WYDANIE XIII

JAVA

Podstawy

ORACLE

Cay S. Horstmann

Tytuł oryginału: Core Java, Volume I: Fundamentals, 13th Edition

Tłumaczenie: Łukasz Piwko z wykorzystaniem fragmentów książki „Java. Techniki zaawansowane. Wydanie XI" w przekładzie Piotra Rajcy

ISBN: 978-83-289-3023-0

Ilustracja z okładki: Jon Chica/Shutterstock
Rysunek 1.1: Sourceforge
Rysunki 2.2, 3.2 - 3.5, 4.9, 5.4, 7.2, 10.5, 10.6, 11.1: Oracle Corporation
Rysunki 2.3 - 2.6, 12.2: Eclipse Foundation, Inc.
Rysunek 4.2: Violet UML Editor

Authorized translation from the English language edition, entitled CORE JAVA VOLUME I: FUNDAMENTALS, 13TH Edition 9780135328378 by Cay S. Horstmann published by Pearson Education, Inc, publishing as Oracle Press Copyright © 2025 Pearson Education, Inc.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/javp13.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/javp13
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

9

Kolekcje

Dobór struktur danych do użycia w programie może mieć niebagatelne znaczenie dla późniejszej implementacji metod i wydajności całej aplikacji. Decydując się na konkretne struktury, należy odpowiedzieć sobie na pytania typu: czy konieczne będzie przeszukiwanie tysięcy (może nawet milionów) posortowanych elementów? Czy konieczne będzie szybkie wstawianie i usuwanie elementów do i ze środka uporządkowanego szeregu elementów? Czy konieczne będzie powiązanie wartości z ich kluczami?

Ten rozdział traktuje o strukturach danych dostępnych w Java Collections Framework. Na studiach informatycznych przedmiot dotyczący *struktur danych* trwa z reguły jeden semestr, dzięki czemu ta ważna tematyka została wyczerpująco przedstawiona w bardzo licznych publikacjach i opracowaniach. W tej książce prezentujemy odmienne podejście w stosunku do innych publikacji z tego zakresu — pomijamy teorię, a koncentrujemy się na zastosowaniu kolekcji dostępnych w API Javy.

9.1. Architektura kolekcji Javy

Początkowo w Javie dostępnych było tylko kilka klas implementujących najbardziej przydatne struktury danych: `Vector`, `Stack`, `Hashtable`, `BitSet` oraz interfejs `Enumerable` odpowiadający za abstrakcyjny mechanizm dostępu do elementów w każdym z tych kontenerów. Było to z pewnością mądre posunięcie ze strony projektantów języka — opracowanie pełnej biblioteki klas kolekcyjnych wymaga czasu i doświadczenia.

Projektanci doszli do wniosku, że czas na zaprezentowanie pełnego zestawu struktur danych nadszedł w Java 1.2. Musieli dokonać wielu trudnych wyborów, ponieważ chcieli stworzyć bibliotekę mającą niewielkie rozmiary i łatwą do opanowania dla programistów. Starali się uniknąć poziomu złożoności charakteryzującego bibliotekę C++ `STL` (ang. *Standard Template Library*), jednocześnie próbując skorzystać z dobrodziejstw algorytmów generycznych, których pionierem była właśnie wymieniona biblioteka. Stare klasy musiały pasować do nowej architektury.

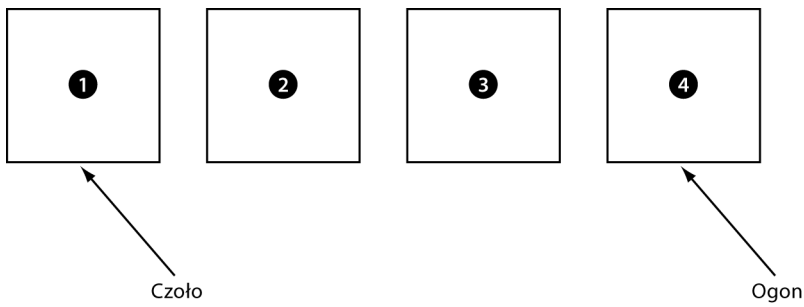
Jak to zwykle bywa przy projektowaniu bibliotek kolekcji, kilkakrotnie stawano przed trudnym wyborem, co zaowocowało kilkoma osobliwymi decyzjami projektowymi. W tym podrozdziale przedstawiamy podstawową strukturę architektury kolekcji Javy, pokazujemy sposoby jej wykorzystania oraz wyjaśniamy, czym kierowali się projektanci, podejmując niektóre bardziej kontrowersyjne decyzje.

9.1.1. Oddzielenie warstwy interfejsów od warstwy klas konkretnych

Podobnie jak większość nowoczesnych bibliotek struktur danych, Java Collections Framework dzieli się na warstwę **interfejsów** i warstwę **implementacji**. Przyjrzymy się temu podziałowi na przykładzie znanej struktury danych o nazwie **kolejka** (ang. *queue*).

Interfejs `Queue` pozwala na dodawanie elementów na końcu kolejki, usuwanie ich z początku struktury danych oraz sprawdzanie liczby elementów w kolejce. Ta struktura danych znajduje zastosowanie przy tworzeniu zbiorów obiektów, z których elementy są pobierane zgodnie z zasadą „pierwszy przyjdzie, pierwszy wyjdzie" (zobacz rysunek 9.1).

Rysunek 9.1. Kolejka



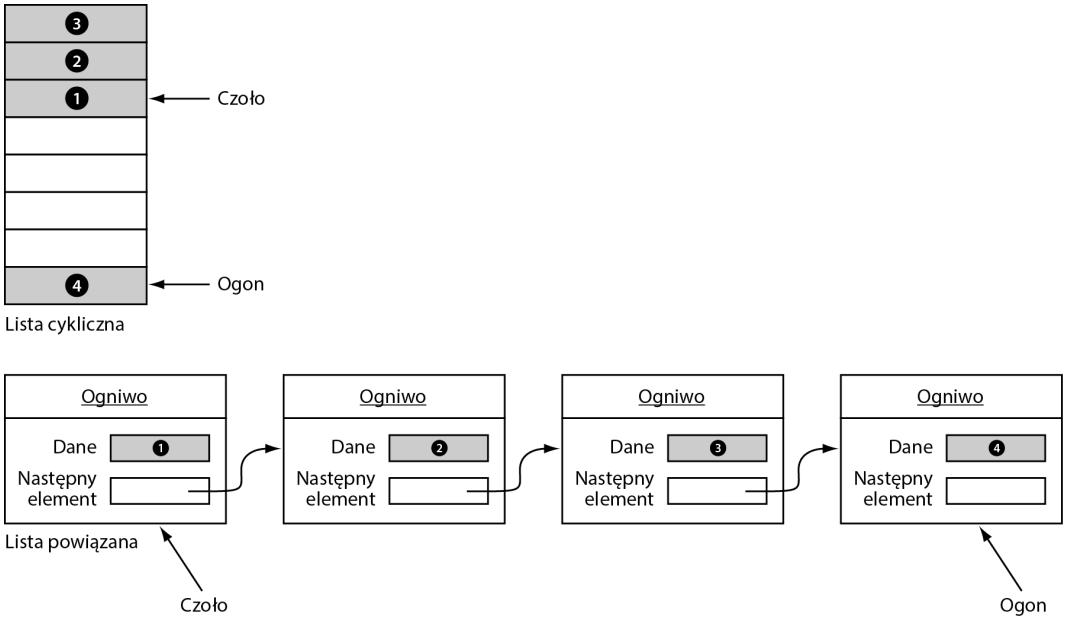
Interfejs `Queue` w swojej uproszczonej formie może wyglądać następująco:

```
public interface Queue<E>     // uproszczona wersja interfejsu z API Javy
{
  void add(E e);
  E remove();
  int size();
}
```

Sam interfejs nie zawiera żadnych danych na temat implementacji kolejki. Z dwóch najczęściej spotykanych implementacji kolejki jedna działa na zasadzie listy cyklicznej, a druga listy powiązanej (rysunek 9.2).

Każdą z tych implementacji można zrealizować za pomocą klasy implementującej interfejs `Queue`. W API Javy klasy te noszą nazwy `ArrayDeque` i `LinkedList`.

```
public class ArrayDeque<E> implements Queue<E> // uproszczona wersja z API Javy
{
    private int first;
    private int last;
    private E[] elements;
```



Rysunek 9.2. Implementacje kolejki

```
    public ArrayDeque(int numElements) { . . . }
    public void add(E e) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    . . .
}

public class LinkedList<E> implements Queue<E> // uproszczona wersja z API Javy
{
    private Node<E> first;
    private Node<E> last;

    public LinkedList() { . . . }
    public void add(E e) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    . . .
}
```

Programista używający w programie kolejki po utworzeniu kolekcji nie musi wiedzieć, która jej implementacja została użyta. Dlatego dobrym rozwiązaniem jest używanie klas konkretnych *tylko* do konstruowania obiektów kolekcyjnych. **Typów interfejsowych** należy używać do przechowywania referencji do tych obiektów.

```
Queue<Customer> expressLane = new ArrayDeque<>(100);
expressLane.add(new Customer("Henryk"));
```

Dzięki takiemu podejściu w razie zmiany zdania można łatwo użyć innej implementacji. Wystarczy tylko jedna zmiana w programie — wywołanie konstruktora. Jeśli na przykład dojdziemy do wniosku, że lepszym wyborem byłaby lista powiązana, zmieniamy kod na następujący:

```
Queue<Customer> expressLane = new LinkedList<>();
expressLane.add(new Customer("Henryk"));
```

Jest jeszcze jeden powód, dla którego należy pozostać przy interfejsie. Zarówno klasa `ArrayDeque`, jak i `LinkedList` ma metody, które nie mają związku z kolejkowaniem. Przypadkowe użycie jednej z nich zostanie zgłoszone jako błąd kompilacji, ponieważ nie można ich wywoływać na zmiennych typu `Queue`.

Co sprawia, że wybieramy jedną implementację zamiast drugiej? Interfejs nie dostarcza żadnych informacji na temat wydajności. Lista cykliczna nieco lepiej wykorzystuje pamięć. Ma jednak ograniczoną pojemność i wymaga realokacji, gdy się zapełni. Pomiary pokazują, że `ArrayDeque` generalnie przewyższa wydajnością `LinkedList`, ponieważ ma mniejszy wpływ na system usuwania nieużytków. Kiedy struktura `ArrayDeque` została dodana do Javy 6, programiści korzystający z interfejsu `Queue` mogli na nią przejść przez zmianę jednego wiersza kodu.

9.1.2. Interfejs Collection

Interfejsem o kluczowym znaczeniu w Java Collections Framework jest interfejs `Collection`. Dwie z jego metod mają fundamentalne znaczenie:

```
public interface Collection<E>
{
  boolean add(E element);
  Iterator<E> iterator();
  . . .
}
```

Pozostałe metody tego interfejsu omawiamy dalej.

Metoda `add` dodaje elementy do kolekcji i zwraca wartość `true`, jeśli wykonana przez nią operacja powoduje zmianę stanu kolekcji, lub wartość `false`, jeśli kolekcja nie ulega zmianie. Jeśli na przykład do zbioru (ang. *set*) spróbujemy dodać obiekt, który już się tam znajduje, metoda `add` nie przyniesie żadnego skutku, ponieważ w zbiorach nie może być duplikatów.

Metoda `iterator` zwraca obiekt implementujący interfejs `Iterator`. Za pomocą tego obiektu można odwiedzić kolejno wszystkie znajdujące się w kolekcji elementy. Opis iteratorów znajduje się w następnej sekcji.

9.1.3. Iteratory

W interfejsie `Iterator` występują cztery metody:

```
public interface Iterator<E>
{
  boolean hasNext();
  E next();
  default void remove();
  default void forEachRemaining(Consumer<? super E> action);
}
```

Wywołując wielokrotnie metodę next, można kolejno odwiedzić wszystkie elementy kolekcji. Jeśli metoda ta napotka koniec kolekcji, zgłosi wyjątek NoSuchElementException. Dlatego przed nią należy zawsze wywoływać metodę hasNext, która zwraca wartość true, jeśli iterator nie dotarł jeszcze do końca i jest przynajmniej jeden element do przejrzenia. Aby przejrzeć wszystkie elementy kolekcji, należy utworzyć obiekt typu Iterator i wywoływać metodę next tak długo, jak metoda hasNext zwraca wartość true. Na przykład:

```
Collection<String> coll = . . .;
Iterator<String> iter = coll.iterator();
while (iter.hasNext())
{
  String element = iter.next();
  obróbka elementu
}
```

Pętlę taką można wyrazić zwięźlej za pomocą konstrukcji typu for each:

```
for (String element : coll)
{
  obróbka elementu
}
```

Kompilator konwertuje pętlę w stylu for each na pętlę z iteratorem.

Pętlę typu for each można stosować do wszystkich obiektów implementujących interfejs Iterable, w którym znajduje się metoda abstrakcyjna:

```
public interface Iterable<E>
{
  Iterator<E> iterator();
  ...
}
```

Interfejs Collection rozszerza interfejs Iterable. Dzięki temu pętli for each można używać do działań na wszystkich kolekcjach z Java Collections Framework.

Nie trzeba nawet pisać pętli. W zamian można wywołać metodę Collection.forEach lub Iterator.forEachRemaining z wyrażeniem lambda wykorzystującym element. Wyrażenie to jest wywoływane na wszystkich elementach kolekcji lub pozostałych elementach, które iterator może odwiedzić.

```
coll.forEach(element -> jakieś działania na elemencie);
iter.forEachRemaining(element -> operacje na elemencie);
```

Kolejność odwiedzania elementów zależy od rodzaju kolekcji. W przypadku listy ArrayList iterator zaczyna od indeksu o numerze 0 i zwiększa ten numer w każdym kolejnym powtórzeniu (iteracji). Natomiast dostęp do elementów w zbiorze HashSet odbywa się w zasadzie losowo. Pewne jest, że przemierzając tę kolekcję, uzyskamy dostęp do każdego jej elementu, ale nie wiadomo, w jakiej kolejności będzie się to odbywać. Nie sprawia to zazwyczaj problemu, ponieważ w działaniach typu obliczanie sumy lub zliczanie elementów pasujących do wzorca kolejność jest nieistotna.

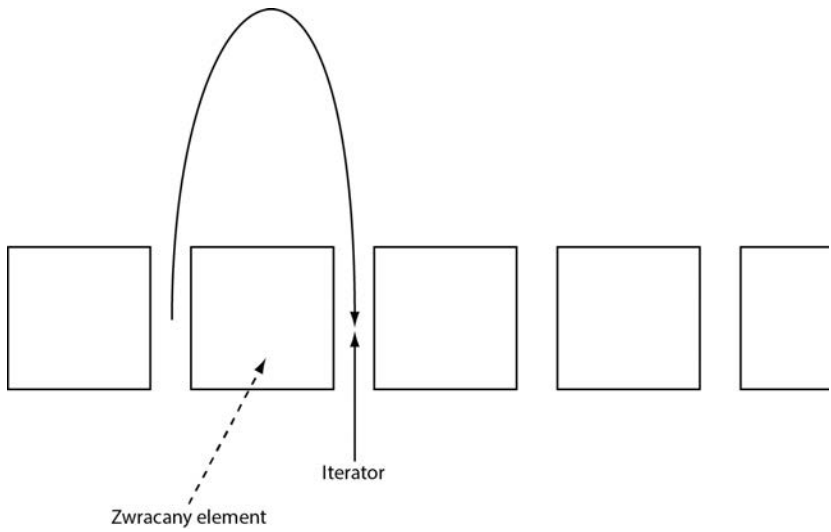
Ci, którzy znają wcześniejsze wersje Javy, zauważą, że metody `next` i `hasNext` interfejsu `Iterator` spełniają tę samą funkcję co `nextElement` i `hasMoreElements` w interfejsie `Enumeration`. Projektanci Java Collections Framework mogli wykorzystać w swojej pracy interfejs `Enumeration`, ale nie podobały im się niezgrabne nazwy jego metod. Dlatego utworzono nowy interfejs z krótszymi nazwami metod.

