Wydanie VII

Rogers Cadenhead

Java™

w 21 dni

SAMS

Helion

Tytuł oryginału: Sams Teach Yourself Java™ in 21 Days

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-2621-7

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ja21d7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Dzień 8.

Struktury danych

W pierwszym tygodniu poznaliśmy podstawowe elementy języka Java: obiekty, klasy, interfejsy, a także słowa kluczowe, instrukcje, wyrażenia i operatory.

W drugim tygodniu przeniesiemy naszą uwagę z klas tworzonych samodzielnie na klasy, które ktoś przygotował dla nas. Standardowa biblioteka klas Javy to zbiór pakietów udostępnionych przez Oracle, który łącznie zawiera ponad 4200 klas gotowych do wykorzystania we własnym kodzie.

Dzisiaj zajmiemy się klasami reprezentującymi dane.

Omówimy następujące struktury danych:

- **zbiory bitów, które przechowują wartości logiczne;**
- **listy tablicowe, czyli tablice, które mogą dowolnie zmieniać swój rozmiar;**
- **stosy, a więc struktury przechowujące dane zgodnie z zasadą LIFO (ang. *Last In, First Out* — ostatni na wejściu, pierwszy na wyjściu);**
- **tablice mieszające, które przechowują elementy na podstawie kluczy.**

Wychodzimy poza tablice

Biblioteka klas Javy udostępnia w pakiecie `java.util` zestaw struktur danych, które pozwalają na dużą elastyczność w organizacji i modyfikacji danych.

Solidne zrozumienie struktur danych i miejsc ich właściwego zastosowania to jeden z ważniejszych elementów programowania w języku Java.

Wiele tworzonych programów Javy wykorzystuje pewne mechanizmy przechowywania i modyfikacji danych wewnątrz klasy. Do tej pory wykorzystywaliśmy w zasadzie trzy struktury danych: zmienne, obiekty `String` i tablice.

Wymienione elementy stanowią tylko niewielki wycinek wszystkich klas dostępnych w Javie. Jeśli nie poznamy dobrze pełnego zestawu struktur danych, może się okazać, że zamiast zastosować bardziej efektywne lub prostsze w implementacji struktury, nadal będziemy korzystać z tablic i tekstów.

Poza typami podstawowymi i tekstami tablice są najprostszą strukturą danych obsługiwaną przez Javę. Tablica to zbiór danych o takim samym typie podstawowym lub takiej

samej klasie. Jest traktowana jako jeden obiekt, choć tak naprawdę zawiera wiele elementów, które można pobrać w sposób niezależny od siebie. Tablice to przydatna struktura, kiedy trzeba przechowywać wiele powiązanych i podobnych do siebie informacji.

Głównym ograniczeniem tablic jest fakt, że nie mogą zmieniać rozmiaru, aby pomieścić więcej lub mniej elementów. Nie można dodać nowych elementów do tablicy, która jest już pełna. Jedna ze struktur danych, którą dziś poznamy, czyli lista tablicowa, nie ma takiego ograniczenia.

Uwaga

> W odróżnieniu od struktur danych dostarczanych przez pakiet `java.util` tablice są uznawane za tak istotny element Javy, że zostały zaimplementowane w samym języku. Oznacza to, że tablice są dostępne bez stosowania dodatkowych obiektów przechowujących ich dane.

Struktury w języku Java

Struktury danych zapewniane przez pakiet `java.util` mogą realizować wiele różnych zadań. W skład pakietu wchodzi interfejs `Iterator`, interfejs `Map`, a także klasy:

- `BitSet`,
- `ArrayList`,
- `Stack`,
- `HashMap`.

Każda z wymienionych struktur danych zapewnia bardzo konkretny mechanizm dotyczący zapisywania i pobierania informacji. Interfejs `Iterator` sam w sobie nie jest strukturą danych, ale definiuje sposób pobierania kolejnych elementów struktury. Definiuje między innymi metodę `next()`, która ma za zadanie pobrać następny element w strukturze zawierającej wiele elementów.

Uwaga

> Interfejs `Iterator` to rozszerzona i ulepszona wersja interfejsu `Enumeration` z wcześniejszych wersji języka. Choć stary interfejs jest nadal obsługiwany, warto używać nowego, bo ma prostsze nazwy metod i obsługuje usuwanie elementów. Poza tym został tak zaprojektowany, aby wykrywać potencjalnie niebezpieczne sytuacje w trakcie korzystania z wątków — zgłasza błąd `ConcurrentModification` ↪`Exception`, gdy jeden wątek modyfikuje element, a drugi wątek właśnie iteruje w pętli po elementach struktury.

Klasa `BitSet` implementuje grupę bitów lub znaczników, które można niezależnie włączać lub wyłączać. Klasa jest użyteczna, gdy trzeba przechowywać zestaw wielu wartości logicznych, a każdą z nich ustawiać lub czyścić, gdy zajdzie taka potrzeba. Znacznik to pojedyncza wartość logiczna informująca o włączeniu lub wyłączeniu pewnej właściwości programu.

Klasa `ArrayList` przypomina w działaniu zwykłą tablicę, ale potrafi automatycznie zwiększać lub zmniejszać swój rozmiar. Podobnie jak w przypadku `ArrayList`, dostęp do poszczególnych elementów odbywa się za pomocą indeksów. Istotną cechą listy tablicowej jest to, że nie wymaga wskazywania rozmiaru w momencie tworzenia — jej rozmiar jest automatycznie zwiększany w trakcie dodawania nowych elementów.

Klasa `Stack` implementuje stos elementów, czyli strukturę LIFO. Potraktuj stos jak zwykłą stertę papierów. Gdy dodajemy nowy element, trafia on na szczyt stosu i przykrywa wszystkie wcześniejsze. Kiedy pobieramy element ze stosu, najpierw bierzemy ten, który trafił na niego jako ostatni. Sposób usuwania elementów ze stosu jest inny niż w przypadku tablicy, w której to każdy element jest dostępny przez cały czas.

Klasa `HashMap` implementuje `Dictionary` — klasę abstrakcyjną definiującą strukturę danych, w której klucze odwzorowuje się na wartości. Strukturę tego typu stosuje się, gdy dostęp do elementów ma się odbywać nie na podstawie indeksu liczbowego, ale pewnego klucza (na przykład tekstu). Ponieważ klasa `Dictionary` jest abstrakcyjna, stanowi jedynie podstawę dla struktur danych z mapowaniem kluczy i nie zapewnia konkretnej implementacji. Kluczem może być dowolny identyfikator służący do odniesienia się do konkretnej wartości zawartej w strukturze danych.

Klasa `HashMap` zapewnia implementację struktury danych z mapowaniem kluczy. Struktura kluczy bazuje na definicji zgłoszonej przez użytkownika. Na przykład w strukturze wykorzystującej kody pocztowe każdy kod pocztowy może posłużyć do pobrania danych innego typu. Znaczenie konkretnych kluczy w tablicy mieszającej zależy od sposobu użycia tablicy i danych, które zawiera.

W następnych punktach każda z wymienionych struktur danych jest omówiona bardziej szczegółowo.

Interfejs Iterator

Interfejs `Iterator` zapewnia pewien standardowy sposób przechodzenia przez listę elementów w zdefiniowanej sekwencji, co jest częstym zadaniem w wielu strukturach danych.

Choć nie można skorzystać z interfejsu poza konkretną strukturą danych, zrozumienie sposobu działania interfejsu `Iterator` pomoże w zrozumieniu innych struktur danych.

Mając to na uwadze, przyjrzyjmy się trzem metodom definiowanym przez interfejs `Iterator`:

```
public boolean hasNext();
public Object next();
public void remove();
```

Metody nie mają kodu, ponieważ interfejsy nie zawierają żadnej implementacji. To klasa implementująca interfejs musi zapewnić kod realizujący wszystkie trzy metody.

Metoda `hasNext()` sprawdza, czy struktura zawiera jakiekolwiek dodatkowe elementy. Używa się jej, aby stwierdzić, czy możliwa jest dalsza iteracja po strukturze danych.

Metoda `next()` pobiera następny element w strukturze. Jeśli nie ma więcej elementów, metoda `next()` zgłasza wyjątek `NoSuchElementException`. Aby uniknąć wyjątku, przed

wywołaniem metody `next()` skorzystaj z metody `hasNext()` i pobierz element tylko wtedy, gdy istnieje.

