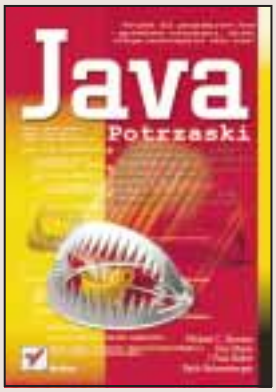
Java. Potrzaski



Autorzy: Michael C. Daconta, Eric Monk, J Paul Keller, Keith Bohnenberger
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-121-5
Tytuł oryginału: Java Pitfalls
Format: B5, stron: 310
Przykłady na ftp: 68 kB

Choć Java to język gwarantujący efektywną pracę, to jednak kryje w sobie wiele pułapek, które mogą zniweczyć jej efekty. Książka ta ma za zadanie oszczędzić Twój czas i zapobiec frustracji przeprowadzając Cię bezpiecznie przez skomplikowane zagadnienia. Zespół ekspertów od języka Java pod wodzą guru programowania w osobie Michaela Daconta proponuje Ci zestaw sprawdzonych rozwiązań 50 trudnych problemów pojawiających się w praktyce każdego programisty. Rozwiązania te pozwolą Ci uniknąć problemów wynikających z niedostatków języka Java oraz jego interfejsów programowych, w tym pakietów java.util, java.io, java.awt i javax.swing. Autorzy dzielą się także z Czytelnikiem swoimi sposobami na poprawę wydajności aplikacji pisanych w Javie.

Oto niektóre z omawianych zagadnień:

- Składnia języka: zastosowanie metody equals() zamiast operatora == do porównywania obiektów klasy String
- Funkcjonalność wbudowana w język: rozdział metod a mechanizm refleksji, interfejsy i klasy anonimowe
- Użyteczne klasy i kolekcje: wybór klasy PropertyFile i ResourceBundle
- Wejście i wyjście, w tym subtelności związane z przesyłaniem serializowanych obiektów za pomocą gniazd sieciowych
- Graficzny interfejs użytkownika: sposoby uniknięcia typowej pułapki polegającej na zastosowaniu metody repaint() zamiast metody validate() w celu uzyskania nowego układu komponentów
- Graficzny interfejs użytkownika – sterowanie: m.in. bardziej funkcjonalna kontrola danych wprowadzanych przez użytkownika
- Wydajność: m.in. zastosowanie odroczonego ładowania, tak by zwiększyć szybkość uruchamiania programów

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

Spis treści

Rozdział 3.

Użyteczne klasy i kolekcje

W językach C++ i Java implementacja list jest dostępna w standardowych bibliotekach, co nie zwalnia nas od znajomości sposobów ich stosowania.

— Brian W. Kernighan i Rob Pike „The Practice Of Programming"

Aby z powodzeniem używać jakiekolwiek klasy, należy dobrze znać sposób, w jaki zostały zaprojektowane ich zadania i ograniczenia. Wymaga to czasu i doświadczenia. Dlatego, aby zaoszczędzić Czytelnikowi czas przedstawiamy w niniejszym rozdziale własne doświadczenia — niektóre klasy pakietu `java.util` ujęte w pięć następujących zagadnień:

Zagadnienie 18. Uporządkowane klucze właściwości? — opisuje sposób udoskonalenia klasy `Properties` dający uporządkowany zbiór właściwości.

Zagadnienie 19. Obsługa kolekcji o znacznych rozmiarach za pomocą mechanizmów buforowania i trwałości — szczegółowo omawia sposób implementacji kolekcji wykorzystujący mechanizmy buforowania i trwałości stanowiącej rozwiązanie rzeczywistego problemu zaczerpniętego z praktyki programistycznej.

Zagadnienie 20. Plik właściwości czy zestaw zasobów? — przedstawia różnice pomiędzy rozwiązaniami wymienionymi w tytule. Mimo że pliki właściwości stanowią często stosowane rozwiązanie, to nie zawsze jest ono optymalne. W zagadnieniu przedstawiono przykład ilustrujący taką sytuację.

Zagadnienie 21. Pułapki klasy Properties — stanowi dokładny przegląd zagadnień i pułapek związanych z zastosowaniem właściwości do przechowywania informacji o konfiguracji programów. W szczególności omawia sposoby umożliwiające pracę aplikacji w różnych systemach niezależnie od sposobu jej instalacji.

Zagadnienie 22. Klasa Vector i nowe kolekcje — szczegółowo opisuje modyfikacje klasy `Vector` na skutek włączenia jej do szkieletu kolekcji udostępnionego w nowszych wersjach języka Java. Zagadnienie ma zachęcić Czytelnika do korzystania z nowego szkieletu kolekcji.

Zagadnienie 18. Uporządkowane klucze właściwości?

Załóżmy, że Czytelnik wrócił właśnie z udanych wakacji i zamierza pochwalić się rodzinie zdjęciami, które zrobił nowym aparatem. Odkurzając rzutnik i zawieszając ekran, zaczyna się jednak zastanawiać, czy taka technologia prezentacji zdjęć nie jest już przestarzała? Zamiast niej decyduje się napisać program w języku Java, który będzie zarządzać pokazem zdjęć. Łącząc umiejętności programowania z cyfrowymi zdjęciami, można zrobić na rodzinie jeszcze większe wrażenie.

Program jest skończony i nadszedł właśnie długo oczekiwany dzień pokazu. Wujek Bob i inni zasiedli wokół komputera. Zostaje uruchomiony program, który wyświetla tytuł pokazu „Wycieczka na Grenadę, wyspę pieprzu".

Autor zdjęć rozpoczyna opowieść o wakacjach, mówiąc: „Następne zdjęcie obrazuje moment, gdy z Patty wsiadamy do samolotu". Jednak po naciśnięciu klawisza, które to wyświetli, pojawia się zdjęcie twoich przyjaciół, Drew i Suzy, w hotelu. Okazuje się, że zdjęcia zupełnie się pomieszały. Gdy wujek Bob zaczyna rzucać pomidorami w ekran monitora, łatwo się domyśleć, że pokaz nie udał się. Jednak zostaje złożone postanowienie dopracowania programu, aby spróbować jeszcze raz.

Każdy byłby zdeterminowany, aby znaleźć błąd, który stał się przyczyną porażki. Kluczową koncepcją programu SlideShow jest abstrakcja ścieżek dostępu do zdjęć. Dzięki temu stworzenie kolejnego pokazu będzie wymagać jedynie zmiany pliku właściwości zawierającego ścieżki dostępu do wyświetlanych zdjęć. Rozwiązanie takie jest o wiele lepsze niż konieczność modyfikowania kodu źródłowego za każdym razem, gdy chce się stworzyć nową prezentację. Ponieważ w pliku właściwości oprócz ścieżek dostępu do poszczególnych zdjęć są zapisane także inne informacje, to trzeba przyjąć pewną konwencję pozwalającą ustalić, która właściwość dotyczy zdjęcia, a która nie. W tym celu wszystkie właściwości zdjęć poprzedzimy przedrostkiem `Image_`. Wydaje się to dobrym rozwiązaniem, jednak zdjęcia nie są wyświetlane we właściwej kolejności. Sprawdźmy zatem, czy plik właściwości zawiera rzeczywiście opis zdjęć w odpowiedniej kolejności. Zawartość pliku wygląda następująco:

```
imageRoot=c:\\images
title=Grenada Vacation
titlePage= Our Trip to Grenada, the Island of Spice
Image_departing=plane.jpg
Image_hotel=hotel.jpg
Image_friends=drew_suzy.jpg
Image_beach=beach.jpg
Image_return=patty_kirsten_keith.jpg
```

Ponieważ kolejność zdjęć w pliku właściwości jest prawidłowa, powodem ich przypadkowego wyświetlania musi być błąd w programie. Przyjrzyjmy się więc fragmentowi kodu, który jest odpowiedzialny za tworzenie obiektów `ImageIcon` na podstawie właściwości zdjęć zapisanych w pliku:

```
01:   public ImageIcon[] getImages()
02:   {
03:        ImageIcon[] images = null;
04:        ArrayList al = new ArrayList();
05:        String imageRoot = properties.getProperty("imageRoot");
06:        Enumeration keys = properties.keys();
07:        While(keys.hasMoreElements())
08:        {
09:             String key = (String)keys.nextElement();
10:             if (key.startsWith("Image_"))
11:             {
12:                ImageIcon icon = new ImageIcon(imageRoot + File.separator
13:                +  properties.getProperty(key));
14:                al.add(icon);
15:             }
16:        }
17:        int size = al.size();
18:        if (size > 0)
19:        {
20:             images = new ImageIcon[size];
21:             al.toArray(images);
22:        }
23:        return images;
24:   }
```

Metoda `getImages()` zwraca tablicę obiektów `ImageIcon`. Tablica ta jest wypełniana w miarę przeglądania kluczy właściwości i sprawdzania, czy rozpoczynają się one od przedrostka `Image_` (wiersz 10.). Jeśli klucz zawiera taki przedrostek, to tworzony jest obiekt klasy `ImageIcon` i wstawiany do tablicy klasy `ArrayList` (wiersze 12. – 14.). Po zakończeniu analizy pliku właściwości tablica `ArrayList` jest konwertowana do zwykłej tablicy obiektów klasy `ImageIcon` i zwraca jako wynik wykonania metody.