Pomiędzy iteratorami w bibliotece kolekcyjnej w Javie a iteratorami z innych bibliotek istnieje ważna różnica koncepcyjna. Modelem iteratorów w tradycyjnych bibliotekach, takich jak Standard Template Library w C++, są indeksy tablicowe. Za pomocą takiego tradycyjnego iteratora element znajdujący się w określonym miejscu można odszukać w podobny sposób jak element `a[i]` w tablicy, jeśli znany jest indeks `i`. Niezależnie od wyszukiwania, iterator taki można przesunąć do kolejnego elementu, co niczym się nie różni od operacji zwiększania indeksu za pomocą instrukcji `i++`, bez wyszukiwania. Iteratory w Javie działają nieco inaczej. Wyszukiwanie i zmiana położenia są ze sobą ściśle związane. Jedynym sposobem na znalezienie elementu jest wywołanie metody `next`, a to powoduje przejście do kolejnego elementu.

Iteratory w Javie należy wyobrażać sobie jako obiekty znajdujące się *pomiędzy elementami*. W chwili wywołania metody `next` iterator *przeskakuje* kolejny element i zwraca referencję do elementu, który właśnie przeskoczył (zobacz rysunek 9.3).

Istnieje jeszcze inna ciekawa analogia. Instrukcję `Iterator.next` można traktować jako odpowiednik instrukcji `InputStream.read`. Odczyt bajta ze strumienia automatycznie oznacza jego „połknięcie". Kolejne wywołanie metody `read` powoduje „połknięcie" i zwrócenie następnego bajta z danych wejściowych. W podobny sposób seria wywołań metody `next` zwraca wszystkie elementy znajdujące się w kolekcji.

Rysunek 9.3. Przesuwanie iteratora



Metoda remove z interfejsu Iterator usuwa element zwrócony przez ostatnie wywołanie metody next. W większości sytuacji jest to rozsądne działanie — aby podjąć decyzję o usunięciu elementu, najczęściej trzeba go wpierw zobaczyć. Aby usunąć element znajdujący się w określonym miejscu, także trzeba za niego przejść. Na przykład poniższa procedura usuwa pierwszy element z kolekcji łańcuchów:

```
Iterator<String> iter = coll.iterator();
iter.next();      // przejście za pierwszy element
iter.remove();    // usunięcie pierwszego elementu
```

Między metodami next i remove istnieje pewna bardzo ważna zależność. Tej drugiej nie można wywołać, jeśli wcześniej nie wywołano pierwszej. Próba zrobienia tego zakończy się rzuceniem wyjątku IllegalStateException.

Aby usunąć dwa kolejne elementy, nie można zastosować dwóch kolejnych wywołań metody remove:

```
iter.remove();
iter.remove();    // Błąd!
```

Najpierw trzeba wywołać metodę next, aby przejść za kolejny element, który ma zostać usunięty.

```
iter.remove();
iter.next();
iter.remove();    // OK
```

9.1.4. Generyczne metody użytkowe

Dzięki temu, że interfejsy Collection i Iterator są generyczne, można pisać metody użytkowe operujące na dowolnych rodzajach kolekcji. Poniżej znajduje się przykładowa metoda generyczna sprawdzająca, czy dowolnego rodzaju kolekcja zawiera określony element:

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Twórcy Java Collections Framework doszli do wniosku, że niektóre z tych metod są tak przydatne, iż powinny być dostępne w bibliotece. Dzięki temu użytkownicy API nie muszą wielokrotnie wynajdywać koła. Jedna z tych metod nosi nazwę contains.

W rzeczywistości w interfejsie Collection<E> znajduje się spora liczba przydatnych metod, które muszą być udostępniane przez wszystkie implementujące go klasy:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean add(E obj)
boolean addAll(Collection<? extends E> from)
```

```
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
Object[] toArray()
T[] toArray(T[] a)
```

Interfejs `Collection` deklaruje metody `equals` i `hashCode`, które są odziedziczone z klasy `Object`. Nie ma technicznego wymogu, aby dodać te deklaracje. Jest to po prostu dobre miejsce na umieszczenie dokumentacji API kolekcji. Podinterfejsy `Set` i `List` umożliwiają specyficzne zachowanie metody `equals`. Inne kolekcje mogą dziedziczyć metodę `Object.equals` lub mogą ją przesłaniać i porównywać elementy w jakiś sposób. W drugim przypadku konieczne jest przesłonięcie także metody `hashCode`.

Przeznaczenie wielu z tych metod łatwo odgadnąć po nazwie. Pełny ich opis znajduje się na końcu podrozdziału w wyciągach z API.

Java Collections Framework zaprojektowano przed dodaniem typów generycznych do Javy. W celu zachowania zgodności wstecznej metody `contains` i `remove` mają parametr typu `Object`, a nie `E`. Metody `containsAll`, `removeAll` i `retainAll` mają parametr typu `Collection<?>`, a nie `Collection<? extends E>`.

To znaczy, że błędy dotyczące typów mogą nie zostać wykryte w czasie kompilacji. Spójrz na przykład na poniższy kod, w którym przypadkowo usunięto `String` z kolekcji obiektów typu `Path`:

```
Collection<Path> paths = . . .;
paths.remove("/tmp"); // Przechodzi kompilację, ale może nie powodować żadnego efektu
```

Oczywiście definiowanie tylu metod we wszystkich klasach implementujących interfejs `Collection` jest bardzo uciążliwe. Aby ułatwić życie programistom, utworzono klasę `Abstract` ↪`Collection` implementującą wszystkie metody tego interfejsu w kategorii metod `size` i `iterator`, które jako jedyne pozostały w niej abstrakcyjne. Na przykład:

```
public abstract class AbstractCollection<E>
            implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();
    public abstract int size();

    public boolean isEmpty()
    {
        return size() == 0;
    }

    public boolean contains(Object obj)
    {
        Iterator<E> it = iterator();
        if (o == null)
        {
            while (it.hasNext()) if (it.next() == null) return true;
        }
```

```
            else
            {
                while (it.hasNext()) if (o.equals(it.next())) return true;
            }
            return false;
        }
        . . .
    }
```

Konkretna klasa kolekcyjna może być rozszerzeniem klasy `AbstractCollection`. W klasie konkretnej konieczne jest zdefiniowanie metod `iterator i size`. Kolekcje modyfikowalne także wymagają metody `add`. O pozostałe metody zadbano w nadklasie abstrakcyjnej `AbstractCollection`. Jeśli jednak w podklasie istnieje możliwość zdefiniowania efektywniejszej metody `contains`, nic nie stoi na przeszkodzie, aby to zrobić.

Takie podejście jest już przestarzałe. Byłoby lepiej, gdyby metody te były metodami domyślnymi interfejsu `Collection`, ale takie metody jeszcze nie istniały, kiedy projektowano Java Collections Framework. Jednak tak nie jest, choć dodano kilka metod domyślnych. Trzy z nich dotyczą strumieni (o których jest mowa w drugim tomie). Metoda `toArray` jest opisana w punkcie 9.6.6. Ponadto następująca bardzo przydatna metoda służy do usuwania elementów spełniających określony warunek:

```
default boolean removeIf(Predicate<? super E> filter)
```

`java.util.Collection<E>` **1.2**

- `Iterator<E> iterator()`

  Zwraca obiekt `Iterator`, za pomocą którego można odwiedzać elementy kolekcji.

- `int size()`

  Zwraca liczbę elementów przechowywanych w kolekcji.

- `boolean isEmpty()`

  Zwraca wartość `true`, jeśli kolekcja nie zawiera żadnych elementów.

- `boolean contains(Object obj)`

  Zwraca wartość `true`, jeśli kolekcja zawiera obiekt identyczny z obiektem `obj`.

- `boolean containsAll(Collection<?> other)`

  Zwraca wartość `true`, jeśli kolekcja zawiera wszystkie elementy znajdujące się w innej kolekcji.

- `boolean add(E element)`

  Próbuje dodać element do kolekcji. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `boolean addAll(Collection<? extends E> other)`

  Próbuje dodać do kolekcji wszystkie elementy z kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `boolean remove(Object obj)`

  Usuwa obiekt `obj` z kolekcji. Zwraca wartość `true`, jeśli obiekt został znaleziony i usunięty.

- `boolean removeAll(Collection<?> other)`

  Usuwa z kolekcji wszystkie elementy, które można znaleźć w kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `default boolean removeIf(Predicate<? super E> filter)` **8**

  Usuwa wszystkie elementy, dla których filtr zwróci wartość `true`. Zwraca `true`, jeżeli w wyniku wywołania tej metody nastąpiły zmiany w kolekcji.

- `void clear()`

  Usuwa wszystkie elementy z kolekcji.

- `boolean retainAll(Collection<?> other)`

  Usuwa z kolekcji wszystkie elementy, które nie są takie same jak jeden z obiektów w kolekcji `other`. Zwraca wartość `true`, jeśli w wyniku wywołania w kolekcji nastąpiły zmiany.

- `Object[] toArray()`

  Zwraca tablicę obiektów zapełnioną elementami z kolekcji.

- `<T> T[] toArray(IntFunction<T[]> generator)` **11**

  Zwraca tablicę obiektów z kolekcji. Tablica ta jest utworzona przez generator, który zazwyczaj jest wyrażeniem konstrukcyjnym `T[]::new`.

`java.util.Iterator<E>` **1.2**

- `boolean hasNext()`

  Zwraca wartość `true`, jeśli są jeszcze elementy do odwiedzenia.

- `E next()`

  Zwraca następny obiekt do odwiedzenia. Wyrzuca wyjątek `NoSuchElementException`, jeśli osiągnie koniec kolekcji.

- `void remove()`

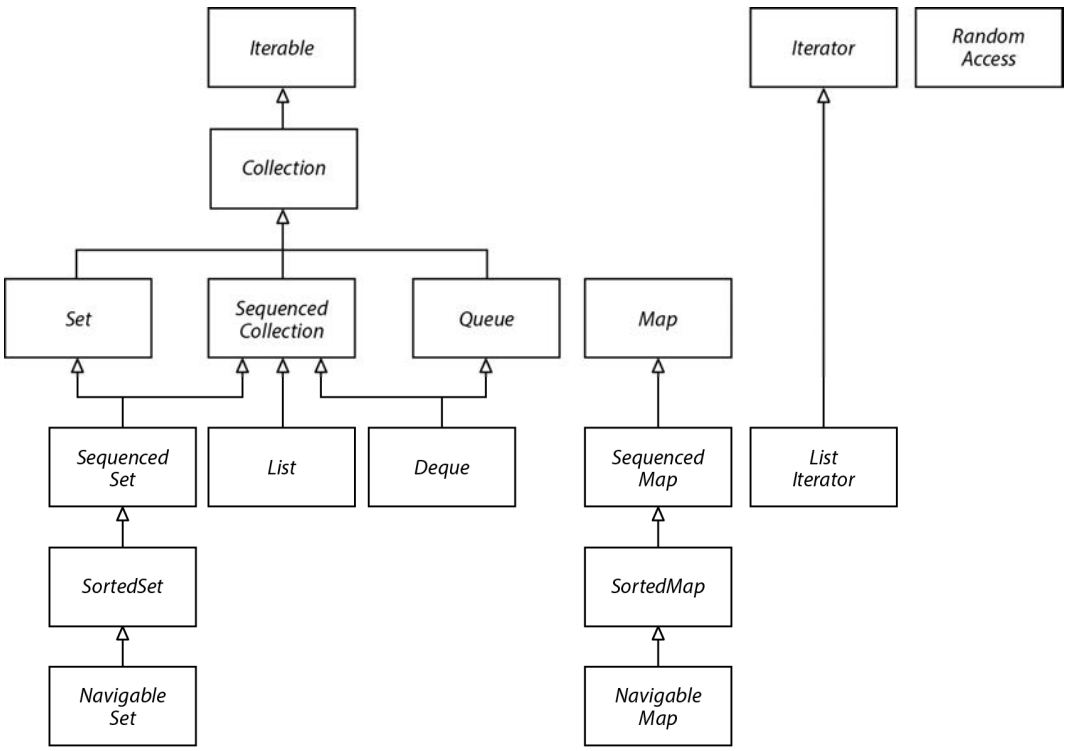
  Usuwa ostatnio odwiedzony obiekt. Wywołanie tej metody musi następować bezpośrednio po odwiedzeniu elementu. Jeśli kolekcja zmieniła się od ostatniego odwiedzenia elementu, metoda ta wyrzuci wyjątek `IllegalStateException`. Iteratory nie muszą obsługiwać tej operacji. Domyślna implementacja zgłasza wyjątek `UnsupportedOperationException`.

- `default void forEachRemaining(Consumer<? super E> action)` **8**

  Przegląda elementy i przekazuje je do określonej akcji tak długo, aż przejrzy wszystkie elementy albo akcja zgłosi wyjątek.

9.2. Interfejsy w systemie kolekcji Javy

System kolekcji Javy (ang. *Java Collections Framework*) zawiera definicje kilku interfejsów dla różnych typów kolekcji. Na rysunku 9.4 są wymienione wszystkie z nich oprócz interfejsów do programowania współbieżnego, które są opisane w rozdziale 10.



Rysunek 9.4. Interfejsy Java Collections Framework

Dwa podstawowe interfejsy kolekcyjne to `Collection` i `Map`. Elementy do kolekcji dodaje się za pomocą metody `add`:

```
boolean add(E element)
```

Ponieważ słowniki przechowują pary klucz – wartość, elementy wstawia się do nich za pomocą metody `put`:

```
V put(K key, V value)
```

Do odczytu elementów z kolekcji służy iterator. Elementy ze słownika można także odczytać za pomocą metody `get`:

```
V get(Object key)
```

Lista to **kolekcja uporządkowana**. Elementy są dodawane w określonych miejscach kontenera, a dostęp do nich można uzyskać na dwa sposoby: za pomocą iteratora lub przy użyciu indeksu

całkowitoliczbowego. Drugi z wymienionych sposobów nazywa się **dostępem swobodnym** (ang. *random access*), ponieważ elementy można przeglądać w dowolnej kolejności. Natomiast iterator umożliwia tylko przeglądanie elementów po kolei.

Interfejs `List` zawiera definicje kilku metod dotyczących dostępu swobodnego:

```
void add(int index, E element)
E remove(int index)
E get(int index)
E set(int index, E element)
```

W `List<Integer>` są dwie metody `remove`:

```
boolean remove(int index) // Usuwa element o danym indeksie
boolean remove(Integer o) // Usuwa element równy o
```

Gdy wywołamy metodę `remove` z parametrem typu `int`, to wybrana zostanie pierwsza wersja. Nie będzie opakowywania ani nie zostanie wyświetlone żadne ostrzeżenie. Nie o to może Ci chodzić:

```
List<Integer> ids = . . .;
int id = . . .;
if (id == 0) ids.remove(id); // Usuwa indeks 0, nie element o wartości 0
```

Dokumentacja API interfejsu `List` definiuje zachowanie metody `equals` dla list. Aby lista została uznana za równą innemu obiektowi, obiekt ten także musi być jakiegoś rodzaju listą. Ponadto obie listy muszą być tego samego rozmiaru. Podczas iteracji przez każdą z nich korespondujące elementy muszą być równe.

Trzeba uczciwie przyznać, że ten składnik systemu kolekcji nie najlepiej udał się projektantom Javy. W praktyce istnieją dwa rodzaje kolekcji uporządkowanych, z których każda charakteryzuje się innymi rozwiązaniami w zakresie optymalizacji wydajności. Kolekcja uporządkowana zbudowana na bazie tablicy zapewnia szybki dostęp swobodny do elementów. Metod klasy `List` dobrze jest używać z indeksami całkowitoliczbowymi. Natomiast lista powiązana, choć też uporządkowana, zapewnia powolny dostęp swobodny i najlepiej ją przeglądać liniowo za pomocą iteratora. Stworzenie dwóch interfejsów byłoby lepszym rozwiązaniem.

Aby uniknąć wykonywania operacji dostępu swobodnego w listach powiązanych, w Java SE 1.4 wprowadzono interfejs znacznikowy o nazwie `RandomAccess`. Nie posiada on żadnych metod, ale można za jego pomocą sprawdzić, czy określona kolekcja umożliwia szybki dostęp swobodny do elementów:

```
if (c instanceof RandomAccess)
  {
    przeglądanie swobodne
  }
  else
  {
    przeglądanie liniowe

  }
```

Interfejs `Set` ma taką samą metodę egzemplarza, jak interfejs `Collection`, ale jego metody są nieco bardziej restrykcyjne. Metoda `add` zbioru nie powinna dodawać duplikatów. Metoda `equals` powinna być zdefiniowana w taki sposób, aby dwa zbiory były identyczne, jeśli mają takie same elementy, ale niekoniecznie w takiej samej kolejności. Metoda `hashCode` dla dwóch zbiorów zawierających takie same elementy powinna zwracać tę samą wartość skrótu.

Po co tworzyć osobny interfejs, skoro sygnatury metod są takie same? Koncepcyjnie nie wszystkie kolekcje są zbiorami. Dzięki istnieniu interfejsu `Set` programiści mogą pisać metody, które działają tylko na zbiorach.

Aby zrozumieć, dlaczego to rozróżnienie jest ważne, pomyśl o porównywaniu. Dwie listy są równe, gdy zawierają te same elementy ułożone w jakiejś kolejności. Metoda `equals` dla klasy implementującej interfejs `Set` musi sprawdzać, czy argument także jest typu `Set`, a nie jakąkolwiek inną kolekcją.

Interfejsy `SortedSet` i `SortedMap` udostępniają obiekt komparatora użyty do sortowania oraz definiują metody tworzące widoki podzbiorów kolekcji. Szerzej na ten temat piszemy w podrozdziale 9.5.3, „Kopie i widoki”.

Interfejsy `NavigableSet` i `NavigableMap` zawierają dodatkowe metody służące do znajdowania następnego lub poprzedniego elementu w posortowanych zbiorach i słownikach. Implementują je klasy `TreeSet` i `TreeMap`. Operacje nawigacji można efektywnie zaimplementować w drzewiastych strukturach danych.

W Javie 21 wprowadzono interfejs `SequencedCollection<E>`, który zapewnia jednolity dostęp do pierwszego i ostatniego elementu kolekcji oraz umożliwia przeglądanie wstecz.

