helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

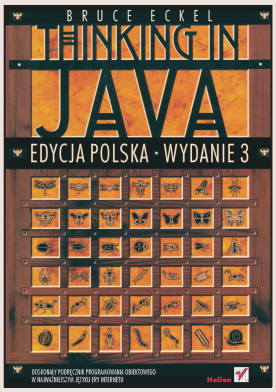
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Thinking in Java. Wydanie 3. Edycja polska



Autor: Bruce Eckel
Tłumaczenie: Adrian Nowak, Szymon Kobalczyk, Łukasz Fryz
ISBN: 83-7361-133-9
Format: B5, stron: 912

Książka „Thinking in Java" wzbudziła entuzjazm i uznanie programistów na całym świecie przede wszystkim dzięki swej wyjątkowej przejrzystości, przemyślanej i uważnej organizacji oraz krótkim, trafnie dobranym przykładom. Książka obejmuje szeroki zakres tematów – zaczynając od podstaw składni Javy, a kończąc na najbardziej zaawansowanych zagadnieniach (dokładny opis pojęć obiektowych, wielowątkowości, zautomatyzowane konstruowanie, testowanie modułów oraz wykrywanie i usuwanie błędów).

- Wprowadzenie do języka Java uaktualnione o prezentację nowych możliwości wprowadzonych w JDK 1.4
- Nowy program testowy weryfikujący poprawność działania każdego z programów i prezentujący generowane przez niego wyniki
- Nowy rozdział poświęcony testowaniu modułów, zautomatyzowanemu konstruowaniu programów, asercjom, rejestracji oraz innym sposobom zapewniania poprawności programów
- Całkowicie zmodyfikowany rozdział poświęcony wielowątkowości pozwalający dobrze poznać podstawy tego zagadnienia
- Ponad 350 działających przykładów uaktualnionych z myślą o niniejszym wydaniu książki; ponad 15 tysięcy wierszy kodu
- Książka zarówno dla początkujących, jak i ekspertów
- Uczy języka Java, a nie rozwiązań zależnych od platformy systemowej
- Szczegółowo omawia podstawowe zagadnienia, nie pomija bardziej zaawansowanych
- Dogłębnie wyjaśnia zasady obiektowości oraz ich zastosowanie w Javie

Bruce Eckel jest prezesem MindView, Inc., firmy prowadzącej zarówno otwarte jak i zamknięte kursy treningowe; zajmującej się też doradztwem, nadzorem i kontrolą nad projektami związanymi z technologiami obiektowymi i wzorcami projektowymi.
Jest autorem książki „Thinking in C++" oraz współautorem książek „Thinking in C#" oraz „Thinking in C++, Volume 2", napisał także kilka innych książek i ponad 150 artykułów. Od ponad 20 lat prowadzi wykłady i seminaria na całym świecie. Był członkiem Komitetu Standardów C++. Zdobył tytuł naukowy w dziedzinie fizyki stosowanej i inżynierii oprogramowania.

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

Spis treści

Podziękowania **17**

Przedmowa **21**
Przedmowa do wydania trzeciego 23
Java 2, JDK 1.4 25

Wprowadzenie **27**
Warunki wstępne 27
Nauka Javy 28
Cele 29
Dokumentacja online 29
Zawartość rozdziałów 30
Ćwiczenia 35
Kody źródłowe 35
Konwencje zapisu 37
Wersje Javy 37
Błędy 37
Projekt okładki 38

Rozdział 1. Wprowadzenie w świat obiektów **39**
Postępująca abstrakcja 40
Obiekt posiada interfejs 41
Obiekt dostarcza usług 44
Ukrywanie implementacji 45
Wielokrotne wykorzystanie implementacji 46
Dziedziczenie: wielokrotne użycie interfejsu 47
„Bycie czymś" a „bycie podobnym do czegoś" 50
Wymienialność obiektów z użyciem polimorfizmu 51
Abstrakcyjne klasy bazowe i interfejsy 54
Tworzenie obiektów, sposób ich stosowania i czas życia 55
Kolekcje i iteratory 56
Hierarchia z pojedynczym korzeniem 57
Rzutowanie w dół kontra wzorce (generalia) 58
Zapewnienie odpowiedniego „sprzątania" 59
Obsługa wyjątków — eliminowanie błędów 60
Współbieżność 61
Trwałość 62
Java i Internet 63
Czym jest Internet? 63
Programowanie po stronie klienta 64

Programowanie po stronie serwera....70
Aplikacje....71
Dlaczego Java odnosi sukcesy....71
Systemy jest łatwiej opisać i zrozumieć....71
Maksymalne zwiększenie wydajności dzięki bibliotekom....72
Obsługa błędów....72
Programowanie na wielką skalę....72
Java kontra C++....73
Podsumowanie....74

Rozdział 2. Wszystko jest obiektem....75
Dostęp do obiektów poprzez referencje....75
Wszystkie obiekty trzeba stworzyć....76
Gdzie przechowujemy dane....77
Przypadek specjalny: typy podstawowe....78
Tablice w Javie....79
Nigdy nie ma potrzeby niszczenia obiektu....80
Zasięg....80
Zasięg obiektów....81
Tworzenie nowych typów danych — klasa....82
Pola i metody....82
Metody, argumenty i wartości zwracane....84
Lista argumentów....85
Tworzenie programu w Javie....86
Widoczność nazw....86
Wykorzystanie innych komponentów....86
Słowo kluczowe static....87
Twój pierwszy program w Javie....89
Kompilacja i uruchomienie....90
Komentarze oraz dokumentowanie kodu....91
Dokumentacja w komentarzach....91
Składnia....92
Osadzony HTML....93
Niektóre znaczniki dokumentacyjne....93
Przykład dokumentowania kodu....96
Styl programowania....96
Podsumowanie....97
Ćwiczenia....97

Rozdział 3. Sterowanie kolejnością wykonania....99
Używanie operatorów Javy....99
Kolejność....100
Przypisanie....100
Operatory matematyczne....103
Operatory zwiększania i zmniejszania....106
Operatory relacji....107
Operatory logiczne....109
Operatory bitowe....111
Operatory przesunięcia....112
Operator trójargumentowy if-else....115
Przecinek....116
Łańcuchowy operator +....116
Najczęstsze pułapki przy używaniu operatorów....117
Operatory rzutowania....117
W Javie nie ma „sizeof”....120

Powtórka z kolejności operatorów....120
Kompendium operatorów....121
Sterowanie wykonaniem....129
Prawda i fałsz....129
if-else....129
return....130
Iteracja....131
do-while....132
for....132
break i continue....134
switch....139
Podsumowanie....143
Ćwiczenia....143

Rozdział 4. Inicjalizacja i sprzątanie....145
Gwarantowana inicjalizacja przez konstruktor....145
Przeciążanie metod....147
Rozróżnianie przeciążonych metod....150
Przeciążanie a typy podstawowe....150
Przeciążanie przez wartości zwracane....155
Konstruktory domyślne....155
Słowo kluczowe this....156
Sprzątanie: finalizacja i odśmiecanie pamięci....159
Do czego służy finalize()....160
Musisz przeprowadzić sprzątanie....161
Warunek zakończenia....162
Jak działa odśmiecacz pamięci....163
Inicjalizacja składowych....166
Określanie sposobu inicjalizacji....167
Inicjalizacja w konstruktorze....168
Inicjalizacja tablic....174
Tablice wielowymiarowe....179
Podsumowanie....182
Ćwiczenia....182

Rozdział 5. Ukrywanie implementacji....185
Pakiet — jednostka biblioteczna....186
Tworzenie unikatowych nazw pakietów....188
Własna biblioteka narzędziowa....191
Wykorzystanie instrukcji import do zmiany zachowania....192
Pułapka związana z pakietami....192
Modyfikatory dostępu w Javie....192
Dostęp pakietowy....193
public: dostęp do interfejsu....194
private: nie dotykać!....195
protected: dostęp „na potrzeby” dziedziczenia....196
Interfejs i implementacja....198
Dostęp do klas....199
Podsumowanie....202
Ćwiczenia....203

Rozdział 6. Wielokrotne wykorzystanie klas....205
Składnia kompozycji....205
Składnia dziedziczenia....208
Inicjalizacja klasy bazowej....210

Łączenie kompozycji i dziedziczenia 212
Zapewnienie poprawnego sprzątania....214
Ukrywanie nazw 217
Wybór między kompozycją a dziedziczeniem....218
protected....220
Przyrostowe tworzenie oprogramowania....221
Rzutowanie w górę....222
Dlaczego „w górę”....223
Słowo kluczowe final....224
Zmienne finalne....224
Metody finalne....227
Klasy finalne....230
Ostrożnie z deklaracją final 230
Inicjalizacja i ładowanie klas 231
Inicjalizacja w przypadku dziedziczenia 232
Podsumowanie 233
Ćwiczenia....233

Rozdział 7. Polimorfizm 237
Rzutowanie w górę raz jeszcze 237
Zapominanie o typie obiektu 239
Mały trik....240
Wiązanie wywołania metody....241
Uzyskiwanie poprawnego działania 241
Rozszerzalność 244
Pułapka: „przesłanianie” metod prywatnych....246
Klasy i metody abstrakcyjne....247
Konstruktory a polimorfizm....250
Kolejność wywołań konstruktorów 250
Dziedziczenie a sprzątanie....252
Zachowanie się metod polimorficznych wewnątrz konstruktorów 255
Projektowanie z użyciem dziedziczenia 257
Czyste dziedziczenie kontra rozszerzanie 259
Rzutowanie w dół a identyfikacja typu w czasie wykonania 260
Podsumowanie 262
Ćwiczenia....262