Wydaje się więc, że kod programu jest napisany prawidłowo. Jednak skoro także plik właściwości jest prawidłowy, to program powinien wyświetlać zdjęcia w odpowiedniej kolejności, a tak nie jest. Należy więc raz jeszcze sprawdzić kod programu, dodając poniższą instrukcję wewnątrz bloku instrukcji warunkowej, rozpoczynającej się w wierszu10.:

```
System.out.println("key  =   '" + key + "'");
```

Dzięki niej dowiemy się, czy pobieramy ścieżki dostępu do zdjęć w tej samej kolejności, w której zostały one zapisane w pliku właściwości. Uruchamiając program, uzyska się następującą informację:

```
key = 'Image_hotel'
key = 'Image_friends'
key = 'Image_departing'
key = 'Image_return'
key = 'Image_beach'
```

Widzimy więc, że kolejność uzyskiwania zdjęć różni się od kolejności ich występowania w pliku. Czy to oznacza, że metoda `keys()` klasy `Properties` posiada błędy? Odpowiedź na to pytanie jest krótka: nie. Źródłem problemu jest przyjęte założenie, że klucze obiektu `Properties` są widziane w tej samej kolejności, w której występują w pliku. Założenie to nie jest prawdziwe. Ponieważ klasa `Properties` stanowi klasę pochodną

klasy Hashtable, to klucze właściwości przechowywane są w tablicy mieszającej, a nie na uporządkowanej liście. Klasa Hashtable przyznaje każdemu kluczowi indeks zależny od wyniku funkcji mieszającej dla danego klucza i od bieżącego rozmiaru tablicy. Indeks ten nie zależy od kolejności, w której klucze są umieszczane w tablicy. Natomiast metoda keys() zwraca klucze zgodnie z numerycznym porządkiem ich indeksów. Dlatego też porządek ich oglądania może różnić się od kolejności, w której zostały umieszczone w tablicy mieszącej.

Czy oznacza to, że musimy umieścić informacje o zdjęciach w kodzie programu, aby uzyskać pożądaną kolejność ich wyświetlania? Na szczęście nie. Istnieją inne sposoby, dzięki którym można osiągnąć pożądany efekt, np. umieścić ścieżki dostępu do zdjęć w zwykłym pliku tekstowym nieposiadającym żadnej struktury i wczytywać jego zawartość. Ponieważ jednak program SlideShow korzysta także z innych właściwości, to musielibyśmy dostarczać mu dwóch różnych plików. Poza tym możliwość przeglądania kluczy obiektu Properties w kolejności, w której występują one w pliku właściwości, może okazać się przydatna w wielu innych zastosowaniach.

W jaki sposób zatem uzyskać uporządkowaną listę kluczy z obiektu Properties? Nie jest to możliwe. Możemy jednak stworzyć własną klasę pochodną klasy Properties, która zrealizuje to zadanie. Klasa ta będzie posiadać zmienną instancji, która przechowa uporządkowane klucze. Aby móc dodawać klucze i usuwać je z uporządkowanej listy, musimy także przesłonić metody put() i remove() własną implementacją oraz dodać metodę, za pomocą której będzie można uzyskać uporządkowaną listę kluczy.

Nową klasę nazwiemy EnhancedProperties (w zagadnieniu 21. przedstawiono szereg przydatnych metod tej klasy). Definicja klasy EnhancedProperties może wyglądać następująco:

```
01:  public class EnhancedProperties extends Properties
02:  {
03:       // konstruktory
04:
05:       ArrayList orderedKeys = new ArrayList();
05:
06:       public synchronized Object put(Object key, Object value)
07:       {
08:            Object object = super.put(key, value);
09:            orderedKeys.add(key);
10:            return object;
11:       }
12:
13:       public synchronized Object remove (Object key)
14:       {
15:            Object object = super.remove(key);
16:            orderedKeys.remove(key);
17:            return object;
18:       }
19:
20:       public synchronized Iterator getOrderedKeys()
21:       {
22:            return orderedKeys.iterator();
23:       }
24:  }
```

Zmienna instancji `orderedKeys` w wierszu 5. jest kontenerem, w którym przechowuje się uporządkowaną listę kluczy. Metoda `put()`, której definicja rozpoczyna się w wierszu 8., wywołuje najpierw metodę `super.put()`, która umieszcza klucz w tablicy mieszającej w sposób właściwy klasie `Properties`. Natomiast wywołanie `orderedKeys.add(key)` w wierszu 9. wstawia ten klucz na uporządkowaną listę kluczy. Wywołanie metody `put()` spowoduje za każdym razem dodanie klucza do tablicy mieszającej `Hashtable` i wstawienie go do uporządkowanej listy kluczy. Metoda `remove()`, której definicja rozpoczyna się w wierszu 13., działa w podobny sposób do metody `put()`. Za każdym razem, gdy wywołana jest metoda `remove()`, klucz zostaje najpierw usunięty z tablicy mieszającej, a następnie z uporządkowanej listy kluczy. Dostęp do tej listy umożliwia metoda `getOrderedKeys()`, która zwraca iterator czytający tę listę.

Ta dość prosta w implementacji klasa pozwoli przeglądać ścieżki dostępu do zdjęć dokładnie w tym samym porządku, w którym zostały one zapisane w pliku właściwości. Metoda `getImages()` tworząca tablicę obiektów klasy `ImageIcon` na podstawie właściwości, których klucz rozpoczyna się od przedrostka `Image_`, musi zostać zmodyfikowana w niewielkim stopniu. Wiersze 6. – 9. trzeba zatem zastąpić następującymi wierszami:

```
06:      Iterator orderedKeys = properties.getOrderedKeys();
07:      While(keys.hasNext())
08:      {
09:            String key = (String)keys.next();
```

Jedyną zmianą, której będziemy musieli wykonać w pozostałej części programu, będzie zastąpienie instancji klasy `Properties` tworzonej podczas wczytywania zawartości pliku właściwości za pomocą instancji nowej klasy `EnhancedProperties`. Po wykonaniu tych zmian i uruchomieniu programu uzyskamy następującą informację o dodawanych kluczach:

```
key = 'Image_departing'
key = 'Image_hotel'
key = 'Image_friends'
key = 'Image_beach'
key = 'Image_return'
```

Ścieżki dostępu do plików są teraz uporządkowane w odpowiedni sposób. Wiadomo już, że klasa `Properties` jest pochodną klasy `Hashtable`, a sposób przyznawania indeksów kluczom umieszczanym w tablicy mieszającej nie jest związany z kolejnością ich wstawiania. Dzięki temu można spokojnie zaprosić wujka Boba na powtórny, udany pokaz zdjęć.

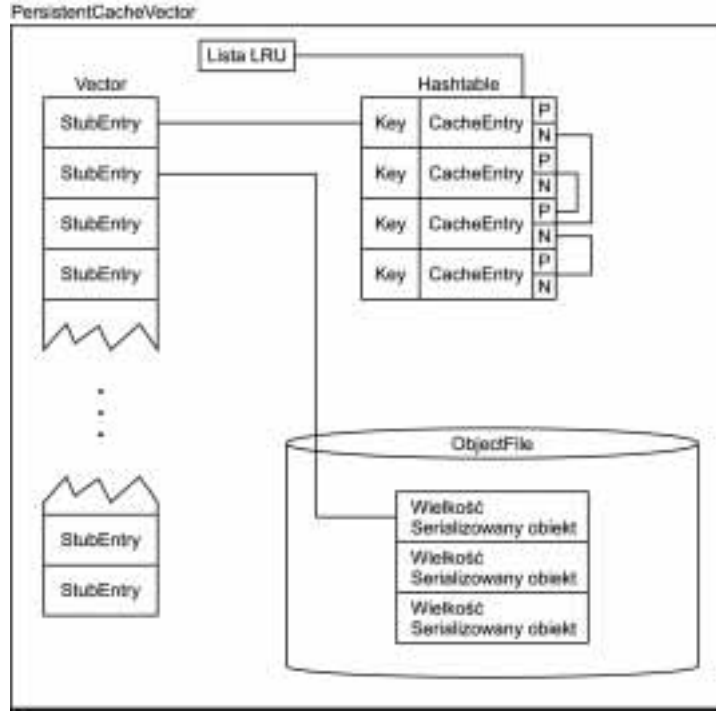
Zagadnienie 19. Obsługa kolekcji o znacznych rozmiarach za pomocą mechanizmów buforowania i trwałości

Czasami zdarza się, że program napisany w języku Java, który wyświetla wyniki zapytań wysyłanych do systemu baz danych, działa doskonale w 9 przypadkach na 10.

W tym jednym nie wyświetla żadnych efektów zapytania, sygnalizując jednocześnie wystąpienie błędu `OutOfMemory`. Wyniki zapytań są umieszczane w klasie `java.util.Vector`, która stanowi efektywną i wygodną metodę krótkotrwałego przechowywania danych. W jaki sposób można pogodzić efektywność tego rozwiązania z koniecznością zapewnienia jego niezawodności w przypadku sporadycznie pojawiających się wyników zapytań o znacznych rozmiarach? Rozwiązaniem będzie struktura danych stanowiąca kombinację bufora LRU (*Least Recently Used*) z trwałym składem obiektów. Jego implementację stanowić będzie klasa `PersistentCacheVector`, którą omówimy w tym zagadnieniu.