```
E getFirst()
E getLast()
void addFirst(E e)
void addLast(E e)
E removeFirst()
E removeLast()
SequencedCollection<E> reversed()
```

Wcześniej te operacje były wykonywane przez różne metody list, zbiorów i kolejek dwukierunkowych. `Podinterfejs` precyzuje typ zwrotny metody `reversed` do `SequencedSet`. Interfejs `SequencedMap` ma analogiczne metody dla słowników.

9.3. Konkretne klasy kolekcyjne

Tabela 9.1 przedstawia zestawienie kolekcji dostępnych w Java Collections Framework oraz ich krótkie opisy (dla uproszczenia pominęliśmy kolekcje bezpieczne dla wątków, które zostały opisane w rozdziale 10.).

Tabela 9.1. Kolekcje konkretne w bibliotece Javy

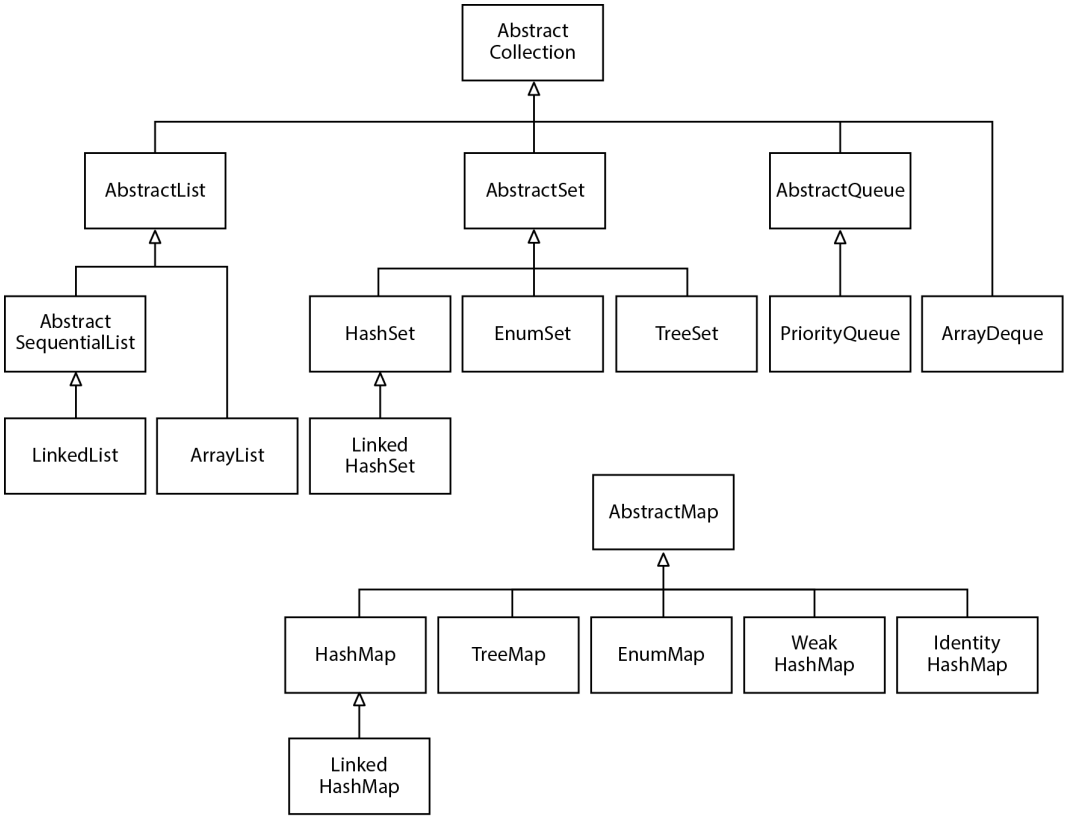
Typ kolekcji	Opis	Zobacz
`ArrayList`	Indeksowana lista o dynamicznie zmieniających się rozmiarach	Punkt 9.3.2
`LinkedList`	Uporządkowana lista pozwalająca na szybkie wstawianie i usuwanie elementów w dowolnej lokalizacji	Punkt 9.3.1
`ArrayDeque`	Nieposiadająca ani początku, ani końca lista cykliczna	Punkt 9.3.5
`HashSet`	Nieuporządkowana kolekcja, w której wszystkie obiekty muszą być unikatowe	Punkt 9.3.3
`TreeSet`	Uporządkowany zbiór	Punkt 9.4.5
`EnumSet`	Zbiór wartości typu wyliczeniowego	Punkt 9.4.6
`LinkedHashSet`	Zbiór pamiętający kolejność wstawianych do niego elementów	Punkt 9.4.5
`PriorityQueue`	Kolekcja pozwalająca na szybkie usunięcie najmniejszego elementu	Punkt 9.3.6
`HashMap`	Struktura danych przechowująca pary klucz – wartość	Punkt 9.4.1
`TreeMap`	Słownik sortujący klucze	Punkt 9.4.1
`EnumMap`	Słownik, w którym klucze są typami wyliczeniowymi	Punkt 9.4.6
`LinkedHashMap`	Słownik pamiętający kolejność wstawianych do niego elementów	Punkt 9.4.5
`WeakHashMap`	Słownik, którego wartości mogą zostać usunięte przez śmieciarkę, jeśli nie są używane gdzieś indziej	Punkt 9.4.4
`IdentityHashMap`	Słownik przechowujący klucze porównywane za pomocą operatora `==` zamiast metody `equals`	Punkt 9.4.7

Wszystkie klasy prezentowane w tabeli, z wyjątkiem tych, których nazwy kończą się słowem `Map`, implementują interfejs `Collection`. Słowniki implementują interfejs `Map` i zostały opisane w podrozdziale 9.4, „Słowniki”.

Na rysunku 9.5 pokazano relacje występujące między tymi klasami.

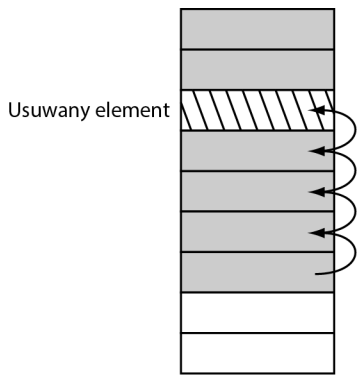
9.3.1. Listy powiązane

W wielu prezentowanych do tej pory przykładach kodu wykorzystywaliśmy tablice i ich dynamicznego krewniaka, czyli klasę `ArrayList`. Niestety tablice i listy tablicowe mają jedną poważną wadę — usuwanie elementów z ich środka jest mało efektywne, ponieważ czynność ta wymaga przesunięcia wszystkich elementów znajdujących się za tym usuwanym w stronę początku (zobacz rysunek 9.6). To samo dotyczy wstawiania elementów w środku struktury.

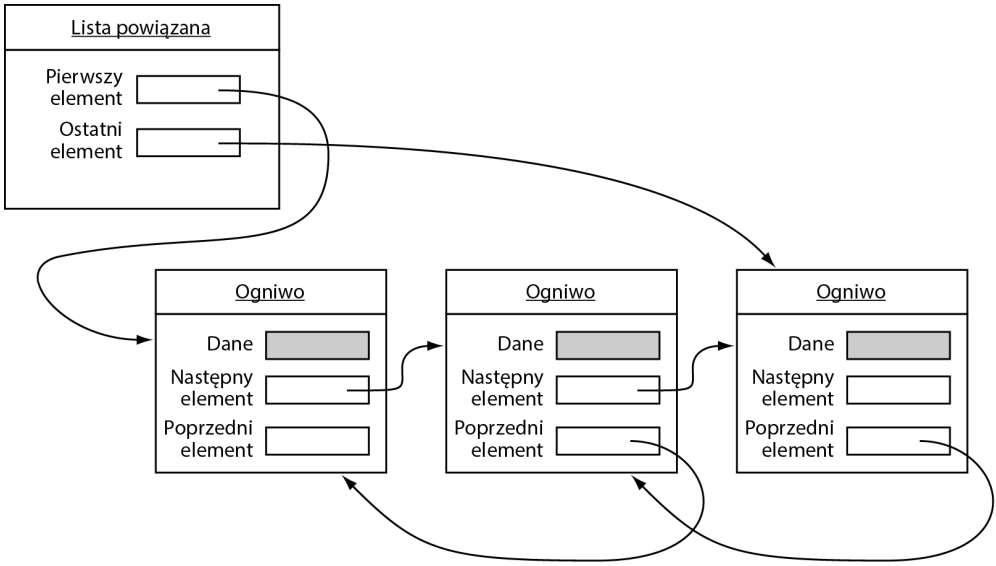


Rysunek 9.5. Klasy z systemu kolekcji Javy

Rysunek 9.6. Usuwanie elementu z tablicy

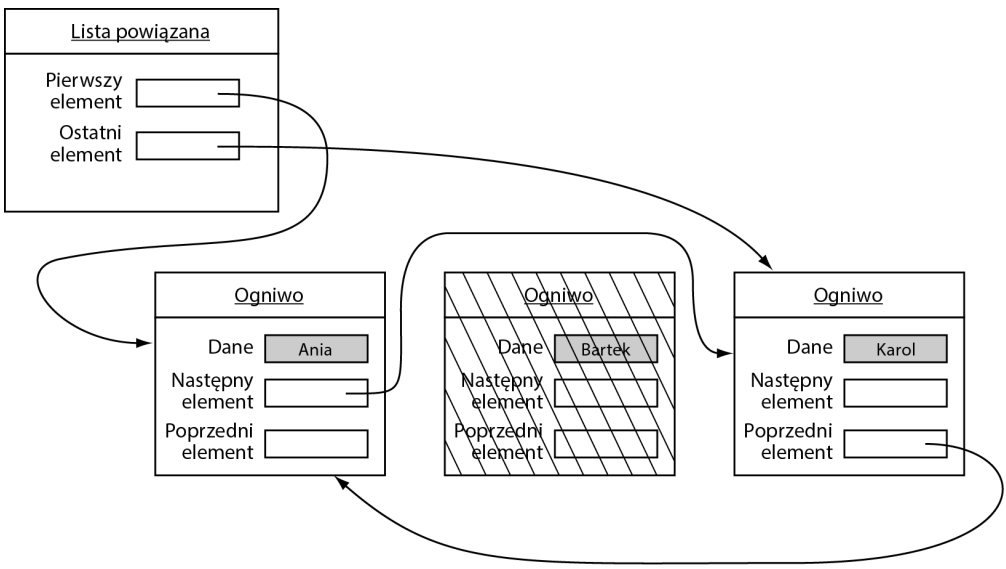


Problem ten pomaga rozwiązać inna powszechnie znana struktura danych, nazywana **listą powiązaną** (ang. *linked list*). Podczas gdy obiekty w tablicy są zapisywane w kolejnych komórkach pamięci, w liście powiązanej znajdują się one w osobnych **ogniwach** (ang. *link*). Każde ogniwo przechowuje także referencję do następnego ogniwa w szeregu. W Javie praktycznie wszystkie listy powiązane są **listami dwukierunkowymi** (ang. *doubly linked list*), co oznacza, że każde ogniwo przechowuje także referencję do ogniwa je poprzedzającego (zobacz rysunek 9.7).



Rysunek 9.7. Lista dwukierunkowa

Usuwanie elementów z środka listy powiązanej charakteryzuje się krótkim czasem wykonania, ponieważ aktualizowane są tylko elementy znajdujące się po obu stronach usuwanego obiektu (zobacz rysunek 9.8).



Rysunek 9.8. Usuwanie elementu z listy powiązanej

Wiele osób ukończyło kurs struktur danych, na którym uczy się implementacji list powiązanych. Niektórzy mogą mieć złe wspomnienia związane z plątaniną połączeń przy usuwaniu lub dodawaniu elementów do list powiązanych. Dla tych osób mamy dobrą wiadomość — w Java Collections Framework znajduje się gotowa do użycia klasa `LinkedList`.

Poniższy fragment programu dodaje do listy trzy elementy, a następnie usuwa drugi element:

```
var staff = new LinkedList<>();    // Klasa LinkedList implementuje interfejs List
staff.add("Ania");
staff.add("Bartek");
staff.add("Karol");
Iterator<String> iter = staff.iterator();
String first = iter.next();        // dojście do pierwszego elementu (Ania)
String second = iter.next();       // dojście do drugiego elementu (Bartek)
iter.remove();                     // usunięcie ostatnio odwiedzonego elementu (Karol)
```

Pomiędzy listami powiązanymi a kolekcjami generycznymi istnieje pewna istotna różnica. Lista powiązana jest **kolekcją uporządkowaną**, w której położenie obiektów ma znaczenie. Metoda `LinkedList.add` dodaje obiekt na końcu listy. Często jednak elementy są wstawiane do środka listy. Metoda `add` związana z położeniem elementu znajduje się w zestawie obowiązków iteratora, ponieważ iteratory służą do opisu położenia elementów w kolekcjach. Zastosowanie iteratorów do dodawania elementów jest uzasadnione tylko w uporządkowanych kolekcjach. Na przykład omawiane w następnej kolejności *zbiory* są strukturami danych, które nie porządkują w żaden sposób swoich elementów. Dlatego Java Collections Framework dostarcza podinterfejs `List➥Iterator`, który zawiera metodę `add`.

```
interface ListIterator<E> extends Iterator<E>
{
  void add(E element);
  . . .
}
```

W przeciwieństwie do metody `Collection.add`, ta metoda nie zwraca wartości logicznej — zakłada się, że metoda `add` zawsze modyfikuje listę.

Dodatkowo interfejs `ListIterator` zawiera dwie metody, za pomocą których można poruszać się po liście wstecz.

