystaniu zasobów wielojęzycznych 470
14.6. Tworzenie wiązki zasobów 471
14.7. Przygotowywanie programów wielojęzycznych 473
14.8. Wykorzystanie konkretnych ustawień lokalnych 474
14.9. Określanie domyślnych ustawień lokalnych 475
14.10. Formatowanie komunikatów 476
14.11. Program MenuIntl 478
14.12. Program BusCard 480

Rozdział 15. ***Klienty sieciowe* *485***

15.0. Wprowadzenie 485
15.1. Nawiązywanie połączenia z serwerem 487
15.2. Odnajdywanie i zwracanie informacji o adresach sieciowych 489
15.3. Obsługa błędów sieciowych 490
15.4. Odczyt i zapis danych tekstowych 491
15.5. Odczyt i zapis danych binarnych 494
15.6. Odczyt i zapis danych serializowanych 496
15.7. Datagramy UDP 498
15.8. Program — klient TFTP wykorzystujący protokół UDP 501
15.9. Program — klient usługi Telnet 505
15.10. Program — klient pogawędek internetowych 507

Rozdział 16. ***Programy Javy działające na serwerze — gniazda*** **513**

16.0. Wprowadzenie 513
16.1. Tworzenie serwera 514
16.2. Zwracanie odpowiedzi (łańcucha znaków bądź danych binarnych) 517
16.3. Zwracanie informacji o obiektach 521
16.4. Obsługa wielu klientów 522
16.5. Rejestracja operacji sieciowych 527
16.6. Program — serwer pogawędek w Javie 531

Rozdział 17. ***Klienty sieciowe II — aplety i klienty WWW*** **537**

17.0. Wprowadzenie 537
17.1. Osadzanie apletów w stronach WWW 538
17.2. Techniki tworzenia apletów 539
17.3. Nawiązywanie połączenia z komputerem, z którego pobrano aplet 542
17.4. Wyświetlanie dokumentu z poziomu apletu 545
17.5. Uruchamianie skryptu CGI z poziomu apletu 547
17.6. Odczyt zawartości zasobu wskazywanego przez adres URL 549
17.7. Pobieranie znaczników HTML z dokumentu o podanym adresie URL 550
17.8. Pobieranie adresów URL zapisanych w pliku 552
17.9. Zamiana nazwy pliku na adres URL 554
17.10. Program MkIndex 555
17.11. Program LinkChecker 559

Rozdział 18. ***Programy działające na serwerach — sewlety i JSP*** **567**

18.0. Wprowadzenie 567
18.1. Pierwszy serwlet — generowanie strony WWW 569
18.2. Serwlety — przetwarzanie danych przekazywanych z formularzy 572
18.3. Cookies 575
18.4. Śledzenie sesji 579
18.5. Generowanie plików PDF przez serwlety 585
18.6. Połączenie HTML-a i Javy — JSP 591
18.7. Dołączanie plików i przekierowania w JSP 596
18.8. Strony JSP wykorzystujące serwlety 597
18.9. Upraszczanie kodu stron JSP dzięki wykorzystaniu komponentów JavaBeans 598
18.10. Składnia JSP 603
18.11. Program CookieCutter 603
18.12. Program — portal informacyjny JabaDot 604

Rozdział 19. *Java i poczta elektroniczna* ... 617
19.0. Wprowadzenie ... 617
19.1. Wysyłanie poczty elektronicznej — wersja działająca w przeglądarkach ... 618
19.2. Wysyłanie poczty elektronicznej — właściwe rozwiązanie ... 622
19.3. Dodawanie możliwości wysyłania poczty do programu działającego na serwerze ... 625
19.4. Wysyłanie wiadomości MIME ... 631
19.5. Tworzenie ustawień poczty elektronicznej ... 634
19.6. Wysyłanie poczty elektronicznej bez użycia JavaMail ... 636
19.7. Odczytywanie poczty elektronicznej ... 640
19.8. Program MailReaderBean ... 645
19.9. Program MailClient ... 648

Rozdział 20. *Dostęp do baz danych* ... 659
20.0. Wprowadzanie ... 659
20.1. Baza danych składająca się z plików tekstowych ... 661
20.2. Bazy danych DBM ... 666
20.3. Konfiguracja i nawiązywanie połączeń JDBC ... 669
20.4. Nawiązywanie połączenia z bazą danych JDBC ... 672
20.5. Przesyłanie zapytań JDBC i pobieranie wyników ... 675
20.6. Wykorzystanie sparametryzowanych poleceń JDBC ... 678
20.7. Wykorzystanie procedur osadzonych w JDBC ... 682
20.8. Modyfikacja danych przy użyciu obiektu ResultSet ... 683
20.9. Modyfikacja danych przy użyciu poleceń SQL ... 685
20.10. Odnajdywanie metadanych JDBC ... 687
20.11. Program JDAdmin ... 693

Rozdział 21. *XML* ... 699
21.0. Wprowadzenie ... 699
21.1. Przekształcanie danych XML przy użyciu XSLT ... 702
21.2. Analiza składniowa XML przy użyciu API SAX ... 705
21.3. Analiza dokumentów XML przy użyciu modelu obiektów dokumentu (DOM) ... 708
21.4. Weryfikacja poprawności struktury przy wykorzystaniu DTD ... 710
21.5. Generowanie własnego kodu XML przy wykorzystaniu DOM ... 711
21.6. Program xml2mif ... 713

Rozdział 22. *Java w aplikacjach rozproszonych — RMI* ... 717
22.0. Wprowadzenie ... 717
22.1. Definiowanie kontraktu RMI ... 719

22.2. Klient RMI ... 721
22.3. Serwer RMI ... 722
22.4. Uruchamianie aplikacji RMI w sieci ... 725
22.5. Program — wywołania zwrotne RMI ... 726
22.6. Program RMIWatch ... 730

Rozdział 23. *Pakiety i ich tworzenie* ... 737
23.0. Wprowadzenie ... 737
23.1. Tworzenie pakietu ... 738
23.2. Tworzenie dokumentacji klas przy użyciu programu Javadoc ... 739
23.3. Stosowanie programu archiwizującego jar ... 743
23.4. Uruchamianie apletu zapisanego w pliku JAR ... 744
23.5. Wykonywanie apletu przy użyciu JDK ... 745
23.6. Uruchamianie programu zapisanego w pliku JAR ... 749
23.7. Tworzenie klasy w taki sposób, by była komponentem JavaBean ... 749
23.8. Umieszczanie komponentów w plikach JAR ... 753
23.9. Umieszczanie serwletów w plikach JAR ... 754
23.10. „Zapisz raz, instaluj wszędzie” ... 755
23.11. Java Web Start ... 756
23.12. Podpisywanie plików JAR ... 762

Rozdział 24. *Stosowanie wątków w Javie* ... 765
24.0. Wprowadzenie ... 765
24.1. Uruchamianie kodu w innym wątku ... 767
24.2. Animacja — wyświetlanie poruszających się obrazów ... 771
24.3. Zatrzymywanie działania wątku ... 775
24.4. Spotkania i ograniczenia czasowe ... 777
24.5. Komunikacja między wątkami — kod synchronizowany ... 779
24.6. Komunikacja między wątkami — metody wait() oraz notifyAll() ... 785
24.7. Zapis danych w tle w programach edycyjnych ... 791
24.8. Wielowątkowy serwer sieciowy ... 792

Rozdział 25. *Introspekcja lub „klasa o nazwie Class”* ... 801
25.0. Wprowadzenie ... 801
25.1. Pobieranie deskryptora klasy ... 802
25.2. Określanie i stosowanie metod i pól ... 803
25.3. Dynamiczne ładowanie i instalowanie klas ... 807
25.4. Tworzenie nowej klasy od podstaw ... 809
25.5. Określanie efektywności działania ... 811

25.6. Wyświetlanie informacji o klasie 815
25.7. Program CrossRef 816
25.8. Program AppletViewer 821

Rozdział 26. ***Wykorzystywanie Javy wraz z innymi językami programowania*** **........ *829***

26.0. Wprowadzenie 829
26.1. Uruchamianie programu 830
26.2. Wykonywanie programu i przechwytywanie jego wyników 833
26.3. Wykorzystanie BSF do łączenia kodu Javy i skryptów 836
26.4. Dołączanie kodu rodzimego (C/C++) 841
26.5. Wywoływanie kodu Javy z kodu rodzimego 847
26.6. Program DBM 848

Posłowie 851

Skorowidz 853

8

Techniki obiektowe

8.0. Wprowadzenie

Java jest językiem zorientowanym obiektowo, czerpiącym z tradycji takich języków jak Simula-67, SmallTalk oraz C++. Składania Javy jest wzorowana na języku C++, natomiast wykorzystywane w niej rozwiązania na języku SmallTalk. Interfejs programistyczny (API) Javy został stworzony w oparciu o model obiektowy. Często są w nim wykorzystywane *wzorce projektowe* (patrz książka „Java. Wzorce projektowe" wydana przez wydawnictwo Helion) takie jak Factory (fabryka) oraz Delegate (delegat). Ich znajomość, choć nie jest konieczna, na pewno pomoże w lepszym zrozumieniu działania API.