Rysunek 3.1 przedstawia strukturę klasy `PersistentCacheVector` z zaznaczeniem, że składa się ona z czterech głównych komponentów: wektora proxy zawierającego namiastki, tablicy mieszającej buforowanych obiektów, listy LRU i obiektu klasy `ObjectFile` umożliwiającego serializację obiektów.

Rysunek 3.1. *Architektura klasy PersistentCacheVector*



Architektura przedstawiona na rysunku 3.1 posiada następujące zalety:

- Prostota użycia podobna do klasy `java.util.Vector`.
- Możliwość osiągania znacznych rozmiarów przez kolekcję (na przykład ponad 50 000 elementów) bez obawy wystąpienia błędu `OutOfMemoryError`.
- Dostępność najczęściej wykorzystywanych elementów kolekcji w pamięci.
- Szczegóły implementacji związane z obsługą kolekcji znacznych rozmiarów nie są widoczne dla użytkownika klasy.

Architekturę tę będzie implementować kod umieszczony w dwóch plikach źródłowych: *PersistentCacheVector.java* i *ObjectFile.java*. Najpierw zostaną omówione najistotniejsze ich fragmenty, a pełny kod źródłowy — pod koniec bieżącego zagadnienia. Zanim przejdziemy do analizy kodu, przyjrzyjmy się zaproponowanemu interfejsowi klasy `PersistentCacheVector`. Posiada on następujące metody o dostępie publicznym:

```
public PersistentCacheVector(int iCacheSize);
public final void add(Serializable o) throws IOException;
public final Object get(int idx) throws IndexOutofBoundsException,
                                        IOException;
public final Object remove(int index) throws IndexOutOfBoundsException,
                                             IOException;
public final void copyTo(Object oArray []) throws IOException;
public final void close();
public final void finalize();
public final int size();
```

Pomiędzy interfejsem klasy `PersistentCacheVector` i metodami klasy `Vector` występują cztery istotne różnice:

Konstruktor klasy `PersistentCacheVector` wymaga określenia maksymalnego rozmiaru bufora. Inaczej niż w przypadku klasy `Vector`, konstruktorowi której przekazujemy początkowy rozmiar wektora, dla klasy `PersistentCacheVector` parametr konstruktora oznacza maksymalną liczbę obiektów, które mogą być przechowywane w pamięci. Obiekty nadmiarowe będą przechowane na dysku.

Niektóre z metod wyrzucają wyjątek `IOException` ze względu na przechowywanie obiektów w plikach. Z punktu widzenia zgodności z klasą `Vector` najlepiej obsługiwać ten wyjątek wewnątrz metod, zamiast przekazywać go do kodu wywołującego metody. Jednak wystąpienie błędu dysku spowodowałoby wtedy nieprzewidziane zachowanie klasy `Vector`. Druga możliwość polega na przechwyceniu wyjątku `IOException` i wyrzuceniu wyjątku `RuntimeException`, który nie musi być obsługiwany. W ten sposób również uzyskalibyśmy przezroczystość fasady klasy `Vector`, gdyż kod wywołujący metody nie musiałby obsługiwać żadnych wyjątków.

Pominięte zostały niektóre metody dostępne w klasie `Vector`. Ich implementację pozostawiono Czytelnikowi jako ćwiczenie do wykonania. Inna możliwość udawania klasy `Vector` polega na utworzeniu jej klasy pochodnej i przesłonięciu wszystkich metod klasy bazowej. Dzięki temu będziemy mogli używać obiektów klasy `PersistentCacheVector` zamiast obiektów klasy `Vector`, jednak w istotny sposób zwiększy to rozmiary implementacji klasy `PersistentCacheVector`.

Parametrem metody `add()` mogą być tylko obiekty implementujące interfejs `Serializable`. Ograniczenie to wynika z konieczności zapewnienia trwałości buforowanych obiektów i oczywiście nie występuje ono w klasie `Vector`. Dla zapewnienia zgodności z klasą `Vector` metoda `add()` mogłaby przyjmować dowolne obiekty, a następnie za pomocą refleksji sprawdzać możliwość ich serializacji. Jednak wydaje się, że lepiej zapewnić kontrolę zgodności typów parametrów kosztem pełnej zgodności z klasą `Vector`.

Implementacja interfejsu klasy PersistentCacheVector wymaga zarządzania strukturami przedstawionymi na rysunku 3.1: wektorem proxy zawierającym obiekty namiastek StubEntry, tablicą mieszającą obiektów CacheEntry, listą LRU i obiektem ObjectFile. Zagadnienia te omówimy szczegółowo.

Klasa wewnętrzna StubEntry stwarza iluzję korzystania z obiektów klasy Vector. Użytkownik spodziewa się zwykłego obiektu klasy Vector, w którym będzie umieszczać swoje obiekty. Jednak w rzeczywistości będą umieszczane tam instancje klasy StubEntry wskazujące, gdzie znajdują się właściwe obiekty. Natomiast właściwe obiekty zostaną umieszczone w buforze lub na dysku (w obiekcie ObjectFile). Klasa StubEntry posiada tylko dwie składowe: znacznik informujący o tym, czy obiekt znajduje się w buforze oraz indeks dostępu do obiektu znajdującego się na dysku:

```
40:    class StubEntry
41:    {
42:       /** Znacznik informujący czy element znajduje się w buforze czy na dysku.
*/
43:       boolean inCache;
44:       /** Wskaźnik pozycji pliku, gdy element znajduje się na dysku. */
45:       long filePointer = -1;
46:    }
```

Klasa wewnętrzna CacheEntry wykorzystywana jest do przechowywania obiektów użytkownika w buforze. Umieszcza także klucz w tablicy mieszającej, w której będzie przechowywany obiekt oraz referencje poprzedniego i następnego elementu listy LRU. Każdy obiekt klasy CacheEntry jest umieszczany w tablicy mieszającej Hashtable przy zastosowaniu klucza, który przechowywany jest także wewnątrz danego obiektu klasy CacheEntry. Efektywność takiego rozwiązania zmniejsza nieco fakt, że klucz tablicy mieszającej musi być obiektem, gdy w rzeczywistości jest on zwykłym indeksem obiektu StubEntry w tablicy. Użycie indeksów tablicy jako kluczy zapewnia przy tym doskonały, bezkolizyjny wynik funkcji mieszającej. Inna, nieco bardziej efektywna możliwość polegałaby na przechowywaniu kluczy w postaci obiektów klasy Integer w obiektach klasy StubEntry. W ten sposób zostałaby ograniczona liczba generowanych kluczy. Umieszczenie w obiektach klasy CacheEntry referencji poprzedniego i następnego elementu listy sprawia, że pełnią one podwójna funkcję elementu tablicy mieszającej i elementu dwukierunkowej listy LRU.

```
51:    class CacheEntry
52:    {
53:       /** Klucz elementu bufora. Stosujemy obiekt Integer
54:           odpowiadający indeksowi elementu wektora. */
55:       Integer key;
56:       /** Obiekt przechowywany w buforze. */
57:       Object o;
58:       /** Referencje poprzedniego i następnego elementu listy. */
59:       CacheEntry prev, next;
60:    }
```

Dostęp do dwukierunkowej listy LRU jest możliwy dzięki referencjom znajdującym się w klasie PersistentCacheVector. Jedna z nich wskazuje początek listy (firstCacheEntry), a druga jej koniec (lastCacheEntry). Zadaniem listy dwukierunkowej jest określenie najrzadziej wykorzystywanego elementu bufora. Za każdym razem, gdy korzystamy z pewnego elementu bufora, zostaje on przesunięty na początek listy. W ten sposób

ostatni element listy reprezentuje najrzadziej używany element. Większość kodu klasy PersistentCacheVector właśnie zarządza listą dwukierunkową. Poniżej zaprezentowano fragment umieszczający instancję klasy CacheEntry (o nazwie ce) na liście LRU. Działa on w następujący sposób: jeśli lista nie jest pusta, to umieszcza nowy element na jej początku i w nim referencje obiektu, który dotychczas stanowił czoło listy jako referencję następnego obiektu listy. Po czym referencję obiektu ce umieszcza w obiekcie, który dotychczas stanowił czoło listy jako referencję poprzedniego elementu listy. Na koniec nadaje nową wartość referencji wskazującej początek listy. Natomiast w przypadku, gdy lista jest pusta, inicjuje referencje początku i końca listy za pomocą referencji nowego elementu.

```
108:          // wstawia element do listy
109:          if (lastCacheEntry != null)
110:          {
111:             // umieszcza go na początku listy
112:             ce.next = firstCacheEntry;
113:             firstCacheEntry.prev = ce;
114:             firstCacheEntry = ce;
115:          }
116:          else
117:          {
118:             // lista jest pusta
119:             firstCacheEntry = lastCacheEntry = ce;
120:          }
```

Klasa ObjectFile przechowuje serializowane obiekty w pliku o dostępie swobodnym, reprezentowanym przez klasę RandomAccessFile. Serializowany obiekt jest przechowywany w postaci tablicy bajtów. Aby zapisać tablicę bajtów za pomocą obiektu klasy RandomAccessFile nie są potrzebne żadne dodatkowe dane. Jednak, aby odczytać tablicę bajtów z pliku, musimy znać jej wielkość. Dlatego też w pliku jest zapisywana najpierw wartość całkowita, a następnie tablica bajtów. Wartość ta określa liczbę bajtów tablicy. Klasa ObjectFile zawiera instancję klasy RandomAccessFile i implementuje metody zapisu i odczytu obiektów w omówionej postaci.

```
07: public class ObjectFile
08: {
09:    RandomAccessFile dataFile;
10:    String sFileName;
11:
12:    public ObjectFile(String sName) throws IOException
13:    {
14:       sFileName = sName;
15:       dataFile = new RandomAccessFile(sName, "rw");
16:    }
```