Poniższa pętla `while` używa dwóch wspomnianych metod, aby przejść przez strukturę danych o nazwie `users`, która implementuje interfejs `Iterator`:

```
while (users.hasNext()) {
    Object ob = users.next();
    System.out.println(ob);
}
```

Przykładowy kod wyświetla zawartość każdego elementu listy za pomocą metod `hasNext()` i `next()`.

Metoda `next()` zwraca obiekt klasy `Object`. Można go rzutować na faktyczną klasę przechowywaną w strukturze danych. Oto przykład dotyczący struktury danych przechowującej obiekty typu `String`:

```
while (users.hasNext()) {
    String ob = (String) users.next();
    System.out.println(ob);
}
```

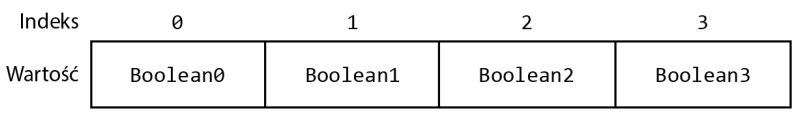
Uwaga

> Ponieważ `Iterator` to interfejs, nigdy nie stosuje się go bezpośrednio. Tak naprawdę metody z `Iterator` wywołuje się dla struktur danych implementujących interfejs. Dzięki zastosowaniu interfejsów iterowanie po tych wszystkich strukturach przebiega w spójny i łatwy do nauczenia sposób.

Zbiory bitów

Klasa `BitSet` przydaje się, gdy trzeba zastosować dużą ilość danych w postaci binarnej, czyli wartości przyjmujących tylko i wyłącznie wartości 0 lub 1. Wartość 0 oznacza wyłączenie lub wartość logiczną `false`, a 1 — włączenie lub wartość logiczną `true`.

Dzięki klasie `BitSet` można przechowywać poszczególne bity i wydobywać ich wartości bez stosowania operacji na bitach. Dostęp do poszczególnych bitów odbywa się przy użyciu indeksu. Ciekawą cechą `BitSet` jest to, że rozrasta się automatycznie, aby pomieścić tyle wartości bitowych, ile wymaga program. Rysunek 8.1 przedstawia logiczną organizację struktury danych.



RYSUNEK 8.1. Organizacja zbioru bitów

Obiekt `BitSet` może posłużyć do przechowywania atrybutów, które można w bardzo łatwy sposób zamodelować za pomocą wartości logicznych. Ponieważ poszczególne bity zbioru

są dostępne jako indeksy, możemy każdemu atrybutowi przypisać pewien stały indeks. Oto przykład:

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

W tej klasie atrybuty otrzymały kolejne wartości całkowite, poczynając od 0. Wartości te możemy wykorzystać do ustawiania i pobierania poszczególnych bitów. Najpierw utwórzmy obiekt `BitSet`:

```
BitSet connex = new BitSet();
```

Konstruktor tworzy zbiór bez określonego rozmiaru. Możliwe jest również utworzenie zbioru o pewnym rozmiarze początkowym:

```
BitSet connex = new BitSet(4);
```

W ten sposób powstał zbiór zawierający cztery wartości bitowe. Niezależnie od zastosowanego konstruktora domyślnie wszystkie bity mają wartość `false`. Po utworzeniu zbioru poszczególne bity możemy ustawiać lub czyścić za pomocą metod `set(int)` i `clear(int)`. Oto przykład:

```
connex.set(ConnectionAttributes.WRITABLE);
connex.set(ConnectionAttributes.STREAMABLE);
connex.set(ConnectionAttributes.FLEXIBLE);

connex.clear(ConnectionAttributes.WRITABLE);
```

Powyższy kod ustawił bity na pozycjach określonych przez atrybuty `WRITABLE`, `STREAMABLE` i `FLEXIBLE`, a następnie wyczyścił atrybut `WRITABLE`. Poszczególne atrybuty zostały poprzedzone nazwą klasy, ponieważ stałe są zmiennymi klasowymi klasy `ConnectionAttributes`.

Aby pobrać wartość bitu na wybranej pozycji, skorzystaj z metody `get()`:

```
boolean isWriteable = connex.get(ConnectionAttributes.WRITABLE);
```

Aby dowiedzieć się, ile bitów zawiera zbiór, użyj metody `size()`:

```
int numBits = connex.size();
```

Klasa `BitSet` zawiera również inne metody służące do przeprowadzania porównań i operacji bitowych na zbiorze, takich jak `AND`, `OR` lub `XOR`. Wszystkie te metody przyjmują jako jedyny argument obiekt `BitSet`.

Pierwszym dziś projektem będzie `HolidaySked`, czyli klasa Javy przechowująca informację o tym, które dni w roku są świętami.

Zastosujemy zbiór bitowy, ponieważ aplikacja musi mieć możliwość pobrania dowolnego dnia w roku i odpowiedzenia na pytanie, czy ten dzień jest świętem.

Wpisz kod z listingu 8.1 do pustego pliku Javy w edytorze NetBeans. Nadaj plikowi nazwę `HolidaySked` i umieść go w pakiecie `com.java21days`.

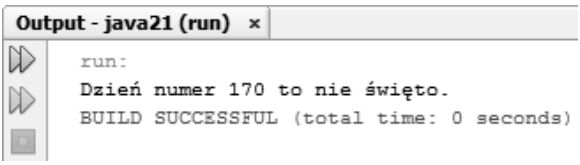
LISTING 8.1. Pełna treść pliku HolidaySked.java

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class HolidaySked {
 6:     BitSet sked;
 7:
 8:     public HolidaySked() {
 9:         sked = new BitSet(365);
10:         int[] holiday = { 1, 15, 50, 148, 185, 246,
11:             281, 316, 326, 359 };
12:         for (int i = 0; i < holiday.length; i++) {
13:             addHoliday(holiday[i]);
14:         }
15:     }
16:
17:     public void addHoliday(int dayToAdd) {
18:         sked.set(dayToAdd);
19:     }
20:
21:     public boolean isHoliday(int dayToCheck) {
22:         boolean result = sked.get(dayToCheck);
23:         return result;
24:     }
25:
26:     public static void main(String[] arguments) {
27:         HolidaySked cal = new HolidaySked();
28:         if (arguments.length > 0) {
29:             try {
30:                 int whichDay = Integer.parseInt(arguments[0]);
31:                 if (cal.isHoliday(whichDay)) {
32:                     System.out.println("Dzień numer " + whichDay +
33:                         " to święto.");
34:                 } else {
35:                     System.out.println("Dzień numer " + whichDay +
36:                         " to nie święto.");
37:                 }
38:             } catch (NumberFormatException nfe) {
39:                 System.out.println("Błąd: " + nfe.getMessage());
40:             }
41:         }
42:     }
43: }
```

Aplikacja wymaga przekazania w argumencie wiersza poleceń pojedynczej liczby całkowitej stanowiącej numer dnia w roku, czyli liczbę z przedziału od 1 do 365. (Dni w roku ze świętami są zdefiniowane w wierszach 10. oraz 11. i będą inne każdego roku). Użyj polecenia *Run/Set Project Configuration/Customize*, aby ustawić argument.

Przetestuj aplikację z różnymi wartościami, na przykład 359 (Boże Narodzenie) lub 103 (dzień moich urodzin). Aplikacja powinna poinformować, że dzień numer 359 to święto, ale dzień 103 nie jest (niestety) świętem.

Rysunek 8.2 przedstawia wynik po przekazaniu dnia o numerze 170.



RYSUNEK 8.2. Testowanie struktury danych BitSet

Klasa `HolidaySked` zawiera tylko jedną zmienną egzemplarza — `sked` — przechowującą obiekt `BitSet` z wartościami dla każdego dnia roku.

Konstruktor tworzy obiekt `sked` z 365 pozycjami. Domyślnie wszystkie pozycje mają ustawiony swój bit na 0. Całość obsługi znajduje się w wierszach od 8. do 15.

Następnie powstaje tablica z liczbami całkowitymi o nazwie `holiday`. Tablica zawiera numery dni w roku, w których są święta, zaczynając od dnia numer 1 (Nowy Rok), a kończąc na 359 (Boże Narodzenie).

Tablica `holiday` służy do dodania każdego ze świąt do zbioru `sked`. Pętla `for` iteruje przez tablicę `holiday` i dla każdego elementu wywołuje metodę `addHoliday(int)` (wiersze od 12. do 14.).

Metoda `addHoliday(int)` została zdefiniowana w wierszach od 17. do 19. Argument reprezentuje dzień, który należy dodać. Kod wywołuje metodę `set(int)`, która ustawia bit na wskazanej pozycji na wartość 1. Na przykład `set(359)` ustawi wartość bitu na 1 na pozycji 359.

Klasa `HolidaySked` potrafi również sprawdzić, czy wskazany dzień roku jest świętem. Za obsługę tego aspektu odpowiada metoda `isHoliday(int)` (wiersze od 21. do 24.). Metoda wywołuje metodę `get(int)`, która zwraca wartość `true`, jeśli na wskazanej pozycji bit jest ustawiony, lub `false` w sytuacji przeciwnej.

Klasę można uruchomić jako aplikację, ponieważ zawiera metodę `main()` (wiersze od 26. do 42.). Aplikacja przyjmuje pojedynczy argument wiersza poleceń — liczbę z zakresu od 1 do 365, reprezentującą dzień roku. Następnie wyświetla na ekranie informację, czy wskazany dzień jest świętem.

Listy tablicowe

Jedną z najpopularniejszych struktur danych w języku Java jest klasa `ArrayList`, ponieważ implementuje ona rozszerzalną wersję zwykłych tablic, a w związku z tym jest znacznie bardziej elastyczna i przydatna. Ponieważ `ArrayList` potrafi w razie potrzeby zmienić rozmiar, wystarczy przekazywać do niej nowe elementy, a samodzielnie się poszerzy lub nawet skróci.

Listę tablicową możemy utworzyć za pomocą konstruktora nieprzyjmującego żadnych argumentów:

```
ArrayList golfer = new ArrayList();
```

Konstruktor tworzy domyślną listę tablicową bez żadnych elementów. Wszystkie listy tuż po utworzeniu są puste. Jednym z atrybutów, który wpływa na automatyczne rozszerzanie się tablicy, jest początkowy rozmiar, czyli liczba elementów, dla których lista przygotowała miejsce w pamięci.

Rozmiar tablicy to liczba faktycznie przechowywanych w niej elementów. Pojemność listy jest zawsze co najmniej równa rozmiarowi tablicy.

Poniższy kod tworzy nową listę tablicową o wskazanej pojemności początkowej:

```
ArrayList golfer = new ArrayList(30);
```

Lista alokuje w pamięci miejsce na 30 elementów. Jeśli całe zarezerwowane miejsce zostanie zużyte, lista rozszerzy się ponownie o połowę rozmiaru początkowego. Jeżeli wstawimy do `golfer` 30. element, lista rozszerzy się tak, aby mogła pomieścić 45 elementów.

Ponieważ alokacja dodatkowego miejsca dla elementów zajmuje czas i pamięć, najlepiej od razu utworzyć listę z tyloma elementami, ilu się spodziewamy.

Dostęp do elementów listy tablicowej nie może się odbywać za pomocą nawiasów kwadratowych ([]), jak to miało miejsce w przypadku zwykłej tablicy. Trzeba zastosować odpowiednie metody klasy `ArrayList`.

Aby dodać element do listy tablicowej, użyj metody `add(Object)`. Oto przykład:

```
golfer.add("Park");
golfer.add("Lewis");
golfer.add("Ko");
```

Metoda `get()` pobiera element listy za pomocą indeksu liczbowego, co przedstawiają poniższe przykłady:

```
String s1 = (String) golfer.get(0);
String s2 = (String) golfer.get(2);
```

Metoda `get()` zwraca typ `Object`, ponieważ powinna obsługiwać wszystkie rodzaje obiektów. Z tego powodu po pobraniu elementu trzeba go rzutować na odpowiedni typ. Ponieważ w `golfer` umieściliśmy tekst, rzutowanie nastąpiło na klasę `String`.

Ponieważ lista jest numerowana od 0, pierwsze wywołanie metody `get()` pobiera tekst `"Park"`, a drugie tekst `"Ko"`.

Możemy nie tylko pobierać element z pozycji wskazanej indeksem, ale również wstawiać element na konkretnej pozycji lub usuwać element z wybranej pozycji. Służą do tego metody `add(int, Object)` oraz `remove(int)`:

```
golfer.add(1, "Kim");
golfer.add(0, "Thompson");
golfer.remove(3);
```

Pierwsze wywołanie `add()` wstawia element na pozycji 1, czyli między `"Park"` i `"Lewis"`. Teksty `"Lewis"` i `"Ko"` zostaną przesunięte w dół listy, aby zrobić miejsce dla `"Kim"`. Drugie wywołanie `add()` dodaje element na początku listy (indeks 0). Wszystkie istniejące elementy przesuną się o jedną pozycję w dół listy, aby zrobić miejsce dla tekstu `"Thompson"`. W tym momencie tablica ma następującą zawartość:

0. `p[;'"Thompson"`

1. `"Park"`

2. `"Kim"`

3. `"Lewis"`

4. `"Ko"`

Wywołanie metody `remove()` usuwa element o indeksie 3, czyli tekst `"Lewis"`. Po tej zmianie lista ma następującą zawartość:

0. `"Thompson"`

1. `"Park"`

2. `"Kim"`

3. `"Ko"`

Metoda `set()` służy do zmiany zawartości konkretnego elementu:

```
golfer.set(1, "Pressel");
```

Metoda zastąpiła tekst `"Park"` tekstem `"Pressel"`, więc nowa lista zawiera następujące elementy:

0. `"Thompson"`

1. `"Pressel"`

2. `"Kim"`

3. `"Ko"`

Aby wyczyścić listę tablicową, czyli usunąć całą jej zawartość, możemy użyć metody `clear()`:

```
golfer.clear();
```

Klasa `ArrayList` zawiera również pewne metody służące do korzystania z elementów, które nie wymagają wskazywania indeksów. Metody te poszukują na liście wskazanego elementu. Pierwszą z takich metod jest `contains(Object)`, która sprawdza, czy obiekt znajduje się na liście:

```
boolean isThere = golfer.contains("Kerr");
```

Inną metodą poszukującą obiektu jest `indexOf(Object)`, która zwraca indeks elementu w tablicy:

```
int i = golfer.indexOf("Ko");
```

Metoda `indexOf()` zwraca indeks elementu lub –1, jeśli obiekt nie znajduje się na liście. Metoda `remove(Object)` znajduje obiekt i usuwa go z listy:

```
golfer.remove("Pressel");
```

Klasa `ArrayList` oferuje również kilka metod służących do sprawdzania i modyfikacji jej rozmiaru. Metoda `size()` zwraca liczbę elementów znajdujących się w tablicy:

```
int size = golfer.size();
```

Przypomnijmy, że lista posiada dwa atrybuty związane z jej wielkością: rozmiar i pojemność. Rozmiar to liczba elementów aktualnie znajdujących się wewnątrz listy, a pojemność to ilość pamięci zarezerwowanej na potrzeby przechowywania elementów. Pojemność jest zawsze większa lub równa rozmiarowi. Można wymusić zmianę pojemności na równą rozmiarowi poprzez użycie metody `trimToSize()`:

```
golfer.trimToSize();
```

Ostrzeżenie

> Biblioteka klas Javy zawiera również klasę `Vector`, która działa bardzo podobnie do listy tablicowej. Próba użycia tej klasy w NetBeans spowoduje wyświetlenie ostrzeżenia, że używa się przestarzałego rozwiązania. Wynika to z faktu, że lista tablicowa stanowi ulepszoną wersję wektorów.

Przejście przez elementy struktury danych w pętli

Jeśli chcemy w sposób sekwencyjny skorzystać ze wszystkich elementów listy, zastosujmy metodę `iterator()`, która zwraca obiekt `Iterator` przechowujący elementy listy w postaci, której można użyć w pętli:

```
Iterator it = golfer.iterator();
```

Zgodnie z informacjami z wcześniejszej części dnia iterator służy do sekwencyjnego przejścia przez elementy. W tym konkretnym przypadku możemy więc użyć `it` wszędzie tam, gdzie obsługiwany jest interfejs `Iterable`.

Poniższa pętla `for` używa iteratora i wszystkich jego metod, aby przejść przez całą zawartość listy:

```
for (Iterator i = golfer.iterator(); i.hasNext(); ) {
    String name = (String) i.next();
    System.out.println(name);
}
```

Następny z dzisiejszych projektów będzie dotyczyć wypełniania i pobierania danych z listy tablicowej. Klasa `CodeKeeper`, przedstawiona na listingu 8.2, przechowuje zestaw kodów tekstowych. Część z nich jest wbudowana, a część dostarcza użytkownik klasy. Ponieważ ilość miejsca niezbędnego do przechowywania kodów nie jest znana do momentu uruchomienia programu, do przechowywania danych wykorzystamy listę tablicową zamiast zwykłej tablicy. Utwórz klasę w NetBeans i pamiętaj, aby umieścić ją w pakiecie `com.java21days`.

LISTING 8.2. Pełna treść pliku CodeKeeper.java

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class CodeKeeper {
 6:     ArrayList list;
 7:     String[] codes = { "alfa", "lambda", "gamma", "delta", "zeta" };
 8:
 9:     public CodeKeeper(String[] userCodes) {
10:         list = new ArrayList();
11:         // wczytaj wbudowane kody
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // wczytaj kody użytkownika
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // wyświetl wszystkie kody
20:         for (Iterator ite = list.iterator(); ite.hasNext(); ) {
21:             String output = (String) ite.next();
22:             System.out.println(output);
23:        }
24:     }
25:
26:     private void addCode(String code) {
27:         if (!list.contains(code)) {
28:             list.add(code);
29:         }
30:     }
31:
32:     public static void main(String[] arguments) {
33:         CodeKeeper keeper = new CodeKeeper(arguments);
34:     }
35: }
```

NetBeans może wyświetlić ostrzeżenie, że klasa zawiera kilka niesprawdzonych lub niebezpiecznych instrukcji. Nie jest to jednak tak poważny problem, jak mogłoby się wydawać. Kod działa poprawnie i nie jest niebezpieczny.