```
boolean hasPrevious()
E Previous()
```

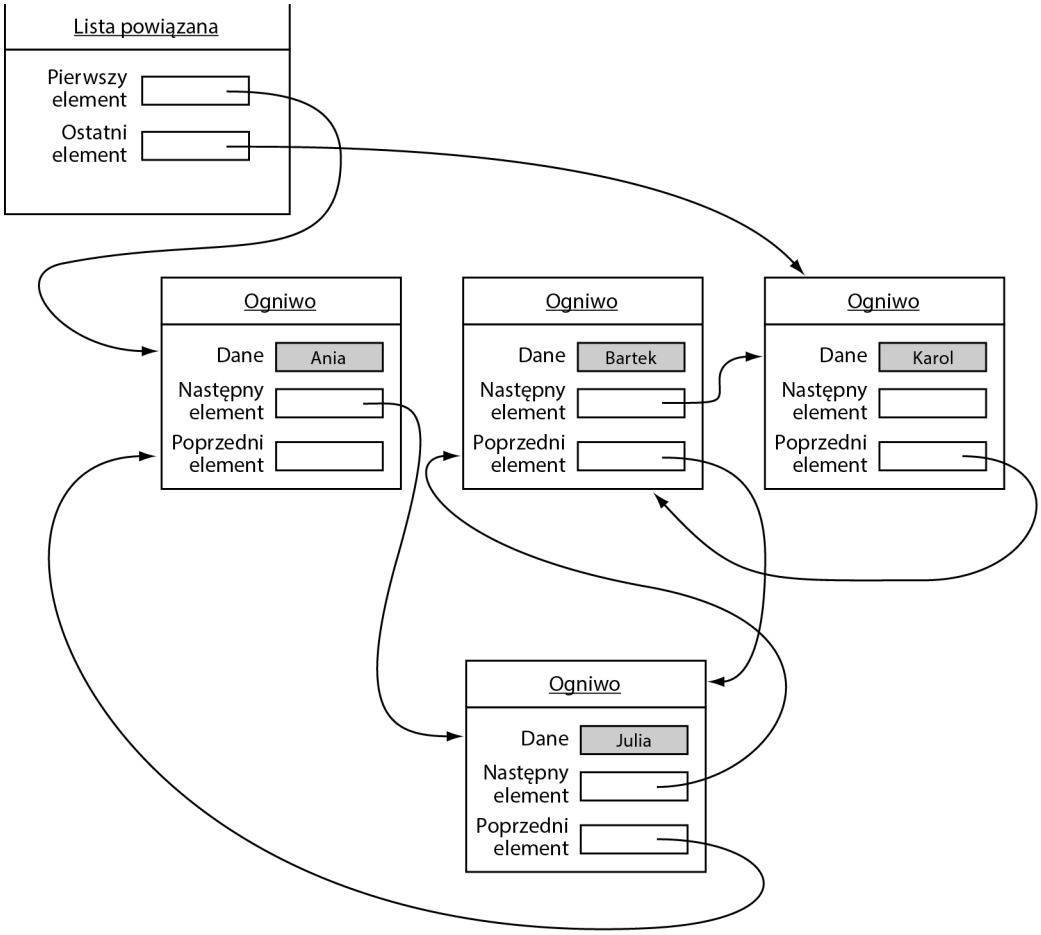
Metoda `previous`, podobnie jak `next`, zwraca obiekt, który właśnie przeskoczyła.

Metoda `listIterator` z klasy `LinkedList` zwraca obiekt iteratora implementujący interfejs `List➥Iterator`.

```
ListIterator<String> iter = staff.listIterator();
```

Metoda `add` dodaje element *przed* iteratorem. Na przykład poniższe instrukcje pomijają pierwszy element na liście powiązanej i dodają element `Julia` przed drugim elementem (zobacz rysunek 9.9):

```
var staff = new LinkedList<String>();
staff.add("Ania");
staff.add("Bartek");
staff.add("Karol");
ListIterator<String> iter = staff.listIterator();
iter.next();    // pominięcie pierwszego elementu
iter.add("Julia");
```



Rysunek 9.9. Dodawanie elementu do listy powiązanej

Jeśli metoda `add` zostanie wywołana kilka razy, elementy zostaną dodane do listy w kolejności podawania. Każdy z nich jest dodawany przed aktualnym miejscem pobytu iteratora.

Jeśli metoda `add` zostanie użyta z nieużywanym jeszcze iteratorem utworzonym przez metodę `listIterator`, wskazującym początek listy powiązanej, nowy element zostanie dodany na samym początku listy. Jeśli iterator przejdzie za ostatni element listy (czyli znajdzie się w miejscu, w którym metoda `hasNext` zwraca wartość `false`), dodawany element będzie stanowił nowy ogon listy. W liście o długości `n` istnieje `n+1` miejsc, w których można wstawiać elementy. Ta liczba odpowiada liczbie położeń, w których może się znaleźć iterator. Jeśli na przykład lista powiązana zawiera trzy elementy `A`, `B` i `C`, to istnieją w niej cztery miejsca (oznaczone znakiem `|`), w których można wstawić nowy element:

```
|ABC
A|BC
AB|C
ABC|
```

Przy stosowaniu analogii do kursora trzeba zachować ostrożność. Metoda `remove` nie działa dokładnie tak jak klawisz *Backspace*. Rzeczywiście bezpośrednio po wywołaniu metody `next` usuwa ona element znajdujący się po lewej stronie iteratora, dokładnie jak klawisz *Backspace*. Jeśli jednak przed nią została wywołana metoda `previous`, zostanie usunięty element znajdujący się po prawej stronie iteratora. Ponadto metody `remove` nie można wywołać dwa razy z rzędu.

Metoda `remove`, w przeciwieństwie do `add`, polegającej wyłącznie na położeniu iteratora, bierze pod uwagę stan iteratora.

W końcu metoda `set` zastępuje ostatni element zwrócony przez metodę `next` lub `previous` nowym elementem. Na przykład poniższa procedura zastępuje pierwszy element listy nową wartością:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next();    // zwraca pierwszy element
iter.set(newValue);               // ustawia pierwszy element na newValue
```

Nietrudno sobie wyobrazić, że w sytuacji, w której jeden iterator przemierza kolekcję, a inny ją modyfikuje, mogą wystąpić nieprzyjemne zdarzenia. Wyobraźmy sobie, że jakiś iterator wskazuje miejsce przed elementem, który został usunięty przez inny iterator. Ten pierwszy iterator traci w takiej sytuacji rację bytu i nie powinien być używany. Iteratory list powiązanych zostały zaprojektowane w taki sposób, aby wykrywać tego typu modyfikacje. Jeśli iterator odkryje, że jego kolekcja została zmodyfikowana przez inny iterator lub metodę samej kolekcji, wyrzuca wyjątek `ConcurrentModificationException`. Przeanalizujmy poniższy przykładowy kod:

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next();              // wyrzuca wyjątek ConcurrentModificationException
```

Wywołanie metody `iter2.next` powoduje wyjątek `ConcurrentModificationException`, ponieważ iterator `iter2` odkrył, że lista została zmodyfikowana przez jakiś czynnik zewnętrzny.

Aby uniknąć tego typu wyjątków, należy stosować się do prostej reguły: do kolekcji można wstawić dowolną liczbę iteratorów, pod warunkiem że służą one tylko do odczytu. Alternatywnie jeden z nich może być zarówno do odczytu, jak i zapisu.

Jednoczesne operacje modyfikujące są wykrywane w bardzo prosty sposób. Kolekcja pamięta liczbę operacji modyfikujących (takich jak dodawanie i usuwanie elementów). Każdy iterator pamięta, ile tego typu operacji *wykonał*. Przed przystąpieniem do działania iterator sprawdza, czy jego liczba zgadza się z liczbą kolekcji. Jeśli nie, zgłasza wyjątek `ConcurrentModification` ↪`Exception`.

Od operacji wykrywania jednoczesnych operacji modyfikujących istnieje jeden ciekawy wyjątek. Lista powiązana zapamiętuje tylko zmiany *struktury*, jak dodawanie i usuwanie ogniw. Metoda `set` *nie* liczy się jako modyfikacja strukturalna. W liście powiązanej może być kilka iteratorów zmieniających zawartość ogniw za pomocą metody `set`. To zachowanie jest potrzebne w kilku algorytmach w klasie `Collections`, które opisujemy dalej.

Poznaliśmy wszystkie podstawowe metody klasy `LinkedList`. Do przemierzania jej w dowolnym kierunku i dodawania oraz usuwania elementów służy obiekt klasy `ListIterator`.

Wiemy już z podrozdziału 9.2, że wiele przydatnych metod do działań na listach powiązanych znajduje się w interfejsie `Collection`. Większość z nich jest zdefiniowana w nadklasie `Abstract ↪Collection` klasy `LinkedList`. Na przykład metoda `toString` wywołuje metodę `toString` na rzecz każdego elementu i tworzy jeden długi łańcuch w formacie `[A, B, C]`, co przydaje się podczas debugowania. Aby sprawdzić, czy na liście znajduje się określony element, należy użyć metody `contains`. Na przykład instrukcja `staff.contains("Henryk")` zwróci wartość `true`, jeśli lista powiązana zawiera łańcuch `Henryk`.

Java Collections Framework zawiera także kilka wątpliwych z teoretycznego punktu widzenia metod. Listy powiązane nie umożliwiają szybkiego dostępu do dowolnego elementu. Aby dotrzeć do *n*-tego elementu listy, trzeba zacząć wędrówkę od początku i ominąć pierwszych *n*–1 elementów. Nie da się nic skrócić. Z tego powodu listy powiązane rzadko są używane w sytuacjach, w których potrzebny jest dostęp do elementów za pomocą indeksu całkowitoliczbowego.

Pomimo tego klasa `LinkedList` udostępnia metodę `get`, która pozwala na dostęp do wybranego elementu:

```
LinkedList<String> list = . . .;
String obj = list.get(n);
```

Oczywiście efektywność tej metody jest niska. Jeśli jej używasz, to zastanów się, czy nie należałoby zmienić zastosowanej struktury danych do rozwiązywanego problemu.

Nigdy nie należy przemierzać listy powiązanej za pomocą tej metody, dającej złudzenie swobodnego dostępu. Poniższy kod jest niebywale powolny:

```
for (int i = 0; i < list.size(); i++)
operacje na elemencie list.get(i);
```

Za każdym razem, kiedy potrzebujemy nowego elementu, poszukiwanie zaczyna się od początku listy. Obiekty `LinkedList` nie zapisują informacji o pozycji.

Metoda `get` posiada jedną niewielką optymalizację — jeśli wartość indeksu wynosi co najmniej `size()/2`, szukanie elementu zaczyna się od końca listy.

Interfejs `Iterator` listy zawiera także metodę informującą o aktualnym położeniu iteratora. W rzeczywistości, ze względu na to, że iteratory w Javie wskazują miejsce pomiędzy elementami, istnieją dwie takie metody. Metoda `nextIndex` zwraca indeks całkowitoliczbowy elementu, który zostałby zwrócony przez następne wywołanie metody `next`. Metoda `previousIndex` zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `previous`. Oczywiście wartość ta jest o jeden mniejsza od wartości `nextIndex`. Metody te są efektywne jako iteratory struktur `LinkedList`, ponieważ iteratory pamiętają swoją aktualną pozycję. W końcu, jeśli mamy indeks `n`, instrukcja `list.listIterator(n)` zwróci iterator wskazujący miejsce bezpośrednio przed elementem znajdującym się w polu o indeksie `n`. Oznacza to, że metoda `next` zwraca ten sam wynik co wywołanie `list.get(n)`.

Dysponując listą powiązaną zawierającą tylko kilka elementów, nie trzeba przesadnie obawiać się braku szybkości metod `get` i `set`. Po co więc w ogóle w takiej sytuacji używać listy powiązanej? Jedynym powodem do używania list powiązanych jest szybkość wstawiania i usuwania elementów do i ze środka kolekcji. Jeśli masz tylko kilka elementów, możesz użyć struktury `ArrayList`.

Zalecamy unikanie wszystkich metod określających położenie na liście za pomocą indeksów. Aby mieć wolny dostęp do wszystkich elementów kolekcji, należy używać obiektów `ArrayList` zamiast list powiązanych.

Program przedstawiony na listingu 9.1 demonstruje praktyczne zastosowanie list powiązanych. Tworzy on dwie listy, scala je, usuwa co drugi element z drugiej z nich, a na końcu testuje działanie metody `removeAll`. Zachęcamy do prześledzenia przepływu sterowania w tym programie i przyjrzenia się iteratorom. Pomocne mogą się okazać rysunki przedstawiające położenie iteratorów:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
...
```

Listing 9.1. linkedList/LinkedListTest.java

```
package linkedList;

import java.util.*;

/**
 * Program demonstrujący działania na listach powiązanych
 * @version 1.13 2023-12-05
 * @author Cay Horstmann
 */
public class LinkedListTest
{
   public static void main(String[] args)
   {
      var a = new LinkedList<String>();
      a.add("Ania");
      a.add("Karol");
      a.add("Eryk");

      var b = new LinkedList<String>();
      b.add("Bartek");
      b.add("Daniel");
      b.add("Franek");
      b.add("Gosia");

      // Scalenie list a i b

      ListIterator<String> aIter = a.listIterator();
      Iterator<String> bIter = b.iterator();

      while (bIter.hasNext())
      {
         if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }
```

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

            // Usunięcie co drugiego słowa z listy b

            bIter = b.iterator();
            while (bIter.hasNext())
            {
               bIter.next(); // Opuszczenie jednego elementu
               if (bIter.hasNext())
               {
                  bIter.next(); // Opuszczenie następnego elementu
                  bIter.remove(); // Usunięcie elementu
               }
            }

            System.out.println(b);

            // Usunięcie wszystkich słów znajdujących się w liście b z listy a

            a.removeAll(b);

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

Zwróć uwagę, że instrukcja `System.out.println(a)` drukuje wszystkie elementy listy powiązanej `a` za pomocą metody `toString` z klasy `AbstractCollection`.

`java.util.List<E>` **1.2**

- `ListIterator<E> listIterator()`

  Zwraca iterator listy.

- `ListIterator<E> listIterator(int index)`

  Zwraca iterator listy, którego pierwsze wywołanie metody `next` zwróci element znajdujący się pod podanym indeksem.

- `void add(int i, E element)`

  Dodaje element w określonym miejscu.

- `boolean addAll(int i, Collection<? extends E> elements)`

  Dodaje wszystkie elementy z kolekcji w określonym miejscu.

- `E remove(int i)`

  Usuwa i zwraca element znajdujący się w określonym miejscu.

- `E get(int i)`

  Zwraca element znajdujący się w określonym miejscu.

- `E set(int i, E element)`

  Zastępuje element znajdujący się w określonym miejscu nowym elementem i zwraca stary element.

- `int indexOf(Object element)`

  Zwraca położenie pierwszego wystąpienia elementu `element` lub wartość `-1`, jeśli element nie zostanie znaleziony.

- `int lastIndexOf(Object element)`

  Zwraca położenie ostatniego wystąpienia elementu `element` lub wartość `-1`, jeśli element nie zostanie znaleziony.

`java.util.ListIterator<E>` **1.2**

- `void add(E newElement)`

  Dodaje element przed bieżącą pozycją.

- `void set(E newElement)`

  Zastępuje ostatni element odwiedzony przez metodę `next` lub `previous` nowym elementem. Zgłasza wyjątek `IllegalStateException`, jeśli lista została zmodyfikowana od czasu ostatniego wywołania metody `next` lub `previous`.

- `boolean hasPrevious()`

  W trakcie przemierzania listy wstecz zwraca wartość `true`, jeśli są jeszcze nieodwiedzone elementy.

- `E previous()`

  Zwraca poprzedni obiekt. W przypadku osiągnięcia początku listy zgłasza wyjątek `NoSuchElementException`.

- `int nextIndex()`

  Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `next`, lub zwraca rozmiar listy, jeśli iterator wyszedł już poza ostatni element.

- `int previousIndex()`

  Zwraca indeks elementu, który zostałby zwrócony przez następne wywołanie metody `previous` lub `-1`, jeśli iterator znajduje się przed pierwszym elementem.

`java.util.LinkedList<E>` **1.2**

- `LinkedList()`

  Tworzy pustą listę powiązaną.

- `LinkedList(Collection<? extends E> elements)`

  Tworzy listę powiązaną i dodaje do niej wszystkie elementy z określonej kolekcji.

- `void addFirst(E element)`
- `void addLast(E element)`

  Dodaje element na początku lub końcu listy.

- `E getFirst()`
- `E getLast()`

  Zwraca element znajdujący się na początku lub końcu listy.

- `E removeFirst()`
- `E removeLast()`

  Usuwa element z początku lub końca listy.

9.3.2. Listy tablicowe

W poprzednim podrozdziale zapoznaliśmy się z interfejsem `List` i klasą `LinkedList`, która go implementuje. Interfejs ten opisuje uporządkowaną kolekcję, w której położenie elementów odgrywa rolę. Elementy można odwiedzać na dwa sposoby: za pomocą iteratora lub dowolnie przy użyciu metod `get` i `set`. Drugi z wymienionych sposobów nie nadaje się do list powiązanych, natomiast doskonale sprawdza się w tablicach. W bibliotece kolekcji znajduje się dobrze znana nam klasa `ArrayList`, która również implementuje interfejs `List`. Obiekt tej klasy zawiera dynamiczną tablicę obiektów.

W Javie 1.0 była inna kolekcja bazująca na listach — klasa `Vector`. Wszystkie metody klasy `Vector` są *synchronizowane*, dzięki czemu można bezpiecznie uzyskać dostęp do jednego obiektu tej klasy z dwóch wątków. Jeśli natomiast pracujemy tylko w jednym wątku — co zdarza się nieporównywalnie częściej — tracimy bardzo dużo czasu na synchronizację. Natomiast metody klasy `ArrayList` nie są synchronizowane. Jak zobaczysz w rozdziale 10., obecnie klasy metody są bezpieczne dla wątków. Nie ma już sensu używać klasy Vector, chyba że program musi współpracować z bardzo starym API.