Rady lub mantry

Jest bardzo wiele drobnych porad, których mógłbym udzielić. Jest też kilka zagadnień, które wciąż powracają podczas poznawania podstaw języka Java i później, na kolejnych etapach nauki.

Wykorzystanie standardowego API

Na ten temat nigdy nie można powiedzieć zbyt dużo. Bardzo wiele zagadnień, z którymi się borykamy, zostało już rozwiązanych przez programistów firmy JavaSoft. Dobre poznanie API jest najlepszym sposobem uniknięcia syndromu „wyważania otwartych drzwi" — czyli wymyślania podrzędnych substytutów doskonałych produktów, z których w każdej chwili można skorzystać. Po części takie jest właśnie przeznaczenie niniejszej książki — ochrona przed ponownym odkrywaniem rzeczy, które już zostały stworzone. Jednym z takich przykładów może być interfejs programistyczny pozwalający na tworzenie i wykorzystanie różnego rodzaju kolekcji, dostępny w pakiecie `java.util`

i opisany dokładniej w poprzednim rozdziale. Cechuje go wysoka uniwersalność i regularność, dzięki czemu potrzeba tworzenia własnego kodu służącego do strukturalizacji danych jest bardzo mała.

Dążenie do ogólności

Istnieje pewien kompromis pomiędzy ogólnością (i wynikającymi z niej możliwościami wielokrotnego używania kodu), na którą kładę nacisk w niniejszej książce, oraz wygodą, jaką daje dostosowanie używanych rozwiązań do tworzonej aplikacji. Jeśli tworzymy pewną niewielką część bardzo dużej aplikacji zaprojektowanej zgodnie z technikami projektowania obiektowego, na pewno będziemy pamiętać o określonym zbiorze tak zwanych *przypadków zastosowania*. Jednak z drugiej strony, jeśli będziemy pisać „pakiet narzędziowy", to warto tworzyć klasy, wykorzystując jak najmniej założeń co do przyszłych sposobów ich wykorzystania. Zapewnienie łatwości wykorzystania kodu w jak największej ilości różnych programów jest najlepszym sposobem tworzenia kodu gwarantującego możliwość wielokrotnego wykorzystania.

Czytanie i tworzenie dokumentacji (Javadoc)

Bez wątpienia Czytelnik przeglądał dokumentację Javy dostępną w formie dokumentów HTML; częściowo zapewne dlatego, że poleciłem dobrze poznać standardowy interfejs programistyczny. Czy firma Sun wynajęła tłumy pisarzy w celu stworzenia tej dokumentacji? Otóż nie. Dokumentacja ta istnieje dzięki temu, iż twórcy Java API poświęcili czas, aby pisać specjalne komentarze dokumentujące — te śmieszne wiesze zaczynające się od znaków `/**`, które można zobaczyć w tak wielu przykładach. A zatem, jeszcze jedna rada: sami też używajmy tych komentarzy. Nareszcie dysponujemy dobrym, standardowym mechanizmem służącym do dokumentowania tworzonych interfejsów programistycznych. I jeszcze uwaga: komentarze te powinny być pisane podczas tworzenia kodu — nie łudźmy się, że uda się napisać je później. W przyszłości nigdy nie będzie na to czasu.

Szczegółowe informacje na temat tworzenia dokumentacji i wykorzystania programu Javadoc można znaleźć w recepturze 23.2.

Klasy potomne powinny być tworzone jak najwcześniej i jak najczęściej

Także o tym wciąż należy przypominać. Tworzenie klas potomnych jest ze wszech miar zalecane. Należy je tworzyć zawsze, kiedy mamy po temu okazję. To najlepszy sposób nie tylko na uniknięcie powtarzania kodu, lecz także na tworzenie działających programów. Informacje na ten temat można znaleźć w wielu dobrych książkach poświęconych technikom projektowania i programowania obiektowego. Wzorce projektowe stały się ostatnio szczególnym przypadkiem „projektowania obiektowego przy jednoczesnym unikaniu wymyślania już opracowanych rozwiązań". Dlatego też połączyłem obie te rady. Na początku warto zajrzeć do dobrej książki.

Korzystanie z wzorców projektowych

We Wstępie, w części zatytułowanej „Inne książki”, wymieniłem pozycję *Java. Wzorce projektowe*, zaznaczając, iż jest to bardzo ważna praca o projektowaniu obiektowym, gdyż zawiera wyczerpujący zbiór informacji o rozwiązaniach, które programiści często wymyślają od nowa. Książka ta jest niezwykle istotna zarówno ze względu na to, że tworzy standardowe słownictwo związane z projektowaniem, jak i dlatego, że w przystępnej formie tłumaczy sposób działania prostych wzorców projektowych oraz pokazuje, jak można je zaimplementować.

Poniżej podałem kilka przykładów wykorzystania wzorców projektowych w standardowym API języka Java.

Wzorzec projektowy	Znaczenie	Przykłady w Java API
Factory (fabryka)	Jedna klasa tworzy kopie obiektów kontrolowane przez klasy potomne.	`getInstance` (w klasach `Calendar`, `Format`, `Locale`, ...); konstruktor gniazd;
Iterator	Pobiera po kolei wszystkie elementy kolekcji, przy czym każdy jest pobierany tylko i wyłącznie raz.	`Iterator` oraz starszy interfejs `Enumeration`
Singleton	Istnieje tylko jedna kopia obiektu.	`java.awt.Toolkit`
Memento	Pobiera i udostępnia stan obiektu, tak aby można go było odtworzyć później.	serializacja obiektów
Command (polecenie)	Hermetyzuje żądania, umożliwiając tworzenie kolejek żądań, odtwarzanie wykonywanych operacji i tak dalej.	`java.awt.Command`
Model-View-Controler (model-widok-kontroler)	Model reprezentuje dane, widok określa to, co widzi użytkownik, a kontroler odpowiada na żądania użytkowników.	`Observer`/`Observable`; patrz także `ServletDispatcher` (receptura 18.8)

8.1. Wyświetlanie obiektów — formatowanie obiektów przy użyciu metody toString()

Problem

Chcemy, aby obiekty mogły być prezentowane w przydatny, intuicyjny sposób.

Rozwiązanie

Należy przesłonić metodę `toString()` odziedziczoną po obiekcie `java.lang.Object`.

Analiza

Za każdym razem, gdy obiekt zostanie przekazany w wywołaniu metody `System.out.println()`, innej analogicznej metody lub użyty w konkatenacji łańcuchów znaków, Java automatycznie wywoła jego metodę `toString()`. Java „wie", że metodę `toString()` posiada każdy obiekt, gdyż posiada ją klasa `java.lang.Object`, a zatem, także wszystkie jej klasy potomne. Domyślna implementacja tej metody użyta w klasie `Object` nie jest ani interesująca, ani przydatna — wyświetla nazwę kasy, znak `@` oraz wartość zwracaną przez metodę `hashCode()` (patrz receptura 8.3). Na przykład, jeśli wykonamy poniższy program:

```
/* Prezentacja metody toString() (bez przesłaniania). */
public class ToStringWithout {
  int x, y;

    /** Prosty konstruktor. */
    public ToStringWithout(int anX, int aY) {
        x = anX; y = aY;
    }

    /** Metoda main - tworzy i wyświetla obiekt. */
    public static void main(String[] args) {
        System.out.println(new ToStringWithout(42, 86));
    }
}
```

przekonamy się, że generuje on niezbyt czytelne wyniki:

```
ToStringWithout@ad3ba4
```

A zatem, aby dane o obiekcie były prezentowane w lepszy sposób, we wszystkich, oprócz najprostszych, klasach należy zaimplementować metodę `toString()`, która będzie wyświetlać nazwę klasy oraz jakieś ważne informacje na temat bieżącego stanu obiektu. W ten sposób można uzyskać lepszą kontrolę nad sposobem formatowania informacji o obiektach w metodzie `println()`, w programach uruchomieniowych oraz wszędzie tam, gdzie odwołanie do obiektu występuje w kontekście łańcucha znaków. Poniżej przedstawiłem zmodyfikowaną wersję poprzedniego programu, w której została zaimplementowana metoda `toString()`:

```
/* Prezentacja metody toString() (przesłoniętej). */
public class ToStringWith {
    int x, y;

    /** Prosty konstruktor. */
    public ToStringWith(int anX, int aY) {
        x = anX; y = aY;
    }

    /** Nowa wersja metody toString. */
    public String toString() {
        return "ToStringWith[" + x + "," + y + "]";
    }
    /** Metoda main tworzy i wyświetla obiekt. */
    public static void main(String[] args) {
        System.out.println(new ToStringWith(42, 86));
    }
}
```

Ta wersja programu wyświetla znacznie bardziej przydatne wyniki:

```
ToStringWith[42,86]
```

8.2. Przesłanianie metody equals

Problem

Chcemy mieć możliwość porównywania obiektów zdefiniowanej przez nas klasy.