Ostrzeżenie to jednak bardzo silna sugestia, że istnieje lepszy sposób korzystania z list tablicowych i innych struktur. Zajmiemy się nim pod koniec dnia.

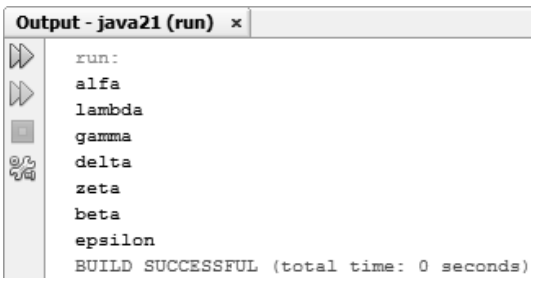
Klasa `CodeKeeper` używa zmiennej egzemplarzowej `list` typu `ArrayList`, aby przechowywać treść kodów.

Najpierw zostają wczytane do listy kody wbudowane (pochodzą ze zwykłej tablicy). Realizujący to zadanie kod znajduje się w wierszach od 12. do 14.

Następnie dodawane są kody wprowadzone przez użytkownika w wierszu poleceń (wiersze od 16. do 18.). Dodanie kodu realizuje metoda `addCode()` znajdująca się w wierszach od 26. do 30. Metoda dodaje kod tylko wtedy, gdy nie ma go jeszcze na liście. W celu realizacji sprawdzenia wykorzystuje metodę `contains(Object)`.

Dodaj odpowiednie argumenty wiersza poleceń, wybierając polecenie *Run/Set Project Configuration/Customize*. Argumentami powinna być lista kodów oddzielona spacjami.

Po dodaniu kodów do listy wyświetlamy jej zawartość. Uruchomienie klasy z argumentami `"beta"` i `"epsilon"` spowoduje wyświetlenie wyniku przedstawionego na rysunku 8.3.



RYSUNEK 8.3. Modyfikacja i wyświetlenie listy tablicowej

Prostsza wersja pętli `for` może posłużyć do iteracji przez zawartość struktury danych. Pętla przyjmuje postać `for (`*`zmienna : struktura`*`)`, gdzie *`struktura`* to struktura danych implementująca interfejs `Iterator`. Część *`zmienna`* deklaruje obiekt, który będzie przechowywał każdy z elementów struktury w trakcie przechodzenia przez pętlę.

Oto wersja pętli `for`, która używa iteratora i jego metod do przejścia przez listę tablicową znajdującą się w zmiennej `golfer`:

```
for (Object name : golfer) {
    System.out.println(name);
}
```

Pętla może być stosowana dla dowolnej struktury danych obsługującej interfejs `Iterator`.

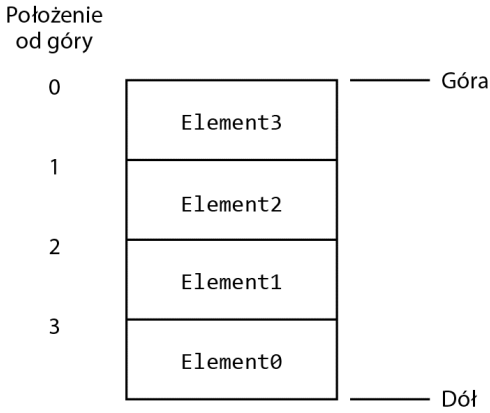
Stos

Stosy to struktur danych modelujące dostęp do informacji w określonej kolejności. Klasa `Stack` w Javie to implementacja typu LIFO, co oznacza, że ostatni dodany element będzie pierwszym usuniętym. Rysunek 8.4 przedstawia logiczną organizację stosu.

Zapewne zastanawiasz się, dlaczego liczby przy elementach nie odpowiadają wartościom liczonym od góry stosu. Pamiętaj, że elementy są dodawane od góry, więc `Element0`, który obecnie znajduje się na samym dnie, był pierwszym elementem dodanym do stosu. Z drugiej strony `Element3` był ostatnim dodanym elementem. Ponieważ to właśnie od znajduje się na szczycie stosu, zostanie usunięty jako pierwszy.

Klasa `Stack` definiuje tylko jeden konstruktor, który jest przy okazji konstruktorem domyślnym. Tworzy on pusty stos. Oto przykład użycia konstruktora:

```
Stack s = new Stack();
```



RYSUNEK 8.4. Organizacja stosu

Stosy w Javie posiadają własne metody służące do modyfikacji stosu.

Do dodawania nowego elementu służy metoda `push()`, która umieszcza go na szczycie stosu:

```
s.push("Jeden");
s.push("Dwa");
s.push("Trzy");
s.push("Cztery");
s.push("Pięć");
s.push("Sześć");
```

Kod umieści na stosie sześć tekstów, a ostatni z nich (`"Sześć"`) znajdzie się na samej górze. Do usuwania elementów ze stosu służy metoda `pop()`, która ściąga element ze szczytu stosu:

```
String s1 = (String) s.pop();
String s2 = (String) s.pop();
```

Kod pobiera ze stosu dwa najwyżej umieszczone teksty i pozostawia na stosie pozostałe cztery. Po jego wykonaniu w zmiennej `s1` znajduje się tekst `"Sześć"`, a w zmiennej `s2` tekst `"Pięć"`.

Aby pobrać element znajdujący się na szczycie stosu bez ściągania go z niego, użyj metody `peek()`:

```
String s3 = (String) s.peek();
```

Wywołanie metody `peek()` zwróci tekst `"Cztery"`, ale pozostawi ten tekst na stosie. Poszukiwanie elementu na stosie odbywa się za pomocą metody `search()`:

```
int i = s.search("Dwa");
```

Metoda `search()` zwraca odległość od szczytu stosu do elementu, jeśli zostanie on znaleziony. W przeciwnym razie zwraca –1. Tekst `"Dwa"` jest trzecim elementem od szczytu, więc metoda `search()` zwróci wartość 2.

Ostatnią metodą zdefiniowaną w klasie `Stack` jest `empty()`. Metoda sprawdza, czy stos jest pusty:

```
boolean isEmpty = s.empty();
```

Uwaga

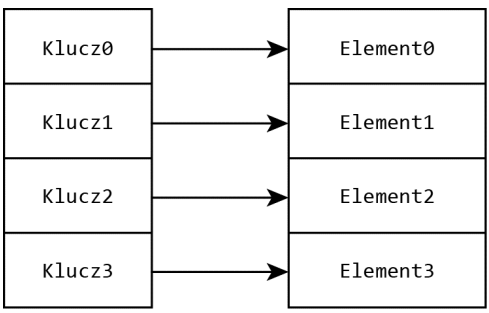
> Podobnie jak we wszystkich innych strukturach danych Javy, które korzystają z indeksów, klasa `Stack` zwraca położenie zaczynające się od indeksu o numerze 0. Element na szczycie ma indeks 0, czwarty element ma indeks 3 itd.

Interfejs Map

Interfejs `Map` definiuje mechanizm do implementacji struktury danych z odwzorowaniem kluczy, czyli do pobierania obiektów na podstawie pewnego klucza. Klucz pełni tę samą rolę co indeks tablicy — to unikatowa wartość służąca do uzyskania dostępu do konkretnej wartości przechowywanej w strukturze danych.

Konkretne wykorzystanie podejścia typu klucz-wartość zapewniają klasy takie jak `HashMap`, które implementują interfejs `Map`. Więcej informacji na temat `HashMap` pojawi się w następnym punkcie.