Rozdział 8. Interfejsy i klasy wewnętrzne 265
Interfejsy 265
„Wielokrotne dziedziczenie” w Javie....268
Rozszerzanie interfejsu poprzez dziedziczenie 271
Grupowanie stałych 272
Inicjalizacja pól interfejsów....274
Zagnieżdżanie interfejsów 275
Klasy wewnętrzne....277
Klasy wewnętrzne a rzutowanie w górę 279
Klasy wewnętrzne w metodach i zasięgach....280
Anonimowe klasy wewnętrzne....282
Połączenie z klasą zewnętrzną....286
Klasy zagnieżdżone 288
Odwoływanie się do obiektu klasy zewnętrznej....290
Sięganie na zewnątrz z klasy wielokrotnie zagnieżdżonej 290
Dziedziczenie po klasach wewnętrznych 291
Czy klasy wewnętrzne mogą być przesłaniane 292
Lokalne klasy wewnętrzne 293
Identyfikatory klas wewnętrznych....295

Dlaczego klasy wewnętrzne ... 296
Domknięcia i wywołania zwrotne ... 298
Klasy wewnętrzne a szkielety sterowania ... 300
Podsumowanie ... 306
Ćwiczenia ... 306

Rozdział 9. Obsługa błędów za pomocą wyjątków ... 309
Podstawy obsługi wyjątków ... 310
Parametry wyjątków ... 311
Przechwytywanie wyjątku ... 312
Blok try ... 312
Obsługa wyjątków ... 312
Tworzenie własnych wyjątków ... 314
Specyfikacja wyjątków ... 317
Przechwytywanie dowolnego wyjątku ... 318
Ponowne wyrzucanie wyjątków ... 320
Sekwencje wyjątków ... 323
Standardowe wyjątki Javy ... 326
Specjalny przypadek RuntimeException ... 326
Robienie porządków w finally ... 328
Do czego służy finally ... 329
Pułapka: zagubiony wyjątek ... 331
Ograniczenia wyjątków ... 332
Konstruktory ... 335
Dopasowywanie wyjątków ... 338
Rozwiązania alternatywne ... 339
Historia ... 341
Perspektywy ... 342
Przekazywanie wyjątków na konsolę ... 345
Zamiana wyjątków sprawdzanych na niesprawdzane ... 345
Wskazówki ... 347
Podsumowanie ... 348
Ćwiczenia ... 348

Rozdział 10. Wykrywanie typów ... 351
Potrzeba mechanizmu RTTI ... 351
Obiekt Class ... 354
Sprawdzanie przed rzutowaniem ... 356
Składnia RTTI ... 366
Refleksje — informacja o klasie w czasie wykonania ... 368
Ekstraktor metod klasowych ... 369
Podsumowanie ... 373
Ćwiczenia ... 374

Rozdział 11. Kolekcje obiektów ... 375
Tablice ... 375
Tablice to obiekty ... 376
Tablice jako wartości zwracane ... 379
Klasa Arrays ... 381
Wypełnianie tablicy ... 387
Kopiowanie tablic ... 389
Porównywanie tablic ... 390
Porównywanie elementów tablic ... 391
Sortowanie tablic ... 394
Przeszukiwanie tablicy posortowanej ... 395
Podsumowanie wiadomości o tablicach ... 397

Wprowadzenie do kontenerów....397
Wypisanie zawartości kontenerów....398
Wypełnianie kontenerów....400
Wada kontenera: nieznany typ....405
Czasami mimo wszystko działa....407
Tworzenie świadomej typu klasy ArrayList....408
Iteratory....410
Rodzaje kontenerów....413
Interfejs Collection....416
Interfejs List....418
Stos na podstawie LinkedList....422
Kolejka na podstawie LinkedList....423
Interfejs Set....423
SortedSet....426
Możliwości funkcjonalne interfejsu Map....427
SortedMap....431
LinkedHashMap....433
Haszowanie i kody haszujące....434
Przesłonięcie metody hashCode()....443
Przechowywanie referencji....446
WeakHashMap....449
Iteratory ponownie....450
Wybór implementacji....451
Wybór między listami....452
Wybór implementacji zbioru....454
Wybór implementacji odwzorowania....456
Sortowanie i przeszukiwanie list....459
Dodatkowe usługi....460
Niemodyfikowalne kontenery Collection i Map....462
Synchronizacja Collection i Map....463
Nieobsługiwane operacje....464
Kontenery Java 1.0 i 1.1....466
Vector i Enumeration....467
Hashtable....468
Stack....468
BitSet....469
Podsumowanie....470
Ćwiczenia....471

Rozdział 12. System wejścia-wyjścia w Javie....477

Klasa File....477
Wypisywanie zawartości katalogu....478
Operacje na katalogach....481
Wejście i wyjście....483
Typy InputStream....483
Typy OutputStream....484
Dodawanie atrybutów i użytecznych interfejsów....485
Czytanie z InputStream za pomocą FilterInputStream....486
Zapis do OutputStream za pomocą FilterOutputStream....486
Klasy Reader i Writer....488
Źródła i ujścia danych....489
Modyfikacja zachowania strumienia....489
Klasy niezmienione....490
Osobna i samodzielna RandomAccessFile....491

Typowe zastosowania strumieni I/O 491
Strumienie wejścia 493
Strumienie wyjścia 495
Strumienie typu „pipe” 497
Narzędzia do zapisu i odczytu danych z plików 497
Standardowe wejście-wyjście 498
Czytanie ze standardowego wejścia 499
Zamiana System.out na PrintWriter 499
Przekierowywanie standardowego wejścia-wyjścia 500
Nowe wejście-wyjście 501
Konwersja danych 505
Pobieranie podstawowych typów danych 508
Widoki buforów 509
Manipulowanie danymi przy użyciu buforów 514
Szczegółowe informacje o buforach 515
Pliki odwzorowywane w pamięci 518
Blokowanie plików 521
Kompresja 524
Prosta kompresja do formatu GZIP 524
Przechowywanie wielu plików w formacie Zip 525
Archiwa Javy (JAR) 528
Serializacja obiektów 529
Odnajdywanie klasy 533
Kontrola serializacji 534
Stosowanie trwałości 542
Preferencje 547
Wyrażenia regularne 549
Tworzenie wyrażeń regularnych 549
Kwantyfikatory 551
Klasy Pattern oraz Matcher 552
metoda split() 559
Operacje zastępowania 560
Metoda reset() 562
Wyrażenia regularne i operacje wejścia-wyjścia Javy 563
Czy klasa StringTokenizer jest potrzebna 564
Podsumowanie 565
Ćwiczenia 565

Rozdział 13. Współbieżność 569

Motywacja 570
Proste wątki 571
Przełączanie 573
Wstrzymywanie wykonywania wątku 574
Priorytet 576
Wątki demony 578
Łączenie wątków 580
Różne sposoby kodowania 582
Tworzenie wrażliwego interfejsu użytkownika 587
Współdzielenie ograniczonych zasobów 588
Niewłaściwy dostęp do zasobów 589
Kolizje podczas dostępu do zasobów 593
Rozwiązywanie współzawodnictwa o zasoby współdzielone 595
Sekcje krytyczne 601
Stany wątków 606
Zablokowanie 606

Współdziałanie wątków 607
Wait i notify 607
Stosowanie potoków do przekazywania danych pomiędzy wątkami 611
Bardziej wyszukane sposoby współdziałania 612
Wzajemna blokada 612
Poprawny sposób zatrzymywania 617
Przerywanie zablokowanego wątku 618
Grupy wątków 619
Podsumowanie 619
Ćwiczenia 621

Rozdział 14. Tworzenie okienek i apletów 623
Podstawy tworzenia apletów 625
Ograniczenia apletów 625
Zalety apletów 626
Szkielet aplikacji 627
Uruchamianie apletów w przeglądarce internetowej 628
Wykorzystanie programu Appletviewer 629
Testowanie apletów 630
Uruchamianie apletów z wiersza poleceń 631
Platforma prezentacyjna 632
Tworzenie przycisku 634
Przechwytywanie zdarzenia 635
Pola tekstowe 637
Rozmieszczenie komponentów 639
BorderLayout 639
FlowLayout 640
GridLayout 641
GridBagLayout 642
Bezpośrednie pozycjonowanie 642
BoxLayout 642
Najlepsze rozwiązanie? 645
Model zdarzeń w Swingu 646
Rodzaje zdarzeń i odbiorców 646
Śledzenie wielu zdarzeń 652
Katalog komponentów Swing 654
Przyciski 655
Ikony 657
Podpowiedzi 659
Pola tekstowe 659
Ramki 661
Panele z paskami przewijania 662
Miniedytor 663
Pola wyboru 664
Przyciski wyboru 666
Listy rozwijane 667
Listy 668
Zakładki 670
Okna komunikatów 671
Menu 672
Menu kontekstowe 677
Rysowanie 679
Okna dialogowe 681
Okna dialogowe plików 685
HTML w komponentach Swing 686

Suwaki i paski postępu 687
Drzewa 688
Tabele 690
Zmiana wyglądu aplikacji 692
Schowek 694
Pakowanie apletu do pliku JAR 696
Cyfrowe podpisywanie apletów 697
JNLP oraz Java Web Start 701
Techniki programowania 706
Dynamiczne dołączanie zdarzeń 706
Oddzielenie logiki biznesowej od interfejsu użytkownika 708
Postać kanoniczna 710
Współbieżność i Swing 710
Interfejs Runnable raz jeszcze 711
Zarządzanie współbieżnością 713
Programowanie wizualne i Beany 716
Czym jest Bean 717
Wydobycie informacji o Beanie poprzez introspektor 719
Bardziej wyszukany Bean 724
Beany i synchronizacja 727
Pakowanie Beana 730
Bardziej złożona obsługa Beanów 732
Więcej o Beanach 732
Podsumowanie 733
Ćwiczenia 734