9.3.3. Zbiór HashSet

W listach powiązanych i tablicach istnieje możliwość ustawienia przechowywanych w nich elementów w określonej kolejności. Aby jednak znaleźć jakiś element o nieznanym położeniu, trzeba przejść przez wszystkie elementy w jego poszukiwaniu. Jeśli kolekcja zawiera dużo elementów, operacja ta może zająć bardzo dużo czasu. Istnieją struktury danych, które pozwalają znacznie szybciej znajdować elementy, ale nie umożliwiają ich ustawiania w wybranej kolejności. Elementy w nich są ustawiane w takiej kolejności, która odpowiada ich własnym celom.

Powszechnie znaną strukturą danych pozwalającą szybko wyszukiwać obiekty jest **tablica skrótów** (ang. *hash table*; nazywana też mieszającą). Tablica ta oblicza liczbę całkowitą, zwaną **skrótem** (ang. *hash code*), dla każdego obiektu. Skrót powstaje w jakiś sposób z pól obiektu, najlepiej w taki, aby obiekty zawierające różne dane dawały różne kody. Tabela 9.2 zawiera kilka przykładowych skrótów zwróconych przez metodę `hashCode` z klasy `String`.

Tabela 9.2. Skróty zwrócone przez metodę hashCode

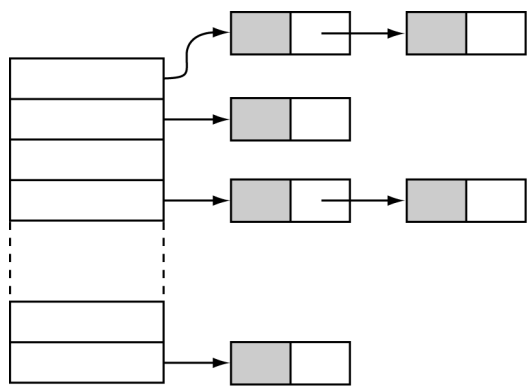
Łańcuch	Skrót
`Lee`	76268
`lee`	107020
`eel`	100300

Definiując własne klasy, należy zaimplementować własną metodę `hashCode` — więcej na ten temat można znaleźć w rozdziale 5. Implementacja ta musi być zgodna z metodą `equals` — jeśli `a` jest równe `b` (`a.equals(b)`), to `a` i `b` muszą mieć ten sam skrót.

To, co jest ważne teraz, to fakt, że obliczanie skrótu przebiega szybko, a wynik tego działania jest uzależniony tylko od stanu obiektu, którego kod jest tworzony, a nie od innych obiektów w tablicy.

Tablice skrótów w Javie są zaimplementowane jako tablice list powiązanych. Listy w tej sytuacji noszą miano **komórek** lub **kubełków** (ang. *bucket*; zobacz rysunek 9.10). Aby określić miejsce do przechowywania obiektu w tablicy, należy obliczyć jego skrót i znaleźć resztę z dzielenia tej liczby przez liczbę wszystkich komórek. Jeśli na przykład obiekt ma skrót 76268, a wszystkich komórek jest 128, zostanie on umieszczony w komórce 108 (ponieważ reszta z dzielenia 76 268 przez 128 wynosi 108). Jeśli mamy szczęście i nie ma w tej komórce żadnego innego elementu, umieszczamy tam nasz obiekt. Oczywiście nie da się uniknąć trafienia komórki ze znajdującym się wewnątrz obiektem. Sytuację taką nazywamy **kolizją**. Wtedy porównujemy nowy obiekt z wszystkimi obiektami w tej komórce, aby sprawdzić, czy go tam jeszcze nie ma. Jeśli skróty są rozsądnie rozłożone losowo, a liczba komórek wystarczająco duża, powinno być konieczne przeprowadzenie tylko kilku porównań.

Rysunek 9.10.
Tablica skrótów



W implementacji `HashMap` kubełki po przekroczeniu pewnego progu zmieniają się z list powiązanych w zbalansowane drzewa binarne. Pozwala to podnieść wydajność w przypadku nieefektywnych funkcji obliczania skrótów, które powodują wiele kolizji, oraz stanowi zabezpieczenie, na wypadek gdyby złośliwy kod próbował zalać tablicę skrótów wieloma wartościami o identycznych skrótach.

Jeśli jest taka możliwość, klucze tablicy skrótów powinny należeć do klasy implementującej interfejs `Comparable`. Daje to gwarancję, że wydajność programu nie ucierpi z powodu niekorzystnej dystrybucji kodów skrótu.

Jeśli tablica zostanie przepełniona, konieczna jest jej **reorganizacja**. Polega to na utworzeniu nowej tablicy z większą liczbą komórek i przeniesieniu wszystkich danych do tej nowej tablicy. Stara tablica zostaje usunięta. O reorganizacji tablicy decyduje **współczynnik zapełnienia** (ang. *load factor*). Jeśli na przykład współczynnik ten wynosi 0,75 (wartość domyślna), a tablica zostanie zapełniona w ponad 75 procentach, następuje jej automatyczna reorganizacja, w wyniku której tworzona jest tablica o dwukrotnie większej liczbie komórek. W większości zastosowań najlepiej pozostawić współczynnik 0,75 bez zmian.

Aby zyskać większą kontrolę nad działaniem tablicy, można określić początkowy licznik komórek. Należy go ustawić na spodziewaną liczbę elementów podzieloną przez współczynnik zapełnienia. Albo, od Javy 19, można użyć statycznej metody pomocniczej `newHashSet`, która wykonuje to obliczenie za nas:

```
HashSet<String> strings = HashSet.newHashSet(expectedElementCount);
```

Niektórzy badacze utrzymują, że dobrze jest ustawić ten licznik na liczbę pierwszą, co pozwoli uniknąć grupowania się kluczy, jednak nie ma na to przekonujących dowodów. Java Collections Framework stosuje liczniki komórek będące potęgami liczby 2, a domyślna liczba to 16 (każda wartość podana jako wielkość tablicy jest automatycznie zaokrąglana do najbliższej potęgi dwójki).

Za pomocą tablic skrótów można zaimplementować kilka bardzo ważnych struktur danych. Najprostszą z nich jest **zbiór** (ang. *set*). Zbiór to kolekcja unikatowych elementów. Metoda `add` zbioru najpierw sprawdza, czy w zbiorze nie ma wstawianego obiektu, i wstawia go, jeśli nic nie znajdzie.

W Java Collections Framework znajduje się klasa `HashSet`, która implementuje zbiór bazujący na tablicy skrótów. Dodawanie elementów do tego zbioru odbywa się za pomocą metody `add`. Metoda `contains` została przedefiniowana w taki sposób, że szybko sprawdza, czy dodawany element nie znajduje się już w zbiorze. Sprawdza tylko elementy w jednej komórce, a nie całej kolekcji.

Iterator zbioru `HashSet` odwiedza wszystkie komórki po kolei. Ponieważ elementy te są porozrzucane po tablicy, iterator odwiedza je w losowej kolejności. Dlatego zbioru `HashSet` należy używać wyłącznie wówczas, gdy kolejność elementów w kolekcji nie jest ważna.

Przykładowy program zaprezentowany na końcu tego podrozdziału (listing 9.2) wczytuje słowa z pliku, dodaje je do zbioru, a następnie drukuje je na ekranie. Można na przykład wprowadzić tekst *Alice in Wonderland* za pomocą poniższego polecenia:

```
java set.SetTest ../gutenberg/alice30.txt
```

Książka *Alice in Wonderland* zawiera 28 195 słów, w tym 5909 unikalnych, wliczając uwagę o prawach autorskich, która znajduje się na początku. Słowa są ułożone w losowej kolejności. Następnie zostają wstawione do zbioru drzewiastego, który przechowuje je w kolejności posortowanej, jak zobaczysz w następnym punkcie.

Listing 9.2. set/SetTest.java

```
package set;

import java.io.IOException;
import java.nio.file.Path;
import java.util.*;

/**
 * Program porównujący operację wstawiania do zbioru skrótów i zbioru drzewiastego.
 * Sposób uruchomienia:
 * java set.SetTest ../gutenberg/alice30.txt
 * java set.SetTest ../gutenberg/crsto10.txt 100
 * @version 1.2 2023.12.07
 * @author Cay Horstmann
 */
public class SetTest
{
   public static void time(Set<String> wordSet, List<String> wordList, int repetitions)
   {
      long totalTime = 0;
      for (int i = 1; i <= repetitions; i++)
      {
         for (String word : wordList)
         {
            long start = System.nanoTime();
            wordSet.add(word);
            long end = System.nanoTime();
            totalTime += end - start;
         }
      }
      Iterator<String> iter = wordSet.iterator();
      for (int i = 1; i <= 20 && iter.hasNext(); i++)
         System.out.print(iter.next() + " ");
      System.out.println("...");
      System.out.printf("%s: %d słów, %d unikalnych, %.3f sekund.%n",
         wordSet.getClass().getSimpleName(), wordList.size(), wordSet.size(),
         totalTime * 1E-9);
   }

   public static void main(String[] args) throws IOException
   {
      List<String> words = new ArrayList<>();
      String filename = args.length > 0 ? args[0] : "../gutenberg/crsto10.txt";
      int repetitions = args.length > 1 ? Integer.parseInt(args[1]) : 1;
      try (var in = new Scanner(Path.of(filename))) {
         while (in.hasNext()) {
            String word = in.next();
            words.add(word);
         }
      }
      time(new HashSet<>(), words, repetitions);
      time(new TreeSet<>(), words, repetitions);
   }
}
```

Uważaj, kiedy modyfikujesz elementy zbioru. Jeśli skrót elementu się zmieni, to element ten przestanie znajdować się na właściwej pozycji w strukturze danych.

```
var rects = new HashSet<Rectangle>();
var rect = new Rectangle(5, 10, 20, 30);
rects.add(rect);
rect.setLocation(0, 0);
    // rects to teraz [java.awt.Rectangle[x=0,y=0,width=20,height=30]]
rects.remove(new Rectangle(0, 0, 20, 30)); // Zwraca false, nie usuwa
```

`java.util.HashSet<E>` **1.2**

- `HashSet()`

  Tworzy pusty obiekt `HashSet`.

- `HashSet(Collection<? extends E> elements)`

  Tworzy obiekt `HashSet` i wstawia do niego wszystkie elementy określonej kolekcji.

- `HashSet(int initialCapacity)`

- `HashSet(int initialCapacity, float loadFactor)`

  Tworzy obiekt `HashSet` o określonej pojemności i z określonym współczynnikiem zapełnienia lub współczynnikiem `0.75`. Jeśli współczynnik rozmiaru do pojemności przekroczy współczynnik zapełnienia, tablica zostaje przeorganizowana i zamieniona na większą.

- `static <E> HashSet<E> newHashSet(int numMappings)` **19**

  Tworzy pusty zbiór skrótów o pojemności wystarczającej do przechowywania spodziewanej liczby elementów (`numMappings`) bez reorganizacji.

`java.lang.Object` **1.0**

- `int hashCode()`

  Zwraca skrót obiektu. Skrót może być dowolną liczbą całkowitą, także ujemną. Definicje metod `equals` i `hashCode` muszą być ze sobą zgodne — jeśli instrukcja `x.equals(y)` zwraca wartość `true`, to instrukcja `x.hashCode()` musi zwracać taką samą wartość jak `y.hashCode()`.

9.3.4. Zbiór TreeSet

Klasa `TreeSet` jest podobna do klasy `HashSet`, ale ma jedną zaletę w stosunku do niej. Zbiór `TreeSet` jest **zbiorem uporządkowanym**. Mimo że elementy dodaje się do niego w dowolnej kolejności w trakcie iteracji przez zbiór, są one automatycznie prezentowane w uporządkowanej kolejności. Wyobraźmy sobie na przykład, że dodajemy do zbioru trzy łańcuchy, a następnie odwiedzamy każdy z nich.

```
var sorter = new TreeSet<String>();
sorter.add("Bartek");
```

```
sorter.add("Ania");
sorter.add("Karol");
for (String s : sorter) System.out.println(s);
```

Wartości zostaną wydrukowane w kolejności alfabetycznej: Ania, Bartek, Karol. Jak wskazuje sama nazwa klasy, elementy w tym zbiorze są przechowywane w strukturze drzewiastej (obecnie używane są **drzewa czerwono-czarne**; ich opis można znaleźć na przykład w książce pod tytułem *Wprowadzenie do algorytmów*, której autorami są Thomas Cormen, Charles Leiserson, Ronald Rivest i Clifford Stein, WNT, Warszawa 2007). Każdy nowy element jest umieszczany w odpowiednim miejscu zgodnie z kolejnością. Dzięki temu elementy odwiedzane przez iterator są zawsze posortowane.

Operacja dodawania elementów do drzewa jest wolniejsza niż w przypadku tablicy skrótów, ale i tak znacznie szybsza niż dodawanie ich w odpowiednim miejscu tablicy lub listy powiązanej. Jeśli drzewo zawiera `n` elementów, znalezienie odpowiedniego położenia dla nowego elementu wymaga około `log2n` testów. Jeśli na przykład drzewo zawiera 1000 elementów, dodanie nowego elementu wiąże się z przeprowadzeniem około 10 porównań.

Ze zbioru drzewiastego można korzystać tylko wtedy, gdy jest możliwość porównywania elementów. Oznacza to, że elementy muszą implementować interfejs `Comparable` lub programista przy tworzeniu zbioru musi dostarczyć komparator (interfejsy `Comparable` i `Comparator` są opisane w rozdziale 6.).

O ile wolniejszy jest zbiór drzewiasty? Uruchom program na listingu 9.2 w następujący sposób:

```
java set.SetTest ../gutenberg/crsto10.txt 100
```

Książka *The Count of Monte Cristo* zawiera 466 300 słów, z których każde zostaje wstawione 100 razy. Na moim testowym komputerze TreeSet jest prawie 4 razy wolniejszy. Jeśli nie potrzebujesz sortowania danych, nie ma sensu marnować na to czasu. Co więcej, w niektórych przypadkach łatwiej jest napisać funkcję obliczającą skróty niż sortującą. Funkcja obliczająca skróty musi tylko dobrze mieszać obiekty, natomiast funkcja porównująca musi je precyzyjnie rozróżniać.

Przeanalizujmy na przykład przypadek zbioru prostokątów. Jeśli użyjemy zbioru `TreeSet`, musimy dostarczyć obiekt `Comparator<Rectangle>`. Powstaje pytanie, jakie kryteria zastosować do porównywania tych figur. Można na przykład porównywać pola powierzchni, ale to nie jest dobre rozwiązanie, ponieważ dwa inaczej wyglądające prostokąty o innych współrzędnych mogą mieć takie samo pole, a my chcemy zachować oba. Komparator musi być kompatybilny z metodą `equals`. To znaczy, że wynikiem porównywania może być zero tylko wtedy, gdy obiekty są równe. Istnieje porządek, który nadaje się do zastosowania z prostokątami (porządek leksykograficzny dla współrzędnych), ale jest on nienaturalny i sprawia trudności obliczeniowe. Funkcja obliczająca skróty jest już natomiast zdefiniowana w klasie `Rectangle`. Jej działanie polega na mieszaniu współrzędnych.

Jeśli komparator jest niekompatybilny z metodą `equals` typu elementów, porównywanie według wzoru może dawać niespójne wyniki. Oto typowy przykład:

```
var words = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
```

Intencją jest nie zbierać elementów, które różnią się tylko wielkością liter:

```
commands.add("ZAMKNIJ");
commands.add("zamknij"); // Nie zostanie dodany.
```

Teraz jednak metoda `equals` nie jest już symetryczna:

```
commands.equals(Set.of("zamknij")) // prawda
Set.of("zamknij").equals(commands) // fałsz
```

`TreeSet` uznaje dwa elementy za równe tylko wtedy, gdy komparator zwróci zero. Natomiast drugi zbiór używa metody `String.equals`.

Lepszym podejściem jest użycie porządku `String.compare` i wstawianie do zbioru tylko łańcuchów składających się z małych liter.

Klasa `TreeSet` implementuje interfejs `NavigableSet`. Znajduje się w nim kilka bardzo przydatnych metod służących do lokalizowania sąsiadujących elementów. Szczegółowe informacje na ten temat znajdują się w wyciągach z API.