Interfejs `Map` definiuje sposób zapisywania i pobierania informacji na podstawie pewnego klucza. Jest w pewien sposób podobny do klasy `ArrayList`, w której to dostęp do elementów odbywa się za pomocą indeksu, choć tym razem jest to pewien klucz. Kluczem w interfejsie `Map` może być w zasadzie wszystko. Można tworzyć własne klasy jako klucze pozwalające pobierać i modyfikować dane w słowniku. Rysunek 8.5 przedstawia, w jaki sposób klucze są odwzorowywane na dane.



RYSUNEK 8.5. Organizacja struktury danych klucz-wartość

Interfejs `Map` deklaruje wiele różnych metod służących do korzystania z danych przechowywanych w słowniku. Klasy implementujące interfejs muszą zaimplementować je wszystkie, aby mogły być w pełni użyteczne. Metody `put(String, Object)` i `get(String, Object)` służą do umieszczania i pobierania elementów ze słownika.

Przyjmijmy, że `look` to obiekt implementujący interfejs `Map`. Poniższy kod ilustruje, w jaki sposób dodać elementy przy użyciu metody `put()`:

```
Rectangle r1 = new Rectangle(0, 0, 5, 5);
look.put("mały", r1);
Rectangle r2 = new Rectangle(0, 0, 15, 15);
look.put("średni", r2);
Rectangle r3 = new Rectangle(0, 0, 25, 25);
look.put("duży", r3);
```

Kod dodaje do słownika trzy obiekty `Rectangle` (z pakietu `java.awt`), wykorzystując teksty jako klucze. Aby pobrać element, użyj metody `get()` i wskaż odpowiedni klucz:

```
Rectangle r = (Rectangle) look.get("średni");
```

Można również usunąć element za pomocą klucza, stosując w tym celu metodę `remove()`:

```
look.remove("duży");
```

Aby dowiedzieć się, ile elementów przechowuje struktura, użyj metody `size()`, która działa w ten sam sposób co dla klasy `ArrayList`:

```
int size = look.size();
```

Można także sprawdzić, czy struktura jest pusta, wywołując metodę `isEmpty()`:

```
boolean isEmpty = look.isEmpty();
```

Tablice mieszające

Klasa `HashMap` implementuje interfejs `Map` i zapewnia pełną implementację struktury danych klucz-wartość. Tablice mieszające umożliwiają przechowywanie danych powiązanych z kluczem i posiadają efektywność zdefiniowaną zgodnie z ustalonym współczynnikiem wypełnienia. Współczynnik wypełnienia to wartość z przedziału od 0,0 do 1,0, która określa, jak i kiedy tablica mieszająca alokuje miejsce na dodatkowe elementy.

Podobnie jak listy tablicowe, tablice mieszające mają określoną pojemność (czyli ilość zarezerwowanej pamięci). Tablica mieszająca alokuje pamięć, porównując aktualny rozmiar z iloczynem pojemności i współczynnika wypełnienia. Jeśli rozmiar tablicy mieszającej przekracza ten iloczyn, tablica zwiększa swoją pojemność, przeliczając skróty (hasze) wszystkich elementów.

Współczynnik wypełnienia bliski 1,0 zapewnia bardziej efektywne użycie pamięci, ale prawdopodobnie wyszukanie konkretnego elementu będzie trwało dłużej. Podobnie, współczynniki wypełnienia bliższe 0,0 zapewnią bardziej efektywne wyszukiwanie, ale spowodują dużą alokację niepotrzebnej nikomu pamięci. Określenie współczynnika dla własnej tablicy mieszającej zależy od tego, czy priorytetem ma być wydajność, czy efektywność pamięciowa.

Utworzenie tablicy mieszającej może się odbyć na jeden z trzech sposobów. Pierwszy z konstruktorów tworzy domyślną tablicę mieszającą o początkowej pojemności 16 elementów i współczynniku wypełnienia wynoszącym 0,75:

```
HashMap hash = new HashMap();
```

Drugi konstruktor tworzy tablicę mieszającą z wskazaną pojemnością początkową i współczynnikiem wypełnienia wynoszącym 0,75:

```
HashMap hash = new HashMap(20);
```

Trzeci i ostatni konstruktor tworzy tablicę mieszającą o określonej pojemności początkowej i wskazanym współczynniku wypełnienia:

```
HashMap hash = new HashMap(20, 0.5F);
```

Wszystkie metody abstrakcyjne wskazane w interfejsie `Map` są oczywiście dostępne w klasie `HashMap`. Klasa definiuje jeszcze kilka innych metod specyficznych dla tablic mieszających. Jedną z nich jest metoda `clear()`, która czyści tablicę mieszającą ze wszystkich kluczy i elementów:

```
hash.clear();
```

Metoda `containsValue(`*`Object`*`)` sprawdza, czy obiekt znajduje się w tablicy mieszającej:

```
Rectangle box = new Rectangle(0, 0, 5, 5);
boolean isThere = hash.containsValue(box);
```

Metoda `containsKey(`*`String`*`)` przeszukuje tablicę w poszukiwaniu wskazanego klucza:

```
boolean isThere = hash.containsKey("mały");
```

Tablice mieszające stosuje się przede wszystkim w sytuacjach, w których przeszukiwanie elementów na podstawie wartości byłoby zbyt kosztowne. Struktura bywa przydatna, gdy korzysta się ze złożonych danych i bardziej efektywne lub wygodne będzie użycie klucza niż porównywanie poszczególnych obiektów.

Klucz, nazywany też kodem haszującym, to specjalna wartość, która unikatowo identyfikuje element w tablicy mieszającej.

Technika wyliczania i korzystania z kodów haszujących do przechowywania obiektów i łatwego odnoszenia się do nich jest powszechnie stosowana w bibliotece klas Javy. Klasa wszystkich innych klas, czyli `Object`, definiuje metodę `hashCode()` przysłanianą przez większość standardowych klas Javy. Dowolna klasa definiująca metodę `hashCode()` może być efektywnie przechowywana i pobierana w tablicy mieszającej. Klasa, która ma trafić do tablicy mieszającej, musi również implementować metodę `equals()`, która sprawdza, czy dwa obiekty są równoważne. Metoda `equals()` najczęściej porównuje wszystkie zmienne egzemplarza pod kątem ich równości.

W następnym projekcie, którym się dziś zajmiemy, użyjemy tablic mieszających w aplikacji sklepu.

Aplikacja `ComicBooks` wyświetla ceny komiksów kolekcjonerskich w zależności od ich wartości bazowej i stanu. Stan określa jedna z następujących wartości: `"jak nowy"`, `"prawie jak nowy"`, `"bardzo dobry"`, `"dobry"`, `"średni"`, `"słaby"`. Każdy ze stanów ma wpływ na wartość komiksu:

- Komiks `"jak nowy"` ma cenę trzy razy wyższą niż bazowa.
- Komiks `"prawie jak nowy"` ma cenę dwa razy wyższą niż bazowa.
- Komiks `"bardzo dobry"` ma cenę 1,5 razy wyższą niż bazowa.
- Komiks `"dobry"` ma cenę odpowiadającą cenie bazowej.
- Komiks `"średni"` ma cenę odpowiadającą połowie ceny bazowej.
- Komiks `"słaby"` ma cenę odpowiadającą ćwiartce ceny bazowej.

Aby powiązać tekst typu `"jak nowy"` lub `"bardzo dobry"` z wartością liczbową, zastosujemy tablicę mieszającą. Kluczami będą opisy stanu komiksu, a wartościami liczby zmiennoprzecinkowe, takie jak 3,0, 1,5 lub 0,25.

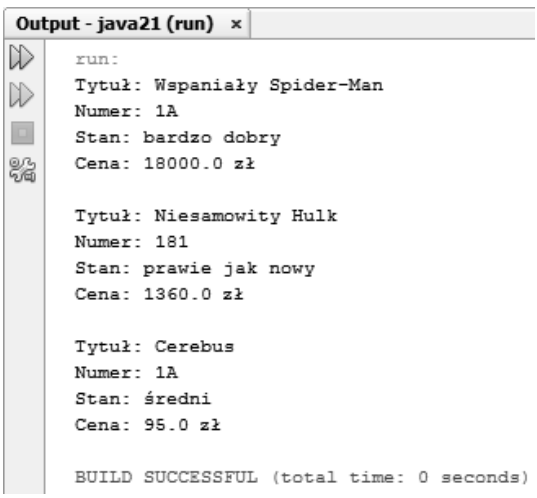
Wpisz kod przedstawiony na listingu 8.3 w NetBeans jako klasę `ComicBooks` z pakietu `com.java21days`.