Dla potrzeb naszego przykładu klasa PersistentCacheArray posiada dodatkowo metodę main() umożliwiającą jej przetestowanie. Poniżej przedstawiamy efekt wykonania testów umieszczonych w tej metodzie:

```
adding 300 objects...
Size: 300
Testing get...
0,1,2,3,4,5,
Now the element at index 5 is : 6
```

```
Size is: 299
296,297,298,299,
1,2,3,4,5,Testing remove...
Size: 299
Removing 10
Size: 289
First 10.
2,4,6,8,10,12,14,16,18,20,
```

Oto pełny kod źródłowy klasy `PersistentCacheVector`:

```
001: /* PersistentCacheVector.java */
002: package jwiley.effective;
003:
004: import java.io.*;
005: import java.util.*;
006:
007: /**
008:  *  Odpowiednik klasy Vector używający bufora LRU i umieszczający
009:  *  na dysku nadmiarowe obiekty.
010:  */
011: public class PersistentCacheVector implements Cloneable, Serializable
012: {
013:    /** Licznik utworzonych obiektów. */
014:    static long lCount;
015:    /** Maksymalny rozmiar bufora. */
016:    int iMaxCacheSize;
017:    /** Bieżący rozmiar bufora. */
018:    int iCacheSize;
019:    /** Bufor. */
020:    Hashtable cache = new Hashtable();
021:    /** Odbicie klasy Vector. Przechowuje jedynie
022:        obiekt klasy VectorEntry dla każdego wstawianego obiektu. */
023:    Vector rows = new Vector();
024:    /** ObjectFile przechowuje obiekty na dysku.
025:        @see ObjectFile
026:    */
027:    ObjectFile of;
028:    /** Nazwa pliku, w którym przechowywane są obiekty. */
029:    String sTmpName;
030:    /** Początek listy LRU.  Pierwszy element reprezentuje
031:     najczęściej (ostatnio) używany element bufora. */
032:    CacheEntry firstCacheEntry;
033:    /** Koniec listy LRU.  Ostatni element reprezentuje.
034:     najrzadziej (ostatnio) używany element bufora. */
035:    CacheEntry lastCacheEntry;
036:
037:    /**
038:     * Klasa wewnętrzna namiastki.
039:     */
040:    class StubEntry
041:    {
042:       /** Znacznik informujący, czy element znajduje się w buforze czy na
dysku. */
043:       boolean inCache;
044:       /** Wskaźnik pozycji pliku, gdy element znajduje się na dysku. */
```

```
045:       long filePointer = -1;
046:    }
047:
048:    /**
049:     * Klasa wewnętrzna reprezentująca element bufora.
050:     */
051:    class CacheEntry
052:    {
053:       /** Klucz elementu bufora. Stosujemy obiekt Integer
054:           odpowiadający indeksowi elementu wektora. */
055:       Integer key;
056:       /** Obiekt przechowywany w buforze. */
057:       Object o;
058:       /** Referencje poprzedniego i następnego elementu listy. */
059:       CacheEntry prev, next;
060:    }
061:
062:    /**
063:     * Konstruktor określający rozmiar bufora. Plik nie jest otwierany,
064:     * dopóki nie zostanie przekroczony rozmiar bufora.
065:     * @param iCacheSize Maksymalny rozmiar bufora.
066:     */
067:    public PersistentCacheVector(int iCacheSize)
068:    {
069:       this.iMaxCacheSize = iCacheSize;
070:    }
071:
072:    private final void openTempFile() throws IOException
073:    {
074:       boolean bInValid = true;
075:
076:       while (bInValid)
077:       {
078:          sTmpName = "tmp" + (System.currentTimeMillis())
079:                           + (++lCount) + ".obf";
080:          File f = new File(sTmpName);
081:          if (!f.exists())
082:             bInValid = false;
083:       }
084:
085:       of = new ObjectFile(sTmpName);
086:    }
087:
088:    /**
089:     * Metoda wstawiania elementu do wektora. Element zostaje
090:     * umieszczony w buforze lub na dysku.
091:     * @param o Obiekt dodawany do wektora.
092:     */
093:    public final void add(Serializable o) throws IOException
094:    {
095:       StubEntry e = new StubEntry();
096:
097:       rows.add(e);
098:
099:       if (iCacheSize < iMaxCacheSize)
100:       {
101:          e.inCache = true;
```

```
102:          CacheEntry ce = new CacheEntry();
103:          ce.o = o;
104:          ce.key = new Integer(rows.size() - 1);
105:          cache.put(ce.key, ce);
106:          iCacheSize++;
107:
108:          // wstawia element do listy
109:          if (lastCacheEntry != null)
110:          {
111:             // umieszcza go na początku listy
112:             ce.next = firstCacheEntry;
113:             firstCacheEntry.prev = ce;
114:             firstCacheEntry = ce;
115:          }
116:          else
117:          {
118:             // lista jest pusta
119:             firstCacheEntry = lastCacheEntry = ce;
120:          }
121:       }
122:       else
123:       {
124:         if (of == null)
125:             openTempFile();
126:
127:         e.filePointer = of.writeObject(o);
128:       }
129:    }
130:
131:    /**
132:     * Metoda dostępu do elementu wektora o podanym indeksie.
133:     * Obiekt pobierany jest z bufora lub pliku.
134:     * @param idx Indeks pobieranego elementu.
135:     * @returns obiekt umieszczony w wektorze.
136:     */
137:    public final Object get(int idx) throws IndexOutOfBoundsException,
138:                                            IOException
139:    {
140:       if (idx < 0 || idx >= rows.size())
141:          throw new IndexOutOfBoundsException("Index: " + idx
142:                                              + " out of bounds.");
143:
144:       StubEntry e = (StubEntry) rows.get(idx);
145:
146:       Object o=null;
147:
148:       if (e.inCache)
149:       {
150:          // pobiera element
151:          CacheEntry ce = null;
152:          ce = (CacheEntry) cache.get(new Integer(idx));
153:
154:          if (ce == null)
155:             throw new IOException("Element at idx " + idx + " is NULL!");
156:
157:          if ((ce != null && ce.o == null))
```

```
158:             throw new IOException("Cache Element's object at idx "
159:                                   + idx + " NOT in cache!");
160:
161:          o = ce.o;
162:
163:          if (ce != firstCacheEntry)
164:          {
165:             // usuwa go z listy
166:             if (ce.next != null)
167:                ce.next.prev = ce.prev;
168:             else // ostatni element listy
169:                 lastCacheEntry = ce.prev;
170:
171:             ce.prev.next = ce.next;
172:
173:             // i wstawia na początek listy
174:             ce.next = firstCacheEntry;
175:             ce.prev = null;
176:             firstCacheEntry.prev = ce;
177:             firstCacheEntry = ce;
178:          }
179:       }
180:       else
181:       {
182:          // obiekt w buforze
183:          e.inCache = true;
184:
185:          // pobieranie i wstawianie do bufora
186:          try
187:          {
188:             o = of.readObject(e.filePointer);
189:          } catch (ClassNotFoundException cnfe)
190:            { throw new IOException(cnfe.getMessage()); }
191:
192:          //*** Sprawdza, czy bufor nie jest pełny!
193:          if (iCacheSize == iMaxCacheSize)
194:          {
195:             // usuwa element znajdujący się na końcu listy.
196:             CacheEntry leastUsed = lastCacheEntry;
197:             if (leastUsed.prev != null)
198:             {
199:                leastUsed.prev.next = null;
200:                lastCacheEntry = leastUsed.prev;
201:                lastCacheEntry.next = null;
202:             }
203:             else
204:             {
205:                // usuwa jedyny element listy
206:                firstCacheEntry = lastCacheEntry = null;
207:             }
208:
209:             // wstawia pobrany element do bufora
210:             CacheEntry ce = new CacheEntry();
211:             ce.o = o;
212:             ce.key = new Integer(idx);
213:             cache.put(ce.key, ce);
214:
```

```
215:             // umieszcza go na liście LRU
216:             if (lastCacheEntry != null)
217:             {
218:                // wstawia na początek listy
219:                ce.next = firstCacheEntry;
220:                firstCacheEntry.prev = ce;
221:                firstCacheEntry = ce;
222:             }
223:             else
224:             {
225:                // lista jest pusta
226:                firstCacheEntry = lastCacheEntry = ce;
227:             }
228:
229:             // pobiera StubEntry usuwanego obiektu
230:             StubEntry outStubEntry = (StubEntry)
231:                 rows.get(leastUsed.key.intValue());
232:
233:             // usuwa obiekt z bufora
234:             CacheEntry outCacheEntry = (CacheEntry)
235:                 cache.remove(leastUsed.key);
236:             if (outCacheEntry == null)
237:                throw new RuntimeException("Cache Entry at "
238:                               + leastUsed.key + " is Null!");
239:
240:             if (outCacheEntry != null && outCacheEntry.o == null)
241:                throw new RuntimeException("Cache object at "
242:                                + leastUsed.key + " is Null!");
243:
244:             Object outObject = outCacheEntry.o;
245:
246:             outStubEntry.inCache = false;
247:
248:             if (outStubEntry.filePointer == -1)
249:             {
250:                // umieszczony w buforze
251:                   outStubEntry.filePointer =
252:                            of.writeObject((Serializable)outObject);
253:             }
254:             else
255:             {
256:                // znajduje już się w pliku - zmiana rozmiaru?
257:                 int iCurrentSize =
258:                   of.getObjectLength(outStubEntry.filePointer);
259:
260:                 ByteArrayOutputStream baos = new ByteArrayOutputStream();
261:                 ObjectOutputStream oos = new ObjectOutputStream(baos);
262:                 oos.writeObject((Serializable) outObject);
263:                 oos.flush();
264:                 int datalen = baos.size();
265:
266:                 if (datalen <= iCurrentSize)
267:                     of.rewriteObject(outStubEntry.filePointer,
268:                                      baos.toByteArray());
269:                 else
270:                     outStubEntry.filePointer =
```

```
                                    of.writeObject((Serializable)outObject);

                    baos = null;
                    oos = null;
                    outObject = null;
                }
            }
            else
            {
                CacheEntry ce = new CacheEntry();
                ce.o = o;
                ce.key = new Integer(idx);
                cache.put(ce.key, ce);
                iCacheSize++;

                // wstawia na listę LRU
                if (lastCacheEntry != null)
                {
                    // na początek listy
                    ce.next = firstCacheEntry;
                    firstCacheEntry.prev = ce;
                    firstCacheEntry = ce;
                }
                else
                {
                    // lista jest pusta
                    firstCacheEntry = lastCacheEntry = ce;
                }
            }

        }

        return o;
    }
    // *** Metody pominięte ze względu na brak miejsca
}
```