Rozwiązanie

Należy stworzyć metodę `equals()`.

Analiza

W jaki sposób określamy równość? W przypadku operatorów arytmetyczny lub logicznych, odpowiedź na to pytanie jest prosta — wystarczy sprawdzić dwie wartości przy użyciu operatora równości (==). Jednak w przypadku porównywania odwołań do obiektów Java udostępnia dwa rozwiązania — operator == oraz metodę `equals()` odziedziczoną po klasie `java.lang.Object`.

Operator równości może być nieco mylący, gdyż jego działanie sprowadza się do porównania dwóch odwołań i sprawdzenia, czy wskazują one ten sam obiekt.

Także odziedziczona metoda `equals()` nie jest aż tak przydatna, jak można by sobie wyobrażać. Niektórzy zaczynają swoją karierę programistów używających Javy, sądząc, że domyślna metoda `equals()` może dokonać jakiegoś magicznego porównania zawartości obiektów pole po polu, a nawet porównać je w sposób binarny. Niemniej jednak metoda ta *nie* porównuje pól! Działa ona w najprostszy z możliwych sposobów — zwraca wartość porównania dwóch obiektów przy użyciu operatora ==. A zatem, aby móc wykonać jakąkolwiek przydatną operację, trzeba samemu stworzyć metodę `equals()` odpowiednią dla danej klasy. Warto zauważyć, że zarówno metoda `equals()`, jak i `hashCode()` są wykorzystywane przez tablice mieszające (klasy `Hashtable` oraz `HashMap`, przedstawione w recepturze 7.6). A zatem, jeśli istnieje prawdopodobieństwo, że inne osoby używające naszych klas mogą chcieć przechowywać je w tablicach mieszających lub tylko porównywać, to powinniśmy, zarówno ze względu na nich, jak i na siebie, poprawnie zaimplementować metodę `equals()`.

Poniżej przedstawiłem zasady działania tej metody:

1. Metoda `equals()` jest zwrotna, czyli `x.equals(x)` musi zwracać wartość `true`.
2. Metoda `equals()` jest symetryczna, czyli `x.equals(y)` musi zwrócić wartość `true` wtedy i tylko wtedy, gdy `y.equals(x)` także zwraca wartość `true`.

3. Metoda `equals()` jest przechodnia, czyli jeśli `x.equals(y)` zwraca wartość `true` oraz `y.equals(z)` także zwraca wartość `true`, to `x.equals(z)` także musi zwracać wartość `true`.
4. Metoda jest spójna, czyli wielokrotne wywoływanie metody `x.equals(y)` zawsze powinno zwracać tę samą wartość (chyba że zmieniły się zmienne określające stan wykorzystywane przy porównywaniu obiektów).
5. Metoda musi być „ostrożna", czyli wywołanie `x.equals(null)` musi zwracać wartość `false`; nie może natomiast zgłaszać wyjątku `NullPointerException`.

Przedstawiona poniżej klasa stara się zaimplementować te zasady:

```
public class EqualsDemo {
    int int1;
    SomeClass obj1;

    /** Konstruktor. */
    public EqualsDemo(int i, SomeClass o) {
        int1 = i;
        obj1 = o;
    }

    public EqualsDemo() {
        this(0, new SomeClass());
    }

    /** Najbardziej typowa implementacja metody equals(). */
    public boolean equals(Object o) {
        if (o == null)        // Zabezpieczenie.
            return false;
        if (o == this)        // Optymalizacja.
            return true;

        // Czy można rzutować do tej klasy?
        if (!(o instanceof EqualsDemo))
            return false;

        EqualsDemo other = (EqualsDemo)o;    // OK, można rzutować.

        // Porównanie poszczególnych pól.
        if (int1 != other.int1)             // Typy proste porównujemy
        bezpośrednio,
            return false;
        if (!obj1.equals(other.obj1))    // a obiekty przy użyciu metody equals()
            return false;
        return true;
    }
}
```

Poniżej przedstawiłem program testowy `junit` (patrz receptura 1.13) dla klasy `EqualsDemo`:

```
import junit.framework.*;
/** Przykładowe testy dla klasy EqualsDemo
 * przeznaczone do wykorzystania z narzędziem junit
 * stworzenie pełnego zbioru testów pozostawiam jako
 * zadanie "do samodzielnego wykonania" dla Czytelnika.
 * Sposób uruchamiania: $ java junit.textui.TestRunner EqualsDemoTest
```

```
 * @version $Id: EqualsDemoTest.java,v 1.2 2002/06/20 20:25:03 ian Exp $
 */
public class EqualsDemoTest extends TestCase {

    /** Testowany obiekt. */
    EqualsDemo d1;
    /** Inny testowany obiekt. */
    EqualsDemo d2;

    /** Metoda init().  */
    public void setUp() {
        d1 = new EqualsDemo();
        d2 = new EqualsDemo();
    }

    /** Konstruktor używany przez junit. */
    public EqualsDemoTest(String name) {
        super(name);
    }

    public void testSymmetry() {
        assertTrue(d1.equals(d1));
    }

    public void testSymmetric() {
        assertTrue(d1.equals(d2) && d2.equals(d1));
    }

    public void testCaution() {
        assertTrue(!d1.equals(null));
    }
}
```

Przy tak dokładnym przetestowaniu, czy jest jeszcze coś, co może w tej metodzie zawieść? Cóż, trzeba zadbać o kilka spraw. Co się stanie, jeśli obiekt przekazany w wywołaniu będzie obiektem klasy potomnej klasy `EqualsDemo`? Wykonamy rzutowanie i… porównamy wyłącznie pola przekazanego obiektu! Jeśli zatem możliwe jest operowanie na obiektach klas potomnych, to prawdopodobnie trzeba będzie jawnie testować obiekty przy użyciu metody `getClass()`. Obiekty klas potomnych powinny natomiast wywoływać metodę `super.equals()`, aby sprawdzać wszystkie pola klasy bazowej.

Co jeszcze może zawieść? A co, jeśli jeden z obiektów, `obj1` lub `inny.obj1`, będzie równy `null`? W takim przypadku możemy zobaczyć śliczny komunikat o wyjątku `NullPointerException`. A zatem, dodatkowo należy sprawdzać wszystkie miejsca, gdzie mogą się pojawić wartości `null`. Problemów tego typu można uniknąć, tworząc dobre konstruktory (co starałem się zrobić w klasie `EqualsDemo`) lub jawnie sprawdzając, czy gdzieś nie pojawiła się wartość `null`.

8.3. Przesłanianie metody hashCode

Problem

Chcemy wykorzystywać obiekty w tablicach mieszających i w tym celu musimy stworzyć metodę `hashCode()`.

Analiza

Według założeń, metoda `hashCode()` ma zwracać liczbę całkowitą typu `int`, której wartości będą w unikalny sposób identyfikować obiekty danej klasy.

Poprawnie napisana metoda `hashCode()` spełnia następujące założenia:

1. Jest powtarzalna — kilkukrotnie wywołana metoda `x.hashCode()` musi zwracać tę samą wartość, chyba że w międzyczasie zmienił się stan obiektu.
2. Metoda `hashCode()` jest symetryczna, czyli: jeśli `x.equals(y)`, to `x.hashCode() == y.hashCode()`; oba wyrażenia muszą zwracać bądź wartość `true`, bądź wartość `false`.
3. Nie ma reguły, która nakazuje, by w przypadku, gdy wywołanie `x.equals(y)` zwraca wartość `false`, także wartości `x.hashCode()` oraz `y.hashCode()` były od siebie różne. Niemniej jednak zapewnienie takiego właśnie działania metody `hashCode()` może poprawić efektywność działania tablic mieszających; na przykład, tablice mogą wywoływać metodę `hashCode()` przed wywołaniem metody `equals()`.

Domyślna implementacja metody `hashCode()` wykorzystana w JDK firmy Sun zwraca adres maszynowy i jest zgodna z przedstawioną powyżej zasadą 1. Zgodność z zasadami 2. i 3., po części zależy od utworzonej metody `equals()`. Poniżej przedstawiłem program wyświetlający kody mieszające kilku obiektów:

```
/** Wyświetla kody mieszające (zwracane przez metodę hashCode())
 * kilku obiektów.
 */
public class PrintHashCodes {

    /** Obiekty, dla których wyświetlimy kody mieszające. */
    protected static Object[] data = {
        new PrintHashCodes(),
        new java.awt.Color(0x44, 0x88, 0xcc),
        new SomeClass()
    };

    public static void main(String[] args) {
        System.out.println("Obliczamy kod mieszający " + data.length + "
        obiektów.");
        for (int i=0; i<data.length; i++) {
            System.out.println(data[i].toString() + " --> " +
                data[i].hashCode());
        }
        System.out.println("Gotowe.");
    }
}
```

Jakie wyniki generuje powyższy program?

```
> jikes +E -d . PrintHashCodes.java
> java PrintHashCodes
Obliczamy kod mieszający 3 obiektów.
PrintHashCodes@7c6768 --> 8152936
java.awt.Color[r=68,g=136,b=204] --> -12285748
SomeClass@690726 --> 6883110
Gotowe.
>
```

Interesująca jest wartość mieszająca dla obiektów klasy `Color`. Jest ona obliczana w następujący sposób:

```
(r<<24 + g<<16 + b<<8 + alpha)
```

Najstarszy bit tego słowa uzyskany w wyniku przesunięcia powoduje, że w przypadku wyświetlania tej wartości jako liczby całkowitej ze znakiem, jest ona wyświetlana jako wartość ujemna. Nic nie stoi na przeszkodzie, by kody mieszające miały wartości ujemne.