LISTING 8.3. Pełna treść pliku ComicBooks.java

```
1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class ComicBooks {
 6:
 7:     public ComicBooks() {
 8:     }
 9:
10:     public static void main(String[] arguments) {
11:         // ustawienie obiektu HashMap
12:         HashMap quality = new HashMap();
13:         float price1 = 3.00F;
14:         quality.put("jak nowy", price1);
15:         float price2 = 2.00F;
16:         quality.put("prawie jak nowy", price2);
17:         float price3 = 1.50F;
18:         quality.put("bardzo dobry", price3);
19:         float price4 = 1.00F;
20:         quality.put("dobry", price4);
21:         float price5 = 0.50F;
22:         quality.put("średni", price5);
23:         float price6 = 0.25F;
24:         quality.put("słaby", price6);
25:         // konfiguracja kolekcji
26:         Comic[] comix = new Comic[3];
27:         comix[0] = new Comic("Wspaniały Spider-Man", "1A", "bardzo dobry",
28:             12_000.00F);
29:         comix[0].setPrice( (Float) quality.get(comix[0].condition) );
30:         comix[1] = new Comic("Niesamowity Hulk", "181", "prawie jak nowy",
31:             680.00F);
32:         comix[1].setPrice( (Float) quality.get(comix[1].condition) );
33:         comix[2] = new Comic("Cerebus", "1A", "średni", 190.00F);
34:         comix[2].setPrice( (Float) quality.get(comix[2].condition) );
35:         for (int i = 0; i < comix.length; i++) {
36:             System.out.println("Tytuł: " + comix[i].title);
37:             System.out.println("Numer: " + comix[i].issueNumber);
38:             System.out.println("Stan: " + comix[i].condition);
39:             System.out.println("Cena: " + comix[i].price + " zł\n");
40:         }
41:     }
42: }
43:
44: class Comic {
45:     String title;
46:     String issueNumber;
47:     String condition;
48:     float basePrice;
```

```
49:     float price;
50:
51:     Comic(String inTitle, String inIssueNumber, String inCondition,
52:         float inBasePrice) {
53:
54:         title = inTitle;
55:         issueNumber = inIssueNumber;
56:         condition = inCondition;
57:         basePrice = inBasePrice;
58:     }
59:
60:     void setPrice(float factor) {
61:         price = basePrice * factor;
62:     }
63: }
```

Po uruchomieniu aplikacji `ComicBooks` na ekranie pojawi się wynik jak na rysunku 8.6.



RYSUNEK 8.6. Przechowywanie wartości w tablicy mieszającej

Aplikacja `ComicBooks` została zaprojektowana jako dwie klasy: główna klasa aplikacyjna o nazwie `ComicBooks` i klasa pomocnicza o nazwie `Comic`.

W aplikacji tablica mieszająca powstaje w wierszach od 12. do 24.

W wierszu 12. tworzony jest sam obiekt tablicy mieszającej.

Następnie powstaje zmienna typu `float` o nazwie `price1`, która otrzymuje wartość 3,0. Wartość trafia do tablicy mieszającej w powiązaniu z kluczem `"jak nowy"`. (Pamiętaj, że tablica mieszająca, podobnie jak wiele innych struktur danych, potrafi przechowywać tylko obiekty. Wartość typu `float` zostanie więc automatycznie zamieniona na obiekt `Float`).

Cały proces jest powtarzany dla wszystkich pozostałych stanów komiksu — od `"prawie jak nowy"` do `"słaby"`.

Po zakończeniu konfiguracji tablicy mieszającej powstaje tablica obiektów `Comic` o nazwie `comix`, która zawiera kilka przykładowych komiksów będących w sprzedaży.

Konstruktor `Comic` zostaje wywołany z czterema argumentami: tytułem książki, numerem wydania, stanem i ceną bazową. Pierwsze trzy argumenty to tekst, czwarty jest typu `float`.

Po utworzeniu obiektu `Comic` wywołujemy jego metodę `setPrice(float)`, aby ustawić cenę sprzedaży na podstawie stanu komiksu. Oto przykład (wiersz 29.):

```
comix[0].setPrice( (Float) quality.get(comix[0].condition) );
```

Metoda `get(String)` tablicy mieszającej służy do pobrania wartości na podstawie klucza, którym jest stan komiksu. Zwrócona wartość jest typu `Object`. (W wierszu 29. `comix[0].`
`↪condition` równa się `"bardzo dobry"`, więc metoda `get()` zwraca wartość zmiennoprzecinkową 1,5).

Ponieważ metoda `get()` zwraca typ `Object`, rzutujemy ją na właściwy typ, czyli `Float`. Dzięki automatycznemu rozpakowywaniu `Float` zamieni się w `float`.

Ten sam schemat działania zastosowano dla dwóch pozostałych komiksów.

Wiersze od 35. do 40. wyświetlają informacje na temat każdego komiksu z tablicy `comix`.

Klasa `Comix` znajduje się w wierszach od 44. do 63. Zawiera pięć zmiennych egzemplarza: obiekty typu `String` o nazwach `title`, `issueNumber` i `condition` oraz wartości zmiennoprzecinkowe `basePrice` i `price`.

Konstruktor klasy znajdujący się w wierszach od 51. do 58. ustawia wartości trzech zmiennych egzemplarza na podstawie argumentów przekazanych do konstruktora.

Metoda `setPrice(float)` z wierszy od 60. do 62. ustawia cenę komiksu. Argumentem przekazanym do metody jest wartość typu `float`. Kod ustawia cenę sprzedaży komiksu na cenę bazową pomnożoną przez przekazaną wartość. Jeśli komiks ma cenę bazową wynoszącą 1000 zł i mnożnik równy 2,0, jego cena sprzedaży wynosi 2000 zł.

Tablice mieszające to bardzo przydatne struktury danych, pozwalające na łatwą modyfikację dużej ilości danych. Tablice mieszające są powszechnie obsługiwane przez standardową bibliotekę Javy dzięki metodzie z klasy `Object`, co pokazuje, jak istotne są one w programach Javy.

Obiekty generyczne

Struktury danych, o których dziś mówimy, należą do najbardziej użytecznych klas standardowej biblioteki klas Javy.

Tablice mieszające, listy tablicowe, stosy i inne struktury zdefiniowane w pakiecie `java.util` są przydatne niezależnie od typu programu, który się aktualnie tworzy. Prawie każdy program komputerowy w jakiś sposób obsługuje dane.

Opisane struktury danych nadają się do użycia w kodzie, który w sposób uogólniony może dotyczyć różnych klas obiektów. Metoda modyfikująca listę tablicową może równie dobrze obsługiwać teksty, bufory tekstów, tablice znaków lub inne obiekty reprezentujące

tekst. Metoda w programie księgowym może przyjmować obiekty reprezentujące liczby całkowite, liczby zmiennoprzecinkowe lub inne obiekty matematyczne, aby wyliczać wynik finansowy.

Elastyczność ma swoją cenę — gdy struktura danych obsługuje każdy rodzaj obiektu, kompilator Javy nie może wyświetlić ostrzeżenia, jeśli struktura nie zostanie prawidłowo zastosowana.

Na przykład aplikacja `ComicBooks` używa tablicy mieszającej o nazwie `quality` do przechowywania mnożników cen uzależnionych od opisu stanu komiksu. Oto przykład umieszczenia pary klucz-wartość dla klucza `"prawie jak nowy"`:

```
quality.put("prawie jak nowy", 1.50F);
```

Zgodnie z przyjętym projektem tablica mieszająca `quality` powinna przechowywać tylko i wyłącznie wartości zmiennoprzecinkowe (jako obiekty `Float`). Klasa skompiluje się jednak prawidłowo niezależnie od tego, jakiej klasy obiekt przekażemy jako wartość. Można się łatwo pomylić i umieścić w odwzorowaniu tekst jako wartość dla pewnego klucza. Oto przykład:

```
quality.put("prawie jak nowy", "1.50");
```

Klasa skompiluje się bez przeszkód i dopiero w momencie wykonywania aplikacji pojawi się błąd `ClassCastException` dla poniższego wiersza kodu:

```
comix[1].setPrice( (Float) quality.get(comix[1].condition) );
```

Powodem błędu jest próba rzutowania wartości dla klucza `"prawie jak nowy"` na typ `Float`, choć otrzymany obiekt jest tekstem o treści `"1.50"`.