Program przedstawiony na listingu 9.3 tworzy dwa zbiory `TreeSet` zapełnione obiektami `Item`. Pierwszy z nich jest sortowany według numerów części, czyli w domyślny sposób. Drugi natomiast posortowano według opisów za pomocą niestandardowego komparatora.

Listing 9.3. treeSet/TreeSetTest.java

```
package treeSet;

import java.util.*;

/**
 * Program sortujący zbiór obiektów typu Item przez porównywanie ich opisów.
 * @version 1.2 2024-12-05
 * @author Cay Horstmann
 */
public class TreeSetTest
{
   public static void main(String[] args)
   {
      var parts = new TreeSet<Item>();
      parts.add(new Item("Toster", 1234));
      parts.add(new Item("Wihajster", 4562));
      parts.add(new Item("Router", 9912));
      System.out.println(parts);

      var sortByDescription = new TreeSet<Item>(Comparator.comparing(Item::description));

      sortByDescription.addAll(parts);
      System.out.println(sortByDescription);
   }
}
```

Na listingu 9.4 przedstawiona jest klasa `Item`.

Listing 9.4. treeSet/Item.java

```
package treeSet;

/**
 * Pozycja z opisem i numerem katalogowym.
 */
public record Item(String description, int partNumber) implements Comparable<Item>
{
   public int compareTo(Item other)
   {
      int diff = Integer.compare(partNumber, other.partNumber);
      return diff != 0 ? diff : description.compareTo(other.description);
   }
}
```

`java.util.TreeSet<E>` **1.2**

- `TreeSet()`
- `TreeSet(Comparator<? super E> comparator)`

  Tworzy pusty zbiór drzewiasty.
- `TreeSet(Collection<? extends E> elements)`
- `TreeSet(SortedSet<E> s)`

  Tworzy zbiór drzewiasty i wstawia do niego wszystkie elementy z kolekcji lub posortowanego zbioru (w drugim przypadku stosuje niezmieniony porządek).

`java.util.SortedSet<E>` **1.2**

- `Comparator<? super E> comparator()`

  Zwraca komparator, który został użyty do posortowania elementów, lub wartość `null`, jeśli elementy są porównywane za pomocą metody `compareTo` z interfejsu `Comparable`.
- `E first()`
- `E last()`

  Zwraca najmniejszy lub największy element posortowanego zbioru.

`java.util.NavigableSet<E>` **6**

- `E higher(E value)`
- `E lower(E value)`

  Zwraca najmniejszy element większy od wartości `value` lub największy element mniejszy od wartości `value`, lub wartość `null`, jeśli nie ma takiego elementu.
- `E ceiling(E value)`

- `E floor (E value)`

  Zwraca najmniejszy element większy od lub równy wartości `value` lub największy element mniejszy od lub równy wartości `value`, lub wartość `null`, jeśli nie ma takiego elementu.

- `E pollFirst()`
- `E pollLast()`

  Usuwa i zwraca najmniejszy lub największy element zbioru lub wartość `null`, jeśli zbiór jest pusty.

9.3.5. Kolejki Queue i Deque

Z wcześniejszych podrozdziałów wiemy już, że kolejka jest strukturą danych pozwalającą na szybkie dodawanie elementów na końcu i usuwanie elementów z początku. Kolejka dwukierunkowa (**deque**) pozwala na szybkie dodawanie i usuwanie elementów po obu stronach. Nie można natomiast dodawać elementów w środku kolejki. W Java SE 6 zaprezentowano nowy interfejs o nazwie `Deque`. Implementują go klasy `ArrayDeque` i `LinkedList`. Służą one do tworzenia kolejek dwukierunkowych, które mogą zmieniać swój rozmiar w zależności od zapotrzebowania. W rozdziale 10. piszemy jeszcze o kolejkach `Queue` i `Deque` z ograniczeniami.

W Javie 21 metody

```
E getFirst()
E getLast()
void addFirst(E e)
void addLast(E e)
E removeFirst()
E removeLast()
```

zostały przeniesione z interfejsu `Deque` do `SequencedCollection`, który rozszerzają także interfejsy `List` i `Set`. Zapewnia to jednolity zestaw metod dostępu do pierwszego i ostatniego elementu kolekcji.

`java.util.Queue<E>` **5.0**

- `boolean add(E e)`
- `boolean offer(E e)`

  Wstawia element za ostatnim elementem kolejki i zwraca wartość `true`, jeśli kolejka nie jest pełna. Jeśli w kolejce nie ma miejsca, pierwsza z powyższych metod zgłasza wyjątek `IllegalStateException`, a druga wartość `false`.

- `E remove()`
- `E poll()`

  Usuwa i zwraca pierwszy element kolejki, jeśli nie jest ona pusta. Jeśli kolejka jest pusta, pierwsza z powyższych metod zgłasza wyjątek `NoSuchElementException`, a druga wartość `null`.

- `E element()`
- `E peek()`

  Zwraca element znajdujący się na czole kolejki (ale go nie usuwa), pod warunkiem że kolejka nie jest pusta. Jeśli kolejka jest pusta, pierwsza z powyższych metod zgłasza wyjątek `NoSuchElementException`, a druga wartość `null`.

`java.util.SequencedCollection<E>` **21**

- `void addFirst(E element)`
- `void addLast(E element)`

  Dodaje dany element przed pierwszym lub za ostatnim elementem kolekcji. Jeśli kolekcja jest pełna, zgłasza wyjątek `IllegalStateException`.

- `E removeFirst()`
- `E removeLast()`

  Usuwa i zwraca pierwszy lub ostatni element kolekcji, pod warunkiem że kolekcja ta nie jest pusta. Jeśli kolekcja jest pusta, zgłasza wyjątek `NoSuchElementException`.

- `E getFirst()`
- `E getLast()`

  Zwraca pierwszy lub ostatni element kolekcji bez jego usunięcia, pod warunkiem że kolekcja nie jest pusta. Jeśli kolekcja jest pusta, zgłasza wyjątek `NoSuchElementException`.

`java.util.Deque<E>` **6**

- `boolean offerFirst(E element)`
- `boolean offerLast(E element)`

  Dodaje element przed pierwszym lub za ostatnim elementem kolejki dwukierunkowej. Jeśli kolejka jest pełna, zwraca `false`.

- `E pollFirst()`
- `E pollLast()`

  Usuwa i zwraca element znajdujący się przed pierwszym lub za ostatnim elementem kolejki dwukierunkowej, jeśli nie jest ona pusta. Jeśli kolejka jest pusta, zwraca `null`.

- `E peekFirst()`
- `E peekLast()`

  Zwraca pierwszy lub ostatni element kolejki dwukierunkowej, nie usuwając go, pod warunkiem że kolejka nie jest pusta. Jeśli kolejka jest pusta, zwraca `null`.

`java.util.ArrayDeque<E>` **6**

- `ArrayDeque()`
- `ArrayDeque(int initialCapacity)`

  Tworzy nieograniczoną kolejkę dwukierunkową o początkowej pojemności 16 elementów lub odpowiadającej wartości `initialCapacity`.

9.3.6. Kolejki priorytetowe

Zasada działania tej struktury danych polega na przyjmowaniu elementów w losowej kolejności i oddawaniu ich w kolejności uporządkowanej. To znaczy, że metoda `remove` wywołana na rzecz kolejki priorytetowej zawsze zwraca najmniejszy element znajdujący się w kolekcji. Nie oznacza to jednak, że elementy w kolejce priorytetowej są przechowywane w kolejności uporządkowanej. Jeśli przejdziemy iteratorem po zawartości kolejki, znajdujące się w niej obiekty mogą nie być posortowane. Jednym ze sposobów realizacji kolejki priorytetowej jest bardzo elegancka i szybka struktura danych o nazwie **sterta** (ang. *heap*). Sterta jest pewnego rodzaju samoorganizującym drzewem binarnym, w którym metody `add` i `remove` powodują przemieszczanie najmniejszego elementu w stronę korzenia bez straty czasu na sortowanie.

Kolejka priorytetowa, podobnie jak struktura `TreeSet`, może przechowywać elementy klasy implementującej interfejs `Comparable` lub obiekt `Comparator` przekazywany do konstruktora.

Typowym zastosowaniem kolejek priorytetowych są harmonogramy zadań. Zadania są dodawane w losowej kolejności i każde z nich ma określony priorytet. Kiedy jakieś zadanie może zostać rozpoczęte, z kolejki usuwane jest zadanie o najwyższym priorytecie (ponieważ tradycyjnie najwyższy priorytet jest oznaczany numerem 1, metoda `remove` usuwa najmniejszy element).

Listing 9.5 demonstruje działanie kolejki priorytetowej. W przeciwieństwie do zbioru `TreeSet`, tutaj iterator nie odwiedza elementów w uporządkowanej kolejności. Natomiast usuwany jest zawsze najmniejszy dostępny element.

Listing 9.5. priorityQueue/PriorityQueueTest.java

```
package priorityQueue;

import java.util.*;
import java.time.*;

/**
 * Program demonstrujący zastosowanie kolejki priorytetowej
 * @version 1.02 2015-06-20
 * @author Cay Horstmann
 */
public class PriorityQueueTest
{
   public static void main(String[] args)
   {
      var pq = new PriorityQueue<LocalDate>();
      pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
```

```
        pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
        pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
        pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse

        System.out.println("Iteracja przez elementy...");
        for (LocalDate date : pq)
           System.out.println(date);
        System.out.println("Usuwanie elementów...");
        while (!pq.isEmpty())
           System.out.println(pq.remove());
     }
  }
```

`java.util.PriorityQueue` **5**

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`

  Tworzy kolejkę priorytetową do przechowywania obiektów implementujących interfejs `Comparable`.

- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`

  Tworzy kolejkę priorytetową i wykorzystuje do jej sortowania określony komparator.

9.4. Słowniki

Zbiór (ang. *set*) to rodzaj kolekcji pozwalający na szybkie wyszukiwanie elementów. Aby jednak znaleźć jakiś element, trzeba posiadać jego wierną kopię. Nie jest to zbyt często wykonywany rodzaj wyszukiwania — zazwyczaj do dyspozycji jest jakiś rodzaj informacji kluczowej i zadanie polega na znalezieniu związanego z nią elementu. Tego typu struktury realizowane są za pomocą **słowników**. Słowniki przechowują pary klucz – wartość. Aby znaleźć element w słowniku, trzeba znać jego klucz. Na przykład w słowniku można przechowywać tabelę danych o pracownikach, gdzie kluczami będą identyfikatory pracowników, a wartościami obiekty typu `Emplo` ↪`yee`. W kilku kolejnych sekcjach opisujemy sposoby posługiwania się słownikami.

9.4.1. Podstawowe operacje słownikowe

W Java Collections Framework znajdują się dwie implementacje słowników ogólnego przeznaczenia: `HashMap` i `TreeMap`. Obie te klasy implementują interfejs `Map`.

Struktura `HashMap` miesza klucze, natomiast w strukturze `TreeMap` klucze są zorganizowane w drzewie poszukiwań posortowanym w porządku liniowym. Funkcja obliczająca skróty lub porównująca jest wywoływana *wyłącznie na rzecz kluczy*. Wartości związane z tymi kluczami nie są mieszane ani porównywane.

Którą strukturę wybrać: HashMap czy TreeMap? Podobnie jak ze zbiorami, mieszanie jest nieco szybsze i należy je wybierać, gdy kolejność kluczy nie ma znaczenia.

Poniżej tworzona jest struktura HashMap do przechowywania danych pracowników:

```
var staff = new HashMap<String, Employee>(); // Klasa HashMap
                                             // implementuje interfejs Map
var harry = new Employee("Henryk Kwiatek");
staff.put("987-98-9996", harry);
. . .
```

Dla każdego obiektu dodawanego do słownika HashMap należy podać jego klucz. W tym przypadku kluczem jest łańcuch, a odpowiadającą mu wartością obiekt typu Employee.

Aby pobrać (i zapisać w pamięci) obiekt ze słownika, należy posłużyć się jego kluczem.

```
var id = "987-98-9996";
Employee e = staff.get(s);    // pobiera obiekt harry
```

Ze względów historycznych metoda get ma w deklaracji parametr typu Object, a nie typu klucza. Na przykład poniższy kod przechodzi kompilację:

```
long numericId = . . .
Employee e = staff.get(numericId); // Przechodzi kompilację, ale nigdy nie pobiera wartości
```

Jeśli z danym kluczem w słowniku nie jest skojarzona żadna wartość, metoda get zwraca wartość null.

Taka wartość zwrotna może być problematyczna. Czasami znana jest dobra inna wartość domyślna, której można by było użyć zamiast null. W takim przypadku można się posłużyć metodą getOrDefault:

```
Map<String, Integer> scores = . . .;
int score = scores.get(id, 0); // jeśli nie ma id, zwraca 0
```

Klucze muszą być unikatowe, to znaczy, że nie można dwóm różnym wartościom przypisać takiego samego klucza. Jeśli metoda put zostanie wywołana dwa razy przy użyciu jednego klucza, wartość z drugiego wywołania zastąpi tę z pierwszego. W rzeczywistości metoda ta zwróci poprzednią wartość przechowywaną pod tym kluczem.

Do usuwania elementów z określonym kluczem ze słownika służy metoda remove. Natomiast metoda size zwraca liczbę elementów znajdujących się w słowniku.

Klucze i wartości słownika najłatwiej przejrzeć iteracyjnie za pomocą metody forEach, przekazując jej wyrażenie lambda odbierające kolejne klucze i wartości. Wyrażenie to jest wywoływane po kolei dla każdej pozycji znajdującej się w słowniku.

```
scores.forEach((k, v) ->
  System.out.println("klucz=" + k + ", wartość=" + v));
```

Na listingu 9.6 pokazano praktyczne zastosowanie słownika. Na początku do słownika dodawane są pary klucz – wartość. Następnie program usuwa jeden klucz ze słownika, co pociąga za

sobą usunięcie skojarzonej z nim wartości. Dalej zostaje zmodyfikowana wartość skojarzona z pewnym kluczem i następuje wywołanie metody get wyszukującej wartość. Na zakończenie program iteruje przez zbiór entrySet.

Listing 9.6. map/MapTest.java

```
package map;

import java.util.*;

/**
 * Program demonstrujący użycie słownika z kluczami typu String i wartościami typu Employee
 * @version 1.12 2015-06-21
 * @author Cay Horstmann
 */
public class MapTest
{
   public static void main(String[] args)
   {
      var staff = new HashMap<String, Employee>();
      staff.put("144-25-5464", new Employee("Anna Kowalska"));
      staff.put("567-24-2546", new Employee("Henryk Kwiatek"));
      staff.put("157-62-7935", new Employee("Marcin Nowak"));
      staff.put("456-62-5527", new Employee("Franciszek Frankowski"));

      // wydruk wszystkich pozycji

      System.out.println(staff);

      // usunięcie wartości

      staff.remove("567-24-2546");

      // podmienienie pozycji

      staff.put("456-62-5527", new Employee("Weronika Kowalska"));
      // wyszukanie wartości

      System.out.println(staff.get("157-62-7935"));

      // iteracja przez wszystkie pozycje

      staff.forEach((k, v) ->
        System.out.println("klucz=" + k + ", wartość=" + v));
   }
}
```

`java.util.Map<K, V>` **1.2**

- `V get(Object key)`

  Zwraca wartość skojarzoną z kluczem key. Jeśli nie znajdzie klucza w słowniku, zwraca wartość null. Klucz może mieć wartość null.

- `default V getOrDefault(Object key, V defaultValue)` **8**

  Pobiera wartość skojarzoną z kluczem i ją zwraca. Jeśli nie znajdzie klucza, zwraca wartość domyślną defaultValue.

- `V put(K key, V value)`

  Wstawia parę klucz – wartość do słownika. Jeśli taki klucz jest już w słowniku, skojarzony z nim obiekt jest zastępowany nowym. Metoda ta zwraca poprzednią wartość skojarzoną z kluczem lub wartość `null`, jeśli wcześniej takiego klucza nie było. Klasy implementujące mogą zabraniać stosowania kluczy lub wartości `null`.

- `void putAll(Map<? extends K, ? extends V> entries)`

  Wstawia do słownika wszystkie pozycje z określonego słownika.

- `boolean containsKey(Object key)`

  Zwraca wartość `true`, jeśli klucz znajduje się w słowniku.