Poniżej przedstawiamy kod źródłowy klasy `ObjectFile`:

```
/* ObjectFile.java */
package jwiley.effective;

import java.io.*;
import java.util.*;

public class ObjectFile
{
    RandomAccessFile dataFile;
    String sFileName;

    public ObjectFile(String sName) throws IOException
    {
        sFileName = sName;
        dataFile = new RandomAccessFile(sName, "rw");
    }

    // zwraca pozycję pliku, na której zapisano obiekt
```

```
19:    public synchronized long writeObject(Serializable obj) throws IOException
20:    {
21:       ByteArrayOutputStream baos = new ByteArrayOutputStream();
22:       ObjectOutputStream oos = new ObjectOutputStream(baos);
23:       oos.writeObject(obj);
24:       oos.flush();
25:
26:       int datalen = baos.size();
27:
28:       // dołącza rekord
29:       long pos = dataFile.length();
30:       dataFile.seek(pos);
31:
32:       // zapisuje rozmiar danych
33:       dataFile.writeInt(datalen);
34:       dataFile.write(baos.toByteArray());
35:
36:       baos = null; oos = null;
37:
38:       return pos;
39:    }
40:
41:
42:    // pobiera bieżący rozmiar obiektu
43:    public synchronized int getObjectLength(long lPos) throws IOException
44:    {
45:       dataFile.seek(lPos);
46:       return dataFile.readInt();
47:    }
48:
49:    public synchronized Object readObject(long lPos)
50:                  throws IOException,ClassNotFoundException
51:    {
52:       dataFile.seek(lPos);
53:       int datalen = dataFile.readInt();
54:       if (datalen > dataFile.length())
55:          throw new IOException("Data file is corrupted. datalen: "
56:                                + datalen);
57:       byte [] data = new byte[datalen];
58:       dataFile.readFully(data);
59:
60:       ByteArrayInputStream bais = new ByteArrayInputStream(data);
61:       ObjectInputStream ois = new ObjectInputStream(bais);
62:       Object o = ois.readObject();
63:
64:       bais = null;    item 20
65:       ois = null;
66:       data = null;
67:
68:       return o;
69:    }
70:
71:    public long length() throws IOException
72:    {
73:       return dataFile.length();
74:    }
75:
76:    public void close() throws IOException
```

```
77:    {
78:       dataFile.close();
79:    }
80:    // Metody pominięte ze względu na brak miejsca
81: } // koniec klasy ObjectFile
```

W zagadnieniu tym przedstawiono połączenie bufora `LRU` i klasy umożliwiającej trwałe przechowywanie obiektów (`ObjectFile`), w wyniku którego uzyskano efektywne rozwiązanie problemu obsługi sporadycznie pojawiających się kolekcji o znacznych rozmiarach. Potrafi ono wydajnie obsłużyć typowy przypadek, w którym występuje niewiele danych oraz charakteryzuje się niezawodnością w momencie pojawienia się wyjątkowo dużej ilości danych. Tworzenie takich niezawodnych, choć nie zawsze efektownych rozwiązań cechuje najlepszych programistów.

Zagadnienie 20. Plik właściwości czy zestaw zasobów?

Proszę sobie wyobrazić, że ktoś rozpoczął właśnie pracę dla nowo powstałej firmy LOA, która zamierza odebrać część internetowego tortu America Online. Dowiedział się, że Sun Microsystems zgodziła się reklamować usługi firmy, w której ten ktoś zacznie pracować, użytkownikom swojego nowego systemu operacyjnego napisanego w całości w Javie. System będzie sprzedawany na całym świecie, a dołączana do niego aplikacja firmy LOA na razie pracuje jedynie w języku angielskim. Zadaniem nowego pracownika jest wyposażenie jej w możliwości obsługi innych języków. Ma czas do końca tygodnia.

Ponieważ jako programista jest on odpowiedzialny jedynie za okno pokazywane użytkownikowi podczas uruchamiania programu, to powierzone zadanie wydaje się wykonalne. W obecnej wersji okno to pobiera wyświetlane informacje z pliku właściwości. Wystarczy więc utworzyć takie pliki zawierające informacje w innych językach i opracować wymienne moduły wyświetlającego je kodu. Jednak menedżer informuje, że jeden i ten sam moduł kodu powinien obsługiwać wszystkie języki. Dlatego kolejnym pomysłem jest wczytywanie właściwości systemu zawierających informacje o wybranym języku. Dzięki tej informacji możliwe będzie następnie wczytanie zawartości odpowiedniego pliku właściwości. W czasie przerwy w pracy nowy pracownik zwierza się ze swojego pomysłu jednemu z bardziej doświadczonych programistów. Pochwala on takie rozwiązanie, informując jednocześnie, że firma Sun dawno je opracowała. Poleca więc zapoznanie się z klasą `ResourceBundle`.

Klasa `ResourceBundle` różni się od klasy `Properties` w wielu aspektach. Klasa `ResourceBundle` i jej klasy pochodne `ListResourceBundle` i `PropertyResourceBundle` zaprojektowano tak, by wykorzystywały klasę `Locale` do obsługi danych zależnych od kraju, w którym mieszka użytkownik programu. Natomiast klasa `Properties` nie używa klasy `Locale`, ponieważ jej zadaniem jest jedynie przechowywanie par obiektów klasy `String` reprezentujących klucz i odpowiadającą mu wartość. Dlatego też klasę `Properties`, w przeciwieństwie do klasy `ResourceBundle`, stosujemy do przechowywania łańcuchów

znaków `String`, które nie podlegają lokalizacji. W ten sposób powinno się oddzielić w programie dane, które podlegają lokalizacji od tych, które są niezależne od języka aplikacji. Kolejna różnica pomiędzy klasami `ResourceBundle` i klasą `Properties` polega na tym, że klasa `ListResourceBundle` umożliwia przechowywanie klucza klasy `String` i wartości klasy `Object`. W praktyce oznacza to, że można przechowywać w niej wartość będącą obiektem dowolnej klasy. Natomiast klasa `Properties` umożliwia przechowywanie jedynie łańcuchów znakowych klasy `String`.

Pierwszym krokiem związanym z internacjonalizacją okna programu będzie określenie, które dane zależą od lokalizacji użytkownika. Aby lepiej zrozumieć jakich danych może to dotyczyć, przyjrzyjmy się bliżej klasie `Locale`. Klasa ta musi uwzględniać nie tylko język, którym posługuje się użytkownik programu, ale także kraj, w którym on mieszka. W wielu krajach używa się bowiem tego samego języka, ale zapisuje liczby i daty w różnych formatach. Trzecim parametrem klasy `Locale` jest wariant. Umożliwia on programiście wyspecjalizowanie dodatkowych różnic w stosunku do podstawowych formatów. W naszym przypadku internacjonalizacji będzie podlegać jedynie tekst wyświetlany w oknie i na przyciskach.

W celu wyświetlania tekstu w różnych językach trzeba stworzyć zestaw zasobów klasy `ResourceBundle` zawierający pliki właściwości. Przez zestaw zasobów rozumiemy w tym przypadku grupę plików zawierających te same dane poddane procesowi lokalizacji dla różnych krajów i języków. Takich zestawów zasobów możemy opracować dowolnie wiele. W naszym przykładzie stworzymy dwa: jeden zawierający tekst powitania wyświetlany w oknie oraz drugi, w którym umieścimy opisy wszystkich przycisków okna. Ponieważ internacjonalizacji podlega jedynie tekst, to zestawy zasobów zawierać będą tylko pliki właściwości (w ogólnym przypadku mogą to być pliki właściwości i klasy języka Java). Zaletą takiego rozwiązania jest to, że, oddzielając w ten sposób kod od danych, możemy przekazać tłumaczom tylko same pliki właściwości.