8.4. Metoda clone

Problem

Chcemy się sklonować. No... przynajmniej nasze obiekty.

Rozwiązanie

Należy przesłonić metodę `Object.clone()`.

Analiza

Sklonowanie oznacza zrobienie duplikatu. Metoda `clone()` dostępna w Javie tworzy dokładną kopię obiektu. Dlaczego mielibyśmy potrzebować takiej możliwości? Otóż w języku Java argumenty wywołań metod są przekazywane przez odwołanie, dzięki czemu wywoływana metoda może zmienić stan przekazanego do niej obiektu. Sklonowanie obiektu przed wywołaniem metody, spowoduje przekazanie do niej kopii obiektu, dzięki czemu jego oryginał będzie bezpieczny.

W jaki sposób można klonować obiekty? Możliwość klonowania nie jest domyślnie dostępna w tworzonych klasach.

```
Object o = new Object();
Object o2 = o.clone();
```

Próba wywołania metody `clone()` bez specjalnego przygotowania spowoduje wyświetlenie stosownego komunikatu. Poniższy przykład przedstawia komunikat wygenerowany przez kompilator Jikes w przypadku próby skompilowania pliku *Clone0.java* (komunikaty generowane przez kompilator *javac* mogą być mniej opisowe):

```
Clone0.java:4:29:4:37: Error: Method "java.lang.Object clone();" in class
"java/lang/Object" has protected or default access. Therefore it is not
accessible in class "Clone0" which is in a different package.
```

W celu zapewnienia możliwości klonowania obiektów tworzonej klasy, należy:

1. Przesłonić metodę `clone()` klasy `Object`.
2. Zaimplementować pusty interfejs `Cloneable`.

Wykorzystanie klonowania

W klasie `java.lang.Object` metoda `clone()` została zadeklarowana jako *chroniona* i. Metody chronione mogą być wywoływane wyłącznie przez obiekty klas potomnych lub innych klas należących do tego samego pakietu (w tym przypadku przez klasy pakietu `java.lang`), jednak nie przez obiekty innych klas, w żaden sposób nie powiązanych z klasą deklarującą te metody. Poniżej przedstawiłem przykład klasy posiadającej metodę `clone()` oraz prosty program, który z tej klasy korzysta:

```
public class Clone1 implements Cloneable {

    /** Klonujemy ten obiekt. W tym celu wystarczy wywołać
     * metodę super.clone().
     */
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    int x;
    transient int y;    // Sklonujemy, jednak nie podlega on serializacji.

    public static void main(String[] args) {
        Clone1 c = new Clone1();
        c.x = 100;
        c.y = 200;
        try {
            Object d = c.clone();
            System.out.println("c=" + c);
            System.out.println("d=" + d);
        } catch (CloneNotSupportedException ex) {
            System.out.println("A oto i niespodzianka!!");
            System.out.println(ex);
        }
    }

    /** Wyświetlamy bieżący obiekt w postaci łańcucha znaków. */
    public String toString() {
        return "Clone1[" + x + "," + y + "]";
    }
}
```

Metoda `clone()` klasy `Object` zgłasza wyjątek `CloneNotSupportedException`. Jest to sposób zabezpieczenia się przed nieostrożnym wywoływaniem metody `clone()` dla klas, których obiekty nie mają być klonowane. Ponieważ wyjątku tego zazwyczaj nie trzeba obsługiwać, wystarczy, że metoda `clone()` zadeklaruje go w klauzuli `throws`, dzięki czemu wyjątek będzie mógł zostać obsłużony przez kod wywołujący tę metodę.

Wywołanie metody `clone()` klasy `Object` tworzy „płytką" kopię obiektu zachowującą jego stan, przy czym operacja ta jest wykonywana na niskim poziomie wirtualnej maszyny Javy (JVM). Oznacza to, że metoda tworzy nowy obiekt i kopiuje do niego wartości wszystkich pól oryginału. Następnie metoda ta zwraca odwołanie do nowego obiektu klasy `Object`, co oznacza, że konieczne będzie wykonanie odpowiedniego rzutowania typu tego nowego obiektu. A zatem, jeśli te wszystkie operacje są wykonywane, to dlaczego musimy tworzyć tę metodę samodzielnie? Otóż jest to konieczne, aby można było wykonać wszelkie czynności związane z zachowaniem stanu, konieczne do

poprawnego skopiowania obiektów. Na przykład, jeśli klasa zawiera odwołania do innych obiektów (a większość klas używanych w realnych zastosowaniach zawiera takie odwołania), to być może będziemy chcieli skopiować także i te obiekty! Domyślna metoda `clone()` kopiuje cały bieżący stan obiektu, co sprawia, że po jej wywołaniu dysponujemy dwoma odwołaniami do każdego z obiektów. Być może trzeba także będzie zamknąć i ponownie otworzyć jakiś plik, aby uniknąć powstania sytuacji, w której dwa wątki (patrz rozdział 24.) odczytują lub zapisują dane w tym samym pliku. Jak widać, operacje, jakie należy wykonać w metodzie `clone()`, zależą od czynności wykonywanych przez pozostałe metody danej klasy.

Załóżmy teraz, że klonujemy klasę zawierającą tablicę obiektów. Po sklonowaniu obiektu takiej klasy będziemy dysponować dwoma odwołaniami do każdego z obiektów przechowywanych w tablicy. Jednak dodawanie kolejnych obiektów spowoduje modyfikację tylko jednej z tych tablic. Wyobraźmy sobie klasę zawierającą obiekty `Vector`, `Stack` lub jakiekolwiek inne kolekcje i pomyślmy, co się stanie, gdy obiekt zostanie sklonowany!

Podsumowując, odwołania do obiektów powinny być klonowane.

Nawet, jeśli tworzona klasa nie potrzebuje metody `clone()`, to może się okazać, że będą jej potrzebować jej klasy potomne. Jeśli w klasie potomnej klasy `Object` nie zostanie zaimplementowana metoda `clone()`, to zapewne będzie w niej dostępna wersja metody `clone()` użyta w klasie `Object`. To z kolei może przysporzyć problemów, jeśli w tworzonej klasie potomnej są wykorzystywane odwołania do kolekcji lub obiektów, których stan może się zmieniać. Ogólnie rzecz biorąc, należy tworzyć metody `clone()`, nawet, jeśli tylko nasze własne klasy potomne będą tej metody potrzebować.

Problemy występujące w standardowym API

Klasa `java.util.Observer` (zaprojektowana w celu implementacji wzorca projektowego model-widok-kontroler w aplikacjach wykorzystujących biblioteki AWT oraz Swing) zawiera prywatne pole klasy `Vector`, lecz nie posiada metody `clone()`, która zapewniałaby poprawne kopiowanie obiektów. Oznacza to, że nie można w bezpieczny sposób kopiować obiektów klasy `Observer`.

8.5. Metoda finalize

Problem

Chcemy, aby w momencie usuwania obiektów z pamięci zostały wykonane pewne czynności.

Rozwiązanie

Można użyć metody `finalize()`, lecz nie należy jej ufać; innym rozwiązaniem jest stworzenie własnej metody wykonywanej w przypadku usuwania obiektu z pamięci.

Analiza

Programiści, którzy wcześniej korzystali z języka C++, mają tendencję, aby utożsamiać destruktory stosowane w C++ z metodami finalizującymi języka Java. W C++ destruktory są wywoływane automatycznie w momencie usuwania obiektu. Jednak w Javie nie występuje żaden operator umożliwiający usunięcie obiektu z pamięci; obiekty są zwalniane automatycznie przez program wchodzący w skład środowiska wykonawczego Javy, odpowiedzialny za oczyszczanie pamięci. Jest on wykonywany jako wątek działający w tle procesów Javy. Program ten, tak często, jak to tylko możliwe, analizuje inne wykonywane programy Javy i sprawdza, czy występują w nich obiekty, do których nie ma już żadnych odwołań. W przypadku odnalezienia takiego obiektu, zanim mechanizm oczyszczający usunie go z pamięci, wywoła jego metodę `finalize()`.

Na przykład, co się stanie, gdy jakiś wywoływany przez nas fragment kodu wywoła metodę `System.exit()`? W takim przypadku zostanie zamknięta cała wirtualna maszyna Javy (oczywiście pod warunkiem, że nie ma żadnego menedżera zabezpieczeń przypominającego te wykorzystywane przy wykonywaniu apletów, który nie zezwoliłby na wykonanie takiej operacji) i metoda `finalize()` nigdy nie zostanie wykonana. Także w przypadku pojawienia się „przecieku pamięci" lub błędnego zapisania odwołania do obiektu jego metoda `finalize()` nie zostanie wykonana.