- `boolean containsValue(Object value)`

  Zwraca wartość `true`, jeśli wartość znajduje się w słowniku.

- `default void forEach(BiConsumer<? super K,? super V> action)` **8**

  Wykonuje działanie (`action`) na wszystkich parach klucz – wartość słownika.

`java.util.HashMap<K, V>` **1.2**

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`

  Tworzy pusty słownik `HashMap` o określonej pojemności i z określonym współczynnikiem zapełnienia (liczba z zakresu od 0,0 do 1,0 określająca stopień zapełnienia tablicy `HashTable`, po przekroczeniu którego następuje reorganizacja na większą jednostkę). Domyślny współczynnik zapełnienia wynosi 0,75.

- `static <K, V> HashMap<K, V> newHashMap(int numMappings)` **19**

  Tworzy pusty słownik skrótów o pojemności wystarczającej do pomieszczenia spodziewanej liczby elementów (`numMappings`) bez reorganizacji.

`java.util.TreeMap<K, V>` **1.2**

- `TreeMap()`

  Tworzy pusty słownik drzewiasty dla wszystkich kluczy implementujących interfejs `Comparable`.

- `TreeMap(Comparator<? super K> c)`

  Tworzy słownik drzewiasty i sortuje jego klucze za pomocą określonego komparatora.

- `TreeMap(Map<? extends K, ? extends V> entries)`

  Tworzy słownik drzewiasty i dodaje do niego wszystkie elementy z innego słownika.

- `TreeMap(SortedMap<? extends K, ? extends V> entries)`

  Tworzy słownik drzewiasty, dodaje do niego wszystkie pozycje z posortowanego słownika oraz wykorzystuje ten sam komparator co określony posortowany słownik.

`java.util.SortedMap<K, V>` **1.2**

- `Comparator<? super K> comparator()`

  Zwraca komparator użyty do sortowania kluczy lub wartość `null`, jeśli klucze są porównywane za pomocą metody `compareTo` z interfejsu `Comparable`.

- `K firstKey()`
- `K lastKey()`

  Zwraca najmniejszy lub największy klucz dostępny w słowniku.

9.4.2. Modyfikowanie wpisów w słowniku

Jedną z trudności, jakie można napotkać w trakcie pracy ze słownikami, jest modyfikowanie wpisów. Normalnie pobiera się starą wartość związaną z kluczem, potem się ją zmienia, a na koniec wstawia się ją z powrotem do słownika. Trzeba jednak pamiętać o specjalnym przypadku, jakim jest pierwsze wystąpienie danego klucza. Powiedzmy, że chcemy wykorzystać słownik do policzenia, ile razy wybrane słowo występuje w pliku. Gdy je napotykamy, zwiększamy wartość licznika w następujący sposób:

```
counts.put(word, counts.get(word) + 1);
```

Technika ta działa prawidłowo w prawie każdym przypadku. Wyjątkiem jest pierwsze napotkanie szukanego słowa, kiedy to następuje zwrócenie przez `get` wartości `null` i wystąpienie wyjątku `NullPointerException`.

Prostym rozwiązaniem jest użycie metody `getOrDefault`:

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

Inna możliwość to wywołanie najpierw metody `putIfAbsent`, która wstawia do słownika wartość tylko wtedy, jeśli uprzednio jego klucz nie istniał (lub był mapowany na `null`).

```
counts.putIfAbsent(word, 0);
counts.put(word, counts.get(word) + 1); // teraz wiadomo, że metoda get zadziała
```

Ale jest jeszcze lepsze rozwiązanie. Tę często wykonywaną operację można uprościć za pomocą metody `merge`. Poniższe wywołanie wiąże `word` z 1, jeśli takiego klucza wcześniej nie było, lub w przeciwnym przypadku łączy poprzednią wartość z 1 za pomocą funkcji `Integer::sum`.

```
counts.merge(word, 1, Integer::sum);
```

Teraz wyobraź sobie inną często spotykaną sytuację. Chcemy powiązać zbiór z każdym kluczem. Na przykład w indeksie książki każdy termin ma przypisany zbiór numerów stron, na których występuje. Oto jak należy zmienić słownik:

```
var index = new TreeMap<String, TreeSet<Integer>>();
. . .
index.computeIfAbsent(term, k -> new TreeSet<>()).add(pageNumber);
```

Tutaj lepiej jest użyć metody computeIfAbsent, ponieważ struktura TreeSet jest tworzona, tylko jeśli z terminem nie jest powiązany wcześniej utworzony zbiór. Metoda computeIfAbsent zwraca nową wartość, dzięki czemu możemy łączyć wywołania metody add w łańcuchy. Z kolei metoda putIfAbsent zwraca poprzedni klucz lub null, więc nie nadaje się do łączenia w łańcuchy.

Kuszące jest zamienienie k -> new TreeSet<>() na TreeSet::new, ale to się nie uda. Funkcja dla danego klucza ma parametr k, który zostałby przekazany do konstruktora. Na szczęście nie ma konstruktora TreeSet z parametrem String i taki kod nie przejdzie kompilacji.

Mogło być gorzej. Wyobraź sobie, że masz słownik Map<Integer, ArrayList<String>> i wywołanie

```
map.computeIfAbsent(n, ArrayList::new)
```

Istnieje konstruktor ArrayList, który przyjmuje liczbę całkowitą jako określenie pojemności. Dla dużych wartości n zostałyby alokowane duże listy tablicowe, a z pewnością nie o to chodziło.

Metody słownika nie są zbyt konsekwentne w kwestii traktowania wartości null. Niektóre traktują ją jako prawidłową wartość, a inne — tak samo jak brakujący klucz.

Metoda getOrDefault należy do pierwszej grupy, a metoda putIfAbsent — do drugiej:

```
counts.put("C++", null);
counts.getOrDefault("C++", -1) // Zwraca null, nie używając wartości domyślnej
counts.putIfAbsent("C++", 1) // Wstawia 1, interpretując null jako brak
```

W dokumentacji API znajdują się opisy także innych, rzadziej używanych metod do modyfikowania wpisów w słownikach.

`java.util.Map<K, V>` **1.2**

- `default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)` **8**

  Jeżeli klucz jest związany z inną niż null wartością v, stosuje funkcję do v i value, a następnie wiąże klucz z wynikiem lub, jeśli ten wynosi null, usuwa klucz. W przeciwnym przypadku wiąże klucz z wartością. Zwraca get(key).

- `default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` **8**

  Stosuje funkcję do key i get(key). Wiąże klucz z wynikiem albo, jeśli ten wynosi null, usuwa klucz. Zwraca get(key).

- `default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` **8**

  Jeżeli klucz jest związany z inną niż null wartością v, stosuje funkcję do key i v, a następnie wiąże klucz z wynikiem lub, jeśli ten wynosi null, usuwa klucz. Zwraca get(key).

- `default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` **8**

  Stosuje funkcję do klucza, chyba że jest on związany z wartością inną niż `null`. Wiąże klucz z wynikiem lub, jeśli ten wynosi `null`, usuwa klucz. Zwraca `get(key)`.

- `default void replaceAll(BiFunction<? super K,? super V,? extends V> function)` **8**

  Wywołuje funkcję na wszystkich pozycjach słownika. Z wynikami różnymi od `null` wiąże klucze, a dla wyników `null` usuwa klucze.

- `default V putIfAbsent(K key, V value)` **8**

  Jeśli klucza nie ma lub jest powiązany z `null`, kojarzy go z `value` i zwraca `null`. W przeciwnym przypadku zwraca powiązaną wartość.

9.4.3. Widoki słowników

W Java Collections Framework słowniki nie są traktowane jako kolekcje. (W innych systemach struktur danych słowniki są traktowane jako kolekcje par klucz – wartość lub kolekcje wartości indeksowanych kluczami). Istnieje jednak możliwość tworzenia **widoków** słowników, czyli obiektów implementujących interfejs `Collection` lub jeden z jego interfejsów podrzędnych.

Wyróżnia się trzy rodzaje widoków: zbiór kluczy, kolekcja wartości (która nie jest zbiorem) oraz zbiór par klucz – wartość. Klucze oraz pary klucz – wartość są reprezentowane jako zbiory, ponieważ w słowniku może się znajdować tylko jeden egzemplarz każdego klucza. Do tworzenia tych trzech widoków służą następujące metody:

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

Elementy zbioru tworzonego przez metodę `entrySet` są obiektami klasy implementującej interfejs `Map.Entry`.

Należy pamiętać, że `keySet` to *nie* `HashSet` ani `TreeSet`, lecz obiekt innej klasy implementującej interfejs `Set`, który jest rozszerzeniem interfejsu `Collection`. W związku z tym obiektów `keySet` można używać tak jak każdej kolekcji.

Można na przykład przejrzeć wszystkie klucze słownika:

```
Set<String> keys = map.keySet();
for (String key : keys)
{
  jakieś operacje na kluczu
}
```

Jeśli trzeba przejrzeć klucze i wartości, można uniknąć operacji wyszukiwania wartości przez enumerację *wpisów*. W tym celu można się posłużyć poniższym ramowym kodem:

```
for (Map.Entry<String, Integer> entry : counts.entrySet())
{
  String k = entry.getKey();
  Integer v = entry.getValue();
  jakieś operacje na k, v
}
```

Egzemplarze `Map.Entry` są połączone ze słownikiem. Przez obiekt pozycji można zmienić wartość w słowniku.

```
for (Map.Entry<String, Integer> entry : counts.entrySet())
{
    String k = entry.getKey();
    Integer v = entry.getValue();
    entry.setValue(v + 1); // to samo, co counts.put(k, v + 1);
}
```

Jeśli natomiast zmodyfikujesz wartość innym sposobem (na przykład za pomocą metody słownika `put`), to pozycja również zostanie zmieniona.

Jeśli chcesz przekazać pozycje do innej metody, to powinieneś je odłączyć od słownika przez wywołanie metody `Map.Entry.copyOf(entry)`. Możesz też tworzyć niepowiązane słowniki egzemplarze `Map.Entry`, wywołując metodę `Map.entry(`*`klucz`*`, `*`wartość`*`)`. Przydaje się to zawsze, gdy potrzebna jest para wartości.

Jeżeli na widoku `keySet` zostanie wywołana metoda `remove` iteratora, to ze słownika zostanie usunięty *klucz wraz z powiązaną wartością*. Nie można natomiast *dodać* elementu do widoku. Dodawanie klucza bez towarzyszącej mu wartości po prostu nie ma sensu. Próba wywołania metody `add` skończy się zgłoszeniem wyjątku `UnsupportedOperationException`. Takie same ograniczenia dotyczą widoku `entrySet`, choć teoretycznie dodanie nowej pary klucz – wartość ma sens.

Żmudnego wpisywania `Map.Entry` można uniknąć dzięki deklaracji `var`.

```
for (var entry : map.entrySet())
{
    jakieś operacje na entry.getKey(), entry.getValue()
}
```

Ewentualnie można skorzystać z metody `forEach`:

```
map.forEach((k, v) -> {
    jakieś operacje na k, v
});
```

`java.util.Map<K, V>` **1.2**

- `Set<Map.Entry<K, V>> entrySet()`

  Zwraca widok w postaci zbioru obiektów `Map.Entry`, czyli par klucz – wartość znajdujących się w słowniku. Ze zbioru tego można usuwać elementy, a skutki tego działania będą uwzględnione w słowniku. Nie można natomiast niczego dodawać.

- `Set<K> keySet()`

  Zwraca widok w postaci zbioru wszystkich kluczy znajdujących się w słowniku. Jeśli jakiś klucz z tego zbioru zostanie usunięty, ze słownika zostaną usunięte ten sam klucz i skojarzona z nim wartość. Nie można niczego dodawać.

- `Collection<V> values()`

  Zwraca widok kolekcyjny wszystkich wartości znajdujących się w kolekcji. Jeśli jakaś wartość z tego zbioru zostanie usunięta, ze słownika zostaną usunięte ta sama wartość i skojarzony z nią klucz. Nie można niczego dodawać.

`java.util.Map.Entry<K, V>` **1.2**

- `K getKey()`
- `V getValue()`

  Zwraca klucz lub wartość określonego wpisu.

- `V setValue(V newValue)`

  Zamienia wartość *znajdującą się w powiązanym słowniku* na nową wartość i zwraca starą wartość.

- `static <K, V> Map.Entry<K,V> copyOf(Map.Entry<? extends K,? extends V> map)` **17**

  Zwraca kopię podanego hasła słownika. W odróżnieniu od elementów zbioru haseł słownika kopia nie jest „żywa”. Wywołanie `setValue` nie powoduje aktualizacji słownika.

9.4.4. Klasa WeakHashMap

W Java Collections Framework znajduje się kilka klas słowników, które mają specjalne przeznaczenie. Omawiamy je krótko w kilku kolejnych podrozdziałach.

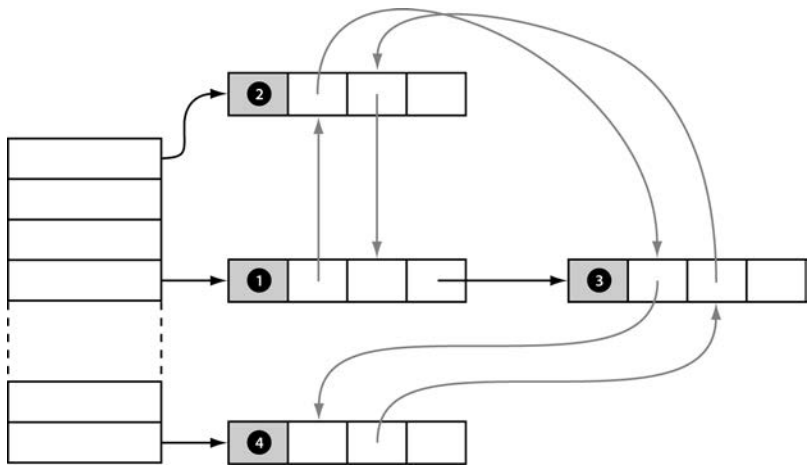
Klasa `WeakHashMap` rozwiązuje pewien bardzo interesujący problem. Spróbujmy sobie wyobrazić, co się dzieje z wartością, której klucz nie jest już używany nigdzie w programie. Załóżmy, że ostatnia referencja do tego klucza została utracona. W takiej sytuacji nie ma już żadnego sposobu dostania się do skojarzonego z tym kluczem obiektu. Ponieważ klucza już nigdzie nie ma, nie można usunąć ze słownika pary klucz – wartość. Nasuwa się myśl, że jest to zadanie dla śmieciarki.

Niestety nie jest to takie proste. System ten przechowuje informacje o aktywnych obiektach. Dopóki słownik jest aktywny, *wszystkie* jego komórki również są aktywne, przez co nie można ich usunąć. Dlatego należy pamiętać, aby usuwać z programu nieużywane wartości z długo wykorzystywanych słowników. Można też wybrać inne rozwiązanie i użyć klasy `WeakHashMap`. Struktura ta współpracuje ze śmieciarką w zakresie usuwania par klucz – wartość, jeśli jedyne odwołanie do klucza pochodzi z pozycji w tablicy.

Oto zasada działania tego mechanizmu. Klucze w `WeakHashMap` są przechowywane w tak zwanych **słabych referencjach** (ang. *weak reference*). Obiekt typu `WeakReference` przechowuje referencję do innego obiektu, w tym przypadku do klucza w tablicy. Obiekty tego typu są traktowane przez moduł śmieciarki (ang. *Garbage Collector — GC*) w specjalny sposób. Standardowo, jeśli GC odkryje, że do jakiegoś obiektu nie ma żadnych referencji, usuwa go. Jeśli natomiast obiekt taki jest dostępny *wyłącznie* za pośrednictwem obiektu `WeakReference`, GC także go usuwa, ale słabą referencję odnoszącą się do niego wstawia do kolejki. Obiekt `WeakHashMap` sprawdza w równych odstępach czasu, czy w kolejce nie pojawiły się jakieś nowe słabe referencje. Pojawienie się takiej referencji w kolejce oznacza, że klucz nie był nigdzie używany i że został zabrany. Wtedy `WeakHashMap` usuwa odpowiednią pozycję.

9.4.5. Klasy LinkedHashSet i LinkedHashMap

Klasy `LinkedHashSet` i `LinkedHashMap`, które pamiętają kolejność wstawiania elementów. Tym samym rozwiązują problem losowej kolejności przechowywanych elementów. Elementy wstawiane do tablicy są łączone w listy dwustronne (zobacz rysunek 9.11).