Rozdział 15. Wykrywanie problemów 737

Testowanie modułów 739
Prosty szkielet testowy 741
JUnit 750
Poprawianie niezawodności poprzez wykorzystanie asercji 754
Składnia asercji 755
Użycie asercji do projektowania przez kontrakt 758
Przykład: projektowanie przez kontrakt i testowanie modułów metodą „białej skrzynki” 761
Konstruowanie przy użyciu narzędzia Ant 765
Zautomatyzujmy wszystko 766
Problemy z make 766
Ant: faktyczny standard 767
Kontrola wersji przy użyciu CVS 771
Prace codzienne 774
Rejestracja 775
Poziomy rejestracji 776
LogRecord 779
Handler 780
Filtry 784
Formatter 785
Przykład: rejestracja informacji poprzez wysyłanie wiadomości pocztą elektroniczną 787
Kontrola poziomu rejestracji przez wykorzystanie przestrzeni nazw 789
Praktyki rejestracyjne stosowane w dużych projektach 791
Podsumowanie 793
Testowanie 794
Testowanie przy wykorzystaniu JDB 794
Graficzne programy uruchomieniowe 799

Profilowanie i optymalizacja 799
Śledzenie zużycia pamięci 800
Śledzenie wykorzystania procesora 800
Testowanie obłożenia 800
Interfejs profilujący JVM 801
Wykorzystanie HPROF 801
Efektywność działania wątków 803
Wytyczne optymalizacji 804
Doclety 804
Podsumowanie 806
Ćwiczenia 808

Rozdział 16. Analiza i projekt 811

Metodologia 811
Faza 0.: stwórz plan 813
Określenie celu 814
Faza 1.: co robimy 814
Faza 2.: jak to stworzymy 817
Pięć etapów projektowania obiektów 819
Wytyczne dotyczące tworzenia obiektów 820
Faza 3.: tworzenie jądra systemu 821
Faza 4.: przejrzenie przypadków użycia 821
Faza 5.: ewolucja 822
Opłaca się planować 824
Programowanie ekstremalne 824
Najpierw pisz testy 825
Programowanie w parach 826
Strategie przejścia 827
Wytyczne 827
Przeszkody w zarządzaniu 829
Podsumowanie 830

Dodatek A Przekazywanie i zwracanie obiektów 831

Przekazywanie referencji 831
Aliasy 832
Tworzenie kopii lokalnych 834
Przekazywanie przez wartość 834
Klonowanie obiektów 835
Dodanie klonowalności do klasy 836
Udane klonowanie 838
Działanie Object.clone() 840
Klonowanie złożonego obiektu 842
Głęboka kopia ArrayList 844
Głęboka kopia poprzez serializację 845
Dodanie klonowalności w dół hierarchii 847
Dlaczego takie dziwne rozwiązanie 848
Sterowanie klonowalnością 848
Konstruktor kopiujący 853
Klasy tylko do odczytu 857
Tworzenie klas tylko do odczytu 858
Wada obiektów odpornych na zmiany 859
Niezmienne obiekty String 861
Klasy String i StringBuffer 864
Łańcuchy są wyjątkowe 866
Podsumowanie 867
Ćwiczenia 868

Dodatek B Wskazówki dla programistów 869
Projekt 869
Implementacja 874

Dodatek C Materiały uzupełniające 879
Szkolenie „Thinking in Java” 879
Szkolenie na CD-ROM-ie „Hands-On Java 3rd edition” 879
Szkolenie „Designing Object & Systems” 880
Thinking in Enterprise Java 880
Szkolenie J2EE 881
Thinking in Patterns (with Java) 881
Szkolenie „Thinking in Patterns” 882
Konsultacja i analiza projektów 882

Dodatek D Zasoby 883
Oprogramowanie 883
Książki 883
Analiza i projektowanie 884
Python 886
Lista moich książek 887

Skorowidz 889

Rozdział 5.

Ukrywanie implementacji

Pierwszą sprawą, braną pod uwagę przy programowaniu obiektowym, jest „oddzielenie rzeczy, które się zmieniają, od rzeczy pozostających bez zmian”.

Jest to szczególnie istotne w przypadku bibliotek. Użytkownicy (programiści-klienci) takiej biblioteki muszą polegać na części, którą wykorzystują, i nie obawiać się, że będą musieli przepisywać kod, jeżeli pojawi się nowa wersja. Z drugiej strony, twórca biblioteki musi mieć swobodę jej modyfikowania i ulepszania przy zachowaniu pewności, iż zmiany te nie będą miały wpływu na kod klientów.

Można to osiągnąć poprzez ustalenie konwencji. Na przykład, programista biblioteki musi zgodzić się na nieusuwanie istniejących metod przy modyfikowaniu klas tej biblioteki, ponieważ uszkodziłoby to kod klientów. Sytuacja odwrotna jest jednak trudniejsza. W przypadku pól klasy twórca biblioteki nie może wiedzieć, które z nich były używane przez klientów. Dotyczy to także metod stanowiących jedynie część implementacji klasy, nieprzewidzianych do bezpośredniego wykorzystania przez programistę-klienta. Co zrobić, gdy twórca biblioteki chciałby wyrzucić starą implementację i zastosować nową? Zmiana któregokolwiek ze wspomnianych składników klas może uszkodzić kod ich użytkownika. Na twórcę biblioteki nałożony jest „gorset” niepozwalający mu niczego zmienić.

W celu rozwiązania tego problemu Java dostarcza *modyfikatory dostępu* (ang. *access modifiers*). Umożliwiają one twórcy biblioteki zaznaczenie, co ma być dostępne dla klienta, a co nie. Poziomy dostępu w porządku od największego do najmniejszego to: `public`, `protected`, pakietowy (który nie posiada słowa kluczowego) i `private`. Na podstawie poprzedniego akapitu można wysnuć wniosek, iż projektant biblioteki powinien wszystko, co się da, uczynić prywatnym, a eksponować jedynie to, co powinno być jego zdaniem wykorzystywane przez klientów. Jest to prawda, mimo że może być to sprzeczne z intuicją osób programujących w innych językach (szczególnie w C) i będących przyzwyczajonymi do nieograniczonego dostępu do wszystkiego. Przed końcem tego rozdziału powinieneś być przekonany o wartości kontroli dostępu.

Koncepcja biblioteki komponentów i kontroli dostępu do nich nie jest jednak kompletna. Pozostaje pytanie: w jaki sposób komponenty są łączone w spójne jednostki biblioteczne? W Javie jest to kontrolowane przez słowo kluczowe `package` (pakiet), na modyfikatory dostępu ma zaś wpływ to, czy klasy znajdują się w tym samym pakiecie czy też w różnych. Zatem na początek dowiemy się, w jaki sposób komponenty bibliotek są umieszczane w pakietach. Dzięki temu możliwe będzie pełne zrozumienie modyfikatorów dostępu.

Pakiet — jednostka biblioteczna

Pakiet jest tym, co otrzymujemy, używając słowa kluczowego `import` do włączenia całej biblioteki, np. poprzez:

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

Powoduje to wprowadzenie do programu całej biblioteki `util`, będącej częścią standardowej dystrybucji Javy. Ponieważ, na przykład, klasa `ArrayList` znajduje się w pakiecie `java.util`, możemy teraz używać zarówno pełnej nazwy `java.util.ArrayList` (co było możliwe także bez instrukcji `import`), jak i napisać po prostu `ArrayList` (dzięki użyciu `import`). Jeżeli chcemy importować pojedynczą klasę, możemy wymienić jej nazwę w instrukcji `import`:

```
import java.util.ArrayList;
```

Możemy teraz używać `ArrayList` bez kwalifikatorów, jednakże inne klasy pakietu `java.util` nie są dostępne.

Uzasadnieniem tego importowania jest dostarczenie mechanizmu zarządzania przestrzeniami nazw. Nazwy składników wszystkich klas są od siebie odizolowane. Metoda `f()` klasy `A` nie wejdzie w konflikt z metodą `f()` klasy `B`, nawet jeśli obie metody mają taką samą sygnaturę (listę argumentów). Co jednak z nazwami klas? Przypuśćmy, iż tworzymy klasę `Stos`, instalowaną na maszynie posiadającej już `Stos` napisany przez kogoś innego. Ten potencjalny konflikt jest przyczyną, dla której w Javie są istotne posiadanie całkowitej kontroli nad przestrzeniami nazw oraz możliwość stworzenia unikatowej nazwy bez względu na ograniczenia Internetu.

Do tej pory większość przykładów w tej książce składała się z jednego pliku i była zaprojektowana do użytku lokalnego — nie musieliśmy przejmować się nazwami pakietów (w takim przypadku nazwa klasy jest umieszczana w „pakiecie domyślnym"). Jest to z pewnością jakieś rozwiązanie i ze względu na swą prostotę będzie stosowane w dalszej części książki, gdziekolwiek będzie to możliwe. Jeżeli jednak planujemy tworzenie bibliotek lub programów przyjaznych dla innych programów Javy na tej samej maszynie, musimy zapobiec konfliktom nazw klas.

Gdy tworzymy plik z kodem źródłowym Javy, jest on powszechnie nazywany *jednostką kompilacji* (czasami również *jednostką translacji*). Nazwa każdej jednostki kompilacji musi mieć rozszerzenie *.java*. Wewnątrz takiej jednostki może znajdować się publiczna klasa, której nazwa musi być taka sama, jak nazwa pliku (włączając w to wielkość liter, wyłączając natomiast rozszerzenie *.java*). W każdej jednostce kompilacji może znajdować się tylko *jedna* klasa publiczna, w przeciwnym razie kompilator zaprotestuje. Ewentualne pozostałe klasy wchodzące w skład jednostki kompilacji są ukryte przed światem zewnętrznym, ponieważ nie są publiczne. Stanowią one „klasy wspierające" zamieszczonej głównej klasy publicznej.