Ale czy nie da się wymusić wykonywania metod finalizujących poprzez wywołanie metody `System.runFinalizersOnExit()`? Niestety nie! Metoda ta została odrzucona (patrz receptura 1.9). W dokumentacji Javy można znaleźć następującą informację na jej temat:

> „Metoda ta jest z założenia niebezpieczna. Jej wywołanie może spowodować wywoływanie metod finalizujących istniejących obiektów w czasie, gdy korzystają z nich inne wątki wykonywane współbieżnie. Może to doprowadzić do dziwnego działania programu lub nawet do jego zawieszenia się".

Co zatem zrobić, jeśli w jakiś sposób powinniśmy „posprzątać" po sobie? Otóż w takich sytuacjach musimy przejąć na siebie odpowiedzialność za stworzenie odpowiedniej metody i wywoływanie jej, zanim obiekt będzie mógł zostać usunięty z pamięci. Metodzie takiej można by nadać nazwę `cleanUp()`[1].

W Java 2 SDK, w wersji 1.3, została wprowadzona metoda czasu wykonania programu (ang. *runtime method*) o nazwie `addShutdownHook()`, do której można przekazać niestandardowy obiekt klasy potomnej klasy `Thread`. Jeśli wirtualna maszyna Javy będzie mieć taką możliwość, to wykona wskazany kod podczas kończenia działania. Metoda ta zazwyczaj działa poprawnie, chyba że wirtualna maszyna Javy zostanie nagle zamknięta, na przykład poprzez sygnał kill w systemach Unix, sygnał KillProcess w 32-bitowych systemach Windows lub samoczynnie zakończy działanie ze względu na wykrycie nieprawidłowości w wewnętrznych strukturach danych.

---

[1] W tłumaczeniu: „czystka" lub „porządek".

A zatem, jakie wnioski? Co prawda nie ma co do tego żadnych gwarancji, niemniej jednak jest spora szansa, że zarówno metody finalizujące, jak i wątki uruchamiane podczas zamykania JVM zostaną poprawnie wykonane.

8.6. Wykorzystanie klas wewnętrznych

Problem

Musimy napisać klasę prywatną lub klasę, która będzie wykorzystywana co najwyżej w jednej innej klasie.

Rozwiązanie

Należy stworzyć klasę „niepubliczną" lub klasę wewnętrzną.

Analiza

Klasę „niepubliczną" można stworzyć, umieszczając ją w pliku źródłowym innej klasy, jednak poza jej definicją. Z kolei *klasa wewnętrzna* oznacza w terminologii języka Java klasę, zdefiniowaną wewnątrz innej klasy. Klasy wewnętrzne zostały spopularyzowane w momencie wprowadzenia JDK 1.1, gdzie były wykorzystywane jako procedury obsługi zdarzeń w aplikacjach o graficznym interfejsie użytkownika (patrz receptura 13.4), jednak możliwości ich zastosowania są znacznie szersze.

W rzeczywistości klasy wewnętrzne można tworzyć w kilku różnych sytuacjach. Jeśli klasa wewnętrzna zostanie zdefiniowana jako członek pewnej klasy, to obiekty tej klasy wewnętrznej będzie można tworzyć w dowolnym miejscu klasy, wewnątrz której została ona zdefiniowana. Z kolei, jeśli klasa wewnętrzna zostanie zdefiniowana w metodzie, to będzie się można do niej odwoływać wyłącznie w tej metodzie. Klasy wewnętrzne mogą posiadać nazwy lub być klasami anonimowymi. Nazwane klasy wewnętrzne posiadają pełne nazwy, których postać jest zależna od kompilatora; standardowa wirtualna maszyna Javy umieszcza takie klasy w plikach o nazwach `KlasaGlowna$KlasaWewnetrzna.class`. Nazwy anonimowych klas wewnętrznych także zależą od używanego kompilatora; standardowa wirtualna maszyna Javy po kompilacji umieszcza ich kod w plikach o nazwach `KlasaGlowna$1.class`, `KlasaGlowna$2.class` i tak dalej.

Obiektów klas wewnętrznych nie można tworzyć w żadnych innych kontekstach; każda próba jawnego odwołania się do takiej klasy, na przykład do klasy `InnaKasaGlowna$KlasaWewnetrzna`, zostanie wykryta w czasie kompilacji i spowoduje zgłoszenie błędu.

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

public class AllClasses {
    /** Klasa wewnętrzna, której można używać w dowolnym
```

```
         * miejscu tego pliku.
         */
        public class Data {
            int x;
            int y;
        }
        public void getResults() {
            JButton b = new JButton("Kliknij mnie");
            b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent evt) {
                    System.out.println("Dziękuję za naciśnięcie");
                }
            });
        }
    }