Zestaw zasobów uzyskujemy, wywołując metodę `ResourceBundle.getBundle()`. Przekazuje się jej jako parametr obiekt klasy `Locale` lub pozwala skorzystać z domyślnego obiektu `Locale`. Aby klasa `ResourceBundle` mogła znaleźć odpowiedni plik właściwości lub klasę, trzeba zachować odpowiednią konwencję tworzenia nazw plików właściwości i klas. Dokumentacja javadoc wyjaśnia dokładnie sposoby tworzenia takich nazw. Poniżej przedstawiono konwencję nazw w kolejności, w której poszukuje jej kod klasy `ResourceBundle`, aby odnaleźć odpowiedni zasób:

```
baseclass + "_" + language1 + "_" + country1 + "_" + variant1
baseclass + "_" + language1 + "_" + country1 + "_" + variant1 + ".properties"
baseclass + "_" + language1 + "_" + country1
baseclass + "_" + language1 + "_" + country1 + ".properties"
baseclass + "_" + language1
baseclass + "_" + language1 + ".properties"
baseclass + "_" + language2 + "_" + country2 + "_" + variant2
baseclass + "_" + language2 + "_" + country2 + "_" + variant2 + ".properties"
baseclass + "_" + language2 + "_" + country2
baseclass + "_" + language2 + "_" + country2 + ".properties"
baseclass + "_" + language2
baseclass + "_" + language2 + ".properties"
baseclass
baseclass + ".properties"
```

Załóżmy na przykład, że metodzie `getBundle()` przekazaliśmy obiekt klasy `Locale` zawierający kod języka niemieckiego (`de`) i kod Szwajcarii (`CH`). Domyślny obiekt klasy `Locale` zawiera natomiast kod języka angielskiego (`en`) i Stanów Zjednoczonych (`US`). Przypuśćmy, że klasę bazową tekstu powitania nazwaliśmy `StartupMessage`. Aby ustalić, jakich nazw zasobów będzie poszukiwać klasa `ResourceBundle`, musimy zastąpić w powyższym schemacie parametr `baseclass` nazwą `"StartupMessage"`, parametr `language1` — łańcuchem `"de"`, parametr `country1` — łańcuchem `"CH"`, parametr `language2` — łańcuchem `"en"` i parametr `country2` — łańcuchem `"US"`. Ustalimy w ten sposób, że program będzie próbować znaleźć kolejno następujące zasoby: `StartupMessage_de_CH`, `StartupMessage_de_CH.properties`, `StartupMessage_de`, `StartupMessage_de.properties`, `StartupMessage_en_US`, `StartupMessage_en_US.properties`, `StartupMessage_en`, `StartupMessage_en.properties`, `StartupMessage`, `StartupMessage.properties`.

Tworzenie zestawu zasobów najlepiej rozpoczynać zawsze od podstawowego. Dzięki temu kod zawsze znajdzie przynajmniej podstawowy zasób w przypadku, gdy nie będzie dostępny zasób odpowiadający przekazanemu lub domyślnemu obiektowi klasy `Locale`. Najpierw utworzymy więc podstawowy plik właściwości. Będzie on posiadać nazwę `StartupMessage` i następującą zawartość:

```
StartupScreen.message = Welcome to LOA: Please press OK to install our software.
Otherwise, press CANCEL to exit.
```

Następnie utworzymy plik właściwości dla domyślnego obiektu klasy `Locale` o nazwie `StartupMessage_en_US.properties`. Zawiera on takie same informacje jak podstawowy plik właściwości. Skoro jednak informacja w obu plikach jest identyczna, to po co tworzyć plik podstawowy. Załóżmy, że obiekt klasy `Locale` przekazany metodzie `getBundle()` dotyczy Chin, a obiekt domyślny Japonii. Ponieważ zestaw zasobów nie zawiera plików właściwości dla języka chińskiego ani japońskiego, to posłuży się właśnie plikiem podstawowym. Gdy w zestawie zasobów nie umieścimy pliku podstawowego, wówczas w opisanym przypadku zostanie wyrzucony wyjątek `MissingResourceException`.

Następnie utworzymy domyślny zasób opisujący teksty przycisków. Nazwiemy go `StartupButton.properties`. Zawiera on następujące informacje:

```
StartupButton.ok=OK
StartupButton.cancel=CANCEL
```

Ponownie utworzymy też zasób dla domyślnego obiektu klasy `Entity` o takiej samej zawartości. Nazwiemy go `StartupButton_en_US.properties`.

Po oddzieleniu zasobów od kodu programu zobaczmy, w jaki sposób może korzystać z nich program.

Poniższy fragment kodu wczytuje zasób `StartupMessage` dla domyślnego obiektu klasy `Locale`:

```
01: ResourceBundle startupMessageBundle;
02: startupMessageBundle = ResourceBundle.getBundle("StartupMessage");
03: String message = startupMessageBundle.getString("StartupScreen.message");
```

Podobnie poniższy fragment kodu wczyta zasób `StartupButton` dla domyślnego obiektu klasy `Locale`:

```
01: ResourceBundle startupButtonBundle;
02: startupButtonBundle = ResourceBundle.getBundle("StartupButton");
03: String okButton = startupButtonBundle.getString("StartupButton.ok");
```

Jeśli domyślny obiekt klasy `Locale` odpowiada językowi angielskiemu (`en`) i USA (`US`), to zostaną załadowane pliki `StartupMessage_en_US.properties` oraz `StartupButton_en_US.properties`. Zaletą przedstawionego rozwiązania jest to, że kod programu nie musi zmieniać się ze zmianą języka wyświetlanych komunikatów. Wystarczy, że tłumacz stworzy nowy plik właściwości i odpowiednio go nazwie. W ten sposób jeden i ten sam fragment kodu może obsługiwać wiele języków.

Problematyka internacjonalizacji oprogramowania jest rozległa. W zagadnieniu tym skoncentrowaliśmy się na omówieniu różnic pomiędzy klasami `Properties` i `ResourceBundle` i przedstawieniu podstawowych sposobów posługiwania się klasą `ResourceBundle`. Można więc potraktować go jedynie jako wprowadzenie do tematu. Język Java dysponuje jeszcze innymi klasami wspierającymi programistę podczas internacjonalizacji programów, na przykład `DateFormat`, `NumberFormat` i `MessageFormat`.

Zagadnienie 21. Pułapki klasy Properties

Jeśli Czytelnik programuje długo w języku Java, to z pewnością używał już obiektów klasy `Properties`. Mógł nawet przyjąć, że klasa ta stanowi cudowny środek, który raz na zawsze uwalnia programistę od konieczności kodowania wartości bezpośrednio w kodzie programu. Jeśli natomiast nie korzystał jeszcze z klasy `Properties` w swoich programach, to może się dowiedzieć, że pozwala ona pobierać klucze i ich wartości zapisane w plikach właściwości. Pliki właściwości są plikami tekstowymi zawierającymi wiersze postaci `klucz=wartość`. Ich zadaniem jest umożliwienie modyfikacji wartości zmiennych klasy `String` bez konieczności wprowadzania zmian w kodzie programu. Klasa `Properties` udostępnia programiście podstawowe narzędzia odczytujące zawartości pliku właściwości oraz pobierające lub nadające dane wartości. Pliki właściwości stanowią wygodny sposób przechowywania informacji o konfiguracji aplikacji lub preferencjach użytkownika. Pliki właściwości wraz z klasą `Properties` stanowią próbę separacji informacji tekstowej i kodu programu. Takie rozwiązanie nie jest jednak pozbawione własnych problemów.

Załóżmy na przykład, że w pliku właściwości będziemy przechowywać dane o systemie bazy danych, z którym łączy się nasza aplikacja. Dzięki takiemu rozwiązaniu uzyskamy możliwość zmiany systemu bazy danych, z którym pracuje nasza aplikacja bez konieczności wprowadzania zmian w kodzie aplikacji. Zawartość takiego pliku właściwości może wyglądać następująco:

```
Default.DbName=Postgres
Default.url=//localhost:5432/bcs
Default.UserName=postgres
Default.Password=postgres
```

Jeśli plik ten umieścimy w katalogu *c:\myApp\properties* i nazwiemy *System.properties*, to jego zawartość można odczytać za pomocą przedstawionego niżej fragmentu kodu:

```
01:  try
02:  {
03:     FileInputStream fis = new FileInputStream("c:\\myApp\\properties
                                                  \\System.properties");
04:     Properties props = new Properties();
05:     props.load(fis);
06:     String dbName = props.getProperty("Default.DbName");
07:  }
08:  catch(FileNotFoundException fnfe){} //obsługa wyjątku
09:  catch(IOException ioe) {} // obsługa wyjątku
```

Powyższy kod tworzy obiekt klasy `Properties` zawierający pary kluczy i odpowiadających im wartości zapisane w pliku *System.properties*. Zmieniając jego zawartość, możemy połączyć naszą aplikację z innym systemem bazy danych, nie zmieniając ani jednego wiersza jej kodu. Doskonałe rozwiązanie, ale gdzie jest problem? Zaletą tego rozwiązania jest oddzielenie informacji o systemie bazie danych i kodu aplikacji. Jednak kod aplikacji zawiera ścieżkę dostępu do pliku. Przypuśćmy, że użytkownik zainstalował naszą aplikację na dysku D zamiast C. Konstruktor `FileInputStream()` wywołany w 3. wierszu wyrzuci wtedy wyjątek `FileNotFoundException`, ponieważ plik *System.properties* nie zostanie znaleziony w katalogu *c:\myApp\properties*. W jaki zatem sposób możemy uniknąć kodowania w programie ścieżki dostępu do pliku właściwości?