Kompilując plik typu *.java*, otrzymujemy plik wynikowy o dokładnie takiej samej nazwie, lecz z rozszerzeniem *.class dla każdej klasy* z pliku *.java*. Możliwe jest więc uzyskanie całkiem sporej liczby plików *.class* z niewielu plików *.java*. Jeżeli pracowałeś kiedyś z językiem kompilowanym, możesz być przyzwyczajony do kompilatora produkującego

formy pośrednie (zwykle pliki typu „obj"), które następnie łączone są przy użyciu linkera (w celu wyprodukowania pliku wykonywalnego) lub bibliotekarza (w celu utworzenia biblioteki). Java nie działa w ten sposób. Działający program to zbiór plików typu *.class*, które mogą zostać spakowane i skompresowane do pliku typu JAR (z wykorzystaniem programu archiwizującego *jar*). Interpreter Javy odpowiada za znajdowanie, ładowanie i interpretowanie tych plików[1].

Biblioteka również stanowi zespół plików zawierających klasy. Każdy plik zawiera jedną klasę publiczną (nie musi zawierać takiej klasy, ale zwykle tak się dzieje), a zatem na każdy plik przypada jeden komponent. Jeżeli chcemy zaznaczyć, że wszystkie te komponenty (znajdujące się w oddzielnych plikach *.java* i *.class*) stanowią jedną całość, używamy słowa kluczowego `package`.

Gdy napiszemy:

```
package mojpakiet;
```

na początku pliku (jeżeli używamy instrukcji `package`, *musi* ona stanowić w pliku pierwszy element nie będący komentarzem), tym samym zaznaczamy, iż ta jednostka kompilacji znajduje się pod parasolem nazwy `mojpakiet`, a zatem każdy, kto chciałby wykorzystać zawarte w niej nazwy, musi albo wyspecyfikować pełną nazwę, albo użyć słowa kluczowego `import` w połączeniu z `mojpakiet` (na jeden z podanych wcześniej sposobów). Zauważmy, że konwencja nazywania pakietów w Javie polega na używaniu wyłącznie małych liter, nawet jeśli nazwa składa się z kilku słów.

Przypuśćmy np., że plik nazywa się `MojaKlasa.java`. Znaczy to, iż może znajdować się w nim jedna i tylko jedna klasa publiczna, a jej nazwą musi być `MojaKlasa` (wielkość liter ma znaczenie):

```
package mojpakiet;
public class MojaKlasa{
  // . . .
```

Teraz, jeżeli ktoś chce wykorzystać klasę `MojaKlasa` lub jakąkolwiek inną klasę publiczną z pakietu `mojpakiet`, musi użyć słowa kluczowego `import`, aby uczynić dostępnymi nazwy zawarte w pakiecie `mojpakiet`. Inną możliwością jest użycie nazwy z pełnym kwalifikatorem:

```
mojpakiet.MojaKlasa m = new mojpakiet.MojaKlasa();
```

Słowo kluczowe `import` czyni ten kod znacznie czystszym:

```
import mojpakiet.*;
// . . .
MojaKlasa m = new MojaKlasa();
```

Warto zapamiętać, że dzięki słowom kluczowym `package` i `import` projektanci bibliotek mogą podzielić pojedynczą globalną przestrzeń nazw, aby zapobiec ich konfliktom bez względu na to, jak wielu ludzi przyłączy się do Internetu i zacznie tworzyć klasy Javy.

[1] Żaden element Javy nie wymusza używania interpretera. Istnieją rodzime kompilatory Javy produkujące pojedynczy plik wykonywalny.

Tworzenie unikatowych nazw pakietów

Można zauważyć, że ponieważ pakiet nigdy nie zostaje „spakowany" do pojedynczego pliku, lecz może się składać z wielu plików *.class*, zaczyna się powoli robić bałagan. Aby temu zapobiec, logiczne wydaje się umieszczenie wszystkich plików *.class*, składających się na dany pakiet, w jednym katalogu — a zatem wykorzystanie hierarchicznej struktury plików dostarczanej przez system operacyjny. Jest to jeden ze sposobów, w jaki Java radzi sobie z problemem bałaganu. Inny sposób poznamy później, przy omawianiu narzędzia *jar*.

Zebranie plików pakietu w pojedynczym podkatalogu rozwiązuje także dwa inne problemy: tworzenie unikatowych nazw pakietów oraz odnajdywanie tych klas, które mogłyby być ukryte gdzieś w strukturze katalogów. Jak powiedziałem w rozdziale 2., osiągane jest to poprzez zakodowanie ścieżki dostępu do pliku *.class* w nazwie pakietu. Zgodnie z konwencją przyjmuje się, iż pierwszą część nazwy pakietu powinna stanowić odwrócona nazwa domeny internetowej twórcy klasy. Ponieważ unikatowość nazw domen internetowych jest gwarantowana, zatem *jeżeli* przestrzegamy konwencji, gwarantowana jest również unikatowość nazwy naszego pakietu, i nie ma możliwości powstania konfliktu (przynajmniej do chwili, gdy utracimy nazwę domeny na rzecz kogoś, kto zacznie pisać kod Javy z takimi samymi nazwami ścieżek jak nasze). Oczywiście jeżeli nie posiadamy własnej domeny, jesteśmy zmuszeni do sfabrykowania jakiejś mało prawdopodobnej kombinacji (np. własnego imienia i nazwiska) w celu nazywania pakietów. Jeżeli decydujemy się na rozpoczęcie publikowania kodu w Javie, warto zdobyć się na stosunkowo niewielki wysiłek pozyskania nazwy domeny.

Drugą częścią sztuczki jest odzyskiwanie nazwy katalogu na naszej maszynie z nazwy pakietu, aby działający program Javy mógł w chwili, gdy tego potrzebuje, załadować plik *.class* (co dzieje się dynamicznie wtedy, gdy program musi utworzyć obiekt danej klasy lub odwołać się do jej składnika statycznego).

Interpreter Javy działa w następujący sposób. Na wstępie odnajduje zmienną środowiskową `CLASSPATH`[2] (ustawianą przez system operacyjny, czasami przez program instalujący Javę lub narzędzie wykorzystujące Javę na naszej maszynie). Zmienna `CLASSPATH` zawiera jeden lub więcej katalogów używanych jako korzenie przy wyszukiwaniu plików *.class*. Zaczynając od takiego korzenia, interpreter zamieni wszystkie kropki w nazwie pakietu na znaki ukośnika w celu utworzenia ścieżki dostępu (a zatem pakiet `foo.bar.baz` zostanie zamieniony na `foo\bar\baz` lub `foo/bar/baz` albo, być może, jeszcze inny w zależności od naszego systemu operacyjnego). Rezultat zostanie następnie dołączony do kolejnych katalogów z `CLASSPATH`. W taki sposób wyznaczone zostają miejsca, w których poszukuje się pliku *.class* o nazwie odpowiadającej nazwie klasy, której instancję chcemy stworzyć (interpreter przeszukuje również pewne standardowe katalogi pod względem miejsca, w którym on sam się znajduje).

Aby to zrozumieć, rozważmy nazwę domeny autora, tj. *bruceeckel.com*. Poprzez jej odwrócenie otrzymujemy `com.bruceeckel`, unikatową globalną nazwę dla moich klas (rozszerzenia com, edu, org itd. były kiedyś pisane wielkimi literami, jednak w Javie 2 zostało to zmienione tak, aby w nazwach pakietów występowały wyłącznie małe litery).

[2] Odwołując się do zmiennych środowiskowych, ich nazwy będę zapisywać wielkimi literami (np.: `CLASSPATH`).

Możliwe jest dalsze podzielenie tej przestrzeni nazw, kiedy chcę np. utworzyć bibliotekę `simple`, przez co powstanie np.:

```
package com.bruceeckel.simple;
```

Taka nazwa pakietu może zostać teraz użyta jako parasol nazw dla następujących dwóch plików:

```
//: com:bruceeckel:simple:Vector.java
// Tworzenie pakietu.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.bruceeckel.util.Vector");
  }
} ///:~
```

Przy tworzeniu własnych pakietów odkryjemy, że instrukcja `package` musi być pierwszym nie będącym komentarzem fragmentem kodu w pliku. Drugi z plików wygląda bardzo podobnie:

```
//: com:bruceeckel:simple:List.java
// Tworzenie pakietu.
package com.bruceeckel.simple;

public class List {
  public List() {
    System.out.println(
      "com.bruceeckel.util.List");
  }
} ///:~
```

Oba pliki zostały umieszczone w systemie autora w podkatalogu:

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Cofając się nieco, możemy odnaleźć nazwę pakietu *com.bruceeckel.simple*, co jednak z pierwszą częścią ścieżki? Tym zajmuje się zmienna środowiskowa `CLASSPATH`, która na maszynie autora ma wartość:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Możemy zaobserwować, iż zmienna `CLASSPATH` może zawierać kilka alternatywnych ścieżek poszukiwań.

Pewne odstępstwo dotyczy plików typu JAR. W zmiennej `CLASSPATH` musimy umieścić nazwę pliku JAR, a nie tylko ścieżkę do miejsca, gdzie się znajduje. A zatem dla pliku nazywającego się *grape.jar* mamy:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Kiedy zmienna `CLASSPATH` jest już poprawnie ustawiona, wtedy następujący plik może być umieszczony w dowolnym katalogu:

```
//: c05:LibTest.java
// Wykorzystanie biblioteki.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.simple.*;

public class LibTest {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
    monitor.expect(new String[] {
      "com.bruceeckel.simple.Vector",
      "com.bruceeckel.simple.List"
    });
  }
} ///:~
```

Gdy kompilator napotyka instrukcję `import`, rozpocznie przeszukiwanie katalogów wymienionych w zmiennej `CLASSPATH` w poszukiwaniu podkatalogu *com\bruceeckel\simple*, a następnie skompilowanych plików o odpowiednich nazwach (*Vector.class* dla klasy `Vector` i *List.class* dla klasy `List`). Należy zauważyć, iż zarówno same klasy, jak i ich pożądane metody muszą być publiczne.