    /** Klasy umieszczone w tym samym pliku co AllClasses, które
     * jednak mogą być używane także w innych kontekstach
     * (powodują jednak wygenerowanie ostrzeżenia).
     */
    class AnotherClass {
        // Metody i pola ...
    }
```

8.7. Tworzenie metod zwrotnych przy wykorzystaniu interfejsów

Problem

Chcielibyśmy stworzyć *metodę zwrotną*, czyli umożliwić innym klasom wywoływanie wskazanego fragmentu kodu.

Rozwiązanie

Jednym z potencjalnych rozwiązań jest wykorzystanie interfejsów.

Analiza

Interfejs to obiekt podobny do klasy, który może zawierać wyłącznie metody abstrakcyjne oraz pola sfinalizowane. Jak można się przekonać, interfejsy są wykorzystywane bardzo często. Poniżej przedstawione zostały często wykorzystywane interfejsy dostępne w standardowym API Javy:

- `Runnable`, `Comparable` oraz `Clonable` (dostępne w pakiecie `java.lang`).
- `List`, `Set`, `Map` oraz `Enumeration`/`Iterator` (dostępne w szkielecie kolekcji, patrz rozdział 7.).
- `ActionListener`, `WindowListener` i inne (dostępne w AWT — grupie pakietów służących do tworzenia aplikacji o graficznym interfejsie użytkownika, patrz receptura 13.4).

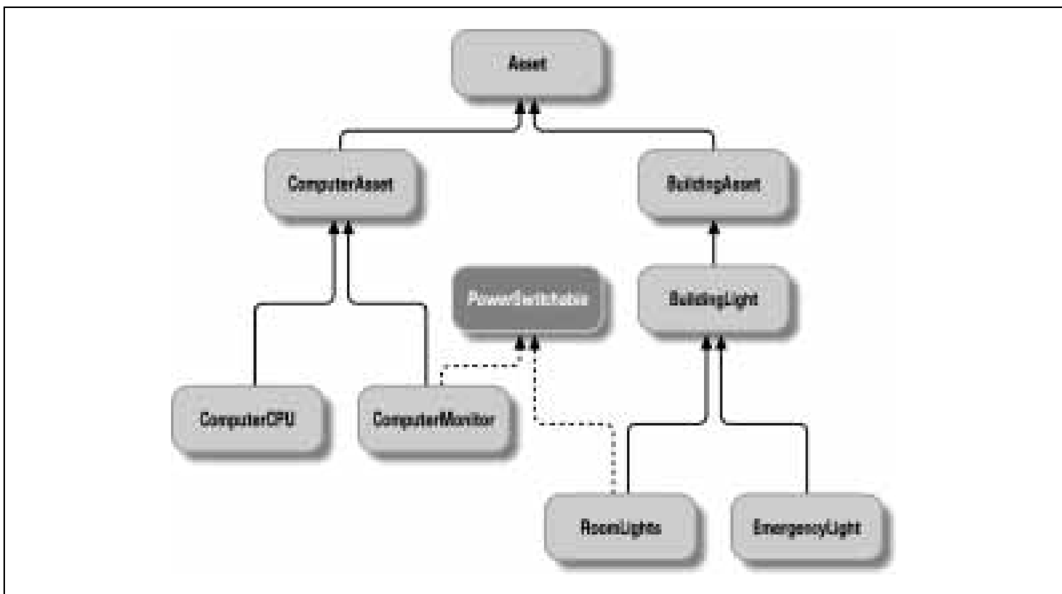
- `Driver`, `Connection`, `Statement` oraz `ResultSet` (dostępne w JDBC, patrz receptura 20.3).
- „Interfejs zdalny" — połączenie pomiędzy klientem i serwerem — został zdefiniowany jako `Interface` (w technologiach RMI, CORBA oraz EJB).

Klasa potomna, klasa abstrakcyjna czy interfejs?

Zazwyczaj każdy problem można rozwiązać na kilka sposobów. Niektóre z nich można rozwiązać dzięki tworzeniu klas potomnych, klas abstrakcyjnych lub też interfejsów. W określeniu optymalnego sposobu rozwiązania mogą pomóc następujące wytyczne:

- *Klasy abstrakcyjnej* można użyć w przypadkach, gdy chcemy stworzyć szablon dla grupy klas potomnych, z których wszystkie mogą dziedziczyć część możliwości funkcjonalnych po klasie nadrzędnej, lecz jednocześnie część tych możliwości muszą samodzielnie zaimplementować. (Każda z klas potomnych „geometrycznej" klasy `Shapes` musi dostarczyć metodę `computeArea()`, a ponieważ sama klasa `Shapes` nie jest w stanie obliczyć swego obszaru, będzie to metoda abstrakcyjna). Zagadnienie to zostało przedstawione w recepturze 8.8).
- *Klasy potomne* należy tworzyć za każdym razem, gdy chcemy rozszerzyć klasę i dodać do niej jakieś możliwości funkcjonalne, niezależnie do tego, czy dana klasa jest klasą abstrakcyjną czy też nie. Rozwiązania tego typu można znaleźć w standardowym API Javy oraz w wielu przykładach przedstawionych w niniejszej książce (między innymi, w przykładach przedstawionych w recepturach 1.13, 5.10, 8.11 oraz 9.7).
- *Klasy potomne* należy także tworzyć w przypadkach, gdy musimy rozszerzyć daną klasę. Aplety (patrz receptura 17.2), serwlety (patrz receptura 18.1) i nie tylko, wykorzystują to rozwiązanie w celu zagwarantowania, że klasy ładowane dynamicznie na pewno będą dysponować pewnymi „bazowymi" możliwościami funkcjonalnymi (patrz receptura 25.3).
- *Interfejsy należy definiować* w sytuacjach, gdy nie ma wspólnej klasy nadrzędnej dysponującej potrzebnymi możliwościami funkcjonalnymi, a potrzebne możliwości mają być dostępne jedynie w pewnych, nie związanych ze sobą, klasach (patrz interfejs `PowerSwitchable` przedstawiony w recepturze 8.7).
- *Interfejsów można także używać jako „znaczników"* przekazujących pewne informacje o danej klasie. Taką rolę pełnią właśnie interfejsy `Cloneable` (patrz receptura 8.4) oraz `Serializable` (patrz receptura 9.16) dostępne w standardowym API Javy.

Załóżmy, że generujemy futurystyczny system do zarządzania budynkiem. Aby zmniejszyć wykorzystanie energii elektrycznej, chcielibyśmy mieć możliwość zdalnego wyłączania (w nocy lub na weekendy) różnego typu urządzeń, które zużywają dużo energii, na przykład monitorów lub oświetlenia pomieszczeń. Załóżmy, że dysponujemy jakąś technologią pozwalającą na „zdalne sterowanie" może to być komercyjna wersja technologii X10 firmy BSR, technologia BlueTooth, 802.11 lub jakakolwiek inna. Stosowana technologia nie ma w tym przypadku żadnego znaczenia, najważniejsze jest to, że musimy zachować najwyższą uwagę podczas określania, jakie urządzenia należy wyłączyć. Automatyczne wyłączanie komputerów mogłoby doprowadzić do niezadowolenia ich użytkowników, gdyż często się zdarza, że celowo nie są one wyłączane na noc. Z kolei wyłączenie oświetlenia awaryjnego w budynku, zawsze stanowiłoby zagrożenie bezpieczeństwa publicznego[2]. Dlatego też stworzyliśmy hierarchię klas przedstawioną na rysunku 8.1.



Rysunek 8.1. Klasy używane przy tworzeniu systemu zarządzania budynkiem

Sam kod tych klas nie został tu przedstawiony (w rzeczywistości jest on całkiem prosty), lecz można go znaleźć w przykładach dołączonych do niniejszej książki. Klasy umieszczone na wyższych poziomach hierarchii — których nazwy kończą się słowem `Asset` oraz klasa `BuildingLight` — to klasy abstrakcyjne. Nie można stworzyć obiektów tych klas, gdyż nie dysponują one żadnymi konkretnymi możliwościami funkcjonalnymi. Aby zagwarantować, i to zarówno w czasie kompilacji programu, jak i jego wykonywania, że wyłączenie oświetlenia awaryjnego budynku nie będzie możliwe, wystarczy upewnić się, że klasa reprezentująca oświetlenie awaryjne — `EmergencyLight` — nie implementuje interfejsu `PowerSwitchable`.

---

[2] Oczywiście oświetlenie awaryjne nigdy nie byłoby podłączane do systemu umożliwiającego zdalne wyłączanie zasilania. Jednak komputery można by podłączyć do takiego systemu ze względów eksploatacyjnych.

Warto zauważyć, że w tym przypadku nie można bezpośrednio wykorzystać mechanizmu dziedziczenia, gdyż nie ma żadnej wspólnej klasy bazowej dla klas `ComputerMonitor` oraz `RoomLight`, która jednocześnie nie byłaby klasą bazową dla klas `ComputerCPU` oraz `EmergencyLight`. A zatem, interfejsów można używać, aby udostępnić te same możliwości funkcjonalne w klasach, które nie są ze sobą w żaden sposób powiązane.

Sposób wykorzystania interfejsu `PowerSwitchable` został przedstawiony w klasie `BuildingManagement`. Klasa ta nie należy do hierarchii przedstawionej na rysunku 8.1, lecz wykorzystuje kolekcję (a w zasadzie to tablicę, aby uprościć kod prezentowanego przykładu) obiektów `Asset`.

Niemniej jednak informacje o urządzeniach, których nie można zdalnie wyłączać, muszą być przechowywane w bazie danych (z różnych powodów, takich jak kontrole, ubezpieczenia i tak dalej). Metoda wyłączająca zasilanie jest ostrożna i sprawdza, czy poszczególne obiekty przechowywane w bazie danych implementują interfejs `PowerSwitchable`. Jeśli tak, to dany obiekt jest rzutowany do tego typu, dzięki czemu można wywołać metodę `powerDown()`. Z kolei w przeciwnym przypadku obiekt jest pomijany, co zapobiega możliwości wyłączenia zasilania oświetlenia awaryjnego lub komputerów z uruchomionym oprogramowaniem Seti@Home, zajętych pobieraniem plików przy użyciu Napstera bądź tworzących kopię bezpieczeństwa danych.

```
/**
 * BuildingManagement - zarządzanie budynkiem oszczędzającym energię.
 * Ta klasa pokazuje, w jaki sposób można zarządzać urządzeniami
 * w biurze, które z nich można bezpiecznie wyłączyć na noc w celu
 * zaoszczędzenia energii elektrycznej - znacznych ilości energii
 * w przypadku dużych biur.
 */
public class BuildingManagement {

    Asset things[] = new Asset[24];
    int numItems = 0;

    /** goodNight wywoływana przez wątek czasomierza o godzinie
     * 22:00 lub gdy system otrzyma polecenie "shutdown" od
     * strażnika.
     */
    public void goodNight() {
        for (int i=0; i<things.length; i++)
            if (things[i] instanceof PowerSwitchable)
                ((PowerSwitchable)things[i]).powerDown();
    }

    // Metoda goodMorning() działałaby bardzo podobnie z tą
    // różnica, że wywoływałaby metodę powerUp().

    /** Dodaj obiekt Asset do tego budynku. */
    public void add(Asset thing) {
        System.out.println("Dodajemy " + thing);
        things[numItems++] = thing;
    }

    /** Program główny. */
    public static void main(String[] av) {
        BuildingManagement b1 = new BuildingManagement();
```

```
            b1.add(new RoomLights(101));    // Oświetlenie w pokoju 101.
            b1.add(new EmergencyLight(101));    // Oświetlenie awaryjne.
            // Dodaj komputer na biurku#4 w pokoju 101.
            b1.add(new ComputerCPU(10104));
            // Dodaj monitor tego komputera.
            b1.add(new ComputerMonitor(10104));

            // Czas mija... słońce zachodzi...
            b1.goodNight();
        }
    }
```

Po uruchomieniu powyższy program wyświetla listę wszystkich dodawanych urządzeń, jednak wyłącznie urządzenia „implementujące" interfejs `PowerSwitchable` są wyłączane:

```
>java BuildingManagement
Dodajemy RoomLights@affc70
Dodajemy EmergencyLight@e63e3d
Dodajemy ComputerCPU@4901
Dodajemy ComputerMonitor@b90b39
Wyłączamy światła w pokoju 101
Wyłączamy monitor na biurku 10104
>
```

8.8. Polimorfizm i metody abstrakcyjne

Problem

Chcemy, aby wszystkie klasy potomne dysponowały własnymi wersjami pewnej metody.

Rozwiązanie

W klasie nadrzędnej należy zdefiniować metodę jako abstrakcyjną, w ten sposób kompilator wymusi, aby każda klasa potomna implementowała tę metodę.

Analiza

W hipotetycznym programie graficznym wszystkie rysowane kształty są reprezentowane przez obiekty klasy `Shape`. Klasa ta posiada abstrakcyjną metodę `computeArea()`, która oblicza dokładną powierzchnię rysowanego kształtu:

```
public class Rectangle extends Shape {
    protected int x, y;
    public abstract double computeArea();
}
```

Na przykład, klasa potomna `Rectangle` posiada metodę `computeArea()`, która mnoży wysokość i szerokość prostokąta i zwraca obliczoną wartość:

```
public class Rectangle extends Shape {
    double width, height;
    public double computeArea() {
        return width * height;
    }
}
```

Z kolei klasa `Circle` zwraca iloczyn $\Pi * r$:

```
public class Circle extends Shape {
    double radius;
    public double computeArea() {
        return Math.PI * radius * radius;
    }
}
```

Taki system zapewnia niezwykle wysoki poziom ogólności. W programie głównym można przekazać kolekcję obiektów `Shape` i (w tym miejscu uwidacznia się piękno całego rozwiązania) wywołać metodę `computeArea()` każdego z nich bez zwracania uwagi na faktyczną klasę danego obiektu. Sposób obsługi metod polimorficznych w Javie zapewni, że automatycznie zostanie wywołana metoda `computeArea()` faktycznej klasy danego obiektu, czyli klasy użytej podczas jego tworzenia.