Alternatywne rozwiązanie będzie wykorzystywać metodę `getResourceAsStream()` klasy `Class`. Metoda ta, poszukując zasobu, wykorzystuje ścieżki dostępu do klas. Dzięki temu możemy umieścić plik właściwości w dowolnym katalogu pod warunkiem, że katalog ten dodamy do ścieżki dostępu do klas. Domyślnie metoda `getResourceAsStream (String resource)` przeszukuje ścieżki dostępu na dwa sposoby. Jeśli przekazany jej jako parametr łańcuch znaków opisujący zasób rozpoczyna się od znaku /, to łańcuch ten nie jest modyfikowany. W przeciwnym razie łańcuch jest dołączany na końcu nazwy pakietu, w nazwie którego wszystkie znaki . zastępowane są znakiem /. Przypuśćmy, że klasa `SysProperties` należy do pakietu `com.mycompany.myapp`. Klasa `SysProperties` może wtedy ładować obiekt klasy `Properties` w następujący sposób:

```
01:  try
02:  {
03:        InputStream is = SysProperties.class.getResourceAsStream
04:        ("/System.properties");
05:        Properties prop = new Properties();
06:        if (is != null)
07:        {
08:              prop.load(is);
09:              String dbName = props.getProperty("Default.DbName");
10:        }
11:  }
12:      catch(IOException ioe) {} // obsługa wyjątku
```

W tym przypadku metoda `getResourceAsStream()`, poszukując pliku *System.properties* sprawdza katalogi umieszczone w ścieżce dostępu do klas. Gdybyśmy jednak zmienili w wierszu 4. parametr wywołania metody z `"/System.properties"` na `"System.properties"`, to, sprawdzając katalogi umieszczone w ścieżce dostępu do klas, metoda

`getResourceAsStream()` poszukiwałaby pliku *com/mycompany/myapp/System.properties*. Obie możliwości są równie dobre. Jeśli chcemy przechowywać razem wszystkie pliki właściwości, to wybierzemy pierwszą z nich. Jeśli natomiast preferujemy umieszczanie plików właściwości w tym samym katalogu, w którym korzystające z nich klasy, właściwe będzie drugie rozwiązanie. Najważniejsze jednak jest to, że w obu przypadkach użytkownik nie musi już umieszczać pliku właściwości *System.properties* w katalogu *c:\myApp\properties*. Możemy także utworzyć plik typu *jar*, który będzie zawierać klasy i pliki właściwości.

Spróbujmy teraz wykorzystać nasz plik właściwości do utworzenia paska narzędzi. Załóżmy, że pliki ikon każdego narzędzia będzie określany w pliku *System.properties*. Pozwoli to zmienić ikonę narzędzia bez konieczności wprowadzania zmian w kodzie programu. Informacje opisujące ikonę narzędzia w pliku właściwości będą miały następującą postać:

```
Save.image=c:\\myApp\\images\\save.gif
New.image=c:\\myApp\\images\\new.gif
```

Właściwości `Save.image` i `New.image` określają kompletne ścieżki dostępu do ikon narzędzi. Możliwość modyfikacji pliku właściwości jest z pewnością lepszym rozwiązaniem niż modyfikacja kodu programu. Jednak także w tym przypadku natrafiamy na ten sam problem co w przypadku kodowania ścieżki dostępu do samego pliku właściwości. Jeśli użytkownik nie zainstaluje naszej aplikacji w katalogu *c:\myApp*, to program nie odnajdzie plików ikon. Lepszym rozwiązaniem będzie więc umieszczenie w pliku właściwości tylko ścieżek dostępu określonych względem katalogu, w którym został zainstalowany program. Musimy wtedy dodatkowo utworzyć właściwość `Application.root`, która będzie przechowywać informacje o katalogu, w którym został zainstalowany program. Wiele programów instalacyjnych, na przykład InstallShield, umożliwia uzyskanie informacji o katalogu, w którym użytkownik zainstalował aplikację. Dzięki temu można prawidłowo zainicjować właściwość `Application.root` i przechowywać jedynie względne ścieżki dostępu do zasobów. Na przykład, gdy użytkownik zainstaluje aplikację myApp w katalogu *d:\apps\myApp*, to zawartość pliku właściwości będzie wyglądać następująco:

```
Application.root=d:\\apps\\myApp
Save.image=\\images\\save.gif
New.image=\\images\\new.gif
```

Oczywiście teraz kod programu po pobraniu ścieżki dostępu do zasobu musi dołączać ją do wartości `Application.root`. Oto fragment kodu pozwalający uzyskać pełną ścieżkę dostępu do zasobu `Save.image`:

```
01:   StringBuffer temp = new StringBuffer();
02:   temp.append(prop.getProperty("Application.root");
03:   temp.append(prop.getProperty("Save.Image");
04:   String saveImagePath = temp.toString();
05:   // kod tworzący obiekt ImageIcon na podstawie saveImagePath
```

Użytkownik może teraz zainstalować aplikację w dowolnym katalogu i będzie ona zawsze mogła uzyskać dostęp do swoich zasobów.

Z instalacją aplikacji mogą być związane jeszcze inne problemy. Na przykład użytkownik zainstalował ją w katalogu */pkgs/programs* systemu Unix. Aplikacja nada więc właściwości `Application.root` wartość `/pkgs/program`. Aby załadować ikonę przycisku *Save*, aplikacja dołączy do właściwości `Application.root` łańcuch `\\images\\save.gif`. Pełna ścieżka dostępu będzie więc miała postać */pkgs/program\\images\\save.gif*, stanowiąc mieszankę sposobu zapisu ścieżek w systemach Unix i Windows. Aby rozwiązać ten problem, napiszemy pomocniczą metodę, która będzie zmieniać format ścieżek w zależności od systemu operacyjnego. System operacyjny, w którym działa aplikacja, ustalimy, korzystając z właściwości systemowej `os.name`. Jeśli będzie nim Unix, to znaki `\\` zastąpimy w ścieżkach znakiem `/`. W ten sposób na przykład ścieżka dostępu do ikony przycisku *Save* uzyska postać */pkgs/programs/images/save.gif*.

Kolejny problem związany z naszym zastosowaniem obiektów klasy `Properties` polega na zapewnieniu, że wszyscy programiści pracujący nad aplikacją będą korzystać z metody `getResourceAsStream()`. Dotychczas omówiliśmy dwa sposoby ładowania obiektów `Properties`, ale istnieje ich znaczniej więcej. Jeśli więc kod aplikacji jest tworzony przez kilku programistów, to istnieje szansa, że przynajmniej jeden z nich będzie ładował obiekty klasy `Properties` inaczej niż pozostali. Taka niespójność może powodować niewłaściwe działanie aplikacji i utrudniać utrzymanie jej kodu. Można jej uniknąć, tworząc na przykład specjalną klasę, która będzie zwracać obiekt klasy `Properties` na podstawie przekazanej jej nazwy pliku właściwości. Poniżej przedstawiono kod klasy `PropertyLoader`:

```
01:  public class PropertyLoader
02:  {
03:        //can't instantiate outside this class
04:        private PropertyLoader() {}
05:        public static Properties load(String propName)
06:        {
07:             try
08:             {
09:                  InputStream is =
10:                  PropertyLoader.class.getResourceAsStream(propName);
11:                  if (is != null)
12:                  {
13:                       Properties props = new Properties();
14:                       props.load(is);
15:                       return props;
16:                  }
17:             }
18:             catch (IOException ioe){} // handle exception
19:             return null;
20:        }
21:  }
```

Klasa `PropertyLoader` posiada metodę statyczna `load()`. Może ona, tak jak w naszym przykładzie, sama obsługiwać wyjątek `IOException` lub tylko go wyrzucać, wymagając obsłużenia go przez kod wywołujący metodę. Niezależnie od wybranego sposobu obsługi wyjątku klasa `PropertyLoader` umożliwia załadowanie obiektu `Properties` przy skorzystaniu ze ścieżki dostępu do klas. Załadowanie obiektu `Properties` odbywa się następująco:

```
Properties props = PropertyLoader.load("/myApp.properties");
```

Wadą klasy PropertyLoader jest to, że ogranicza nas tylko do obiektów klasy Properties. Chociaż klasa Properties posiada szereg użytecznych metod, to czasami przydatne okazują się jeszcze inne. Na przykład, gdy zachodzi potrzeba zamiany wartości właściwości klasy String na wartość typu int lub boolean. Klasy Integer i Boolean dostarczają co prawda metody umożliwiającej przekształcenie wartości właściwości System.prop na typ int lub boolean, ale nie jest możliwe jej zastosowanie do dowolnego pliku właściwości. Kolejną przydatną metodą jest getPathProperty(), która automatyzuje konwersję formatu ścieżek dostępu dla różnych systemów operacyjnych. Zamiast zmuszać użytkownika klasy PropertyLoader do samodzielnej implementacji wspomnianych metod, możemy utworzyć klasę EnhancedProperties jako pochodną klasy Properties zawierającą dodatkowe metody. Klasa EnhancedProperties może wyglądać następująco:

```
01:  public class EnhancedProperties extends Properties
02:  {
03:      public EnhancedProperties()
04:      {
05:         super();
06:      }
07:      public EnhancedProperties(Properties defaults)
08:      {
09:         super(defaults);
10:      }
11:      public EnhancedProperties(InputStream is)  throws IOException
12:      {
13:         load(is);
14:      }
15:      public boolean propertyToBoolean(String key)
16:      {
17:         // code to convert a String value to a boolean value.
18:      }
19:      public int propertyToInt(String key)
20:      {
21:         // code to convert a String value to an int value.
22:      }
23:      public String getPathProperty(String key)
24:      {
25:         // code to convert the returned path value to the appropriate
26:         system path value.
27:      }
28:  }
```

Musimy jeszcze zmodyfikować klasę PropertyLoader tak, by zwracała obiekt klasy EnhancedProperties zamiast Properties:

```
return  new EnhancedProperties(is);
```

Ponieważ w wierszu 11. klasy EnhancedProperties dodaliśmy konstruktor, którego parametrem jest InputStream, to wiersze 12. – 14. klasy PropertyLoader możemy zastąpić pojedynczym wierszem pokazanym wyżej. W klasie EnhancedProperties umieściliśmy tylko kilka propozycji dodatkowych metod, pozostawiając pozostałe inwencji Czytelnika.