Ustawianie zmiennej `CLASSPATH` było tak trudne dla początkujących użytkowników Javy (także dla autora), iż firma Sun uczyniła pakiet JDK dla Java 2 nieco sprytniejszym. Po jego zainstalowaniu odkryjemy, że skompilowanie i uruchomienie prostych programów jest możliwe bez ustawiania `CLASSPATH`. W celu skompilowania i uruchomienia kodu źródłowego (dostępnego pod adresem *www.BruceEckel.com*) konieczne będzie dodanie do zmiennej `CLASSPATH` ścieżki dostępu do głównego katalogu zawierającego kody przykładów.

Kolizje

Co się dzieje, gdy program importuje z użyciem znaku * dwie biblioteki zawierające te same nazwy? Na przykład, przypuśćmy, że program zawiera instrukcje:

```
import com.bruceeckel.simple.*;
import java.util.*;
```

Ponieważ pakiet `java.util.*` również zawiera klasę `Vector`, powoduje to potencjalną kolizję. Jednak wszystko jest w porządku, dopóki nie napiszemy kodu, który rzeczywiście spowoduje kolizję — jest to dobre rozwiązanie, ponieważ w przeciwnym razie zmuszeni bylibyśmy tworzyć rozbudowany kod, aby uniemożliwić kolizje, które nigdy by się nie zdarzyły.

Kolizja *ma* miejsce, gdy spróbujemy teraz stworzyć `Vector`:

```
Vector v = new Vector();
```

Do której klasy `Vector` to się odnosi? Kompilator nie może tego wiedzieć, podobnie jak czytający kod. Zatem kompilator protestuje i zmusza do jawności. Jeżeli chcemy, na przykład, stworzyć standardowy `Vector` Javy, musimy napisać:

```
java.util.Vector v = new java.util.Vector();
```

Ponieważ powyższa instrukcja (wraz z wartością CLASSPATH) dokładnie określa położenie klasy Vector, nie ma zatem potrzeby poprzedzać jej konstrukcją import java.util.*, chyba że używamy jeszcze innych elementów pakietu java.util.

Własna biblioteka narzędziowa

Uzbrojeni w tę wiedzę możemy stworzyć własne biblioteki pomocnicze w celu zredukowania lub wyeliminowania powtarzania kodu. Można, na przykład, rozważyć wprowadzenie innej nazwy dla System.out.println() w celu ograniczenia ilości pisanego kodu. Nasz skrót mógłby stać się częścią pakietu o nazwie tools (narzędzia):

```
//: com:bruceeckel:tools:P.java
// Skróty P.rint i P.rintln.
package com.bruceeckel.tools;

public class P {
  public static void rint(String s) {
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
} ///:~
```

Tak zdefiniowanego skrótu możemy używać do wypisania obiektu klasy String z dodanym znakiem nowego wiersza (wykorzystując P.rintln()) lub bez tego znaku (z wykorzystaniem P.rint()).

Można się domyślić, iż lokalizacją powyższego pliku musi być katalog o nazwie zaczynającej się od jednej z lokalizacji podanych w zmiennej CLASSPATH, po której następuje *com/bruceeckel/tools*. Po kompilacji plik *P.class* może być wykorzystywany wszędzie w naszym systemie z użyciem instrukcji import:

```
//: c05:ToolTest.java
// Wykorzystanie biblioteki narzędziowej.
import com.bruceeckel.tools.*;
import com.bruceeckel.simpletest.*;

public class ToolTest {
  static Test monitor = new Test();
  public static void main(String[] args) {
    P.rintln("Dostępne od teraz!");
    P.rintln("" + 100); // Wymuś konwersję na String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
    monitor.expect(new String[] {
      "Dostępne od teraz!",
      "100",
      "100",
      "3.14159"
    });
  }
} ///:~
```

Zauważmy, że wszystkie obiekty mogą z łatwością zostać zmuszone do wyprodukowania swej reprezentacji w postaci łańcucha znaków poprzez umieszczenie ich w wyrażeniu typu `String` — w powyższym przykładzie sztuki tej dokonuje się poprzez rozpoczęcie wyrażenia od pustego łańcucha znakowego. Prowadzi to do interesującej obserwacji. Jeżeli wywołamy `System. out.println(100)`, zadziała bez rzutowania do typu `String`. Stosując pewne dodatkowe przeciążanie, możemy sprawić, aby klasa `P` również to potrafiła (jest to jedno z ćwiczeń na końcu rozdziału).

Od tej pory, kiedy tylko stworzysz jakieś użyteczne narzędzie, będziesz mógł dodać je do swojego własnego katalogu `util` lub `tools`.

Wykorzystanie instrukcji import do zmiany zachowania

Jednym z elementów języka C, niewystępującym w Javie, jest *kompilacja warunkowa* pozwalająca na uzyskanie innego zachowania za pomocą pojedynczego przełącznika bez zmieniania reszty kodu. Powodem, dla którego rozwiązanie to nie zostało włączone do Javy, jest prawdopodobnie fakt, iż w C używane było głównie do rozwiązywania problemu przenośności na różne platformy — różne fragmenty kodu włączane były w zależności od docelowej platformy. Ponieważ Java miała być z założenia automatycznie przenośna, zatem kompilacja warunkowa nie powinna być potrzebna.

Istnieją jednak inne cenne zastosowania kompilacji warunkowej. Bardzo przydatne jest jej wykorzystanie przy usuwaniu błędów. Elementy odpowiedzialne za wykrywanie błędów są włączone podczas pracy nad programem, wyłączone zaś w produkcie dostarczanym klientom. Można to osiągnąć, zmieniając importowany pakiet, a tym samym kod używany w testowej oraz finalnej wersji programu. Rozwiązanie to można zastosować do dowolnego kodu, z którego chcemy korzystać warunkowo.

Pułapka związana z pakietami

Warto zapamiętać, że zawsze, gdy tworzymy pakiet, niejawnie określamy strukturę katalogów przez nadanie mu nazwy. Pakiet *musi* znajdować się w katalogu wskazywanym przez jego nazwę i powinien być dostępny przy przeszukiwaniu katalogów za pomocą zmiennej `CLASSPATH`. Eksperymentowanie ze słowem kluczowym `package` może być na początku nieco frustrujące, ponieważ jeśli nie stosujemy się ściśle do reguły zgodności między nazwami pakietów i ścieżkami do katalogów, otrzymujemy dużo tajemniczych komunikatów czasu wykonania dotyczących niemożności odnalezienia określonych klas, nawet jeżeli klasy te znajduje się w tym samym katalogu. Po otrzymaniu takiej wiadomości można spróbować wykomentować instrukcję `package` — jeżeli po tym zabiegu wszystko zacznie działać, wtedy wiadomo już, w czym tkwi problem.

Modyfikatory dostępu w Javie

Używając w Javie modyfikatorów dostępu: `public`, `protected` lub `private`, umieszczamy je przed definicją każdego składnika klasy, bez względu na to, czy jest to pole czy metoda. Występuje tu znacząca różnica w stosunku do C++, gdzie modyfikator dostępu kontroluje wszystkie następujące po nim definicje, aż do wystąpienia innego modyfikatora.

W ten czy inny sposób wszystko ma określony stopień dostępu. Zajmiemy się teraz różnymi ich typami, począwszy od dostępu domyślnego.

Dostęp pakietowy

Co się dzieje, jeśli nie podamy żadnego modyfikatora dostępu, tak jak miało to miejsce we wszystkich przykładach przed tym rozdziałem? Domyślny stopień dostępu nie posiada swojego słowa kluczowego, określany jest jednak często jako *dostęp pakietowy* (a czasami także jako dostęp „przyjazny"). Oznacza to, iż inne klasy tego samego pakietu posiadają dostęp do elementu, jednak dla klas poza pakietem wydaje się on być prywatny. Ponieważ jednostka kompilacji — plik — może należeć jedynie do jednego pakietu, zatem, dzięki zastosowaniu dostępu pakietowego, wszystkie klasy wewnątrz tej samej jednostki kompilacji są automatycznie dostępne dla siebie nawzajem.

Dostęp pakietowy pozwala na grupowanie spokrewnionych klas w pakiety, dzięki czemu mogą one z łatwością ze sobą współdziałać. Gdy programista umieszcza pewne klasy razem we wspólnym pakiecie (tym samym umożliwiając im wzajemny dostęp do zdefiniowanych w nich składowych o dostępie pakietowym), musi być „właścicielem" kodu tego pakietu. Wydaje się sensowne, że tylko kod „posiadany" przez nas powinien mieć dostęp w ramach pakietu do innego „posiadanego" przez nas kodu. Można powiedzieć, iż taki dostęp nadaje sens (lub że stanowi przyczynę) grupowaniu klas w pakiety. W wielu językach sposób organizacji definicji w plikach jest całkowicie dowolny — w Javie jesteśmy zmuszeni do organizowania ich w sposób rozsądny. Dodatkowo będziemy prawdopodobnie chcieli wyłączyć klasy, które nie powinny mieć dostępu do klas zdefiniowanych w aktualnym pakiecie.