```
import java.util.*;

/** Część programu głównego wykorzystującego obiekty
 * klasy Shape.
 */
public class Main {

    Collection allShapes;   // Kolekcja tworzona w konstruktorze,
                            // który nie jest przedstawiony w tym przykładzie.

    /** Odwołuje się kolejno do wszystkich obiektów Shape
     * w kolekcji i oblicza ich pola.
     */
    public double totalAreas() {
        Iterator it = allShapes.iterator();
        double total = 0.0;
        while (it.hasNext()) {
            Shape s = (Shape)it.next();
            total += s.computeArea();
        }
        return total;
    }
}
```

Możliwości te są niezwykle przydatne z punktu widzenia utrzymania i rozwijania programów, gdyż w przypadku dodawania nowych klas potomnych kod programu głównego nie musi być w żaden sposób modyfikowany. Co więcej, cały kod związany w obsługą konkretnych figur, na przykład wielokątów, jest umieszczany w jednym miejscu — pliku źródłowym klasy `Polygon` (ang. *wielokąt*). To wielkie usprawnienie w porównaniu z wcześniej stosowanymi językami programowania, w których pola struktur lub rekordów określające typ były używane w instrukcjach wyboru lub złożonych konstrukcjach warunkowych, umieszczanych w różnych miejscach programu. Dzięki wykorzystaniu polimorfizmu oprogramowanie pisane w języku Java jest bardziej niezawodne i łatwiejsze do utrzymania.

8.9. Przekazywanie wartości

Problem

Musimy przekazać do metody wartość typu podstawowego, na przykład `int`, i pobrać z metody nie tylko zwracany przez nią wynik, lecz także zmodyfikowaną wartość, która została do niej przekazana.

Opisana sytuacja często występuje w przypadku przetwarzania łańcuchów znaków, gdy metoda musi zwrócić, na przykład, wartość logiczną lub ilość przekazanych znaków, a jednocześnie powiększyć indeks tablicy lub łańcucha znaków przechowywany w klasie wywołującej tę metodę.

Możliwości takie mogą się także przydać w konstruktorach, które nie zwracają żadnych wartości, lecz powinny przekazywać informacje o „wykorzystaniu" lub przetworzeniu pewnej ilości znaków. Przykładowo, informacje takie mogą być konieczne w sytuacjach, gdy łańcuch znaków jest wstępnie przetwarzany w konstruktorze, a kolejne etapy przetwarzania będą realizowane przez metody wywoływane w dalszych fragmentach programu.

Rozwiązanie

Należy wykorzystać wyspecjalizowaną klasę, taką jak przedstawiona w dalszej części receptury.

Analiza

`Integer` to jedna z predefiniowanych klas potomnych klasy `Number`, o których wspominałem we Wstępie oraz w pierwszych pięciu rozdziałach niniejszej książki. Jej celem jest reprezentacja wartości typu `int`, a dodatkowo zawiera także metody statyczne służące do zamiany łańcuchów znaków na liczby oraz formatowania wyświetlanych liczb całkowitych.

Klasa ta jest bardzo dobra, jednak nam wystarczyłoby coś prostszego.

Poniżej przedstawiłem stworzoną przeze mnie klasę `MutableInteger`, która przypomina nieco klasę `Integer`, lecz jest bardziej wyspecjalizowana, gdyż pomija wszelkie niepotrzebne nam możliwości funkcjonalne. Klasa `MutableInteger` implementuje wyłącznie metody `set`, `get` oraz przeciążoną metodę `incr`, którą można wywołać bez podawania argumentu (w tym przypadku metoda ta odpowiada operatorowi ++) lub podając argument liczbowy (w tym przypadku przekazana wartość jest dodawana do wartości zapisanej w obiekcie, co sprawia, że ta wersja metody odpowiada operatorowi +=). Język Java nie daje możliwości przeciążania operatorów, a zatem klasa wywołująca musi użyć odpowiedniej metody, zamiast skorzystać z odpowiedniego operatora. W aplikacjach,

w których potrzebna jest możliwość przekazywania wartości do i z metod, korzyści, jakie daje użycie tej klasy, są znacznie cenniejsze niż drobne ograniczenia syntaktyczne związane z koniecznością posługiwania się metodami. W pierwszej kolejności przeanalizujmy przykład wykorzystania klasy `MutableInteger`. Załóżmy, że musimy wywołać metodę skanującą, na przykład o nazwie `parse()`, która zwraca wartość logiczną określającą, czy liczba została odnaleziona oraz wartość całkowitą określającą położenie odnalezionej liczby:

```
import com.darwinsys.util.*;

/** Prezentacja wykorzystania klasy MutableInteger
 * w celu przekazywania dodatkowej wartości, oprócz
 * właściwej wartości wynikowej zwracanej przez metodę.
 */
public class StringParse {
    /** Oto metoda, która zwraca wartość logiczną, a jednocześnie
     * przekazuje wartość całkowitą określającą położenie
     * w łańcuchu znaków, gdzie odnaleziono poszukiwaną liczbę.
     */
    public static boolean parse(String in, char lookFor, MutableInteger
    whereFound) {
        int i = in.indexOf(lookFor);
        if (i == -1)
            return false;    // Nie znaleziono.
        whereFound.setValue(i);    // Zwróć miejsce, gdzie znaleziono.
        return true;        // Poinformuj, że znaleziono.
    }

    public static void main(String[] args) {
        MutableInteger mi = new MutableInteger();
        String text = "Witaj Świecie";
        char c = 'Ś';
        if (parse(text, c, mi)) {
            System.out.println("Litera " + c + " została odnaleziona na
            pozycji " + mi + " w łańcuchu " + text);
        } else {
            System.out.println("Nie odnaleziono litery.");
        }
    }
}
```

Wielu „purystów obiektowych" mogłoby stwierdzić — i co więcej, mieliby rację — że nie należy stosować takich rozwiązań. Zawsze można by bowiem napisać metodę w taki sposób, aby była z niej zwracana tylko jedna wartość (w tym przypadku byłaby to wartość określająca położenie odnalezionego fragmentu łańcucha lub wartość `-1`, która oznaczałaby, że poszukiwany fragment nie został odnaleziony) lub stworzyć klasę pomocniczą zawierającą obie zwracane informacje — liczbę całkowitą oraz wartość logiczną. Jednak w standardowym API Javy można znaleźć precedens — przedstawiony przykład w dużym stopniu przypomina sposób wykorzystania klasy `ParsePosition` (przedstawiony w recepturze 6.5). W każdym razie, omawiane tu możliwości funkcjonalne są tak często przydatne i wykorzystywane, że doszedłem do wniosku, że ich przedstawienie jest w pełni usprawiedliwione. Jednocześnie informuję, że w tworzonych programach należy unikać stosowania rozwiązań tego typu!

Po tych wszystkich wyjaśnieniach, mogę w końcu przedstawić kod klasy `MutableInteger`:

```
package com.darwinsys.util;