Zagadnienie 22. Klasa Vector i nowe kolekcje

Stare przyzwyczajenia ciężko zmienić. Dlatego też wielu programistów, którzy intensywnie używali kolekcji w języku Java 1.0, niechętnie decyduje się porzucić swoje doświadczenia i rozpocząć programowanie z wykorzystaniem nowych kolekcji zaproponowanych w następnych wersjach języka Java. Zadaniem tego zagadnienia jest pomóc wykonać pierwszy krok w tym kierunku. Zamiast więc atakować Czytelnika ogromną liczbą informacji o nowych kolekcjach i ich wspaniałych możliwościach, wybraliśmy zdecydowanie prostszy sposób. Pokażemy przykład zastosowania poprzedniej wersji klasy `Vector`, a następnie ten sam przykład wykorzystujący nowy interfejs klasy `Vector`. To najłatwiejszy sposób zapoznania się z podobieństwami i różnicami obu klas, który umożliwi samodzielne poznanie pozostałych ponad 25 klas zawierających nowy zbiór kolekcji.

Zanim przejdziemy do omówienia przykładów, przedstawmy krótko nowy interfejs kolekcji. Stanowi on szkielet umożliwiający tworzenie kolekcji i wykonywanie na nich operacji. Przez kolekcję rozumiemy w tym przypadku grupę obiektów. Nowy interfejs kolekcji definiuje kilka rodzajów takich grup — kolekcje, listy, mapy i zbiory. Każda z tych kategorii jest reprezentowana przez interfejs, klasę abstrakcyjną i jedną klasę konkretną lub więcej. Uzupełnienie stanowią operacje oglądania kolekcji za pomocą iteratorów, porównań wewnątrz kolekcji, wyszukiwania elementów kolekcji oraz sortowania kolekcji.

W pierwszym programie zademonstrowano najczęściej używane metody klasy `Vector`. Zawiera on przykłady dodawania i usuwania elementów wektora, wstawiania elementów, pobierania elementów, kopiowania elementów wektora do tablicy obiektów, zastosowania obiektu `Enumeration` do przeglądania wektora oraz wywołania metody usuwającej wszystkie elementy wektora.

```
01: import java.util.*;
02:
03: public class OldVectorAPI
04: {
05:     public static void main (String args[])
06:     {
07:         Vector v = new Vector();
08:         v.addElement("M. Elaine");
09:         v.addElement("D. Allison");    item 22
10:         v.addElement("J. Eric");
11:         v.addElement("A. Jewell");
12:         v.addElement("N. Anh");
13:
14:         v.removeElementAt(4);
15:         v.removeElement("D. Allison");
16:         v.insertElementAt("C. Thomas", 1);
17:
18:         print(v);
19:         printElements(v.elements());
20:
```

```
21:         Object[] middleNames = new Object[v.size()];
22:         v.copyInto(middleNames);
23:         v.removeAllElements();
24:     }
25:
26:     public static void print (Vector v)
27:     {
28:         int numItems = v.size();
29:         for (int i = 0; i < numItems; i++)
30:             System.out.println(v.elementAt(i));
31:     }
32:
33:     public static void printElements (Enumeration e)
34:     {
35:         while (e.hasMoreElements())
36:             System.out.println(e.nextElement());
37:     }
38: }
```

Modyfikacja tego programu tak, aby wykorzystywał nowy interfejs wektorów, jest bardzo prosta. Odpowiedniki metod zawiera tabela 3.1.

Tabela 3.1. *Porównanie poprzedniego i bieżącego interfejsu wektorów*

Poprzedni interfejs	Bieżący interfejs
`void addElement(Object)`	`boolean add(Object)`
`void copyInto(Object[])`	`Object[] toArray()`
`Object elementAt(int)`	`Object get(int)`
`Enumeration elements()`	`Iterator iterator()`
`void insertElementAt(Object, int)`	`void add(index, Object)`
`void removeAllElements()`	`void clear()`
`boolean removeElement(Object)`	`boolean remove(Object)`
`void removeElementAt(int)`	`void remove(int)`
`int size()`	`int size()`

Klasa `Vector` w nowym zestawie kolekcji implementuje interfejs `List`. W następnej wersji programu można więc upewnić się, że korzystamy z nowych kolekcji, podstawiając referencję nowo utworzonego wektora do zmiennej typu `List`. Oczywiście, metody specyficzne dla klasy `Vector` nie będą wtedy dostępne bez jawnego zastosowania rzutowania. Oto wersja programu wykorzystująca nowe kolekcje:

```
01: import java.util.*;
02:
03: public class NewVectorAPI
04: {
05:     public static void main (String args[])
06:     {
07:         List v = new Vector();
08:         v.add("M. Elaine");
09:         v.add("D. Allison");
10:         v.add("J. Eric");
11:         v.add("A. Jewell");
```

```
12:         v.add("N. Anh");
13:
14:         v.remove(4);
15:         v.remove("D. Allison");
16:         v.add(1, "C. Thomas");
17:
18:         print(v);
19:         printElements(v.iterator());
20:
21:         Object[] middleNames = v.toArray();
22:         v.clear();
23:     }
24:
25:     public static void print (List v)
26:     {
27:         int numItems = v.size();
28:         for (int i = 0; i < numItems; i++)
29:             System.out.println(v.get(i));
30:     }
31:
32:     public static void printElements (Iterator it)
33:     {
34:         while (it.hasNext())
35:             System.out.println(it.next());
36:     }
37: }
```

Oprócz różnic interfejsu klasy `Vector` podanych w tabeli 3.1 należy wskazać jeszcze jedną. Nowe kolekcje bazują na koncepcji iteratora służącego do czytania ich zawartości. W zależności od typu kolekcji iteratory mogą posiadać różne poziomy funkcjonalności. Klasa `Vector` implementuje interfejs `List`, który specyfikuje dwa rodzaje iteratorów. Pierwszy z nich, klasy `Iterator`, musi być implementowany przez wszystkie kolekcje. Drugi, klasy `ListIterator`, posiada dodatkową funkcjonalność.

Porównując wiersze 35. i 36. pierwszej wersji programu z wierszami 34. i 35. drugiej wersji, zauważymy, że metody klas `Enumeration` i `Iterator` są bardzo zbliżone. Metoda `Enumeration.hasMoreElements()` stanowi odpowiednik metody `Iterator.hasNext()`, a metoda `Enumeration.nextElement()` — odpowiednik metody `Iterator.next()`. Klasa `Iterator` posiada dodatkowo metodę `remove()`, co daje jej pewną przewagę nad klasą `Enumeration`. Klasa `Iterator` stanowi najmniejszy wspólny mianownik w dostępie do elementów różnych typów kolekcji.

Klasa `ListIterator` udostępnia bardziej rozbudowaną funkcjonalność: pozwala modyfikować listę (dodawać, modyfikować i usuwać elementy), a także przeglądać ją w obu kierunkach. Programista ma więc do wyboru operacje na liście wykonywane za pomocą metod iteratora `ListIterator` lub metod interfejsu `List`.

Stosowanie większości nowych kolekcji nie jest bezpieczne w programach wielowątkowych. Dotyczy to na przykład klasy `ArrayList` będącej klasą siostrzaną klasy `Vector` (czyli klasą implementującą interfejs `List`). Natomiast użycie klasy `Vector` jest nadal bezpieczne z punktu widzenia wątków. Programista dokonuje więc wyboru, czy korzysta z klasy `ArrayList`, której zastosowanie w programie wielowątkowym nie jest bezpieczne, czy też klasy `Vector`, którą może bez obaw używać w wielu wątkach.

Zastosowanie klasy `Vector` pochodzącej z nowego zestawu kolekcji nie daje większej efektywności w porównaniu z poprzednią wersją, ale zapewnia większą modyfikowalność i uniwersalność kodu. Jeśli na przykład okaże się, że wydajność programu jest za mała, to (zamiast klasy `Vector`) możemy wykorzystać inną implementację ze zbioru nowych kolekcji. W przypadku poprzedniej wersji klasy `Vector` oznaczałoby to konieczność modyfikacji programu. Warto więc przekonać się do stosowania nowych kolekcji, ponieważ ułatwia to utrzymanie kodu i zwiększa możliwości jego modyfikacji.