Klasa ma kontrolę nad tym, który kod ma dostęp do jej składników. Nie istnieje magiczny sposób „włamania się". Kod z innego pakietu nie może się pojawić i powiedzieć: „Cześć, jestem przyjacielem Boba!" i oczekiwać, że zostaną mu udostępnione chronione, „przyjacielskie" i prywatne składniki klasy Bob. Jedynymi sposobami przyznania dostępu do składnika są:

1. Uczynienie składnika publicznym (przez podanie modyfikatora `public`). Od tej chwili dostęp do niego będzie miał każdy, z dowolnego miejsca.
2. Zapewnienie dostępu do składnika w ramach pakietu poprzez rezygnację z podania modyfikatora dostępu oraz umieszczenie klas mających mieć dostęp do składnika w tym samym pakiecie.
3. Jak przekonamy się w rozdziale 6. (po wprowadzeniu dziedziczenia), klasa pochodna ma dostęp do składników chronionych (zadeklarowanych z modyfikatorem `protected`) oraz publicznych (modyfikator `public`). Nie ma natomiast dostępu do składników prywatnych (modyfikator `private`). Do składników „pakietowych" ma dostęp tylko wtedy, gdy obie klasy znajdują się w tym samym pakiecie. Nie musisz jednak na razie zaprzątać sobie tym głowy.
4. Dostarczenie akcesorów i modyfikatorów — metod udostępniających i zmieniających wartość składnika (w języku angielskim określane są one jako metody typu *accessor/mutator* lub *get/set*). W kategoriach programowania zorientowanego obiektowo jest to rozwiązanie najbardziej cywilizowane. Ma ono także fundamentalne znaczenie dla komponentów JavaBeans, o czym przekonamy się w rozdziale 14.

public: dostęp do interfejsu

Używając słowa kluczowego `public`, czynimy deklarację składnika następującego po nim dostępną dla każdego, w szczególności dla programisty-klienta wykorzystującego bibliotekę. Przypuśćmy, iż definiujemy pakiet `dessert` zawierający następującą jednostkę kompilacji:

```
//: c05:dessert:Cookie.java
// Tworzenie biblioteki.
package c05.dessert;


public class Cookie {
  public Cookie() {
   System.out.println("Konstruktor klasy Cookie");
  }
  void bite() { System.out.println("metoda bite"); }
} ///:~
```

Pamiętaj, iż plik *Cookie.java* musi znajdować się w podkatalogu o nazwie *dessert*, umieszczonym w katalogu *c05* (oznaczającym piąty rozdział książki), który z kolei musi być podkatalogiem jednego z katalogów umieszczonych w zmiennej `CLASSPATH`. Błędem jest zakładanie, że Java zawsze traktuje katalog aktualny jako jeden z punktów startowych dla poszukiwań. Jeżeli nie umieścimy katalogu „.” w zmiennej `CLASSPATH`, Java nie będzie postępować w ten sposób.

Stwórzmy teraz program wykorzystujący klasę `Cookie`:

```
//: c05:Dinner.java
// Zastosowanie biblioteki.
import com.bruceeckel.simpletest.*;
import dessert.*;

public class Dinner {
  static Test monitor = new Test();
  public Dinner() {
   System.out.println("Konstruktor klasy Dinner");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Brak dostępu
    monitor.expect(new String[] {
      "Konstruktor klasy Cookie"
    });
  }
} ///:~
```

Stworzenie obiektu klasy `Cookie` było możliwe, ponieważ zarówno sama klasa, jak i jej konstruktor są publiczne (koncepcji klasy publicznej przyjrzymy się trochę później). Składnik `bite()` jest jednak niedostępny z wnętrza pliku *Dinner.java*, ponieważ dostęp do niego jest możliwy jedynie wewnątrz pakietu `dessert` (o sprawdzenie praw dostępu do tego składnika zadba kompilator).

Pakiet domyślny

Może być zaskakujące, iż poniższy kod kompiluje się mimo wrażenia, że łamie zasady:

```
//: c05:Cake.java
// Dostęp do klasy w oddzielnej
// jednostce kompilacji.
import com.bruceeckel.simpletest.*;

class Cake {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
    monitor.expect(new String[] {
      "Pie.f()"
    });
  }
} ///:~
```

W drugim pliku, znajdującym się w tym samym katalogu, umieszczamy:

```
//: c05:Pie.java
// Druga z klas.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

Początkowo można przypuszczać, iż powyższe pliki są sobie zupełnie obce, a mimo to obiekt typu `Cake` może utworzyć obiekt `Pie` i wywołać jego metodę `f()`! (Aby możliwe było skompilowanie tego pliku, należy umieścić katalog `"."` w zmiennej `CLASSPATH`). Zwykle narzuca się przypuszczenie, że `Pie` i `f()` są składowymi o dostępie pakietowym, i przez to niedostępnymi dla klasy `Cake`. Istotnie są one dostępne w ramach pakietu — to założenie jest poprawne. Powodem, dla którego są dostępne z pliku *Cake.java*, jest to, że znajdują się w tym samym co on katalogu i nie są jawnie przypisane do żadnego pakietu. Java traktuje takie pliki jako należące do „pakietu domyślnego" dla tego katalogu, a przez to dostępne w ramach pakietu dla innych plików w tym samym katalogu.

private: nie dotykać!

Słowo kluczowe `private` oznacza, że do danego składnika klasy nie ma dostępu nikt poza jego własną klasą, z wnętrza jej metod. Inne klasy tego samego pakietu nie mają dostępu do składowych prywatnych — to tak jakbyśmy izolowali klasę nawet przed nami. Z drugiej strony, nie jest nieprawdopodobne, że pakiet będzie tworzony przez grupę współpracujących osób, a w takiej sytuacji modyfikator `private` pozwoli na swobodne zmienianie składnika bez martwienia się o wpływ tych zmian na inne klasy w pakiecie.

Domyślny dostęp pakietowy daje wystarczający stopień ukrywania — pamiętajmy, iż składnik „pakietowy" jest niedostępny dla programisty używającego klasy. Jest to bardzo przyjemne, ponieważ domyślny stopień dostępu jest tym, którego będziemy najczęściej używać (a także tym, który otrzymamy, jeśli zapomnimy o podaniu jakiegokolwiek modyfikatora dostępu). W typowej sytuacji będziemy zatem zastanawiać się nad składnikami,

które chcielibyśmy uczynić publicznymi dla programisty-klienta. W rezultacie można początkowo dojść do wniosku, że słowa kluczowego `private` nie będziemy używać często, ponieważ można sobie bez niego poradzić (widać tu wyraźną różnicę w stosunku do C++). Spójne używanie modyfikatora `private` jest jednak bardzo istotne, szczególnie jeśli w grę wchodzi wielowątkowość (przekonamy się o tym w rozdziale 13.).

Oto przykład użycia słowa kluczowego `private`:

```
//: c05:IceCream.java
// Demonstracja słowa kluczowego "private".

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

Widzimy tu, jak pomocny może być modyfikator `private`: chcielibyśmy kontrolować sposób tworzenia obiektu i uniemożliwić bezpośredni dostęp do określonego konstruktora (lub wszystkich konstruktorów). W powyższym przykładzie nie możemy stworzyć obiektu typu `Sundae` poprzez konstruktor — musimy zamiast tego wywoływać metodę `makeASundae()`, aby stworzyła go za nas[3].

Każdą metodę, co do której jesteśmy pewni, iż jest jedynie pomocnicza w danej klasie, możemy uczynić prywatną, aby upewnić się, że nie użyjemy jej przypadkowo w innym miejscu pakietu, uniemożliwiając sobie tym samym jej zmianę lub usunięcie — czyniąc ją prywatną, gwarantujemy sobie zachowanie takich możliwości.

To samo odnosi się do prywatnych pól klasy. Jeżeli nie jesteśmy zmuszeni eksponować wewnętrznej implementacji (co zdarza się znacznie rzadziej, niż można przypuszczać), powinniśmy wszystkie pola uczynić prywatnymi. Jednakże to, że referencja do obiektu jest prywatna wewnątrz klasy, nie oznacza, iż jakiś inny obiekt nie może posiadać publicznej referencji do tego samego obiektu (dokładniejsze omówienie „odwołań" znajduje się w dodatku A).

protected: dostęp „na potrzeby" dziedziczenia

By zrozumieć działanie modyfikatora `protected`, musimy wybiec nieco do przodu. Po pierwsze, powinieneś być świadomy, iż zrozumienie poniższego fragmentu nie jest konieczne do dalszej pracy z książką, aż do wprowadzenia pojęcia *dziedziczenia* (rozdział 6.).

[3] W tym przypadku występuje także inny efekt: konstruktor domyślny jest jedynym zdefiniowanym, a jednocześnie jest on prywatny. W rezultacie niemożliwe będzie dziedziczenie po tej klasie (ten temat zostanie wprowadzony w rozdziale 6.).

Jednak w celu uzyskania pełnego obrazu zamieszczam tutaj pobieżny opis i przykład użycia modyfikatora `protected`.

Słowo kluczowe `protected` jest związane z koncepcją *dziedziczenia*, polegającą na tworzeniu nowej klasy poprzez dodanie nowych składników do klasy już istniejącej (określanej jako *klasa bazowa*), bez jakichkolwiek modyfikacji tej ostatniej. Możliwa jest również zmiana zachowania istniejących składowych. By umożliwić dziedziczenie po istniejącej klasie, musimy zaznaczyć, że nowa klasa rozszerza tę pierwszą. Robimy to w następujący sposób:

```
class Foo extends Bar {
```

Reszta definicji klasy powinna wyglądać normalnie.

Jeżeli tworzymy nowy pakiet i dziedziczymy po klasie z innego pakietu, wtedy jedynymi składowymi, do których mamy dostęp, są składowe publiczne pakietu oryginalnego (oczywiście jeżeli przeprowadzimy dziedziczenie *w tym samym* pakiecie, będziemy korzystać ze wszystkich składowych dostępnych w ramach pakietu). Czasami twórca klasy bazowej pragnie udzielić dostępu do określonego składnika klasom pochodnym, jednak nie chce go udzielać całemu światu. Do tego właśnie służy modyfikator `protected`.