Błędy wykonania są znacznie bardziej problematyczne od błędów kompilacji. Błąd kompilacji uniemożliwi uruchomienie programu, więc trzeba go poprawić, aby móc kontynuować. Błąd wykonania może się wkraść do kodu i pozostawać niezauważony przez programistę, choć będzie sprawiał problemy użytkownikom programu.

Na szczęście możemy wskazać w strukturze danych klasę, której obiekty ma przechowywać. Ta funkcja języka Java nosi nazwę programowania uogólnionego (generycznego).

Dodatkową informację umieszcza się w instrukcjach, które przypisują strukturę danych do zmiennej lub też tworzą obiekt. Klasę lub klasy umieszcza się wewnątrz znaków < i >, tuż po nazwie klasy ze strukturą danych. Oto przykład:

```
ArrayList<Integer> zipCodes = new ArrayList<>();
```

Powyższy kod tworzy listę tablicową, która służy do przechowywania obiektów `Integer`. Kompilator automatycznie zgaduje typ klasy, gdy w tej samej instrukcji widzi ponowne użycie znaków < i >. Połączenie znaków <> po nazwie klasy nazywa się czasem **operatorem diamentowym**. Oto inny przykład:

```
HashMap<String, Float> quality = new HashMap<String, Float>();
```

Operator diamentowy automatycznie tak dopasowuje klasę, która powinna się pojawić między znakami, aby cała instrukcja miała sens.

Ponieważ lista została zadeklarowana dla określonej klasy, poniższe polecenia spowodują zgłoszenie przez NetBeans błędu kompilacji i podświetlenie wybranego wiersza:

```
zipCodes.add("90210");
zipCodes.add("02134");
zipCodes.add("20500");
```

Kompilator automatycznie rozpoznaje, że obiekty `String` nie pasują do tej listy tablicowej. Na liście wartości mogą się pojawić jedynie liczby całkowite:

```
zipCodes.add(90210);
zipCodes.add(02134);
zipCodes.add(20500);
```

Przekazywane liczby zostaną automatycznie zamienione na obiekty `Integer`.

Struktury danych, które obsługują kilka klas — na przykład tablice mieszające — przyjmują więcej nazw klas, a poszczególne nazwy rozdziela się znakami przecinka wewnątrz nawiasów `<i>`.

Aplikacja `ComicBooks` może w łatwy sposób skorzystać z programowania uogólnionego. Wystarczy zmienić wiersz 10. z listingu 8.3 na następujący:

```
HashMap<String, Float> quality = new HashMap<>();
```

Powyższy zapis wymusza stosowanie obiektów `String` jako kluczy i obiektów `Float` jako wartości. Po dodaniu takiego zapisu nie uda się dodać jako wartości elementu innego niż liczba zmiennoprzecinkowa, bo pojawi się błąd kompilacji.

Programowanie generyczne upraszcza również pobieranie obiektu ze struktury danych, ponieważ nie trzeba już rzutować pobranej wartości na odpowiednią klasę. Tablica mieszająca `quality` nie wymaga zastosowania rzutowania na obiekt `Float` w instrukcji takiej jak poniższa:

```
comix[1].setPrice(quality.get(comix[1].condition));
```

Od strony stylistycznej dodawanie nazw klas generycznych w deklaracjach zmiennych i konstruktorach nie jest szczególnie przyjemne. Gdy jednak przyzwyczaisz się do korzystania z tych mechanizmów (oraz automatycznego otaczania obiektem, automatycznego rozpakowywania i nowej pętli `for`), zauważysz, że struktury danych są łatwe w użyciu, a dzięki nim mniej narażamy się na błędy.

Klasa `CodeKeeper2` z listingu 8.4 to nowa wersja klasy `CodeKeeper`, która została zmodyfikowana tak, aby używała programowania generycznego, zgadywała typy i korzystała z pętli `for`, potrafiącej iterować przez strukturę danych, na przykład listę tablicową.

LISTING 8.4. Pełna treść pliku CodeKeeper2.java

```
1: package com.java21days;
2:
3: import java.util.*;
4:
5: public class CodeKeeper2 {
6:     ArrayList<String> list;
```

```
 7:     String[] codes = { "alfa", "lambda", "gamma", "delta", "zeta" };
 8:
 9:     public CodeKeeper2(String[] userCodes) {
10:         list = new ArrayList<>();
11:         // wczytaj wbudowane kody
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // wczytaj kody użytkownika
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // wyświetl wszystkie kody
20:         for (String code : list) {
21:             System.out.println(code);
22:        }
23:     }
24:
25:     private void addCode(String code) {
26:         if (!list.contains(code)) {
27:             list.add(code);
28:         }
29:     }
30:
31:     public static void main(String[] arguments) {
32:         CodeKeeper2 keeper = new CodeKeeper2(arguments);
33:     }
34: }
```

Jedyne modyfikacje pojawiają się w wierszu 6., który dodaje deklarację listy tablicowej jako zbioru tekstów; wierszu 10., który automatycznie zgaduje właściwy typ; a także w wierszach od 20. do 22., gdzie zastosowano pętlę `for` do wyświetlenia wszystkich kodów.

Wyliczenia

Typowym zastosowaniem stałych w języku Java bywa nadanie sensownej etykiety zbiorowi liczb całkowitych. Skorzystaliśmy z tej sztuczki już wcześniej, w trakcie obsługi zbiorów bitów:

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

Stałe są bardzo użyteczne, bo zapewniają dodatkową informację w instrukcji, która je zawiera. Porównaj dwie poniższe instrukcje realizujące to samo zadanie:

```
setConnectionType(1);
setConnectionType(ConnectionAttributes.WRITABLE);
```

Druga wersja jest znacznie łatwiejsza do zrozumienia dla programisty czytającego kod.

Java posiada specjalny typ danych przeznaczony dla zbiorów stałych. Ma on pewne zalety w porównaniu ze stałymi umieszczonymi w klasie. Słowo kluczowe `class` zamienia się na `enum`, a poszczególne nazwy wartości oddziela się znakami przecinka:

```
public enum Compass {
    NORTH,
    EAST,
    SOUTH,
    WEST,
    NORTHEAST,
    SOUTHEAST,
    SOUTHWEST,
    NORTHWEST
}
```

Każda z wartości jest niejawnie typu `static` i `final`, podobnie jak w stałych. Może być wykorzystywana w instrukcjach, wywołaniach metod i innym kodzie w dokładnie taki sam sposób jak stałe klasowe. Oto przykład aplikacji, która korzysta z wyliczenia:

```
public class DirectionSetter {
    Compass current;
    public void setDirection(Compass dir) {
        current = dir;
    }

    public static void main(String[] arguments) {
        DirectionSetter app = new DirectionSetter();
        app.setDirection(Compass.WEST);
        System.out.println(app.current);
    }
}
```

Klasa ustawia aktualną zmienną egzemplarza na `WEST` z wyliczenia `Compass`, a następnie wyświetla zmienną, czyli tekst `"WEST"`.

Zaletą stosowania `enum` zamiast klasy i stałych jest możliwość wyłapania przez kompilator błędów, gdy spróbuje się użyć niewłaściwej wartości. Jedynymi wartościami możliwymi do wysłania do `setDirection(Compass)` są wartości dotyczące wyliczenia `Compass`.

Dla porównania metoda, która przyjmowała wartości `ConnectionAttributes`, tak naprawdę mogła przyjąć dowolną wartość będącą liczbą całkowitą.

Istnieją też inne zalety wyliczeń — mogą one istnieć w podobny sposób jak klasy i posiadać własne metody.

Jeśli kiedykolwiek będziesz potrzebował zestawu stałych, skorzystaj z wyliczeń.

Podsumowanie

Dziś poznaliśmy kilka struktur danych, z których możemy korzystać we własnych programach Javy:

- **zestawy bitów** — duże zbiory wartości typu włączony-wyłączony;
- **listy tablicowe** — tablice, które potrafią się w razie potrzeby dynamicznie rozszerzać i zwężać;
- **stosy** — struktury, w których ostatnio dodany element jest pierwszym pobieranym;
- **tablice mieszające** — obiekty pobierane i zapisywane za pomocą unikatowych kluczy.

Wymienione struktury danych stanowią część pakietu `java.util`, który zawiera wiele przydatnych klas służących do obsługi danych, dat, tekstów itp. Dodanie elementów generycznych oraz nowej pętli `for` rozbudowuje możliwości i elastyczność języka.

Wprowadziliśmy również wyliczenia — nowy typ danych służący do reprezentowania zestawu powiązanych wartości jako stałych.