/** Klasa MutableInteger jest podobna do klasy Integer,
 * lecz daje możliwość zmieniania przechowywanych liczb
 * całkowitych w celu uniknięcia zbyt częstego
 * tworzenia obiektów związanych z wykonywaniem operacji
 * typu:
 * c = new Integer(c.getInt()+1),
 * które mogą być bardzo kosztowne, jeśli będą wykonywane
 * zbyt często.
 * Nie jest to klasa potomna klasy Integer, gdyż Integer
 * jest klasą sfinalizowaną (dla uzyskania wysokiej
 * efektywności działania:-))
 */
public class MutableInteger {
    private int value = 0;

    public MutableInteger() {
    }

    public MutableInteger(int i) {
        value = i;
    }

    public void incr() {
        value++;
    }

    public void incr(int amt) {
        value += amt;
    }

    public void decr() {
        value--;
    }

    public void setValue(int i) {
        value = i;
    }

    public int getValue() {
        return value;
    }

    public String toString() {
        return Integer.toString(value);
    }

    public static String toString(int val) {
        return Integer.toString(val);
    }

    public static int parseInt(String str) {
        return Integer.parseInt(str);
    }
}
```

Patrz także

Jak już wspominałem, zamiast klasy `MutableInteger` można posłużyć się klasą `ParsePosition`. Niemniej jednak klasy `MutableInteger` można z powodzeniem użyć także w innych sytuacjach, na przykład doskonale nadaje się ona do stworzenia liczników wykorzystywanych w serwletach.

8.10. Zgłaszanie własnych wyjątków

Problem

Chcielibyśmy stworzyć i wykorzystać jedną lub kilka klas reprezentujących wyjątki, charakterystycznych dla danej aplikacji.

Rozwiązanie

Należy zdefiniować klasy potomne klas `Exception` lub `RuntimeException`.

Analiza

Teoretycznie rzecz biorąc, można by stworzyć klasę potomną klasy `Throwable`, niemniej jednak rozwiązanie takie jest uważane za nieodpowiednie. Zazwyczaj, tworząc własne wyjątki, definiuje się klasy potomne klasy `Exception` (jeśli mają to być wyjątki sprawdzane) lub klasy `RuntimeException` (jeśli mają to być wyjątki niesprawdzane). Wyjątki sprawdzane to takie, które programiści tworzący aplikację muszą przechwytywać i obsługiwać lub „przekazać dalej", podając je w klauzuli `throws` w definicji metody.

W przypadku tworzenia klas potomnych klas `Exception` oraz `RuntimeException` przyjęło się implementować przynajmniej dwa konstruktory — bezargumentowy oraz umożliwiający przekazanie jednego łańcucha znaków:

```
/** A ChessMoveException jest zgłaszany, gdy użytkownik
 * wykona nieprawidłowy ruch.
 */
public class ChessMoveException extends Exception {
    public ChessMoveException () {
        super();
    }
    public ChessMoveException (String msg) {
        super(msg);
    }
}
```

Patrz także

W dokumentacji klasy `Exception` podano bardzo wiele przykładów jej klas potomnych; przed tworzeniem własnych klas wyjątków warto tam zajrzeć i sprawdzić, czy już nie ma klasy, której można by użyć.

8.11. Program Plotter

Program przedstawiony w tej recepturze nie jest złożony, a wręcz przeciwnie — jest prosty i właśnie dlatego posłuży nam jako przykład prezentujący niektóre zagadnienia przedstawione w tym rozdziale oraz będzie punktem wyjściowym do dalszych dyskusji. Przedstawiona klasa opisuje grupę starych (używanych w latach 70-tych i 80-tych) ploterów piórowych. Tych, którzy nigdy nie widzieli takiego urządzenia informuję, że ploter piórowy to urządzenie, które przesuwa specjalne pióro nad kartką papieru, rysując na niej zadane kształty. Taki ploter jest w stanie podnieść pióro, opuścić je na papier, rysować odcinki proste, litery i tak dalej. Przed pojawieniem się drukarek laserowych i atramentowych plotery piórowe były najpopularniejszym sposobem przygotowywania wszelkiego typu wykresów oraz slajdów używanych w prezentacjach (cóż, było to na długo przed pojawieniem się takich programów jak Harvard Presents oraz Microsoft Power Point). Aktualnie tylko kilka firm wciąż produkuje plotery piórowe, a jednak zdecydowałem się podać je jako przykład, gdyż są one na tyle proste, że łatwo można zrozumieć zasady ich działania nawet na podstawie krótkiego opisu.

Poniżej przedstawiłem klasę wysokiego poziomu, która w abstrakcyjny sposób opisuje kluczowe cechy ploterów produkowanych przez różne firmy. Można by jej użyć w programie analitycznym lub służącym do wyszukiwania danych w celu rysowania kolorowych wykresów prezentujących wzajemne związki pomiędzy danymi. Jednak nie chcę, aby mój program główny musiał zwracać uwagę na wszelkie szczegóły związane obsługą poszczególnych typów ploterów i z tego powodu klasa `Plotter` przedstawiona poniżej, jest klasą abstrakcyjną.

```
/**
 * Klasa abstrakcyjna Plotter. Należy tworzyć jej klasy
 * potomne obsługujące różnego typu plotery:
 * X, DOS, Penman, HP i tak dalej.
 *
 * Układ współrzędnych: X = 0 z lewej strony, wartości współrzędnych rosną
 * ku prawej; Y = 0 u góry, wartości współrzędnych rosną ku dołowi rysunku
 * (tak samo jak w AWT).
 */
public abstract class Plotter {
    public final int MAXX = 800;
    public final int MAXY = 600;
    /** Bieżąca współrzędna X (ten sam sposób wykorzystania jak w AWT!). */
    protected int curx;
    /** Current Y co-ordinate (ten sam sposób wykorzystania jak w AWT!). */
    protected int cury;
    /** Bieżący stan: u góry lub na dole. */
    protected boolean penIsUp;
    /** Aktualnie używany kolor. */
    protected int penColor;

    Plotter() {
        penIsUp = true;
        curx = 0; cury = 0;
    }
    abstract void rmoveTo(int incrx, int incry);
    abstract void moveTo(int absx, int absy);
```

```
        abstract void penUp();
        abstract void penDown();
        abstract void penColor(int c);

        abstract void setFont(String fName, int fSize);
        abstract void drawString(String s);

        /* Metody nieabstrakcyjne. */

        /** Rysujemy prostokąt o szerokości w i wysokości h. */
        public void drawBox(int w, int h) {
            penDown();
            rmoveTo(w, 0);
            rmoveTo(0, h);
            rmoveTo(-w, 0);
            rmoveTo(0, -h);
            penUp();
        }

        /** Rysujemy prostokąt na podstawie obiektu Dimension
         * określającego wymiary prostokąta. Klasa ta jest dostępna
         * w bibliotece AWT.
         */
        public void drawBox(java.awt.Dimension d) {
            drawBox(d.width, d.height);
        }

        /** Rysujemy prostokąt na podstawie obiektu Rectangle
         * określającego wymiary prostokąta. Klasa ta jest dostępna
         * w bibliotece AWT.
         */
        public void drawBox(java.awt.Rectangle r) {
            moveTo(r.x, r.y);
            drawBox(r.width, r.height);
        }
    }
```

W powyższej klasie warto zwrócić uwagę na sporą grupę metod abstrakcyjnych. Metody odpowiadające za poruszanie piórem, jego kontrolę oraz rysowanie zostały zdefiniowane jako metody abstrakcyjne, ze względu na możliwość realizacji tych czynności na wiele różnych sposobów. Jednak metoda rysująca prostokąt (`drawBox()`) posiada domyślną implementację, która powoduje opuszczenie pióra w ostatnim wybranym położeniu, narysowanie czterech krawędzi prostokąta i podniesienie pióra. Klasy potomne obsługujące bardziej „inteligentne" plotery zapewne przesłonią tę metodę, jednak klasy reprezentujące mniej zaawansowane urządzenia mogą korzystać z tej domyślnej implementacji. Dostępne są także dwie przeciążone wersje metody `drawBox()`, z których można skorzystać, gdy wymiary prostokąta są określone za pomocą obiektu `Dimension` lub gdy jego położenie i wymiary opisuje obiekt `Rectangle`.

Poniższy prosty program przedstawia sposób wykorzystania klas potomnych klasy `Plotter`. Metoda `Class.forName()` wywoływana na początku programu głównego zostanie dokładniej opisana w recepturze 25.3. Jak na razie wystarczy, abyś wiedział, że powoduje ona utworzenie obiektu podanej klasy, który w naszym przypadku jest zapisywany w zmiennej `r` i używany do sporządzenia rysunku:

```
/** Program główny, "sterownik" dla klasy Plotter.
 * Program symuluje większe aplikacje graficzne, takie jak GnuPlot.
 */
public class PlotDriver {

    /** Tworzymy obiekt (sterownik) Plotter i sprawdzamy, jak działa. */
    public static void main(String[] argv)
    {
        Plotter r ;
        if (argv.length != 1) {
            System.err.println("Sposób użycia: PlotDriver driverclass");
            return;
        }
        try {
            Class c = Class.forName(argv[0]);
            Object o = c.newInstance();
            if (!(o instanceof Plotter))
                throw new ClassNotFoundException("To nie jest instanceof
                Plotter.");
            r = (Plotter)o;
        } catch (ClassNotFoundException e) {
            System.err.println("Przykro mi, "+argv[0]+" nie jest klasą
            reprezentującą ploter.");
            return;
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
        r.penDown();
        r.penColor(1);
        r.moveTo(200, 200);
        r.penColor(2);
        r.drawBox(123, 200);
        r.rmoveTo(10, 20);
        r.penColor(3);
        r.drawBox(123, 200);
        r.penUp();
        r.moveTo(300, 100);
        r.penDown();
        r.setFont("Helvetica", 14);
        r.drawString("Witaj Świecie");
        r.penColor(4);
        r.drawBox(10, 10);
    }
}
```

W kilku dalszych rozdziałach przedstawię kolejne przykłady wykorzystujące klasę `Plotter` oraz inne klasy z nią związane.