Odwołamy się teraz jeszcze raz do pliku *Cookie.java*. Następująca klasa *nie posiada* dostępu do składnika „przyjaznego”:

```
//: c05:ChocolateChip.java
// Nie ma dostępu do składowej pakietowej z innego pakietu.
import com.bruceeckel.simpletest.*;
import dessert.*;

public class ChocolateChip extends Cookie {
  private static Test monitor = new Test();
  public ChocolateChip() {
   System.out.println("Konstruktor klasy ChocolateChip");
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Nie ma dostępu do bite
    monitor.expect(new String[] {
      "Konstruktor klasy Cookie",
      "Konstruktor klasy ChocolateChip"
    });
  }
} ///:~
```

Jedną z interesujących spraw dotyczących dziedziczenia jest to, że jeżeli metoda `bite()` istnieje w klasie `Cookie`, to istnieje również w każdej klasie dziedziczącej po `Cookie`. Ponieważ jednak `bite()` jest składową dostępną w ramach pakietu, lecz zdefiniowaną w innym pakiecie, w naszym pakiecie nie możemy z niej korzystać. Moglibyśmy oczywiście uczynić ją publiczną, jednak to dałoby do niej dostęp każdemu, a być może tego nie chcemy. Zmieniając klasę `Cookie` w następujący sposób:

```
public class Cookie {

  public Cookie() {
    System.out.println("Obiekt Cookie skonstruowany");
  }
```

```
  protected void bite() {
    System.out.println("metoda bite");
  }
}
```

powodujemy, że metoda `bite()` wciąż ma „pakietowy" stopień dostępu wewnątrz pakietu `dessert`, ale równocześnie jest dostępna dla każdej klasy dziedziczącej po `Cookie`. *Nie jest* jednak publiczna.

Interfejs i implementacja

Kontrola dostępu jest często nazywana *ukrywaniem implementacji*. Grupowanie danych i metod w klasy w połączeniu z ukrywaniem danych jest często nazywane *hermetyzacją* lub *enkapsulacją* (ang. *encapsulation*)[4]. W rezultacie otrzymujemy typ danych o pewnej charakterystyce i możliwych zachowaniach.

Kontrola dostępu wprowadza rozgraniczenia wewnątrz takiego typu z dwóch istotnych powodów. Pierwszym jest określenie, czego programista-klient powinien używać, a czego nie. Możemy dzięki temu wbudować nasze wewnętrzne mechanizmy w tworzoną strukturę bez przejmowania się tym, iż klient przypadkowo potraktuje „wnętrzności" jako część interfejsu, którego ma używać.

Łączy się to bezpośrednio z drugim powodem — chęcią oddzielenia interfejsu od implementacji. Jeżeli nasza struktura jest wykorzystywana w pewnym zbiorze programów, jednak programiści-klienci nie mogą zrobić niczego poza wysyłaniem komunikatów do publicznego interfejsu, wtedy możemy zmienić wszystko, co *nie* jest publiczne (na przykład jest dostępne w ramach pakietu, chronione lub prywatne) bez konieczności zmian w kodzie klientów.

Znajdujemy się obecnie w świecie programowania zorientowanego obiektowo, gdzie słowo `class` opisuje w rzeczywistości „klasę obiektów", tak jak opisalibyśmy klasę ryb lub klasę ptaków. Każdy obiekt należący do takiej klasy ma tę samą charakterystykę i wspólne zachowania. Klasa opisuje, jak obiekty danego typu będą wyglądać i działać.

W pierwszym języku zorientowanym obiektowo, Simuli-67, słowo kluczowe `class` było używane do opisania nowego typu danych. To samo słowo jest używane w większości języków obiektowych. Stanowi to najistotniejsze zagadnienie języka: tworzenie nowych typów danych będących czymś więcej niż pudełkami zawierającymi dane i metody.

Klasa jest fundamentalnym pojęciem programowania zorientowanego obiektowo w Javie. Słowo kluczowe `class` nie będzie wyróżniane dalej w książce — byłoby to zbyt irytujące w przypadku czegoś powtarzanego tak często.

Dla czytelności można przyjąć styl tworzenia klas polegający na umieszczeniu najpierw składowych publicznych, potem chronionych, „przyjaznych", i na końcu prywatnych. Zaletą takiego rozwiązania jest to, iż użytkownik klasy może wtedy — czytając od góry

[4] Niektórzy określają tym terminem samo ukrywanie implementacji.

do dołu — zobaczyć na początku to, co jest dla niego istotne (składowe publiczne, ponieważ ma do nich dostęp spoza pliku), a następnie przerwać czytanie, gdy napotka składniki niepubliczne będące częścią wewnętrznej implementacji:

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

Metoda ta czyni kod tylko częściowo prostszym w czytaniu, ponieważ interfejs i implementacja są wciąż ze sobą wymieszane, tzn. kod źródłowy — implementacja — znajduje się wewnątrz klasy. Dokumentacja tworzona na podstawie komentarzy, generowana z użyciem narzędzia javadoc (opisanego w rozdziale 2.), zmniejsza dodatkowo znaczenie czytelności kodu dla programisty-klienta. Wyświetlanie interfejsu dla użytkownika klasy jest tak naprawdę zadaniem *przeglądarki klas* (ang. *class browser*) — narzędzia potrafiącego przyjrzeć się wszystkim dostępnym klasom i pokazać w użyteczny sposób, co można z nimi zrobić (tj. jakie ich składniki są dostępne). W chwili ukazania się tej książki przeglądarki takie powinny być już niemal obowiązkowym elementem każdego narzędzia programistycznego Javy.

Dostęp do klas

W Javie możliwe jest również użycie modyfikatorów dostępu w celu określenia, które klasy w bibliotece będą dostępne dla użytkowników tej biblioteki. Jeżeli chcemy, aby klasa była dostępna dla programisty-klienta, wtedy umieszczamy słowo kluczowe `public` gdzieś przed jej definicją. Konstrukcja taka decyduje, że klient będzie w ogóle w stanie stworzyć obiekt takiej klasy.

Aby udzielić dostępu do klasy, specyfikator musi pojawić się przed słowem kluczowym `class`. Możemy zatem napisać:

```
public class Widget {
```

Teraz, zakładając, iż nasza biblioteka nazywa się `mylib`, każdy klient może zacząć używać klasy `Widget`, pisząc:

```
import mylib.Widget;
```

lub

```
import mylib.*;
```

Nałożono jednak szereg dodatkowych ograniczeń:

1. W jednostce kompilacji (pliku) może znajdować się tylko jedna klasa publiczna. Idea polega na tym, że każda jednostka kompilacji ma jeden publiczny interfejs

reprezentowany przez klasę publiczną. Może posiadać dowolnie wiele pomocniczych klas dostępnych w obrębie pakietu. Jeżeli jednak w jednostce kompilacji umieścimy więcej niż jedną klasę publiczną, kompilator zaprotestuje.

2. Nazwa klasy publicznej musi dokładnie pasować do nazwy pliku zawierającego jednostkę kompilacji, łącznie z wielkością liter. Tak więc dla klasy `Widget` plik musi nazywać się *Widget.java*, nie zaś *widget.java* czy *WIDGET.java*. Także w tym przypadku otrzymamy błąd kompilacji, jeśli na to nie przystaniemy.
3. Możliwe, choć nietypowe, jest posiadanie jednostki kompilacji niezawierającej żadnej klasy publicznej. W takim przypadku nazwa pliku może być dowolna.

Co się dzieje, gdy wewnątrz biblioteki `mylib` mamy jakąś klasę wykorzystywaną jedynie do osiągnięcia celu realizowanego przez `Widget` lub inną publiczną klasę z `mylib`? Nie chcemy zajmować się przygotowaniem jej dokumentacji dla programisty-klienta, a poza tym wydaje nam się, iż kiedyś być może wszystko kompletnie zmienimy albo w ogóle pozbędziemy się tej klasy, zastępując ją inną. Aby zapewnić sobie taką elastyczność, musimy upewnić się, iż żaden klient nie uzależni się od szczegółów implementacyjnych ukrytych wewnątrz `mylib`. W tym celu po prostu nie umieszczamy słówka `public` przed deklaracją klasy (przez co staje się ona widoczna jedynie z wnętrza pakietu).

Tworząc klasę dostępną w obrębie pakietu, wciąż warto niektóre z jej pól uczynić prywatnymi — pola zawsze powinny być jak najbardziej „prywatne"; jeśli natomiast chodzi o metody, to przeważnie rozsądnym rozwiązaniem jest stosowanie tego samego dostępu do nich jak klas, w których się znajdują (czyli dostępu pakietowego). Klasy dostępne w obrębie pakietu są zazwyczaj używane wyłącznie wewnątrz niego, a zatem metody tych klas należy definiować jako publiczne wyłącznie w razie konieczności, a w tych przypadkach kompilator nas o tym poinformuje.

Zauważmy, że klasa nie może być prywatna (uczyniłoby to ją niedostępną dla wszystkich z wyjątkiem jej samej) lub chroniona[5]. Tak więc w kwestii stopnia dostępności klasy istnieją jedynie dwie możliwości: klasa dostępna w obrębie pakietu i klasa „publiczna". Jeżeli nie chcemy, aby ktokolwiek miał dostęp do klasy, wtedy możemy uczynić wszystkie konstruktory prywatnymi, uniemożliwiając tym samym wszystkim, poza nami, utworzenie obiektu naszej klasy wewnątrz statycznej składowej klasy. Oto przykład:

```
//: c05:Lunch.java
// Demonstracja modyfikatorów dostępu do klas.
// Tworzenie klasy efektywnie prywatnej z wykorzystaniem
// prywatnych konstruktorów:

class Soup {
  private Soup() {}
  // (1) Umożliwienie tworzenia obiektu poprzez metodę
  // statyczną:
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Stworzenie statycznego obiektu i zwracanie na
  // żądanie referencji do niego.
```

[5] Tak naprawdę *klasy wewnętrzne* mogą być prywatne lub chronione, jednak jest to przypadek specjalny. Zostanie to opisane w rozdziale 7.

```
  // (Wzorzec projektowy "Singleton"):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}

class Sandwich { // Wykorzystuje klasę Lunch
  void f() { new Lunch(); }
}

// W pliku możliwa jest tylko jedna klasa publiczna:
public class Lunch {
  void test() {
    // Tego nie wolno! Konstruktor jest prywatny:
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

Do tej pory metody zwracały jedynie `void` lub typy proste, dlatego definicja:

```
public static Soup access() {
    return ps1;
  }
```

może się z początku wydawać nieco dziwna. Słowo znajdujące się przed nazwą metody (`access`) mówi, co metoda zwraca. Do tej pory było to zwykle słowo `void` oznaczające, iż nic nie jest zwracane. Możemy jednak również zwracać referencję do obiektu, co właśnie ma miejsce w powyższym przykładzie. Ta metoda zwraca referencję do obiektu klasy `Soup`.

Klasa `Soup` pokazuje, jak uniknąć bezpośredniego tworzenia klasy poprzez uczynienie wszystkich jej konstruktorów prywatnymi. Należy pamiętać, że jeżeli nie stworzymy jawnie choćby jednego konstruktora, wtedy zostanie dla nas wyprodukowany konstruktor domyślny (niepobierający argumentów). Dzięki napisaniu konstruktora domyślnego zapobiegamy jego automatycznemu utworzeniu. Czyniąc go prywatnym, uniemożliwiamy wszystkim utworzenie obiektu naszej klasy. Jak w takim razie ktoś może ją wykorzystać? Wcześniejszy przykład pokazuje dwie możliwości. Po pierwsze, obiekt klasy `Soup` jest tworzony wewnątrz statycznej metody tej klasy, a następnie metoda taka zwraca referencję do utworzonego obiektu. Może to być użyteczne, jeżeli chcemy wykonać jakieś dodatkowe operacje na obiekcie przed zwróceniem referencji do niego lub jeśli chcemy zliczać stworzone obiekty typu `Soup` (np. w celu ograniczenia ich populacji).

Druga opcja polega na wykorzystaniu jednego z tzw. *wzorców projektowych* (ang. *design patterns*) — opisane są one w książce *Thinking in Patterns (in Java)*, którą można ściągnąć z witryny *www.BruceEckel.com*. Ten szczególny użyty tutaj wzorzec nazywany jest „singletonem", ponieważ polega na umożliwieniu stworzenia tylko jednego obiektu klasy. Obiekt typu `Soup` jest tworzony jako statyczny i prywatny składnik klasy `Soup`, a zatem powstaje w ten sposób jeden i tylko jeden egzemplarz. Następnie możemy go otrzymać poprzez wywołanie publicznej metody `access()`.

Jak już wspomniałem, jeżeli nie podamy żadnego modyfikatora dostępu do klasy, przyjmowany jest domyślny dostęp pakietowy. Oznacza to, iż obiekt naszej klasy może zostać stworzony przez każdą inną klasę pakietu, jednak nie poza pakietem.

(Należy pamiętać, że pliki znajdujące się w tym samym katalogu, nieposiadające jawnego określenia pakietu stają się niejawnie częścią pakietu domyślnego dla tego katalogu). Jeżeli jednak statyczny składnik klasy jest publiczny, wtedy programista-klient może nadal mieć dostęp do tego składnika, mimo iż nie może stworzyć obiektu klasy.

Podsumowanie

W każdej relacji istotna jest obecność rozgraniczeń respektowanych przez wszystkie uczestniczące w niej strony. Tworząc bibliotekę, ustanawiamy relację z jej użytkownikiem — programistą, który próbuje napisać aplikację lub stworzyć większą bibliotekę z wykorzystaniem naszej.

Gdybyśmy nie ustanowili reguł, klient mógłby zrobić wszystko z każdą składową klas, nawet jeśli wolelibyśmy, aby nie manipulował bezpośrednio którąś z nich. Wszystko byłoby odsłonięte przed światem.

Rozdział ten pokazał, jak z klas formuje się biblioteki — po pierwsze, w jaki sposób grupa klas jest pakowana w bibliotekę, i po drugie, w jaki sposób klasa kontroluje dostęp do swoich składników.

Ocenia się, że projekt programistyczny prowadzony w języku C zaczyna załamywać się gdzieś pomiędzy 50 a 100 tysiącami wierszy kodu, ponieważ C posiada pojedynczą „przestrzeń nazw", a zatem nazwy zaczynają wchodzić w kolizje, powodując dodatkowy narzut związany z zarządzaniem nimi. W Javie słowo kluczowe `package`, schemat nazywania pakietów oraz instrukcja `import` dają nam całkowitą kontrolę nad nazwami, zatem problem ich kolizji nie występuje.

Istnieją dwa powody wprowadzenia kontroli dostępu do składowych klas. Pierwszym jest trzymanie użytkowników z daleka od narzędzi, których nie powinni dotykać — niezbędnych dla wewnętrznych działań typu danych, nie będących jednak częścią interfejsu, który klienci powinni wykorzystywać w celu rozwiązywania konkretnych problemów. Uczynienie metod i pól prywatnymi służy więc użytkownikom, ponieważ pozwala im na łatwe odróżnienie tego, co jest dla nich istotne, od tego, co mogą zignorować. Ułatwia im zrozumienie klasy.

Drugim z najważniejszych powodów wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki zmiany wewnętrznego sposobu działania klasy bez przejmowania się wpływem, jaki będzie to miało na programistę-klienta. Możemy zbudować najpierw klasę w jakiś sposób, a następnie odkryć, iż zmiana struktury kodu przyniesie znacznie większą szybkość. Jeżeli interfejs i implementacja są od siebie wyraźnie oddzielone i chronione, można to osiągnąć bez zmuszania użytkowników do przepisywania ich kodu.

Modyfikatory dostępu umożliwiają twórcy klasy ważną kontrolę. Użytkownicy wyraźnie widzą, czego wolno im używać, a co mają ignorować. Ważniejsza nawet jest jednak możliwość zapewnienia, że użytkownik nie uzależni się od żadnej części wewnętrznej implementacji klasy. Jeżeli posiadamy taką gwarancję jako twórcy klasy, wtedy możemy zmienić tę implementację, wiedząc, iż nie będzie to miało wpływu na żadnego programistę-klienta, ponieważ nie mają oni dostępu do zmienianego fragmentu klasy.

Dzięki możliwości zmiany wewnętrznej implementacji możemy nie tylko ulepszać nasz projekt w późniejszym czasie, ale zyskujemy także prawo popełniania błędów. Błędy zostaną popełnione bez względu na to, jak starannie planujemy i projektujemy. Wiedząc, iż popełnianie błędów jest stosunkowo bezpieczne, stajemy się bardziej skorzy do eksperymentowania, szybciej się uczymy i szybciej kończymy nasze zadanie.

Publiczny interfejs jest tym, *co widzi* użytkownik klasy, dlatego w jego przypadku najważniejsze jest poprawne zdefiniowanie go już w fazach analizy i projektowania. Nawet tu jednak mamy pewien margines dla zmian. Jeśli zdefiniowany za pierwszym interfejs nie jest dobry, możemy bezpiecznie *dodawać* do niego metody, jeżeli tylko nie usuwamy żadnej, która była już wykorzystana w kodzie programistów-klientów.

Ćwiczenia

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.BruceEckel.com*.

1. Napisz program tworzący obiekt typu `ArrayList` bez jawnego importowania pakietu `java.util.*`.
2. W części zatytułowanej „Pakiet — jednostka biblioteczna” zmień fragmenty kodu dotyczące pakietu `mypackage` w zestaw kompilujących i wykonujących się plików Javy.
3. W części zatytułowanej „Kolizje” zmień fragmenty kodu w program i sprawdź, czy kolizje rzeczywiście występują.
4. Uogólnij klasę `P`, zdefiniowaną w tym rozdziale, poprzez dodanie przeciążonych wersji metod `rint()` i `rintln()` niezbędnych do obsługi wszystkich podstawowych typów Javy.
5. Stwórz klasę z publicznymi, prywatnymi, chronionymi i „przyjaznymi” składnikami danych i metodami. Stwórz obiekt tej klasy i zaobserwuj, jakie komunikaty kompilatora otrzymasz, jeżeli spróbujesz dostać się do każdej ze składowych. Bądź świadom, iż klasy w tym samym katalogu są częścią „pakietu domyślnego”.
6. Stwórz klasę zawierającą dane chronione. Stwórz drugą klasę w tym samym pliku, zawierającą metodę manipulującą chronionymi danymi pierwszej z klas.
7. Zmień klasę `Cookie` tak, jak zalecono w części zatytułowanej „protected: dostęp „na potrzeby” dziedziczenia”. Sprawdź, że metoda `bite()` nie jest publiczna.

8. W części zatytułowanej „Dostęp do klas” znajdziemy fragmenty kodu opisujące bibliotekę `mylib` i klasę `Widget`. Stwórz opisaną bibliotekę, a następnie stwórz obiekt typu `Widget` wewnątrz klasy nie będącej częścią pakietu `mylib`.
9. Utwórz nowy katalog, a następnie dokonaj edycji zmiennej `CLASSPATH` w celu włączenia do niej tego katalogu. Skopiuj plik *P.class* (wyprodukowany w wyniku kompilacji *com.bruceeckel.tools.P.java*) do nowo powstałego katalogu, a następnie zmień nazwy: pliku, klasy `P` w jego wnętrzu oraz nazwy metod. Stwórz kolejny program w innym katalogu, wykorzystujący tę nową klasę.
10. Wykorzystując przykład *Lunch.java*, stwórz klasę o nazwie `ConnectionManager`, zarządzającą ustaloną tablicą obiektów typu `Connection`. Klient nie może mieć możliwości bezpośredniego stworzenia obiektu `Connection`, w zamian powinien otrzymywać je poprzez wywołanie statycznej metody klasy `ConnectionManager`. Gdy klasie `ConnectionManager` zabraknie obiektów `Connection`, powinna zwracać wartość `null`. Przetestuj stworzone klasy w metodzie `main()`.
11. Stwórz następujący plik w katalogu *c05/local* (zakładamy, iż znajduje się on w zmiennej `CLASSPATH`):

```
///: c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
  public PackagedClass() {
    System.out.println(
      "Tworzenie klasy należącej do pakietu");
  }
} ///:~
```

Następnie stwórz następujący plik w katalogu innym niż *c05*:

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```

Wyjaśnij, dlaczego kompilator generuje błąd. Czy uczynienie klasy `Foreign` częścią pakietu `c05.local` coś zmieni?