Nauka sposobu organizacji danych w języku Java ma znaczenie dla ogólnej płynności programowania. Tego rodzaju wiedza przyda się niezależnie od tego, czy tworzy się serwlety, aplikacje typu desktop, czy cokolwiek innego, co wymaga reprezentowania danych na różne sposoby.

Pytania i odpowiedzi

Pytanie: Projekt `HolidaySked` mógłby zostać zaimplementowany jako tablica wartości logicznych. Czy zastosowanie tego rozwiązania jest mniej preferowane od zbioru bitów?

Odpowiedź: To zależy. W trakcie prac ze strukturami danych zauważysz, że istnieje wiele sposobów na implementację pewnych zagadnień. Zbiór bitów jest preferowanym rozwiązaniem, jeśli ilość pamięci zajmowanej przez program ma znaczenie, bo zajmuje jej po prostu mniej. Z drugiej strony tablica wartości logicznych jest lepsza, jeśli znaczenie ma szybkość działania programu. W przedstawionym przykładzie `HolidaySked` rozmiar tablicy jest tak mały, że różnica w rozmiarze i szybkości byłaby pomijalna, ale gdy tworzy się bardziej zaawansowane aplikacje, wybór odpowiedniego rozwiązania ma duże znaczenie.

Pytanie: Ostrzeżenie kompilatora Javy dotyczące struktur danych, które nie stosują rozwiązań generycznych, jest dosyć przekonujące. Tworzenie klasy, która zawiera „niesprawdzone lub niebezpieczne operacje", nie wydaje się zbyt dobrym rozwiązaniem. Czy istnieje jakiś powód, by używać starego kodu i zrezygnować z wersji generycznej?

Odpowiedź: Ostrzeżenie kompilatora dotyczące bezpieczeństwa jest nieco przesadzone. Programiści Javy przez wiele lat wykorzystywali listy tablicowe, tablice mieszające i inne podobne struktury podczas tworzenia oprogramowania, które działało pewnie i bezpiecznie. Brak elementów generycznych powodował jedynie, że trzeba było sobie zadać więcej

trudu, aby nie pojawiły się problemy i błędy wykonania związane z umieszczeniem w strukturze klas nieodpowiedniego typu.

W zasadzie należałoby powiedzieć, że zastosowanie generycznych struktur danych zwiększa bezpieczeństwo, zamiast sugerować, że stare rozwiązanie jest niebezpieczne.

Osobiście stosuję wersję generyczną w nowym kodzie lub w starym kodzie, który jest znacząco modyfikowany, ale pozostawiam nietknięty stary kod, który działa prawidłowo.

Quiz

Oceń przyswojenie dzisiejszego materiału, odpowiadając na poniższe pytania.

Pytania

1. Którego z rodzajów danych nie można przechowywać w tablicy mieszającej?
   a) `String`
   b) `int`
   c) Oba typy można przechowywać w tablicy mieszającej.
2. Tworzymy listę tablicową i dodajemy do niej trzy teksty: `"Tomasz"`, `"Edward"` i `"Jacek"`. Została wywołana metoda `remove("Edward")`. Która z poniższych metod `ArrayList` pobierze tekst `"Jacek"`?
   a) `get(1);`
   b) `get(2);`
   c) `get("Jacek");`
3. Która z tych klas implementuje interfejs `Map`?
   a) `Stack`
   b) `HashMap`
   c) `BitSet`

Odpowiedzi

1. Odpowiedź: c. W starszych wersjach języka Javy do reprezentacji typów podstawowych, takich jak `int`, niezbędne było ich opakowanie w obiekt (`Integer` dla liczb całkowitych). Obecnie nie jest to konieczne. Typy podstawowe są automatycznie konwertowane na obiekt w procesie nazywanym opakowywaniem.
2. Odpowiedź: a. Indeksy poszczególnych elementów w tablicy mogą ulec zmianie w momencie dodawania lub usuwania nowych elementów. Ponieważ `"Jacek"` stanie się drugim elementem tablicy po usunięciu elementu `"Edward"`, pobierze go wywołanie `get(1)`.
3. Odpowiedź: b. Interfejs ten implementuje klasa `HashMap`, jak i bardzo podobna do niej klasa `Hashtable`.

Zadania z certyfikacji

Poniższe pytanie może się pojawić na teście ze znajomości języka programowania Java. Odpowiedz na nie bez zaglądania do przerobionego materiału i bez używania kompilatora Javy do przetestowania kodu.

Przy założeniu, że:

```
public class Recursion {
    public int dex = -1;

    public Recursion() {
        dex = getValue(17);
    }

    public int getValue(int dexValue) {
        if (dexValue > 100) {
            return dexValue;
        } else {
            return getValue(dexValue * 2);
        }
    }

    public static void main(String[] arguments) {
        Recursion r = new Recursion();
        System.out.println(r.dex);
    }
}
```

Jaki będzie wynik działania aplikacji?

A. –1

B. 17

C. 34

D. 136

Odpowiedź na pytanie znajduje się w przykładach dostępnych do pobrania z witryny wydawnictwa. Zajrzyj do pliku *certyfikacja8.txt*.

Ćwiczenia

Aby poszerzyć swoją wiedzę o poruszanych dziś tematach, wykonaj poniższe ćwiczenia.

1. Dodaj dwa dodatkowe stany komiksów do aplikacji `ComicBooks`: `"całkowicie nowy"`, który sprzedaje się po cenie pięciokrotnie wyższej niż bazowa, i `"bez okładki"`, który sprzedaje się po jednej dziesiątej ceny bazowej.
2. Zmodyfikuj tak aplikację `ComicBooks`, aby przy obsłudze stanu komiksu korzystała z wyliczeń.

Rozwiązania ćwiczeń znajdują się w przykładach dostępnych do pobrania z witryny wydawnictwa pod adresem *http://helion.pl/ksiazki/ja21d7.htm*.

Skorowidz

K

L, Ł

M

N

O

P

R

S, Ś

T

U

W

X

Z

Ż

Java™

w 21 dni

Już za 21 dni możesz być znakomitym programistą Javy!

Java jest zorientowanym obiektowo i bezpiecznym językiem programowania. Użytkownicy uważają go za łatwiejszy do nauki od języka C++ i taki, który jednocześnie pozwala na unikanie istotniejszych błędów. W Javie można tworzyć kod działający na dowolnym komputerze lub urządzeniu z maszyną wirtualną Javy, niezależnie od tego, czy jest to serwer z Linuksem, komputer Apple Mac z OS, czy telefon z Androidem. Co ważne, Java jest cały czas konsekwentnie rozwijana przez firmę Oracle. A przy tym wszystkim programowania w Javie można się bardzo szybko nauczyć!

Dzięki niniejszej książce bez problemu zrozumiesz wszystkie najważniejsze elementy najnowszej wersji języka Java 8. Dogłębnie poznasz sam język, a także jego podstawowe biblioteki. Książkę podzielono na 21 lekcji, kładąc nacisk na praktyczne aspekty programowania. Właściwie od początku będziesz pisać aplikacje dla różnych środowisk, w tym mobilnych. Niezależnie od tego, czy dopiero zaczynasz przygodę z programowaniem, czy też znasz już inne języki, dzięki tej książce nabierzesz biegłości w posługiwaniu się Javą i przygotujesz się do rozwijania własnych projektów — nawet tych bardzo ambitnych!

W tej książce zawarto:

- gruntowne podstawy języka Java, opis obiektów Javy, klas, interfejsów i pakietów
- wyjaśnienie zasad rządzących wątkami, asercją i obsługą wyjątków w Javie
- opis najbardziej użytecznych klas Javy, w tym klas Swing, do budowy interfejsów graficznych
- wyjaśnienie specyficznych struktur danych, takich jak listy tablicowe, stosy, mapy, tablice mieszające i zbiory bitów
- omówienie zaawansowanych funkcji Javy, takich jak obsługa wejścia i wyjścia za pomocą strumieni, domknięcia, korzystanie z baz danych i obsługa kanałów RSS za pomocą XOM
- wprowadzenie do budowania aplikacji dla urządzeń z Androidem

Rogers Cadenhead — jest programistą aplikacji internetowych z wieloletnim doświadczeniem. Ukończył studia na University of North Texas. W latach 2006 – 2008 brał udział w pracach nad specyfikacją RSS 2.0. Jest autorem licznych, bardzo popularnych książek dotyczących programowania i internetu.

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Informatyka w najlepszym wydaniu

SAMS

ISBN 978-83-283-2621-7

9 788328 326217

cena: 99,00 zł

ALWAYS LEARNING

PEARSON