Rysunek 9.11. Powiązana tablica skrótów

Przeanalizujmy na przykład poniższe instrukcje z listingu 9.6, wstawiające elementy do słownika:

```
var staff = new LinkedHashMap<String, Employee>();
staff.put("144-25-5464", new Employee("Anna Kowalska"));
staff.put("567-24-2546", new Employee("Henryk Kwiatek"));
staff.put("157-62-7935", new Employee("Marcin Nowak"));
staff.put("456-62-5527", new Employee("Franciszek Frankowski"));
```

Instrukcja `staff.keySet().iterator()` przegląda klucze w następującej kolejności:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

Następnie `staff.values().iterator()` przegląda wartości w takim porządku:

```
Anna Kowalska
Henryk Kwiatek
Marcin Nowak
Franciszek Frankowski
```

Po elementach struktur `LinkedHashMap` można także iterować według *kolejności dostępu* zamiast kolejności wstawiania. Każde wywołanie metody `get` lub `put` na rzecz jakiegoś elementu powoduje jego przeniesienie na *koniec* listy powiązanej (zmienia się tylko kolejność w liście powiązanej, nie w komórce w tablicy; pozycja pozostaje w komórce, która odpowiada skrótowi klucza). Aby utworzyć taką strukturę danych, można użyć następującej instrukcji:

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Porządek według kolejności dostępu znajduje zastosowanie w usuwaniu najdłużej nieużywanych elementów z pamięci podręcznej. Na przykład często używane obiekty mogą być przechowywane w pamięci, a rzadziej używane w bazie danych. Jeśli nie uda się znaleźć jakiegoś obiektu w tablicy i jest ona zapełniona, można do niej wpuścić iterator, który usunie kilka pierwszych elementów. Będą to te obiekty, które były używane w najdalszej przeszłości.

Proces ten można nawet zautomatyzować. W tym celu należy utworzyć podklasę klasy `Linked`
`↪HashMap` i przedefiniować poniższą metodę:

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Ta metoda jest wywoływana po dodaniu nowej pozycji. Jeśli zwróci `true`, to powoduje usunięcie najstarszego elementu (`eldest`). Poniższa przykładowa pamięć podręczna przechowuje maksymalnie sto elementów:

```
var cache = new LinkedHashMap<K, V>(128, 0.75F, true)
{
  protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
  {
    return size() > 100;
  }
};
```

Możesz też sprawdzić najstarszy element, aby podjąć decyzję, czy go usunąć. Można na przykład sprawdzić jego znacznik czasu i usunąć go tylko wtedy, gdy jest wystarczająco stary. Nie spowoduje to automatycznego usunięcia wszystkich starych elementów, ponieważ metoda `removeEldestEntry` jest wywoływana tylko raz dla każdej nowej pozycji. W metodzie `removeEldestEntry` możesz modyfikować słownik, na przykład przez usuwanie początkowych elementów, które są wystarczająco stare. W takim przypadku musisz zwracać `false`.

Jeśli potrzebujesz zbioru ostatnio dodanych elementów zamiast słownika, zobacz punkt 9.5.4.

9.4.6. Klasy EnumSet i EnumMap

Struktura danych realizowana przez klasę `EnumSet` przechowuje elementy typu wyliczeniowego. Ponieważ typ wyliczeniowy ma skończoną liczbę egzemplarzy, wewnętrzna implementacja struktury `EnumSet` jest szeregiem bitów. Jeśli odpowiednia wartość znajduje się w zbiorze, włączany jest jej bit.

Klasa EnumSet nie posiada konstruktora publicznego. Do utworzenia tego typu zbioru trzeba użyć statycznej metody fabrycznej:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Do modyfikacji zbioru EnumSet można używać zwykłych metod z interfejsu Set.

EnumMap to słownik, w którym klucze są typu wyliczeniowego. Jest on prostą i wydajną tablicą wartości. Typ klucza należy określić w konstruktorze:

```
var personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```

W dokumentacji API klasy EnumSet i EnumMap znajdują się dziwnie wyglądające parametry typowe w formie E extends Enum<E>. Zapis ten oznacza, że E jest typem wyliczeniowym. Wszystkie typy wyliczeniowe dziedziczą po generycznej klasie Enum. Na przykład klasa Weekday dziedziczy po klasie Enum<Weekday>.

9.4.7. Klasa IdentityHashMap

Klasa IdentityHashMap również jest przeznaczona do specjalnych celów. Wartości haszowe kluczy nie powinny być w niej obliczane przez metodę hashCode, ale przez System.identityHash
↪Code. Metody tej używa metoda Object.hashCode do obliczania skrótu z adresu obiektu w pamięci. Ponadto klasa ta porównuje obiekty za pomocą operatora == zamiast metody equals.

Dzięki temu dwa obiekty są uznawane za różne, nawet jeśli mają taką samą zawartość. Klasa ta znajduje zastosowanie w implementacji algorytmów przemierzających obiekty (na przykład serializujących), gdzie konieczne jest pamiętanie, które obiekty zostały już przetworzone.

W Javie 20 naprawiono subtelny błąd z IdentityHashMap, który pojawił się w Javie 8 (*https://bugs.openjdk.org/browse/JDK-8284901*). Dwie nieczęsto używane metody błędnie używały metody equals zamiast operatora ==:

```
void remove(Object key, Object value)
void replace(K key, V oldValue, V newValue)
```

Metody te są czasami używane w algorytmach współbieżnych, w przypadkach, w których słownik powinien być modyfikowany tylko wtedy, gdy nie został zmieniony przez inny wątek. Można sobie tylko wyobrazić frustrację pierwszego programisty, który natknął się na ten błąd.

Żaden system oprogramowania, który jest tak skomplikowany jak platforma Java, nie może być całkowicie wolny od błędów. Dlatego tak ważne jest regularne aktualizowanie do najnowszej wersji.

`java.util.WeakHashMap<K, V>` **1.2**

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`

  Tworzy pusty słownik `WeakHashMap` o określonej pojemności i określonym współczynniku zapełnienia.

`java.util.LinkedHashSet<E>` **1.4**

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`

  Tworzy pusty zbiór `LinkedHashSet` o określonej pojemności i określonym współczynniku zapełnienia.

`java.util.LinkedHashMap<K, V>` **1.4**

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`

  Tworzy pusty słownik `LinkedHashMap` o określonej pojemności oraz określonym współczynniku zapełnienia i porządku. Jeśli parametr `accessOrder` ma wartość `true`, stosowany jest porządek według kolejności dostępu. Wartość `false` oznacza porządek w kolejności wstawiania.

- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`

  Tę metodę należy przedefiniować, aby zwracała wartość `true`, jeśli najstarsza pozycja ma być usuwana. Parametr `eldest` określa pozycję, której usunięcie jest rozważane. Metoda ta jest wywoływana po dodaniu pozycji do słownika. Domyślna implementacja zwraca wartość `false` — stare elementy nie są domyślnie usuwane. Można jednak ją przedefiniować, aby selektywnie zwracała wartość `true`, na przykład gdy najstarsza pozycja spełnia określone warunki lub słownik przekracza określony rozmiar.

`java.util.EnumSet<E extends Enum<E>>` **5.0**

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`

  Zwraca zbiór zapełniony wartościami z określonego typu wyliczeniowego.

- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`

  Zwraca zmienny zbiór, który początkowo jest pusty.

- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`

  Zwraca zmienny zbiór zapełniony wartościami z zakresu od `from` do `to` (włącznie).

- `static <E extends Enum<E>> EnumSet<E> of(E e)`

  `...`

- `static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)`
- `static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)`

  Zwraca zmienny zbiór zawierający określone elementy, które nie mogą być `null`.

- `public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)`
- `public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`

  Tworzy zmienny zbiór, który początkowo zawiera podane elementy. W drugiej metodzie argument `c` musi być typu `EnumSet` lub musi coś zawierać (aby można było określić typ elementów).

`java.util.EnumMap<K extends Enum<K>, V>` **5.0**

- `EnumMap(Class<K> keyType)`

  Tworzy słownik, którego klucze są określonego typu.

`java.util.IdentityHashMap<K, V>` **1.4**

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`

  Tworzy pusty zmienny słownik `IdentityHashMap` o pojemności równej najmniejszej potędze cyfry 2 większej niż `1,5*expectedMaxSize` (domyślna wartość parametru `expectedMaxSize` to 21).

`java.lang.System` **1.0**

- `static int identityHashCode(Object obj)` **1.1**

  Zwraca ten sam skrót (obliczony na podstawie adresu w pamięci obiektu) co metoda `Object.hashCode`, nawet jeśli w klasie, do której należy obiekt `obj`, przedefiniowano metodę `hashCode`.

9.5. Kopie i widoki

Z rysunków 9.4 i 9.5 można wyciągnąć wniosek, że projektanci Javy wpadli w lekką przesadę, tworząc tak wiele interfejsów i klas abstrakcyjnych do implementacji raczej umiarkowanej liczby konkretnych klas kolekcyjnych. Jednak rysunki te nie pokazują wszystkiego. Za pomocą **widoków** (ang. *view*) można tworzyć inne obiekty implementujące interfejsy `Collection` i `Map`. Przykład tego przedstawiliśmy na podstawie metody `keySet` z klas słownikowych. Na pierwszy rzut oka wydaje się, że to ta metoda tworzy nowy zbiór, zapełnia go wszystkimi kluczami ze słownika i zwraca go. Nie jest to jednak prawda. Metoda `keySet` zwraca obiekt klasy implementującej interfejs `Set`, którego metody operują na oryginalnym słowniku. Taka kolekcja nazywana jest **widokiem**.

Widoki mają kilka ważnych zastosowań w Java Collections Framework. Opisujemy je w poniższych podrozdziałach.

9.5.1. Małe kolekcje

Dostępne są metody statyczne zwracające zbiory lub listy określonych elementów oraz słowniki zawierające dane pary klucz-wartość.

Na przykład:

```
List<String> names = List.of("Piotr", "Paweł", "Maria");
Set<Integer> numbers = Set.of(2, 3, 5);
```

Powyższy kod zwraca listę i zbiór zawierające po trzy elementy.

Można też przekazać tablicę obiektów:

```
String[] namesArray = { "Piotr", "Paweł", "Maria" };
List<String> names = List.of(namesArray); // Lista zawierająca trzy obiekty
```

Jeśli przekażesz tablicę wartości typu podstawowego, wynikiem będzie lista zawierająca jeden element — tablicę:

```
int[] numbersArray = { 2, 3, 5 };
List<int[]> arrays = List.of(numbersArray);
    // Lista zawierająca jeden obiekt — tablicę int[]
```

Nie ma metody pomocniczej, która zamienia tablicę wartości typu podstawowego na listę elementów typu opakowaniowego.

Aby utworzyć słownik, należy przekazać klucze i wartości, na przykład:

```
Map<String, Integer> scores = Map.of("Piotr", 2, "Paweł", 3, "Maria", 5);
```

Elementy, klucze i wartości nie mogą być `null`. Klucze w zbiorach i słownikach nie mogą się powtarzać:

```
numbers = Set.of(13, null); // Błąd -- element null
scores = Map.of("Piotr", 4, "Piotr", 2); // Błąd -- powtórzony klucz
```

Nie ma żadnej gwarancji co do kolejności iteracji tych zbiorów i słowników. Nawet celowo jest ona pomieszana za pomocą ziarna, które jest randomizowane przy każdym uruchomieniu maszyny wirtualnej. Spójrz na dwa poniższe wykonania polecenia `jshell`:

```
$ jshell -q
jshell> Set.of("Piotr", "Paweł", "Maria")
$1 ==> [Piotr, Maria, Paweł]
jshell> /exit
$ jshell -q
jshell> Set.of("Piotr", "Paweł", "Maria")
$1 ==> [Paweł, Maria, Piotr]
```

> Niektórzy programiści Javy piszą programy, których poprawność opiera się na założeniu, że szczegóły implementacyjne nigdy się nie zmienią. To bardzo utrudnia implementatorom bibliotek wprowadzanie praktycznych zmian. W tym przypadku przesłanie jest jasne — nie zakładaj w programach jakiejkolwiek kolejności elementów.

Interfejsy `List` i `Set` mają po jedenaście metod przyjmujących od zera do dziesięciu argumentów oraz metodę `of` przyjmującą zmienną liczbę argumentów. Specjalizacje utworzono w celu optymalizacji wydajności.

W interfejsie `Map` nie może być wersji ze zmienną liczbą argumentów, ponieważ typy argumentów dotyczących kluczy i wartości przeplatają się po kolei. Istnieje statyczna metoda `ofEntries` przyjmująca dowolną liczbę obiektów typu `Map.Entry <K, V>`, które można utworzyć za pomocą statycznej metody `entry`. Na przykład:

```
import static java.util.Map.*;
. . .
Map<String, Integer> scores = ofEntries(
entry("Piotr", 2),
entry("Paweł", 3),
entry("Maria", 5));
```

Metody `of` i `ofEntries` tworzą obiekty klas, które mają zmienną egzemplarzową dla każdego elementu lub opierają się na tablicy.

Takie obiekty kolekcji są **niemodyfikowalne**. Jakakolwiek próba zmiany ich zawartości skutkuje wyjątkiem `UnsupportedOperationException`.

Jeśli potrzebujesz zmiennej kolekcji, możesz przekazać niezmienną kolekcję do konstruktora:

```
var names = new ArrayList<>(List.of("Piotr", "Paweł", "Maria")); // Zmienna lista imion
```

Poniższa instrukcja zwraca niemodyfikowalny obiekt implementujący interfejs `List` oraz tworzy złudzenie istnienia `n` elementów typu `anObject`.

```
Collections.nCopies(n, anObject)
```

Na przykład poniższa instrukcja tworzy listę zawierającą 100 łańcuchów `DOMYŚLNY`:

```
List<String> settings = Collections.nCopies(100, "DOMYŚLNY");
```

Struktura ta zajmuje bardzo mało miejsca w pamięci — obiekt jest zapisany tylko w jednym miejscu. Jest to sprytne zastosowanie techniki widoków.

Klasa `Collections` zawiera kilka metod, których parametry lub wartości zwrotne są kolekcjami. Nie należy jej mylić z interfejsem `Collection`.

W Javie nie ma klasy `Pair`, więc niektórzy programiści z braku lepszego rozwiązania używają `Map.Entry`.

9.5.2. Niemodyfikowalne kopie i widoki

Aby utworzyć **niemodyfikowalną kopię** kolekcji, należy użyć metody `copyOf` z typu kolekcji:

```
ArrayList<String> names = . . .;
Set<String> nameSet = Set.copyOf(names); // Imiona jako niemodyfikowalny zbiór
List<String> nameList = List.copyOf(names); // Imiona jako niemodyfikowalna lista
```

Tak jak w przypadku metod `of`, metody `thecopyOf` nie tworzą kolekcji elementów `null`, lecz zgłaszają wyjątek `NullPointerException`.

Każda metoda `copyOf` tworzy kopię kolekcji. Jeśli oryginalna kolekcja zostanie zmodyfikowana, nie ma to wpływu na kopię.

Jeśli oryginalna kolekcja jest niemodyfikowalna i prawidłowego typu, wówczas metoda `copyOf` po prostu ją zwraca:

```
Set<String> names = Set.of("Piotr", "Paweł", "Maria");
Set<String> nameSet = Set.copyOf(names); // Nie trzeba wykonywać kopii: names == nameSet
```

Klasa `Collections` zawiera metody, które tworzą **niemodyfikowalne widoki** kolekcji. Stanowią one dodatkowy element kontrolny w istniejącej kolekcji w czasie działania programu. Jeśli zostaje wykryta próba modyfikacji niemodyfikowalnej kolekcji, następuje zgłoszenie wyjątku.

Jeśli jednak oryginalna kolekcja się zmieni, to widok odzwierciedli te zmiany. To odróżnia widoki od kopii.

Niemodyfikowalne widoki można tworzyć za pomocą ośmiu metod:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableNavigableSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
Collections.unmodifiableNavigableMap
```

Każda z nich jest zdefiniowana tak, aby działała na podstawie interfejsu. Na przykład `Collections.unmodifiableList` działa z `ArrayList`, `LinkedList` i każdą inną klasą implementującą interfejs `List`.

Powiedzmy na przykład, że chcesz, aby pewna część Twojego programu miała dostęp do zawartości kolekcji, ale jej nie modyfikowała. Oto co możesz zrobić:

```
var staff = new LinkedList<String>();
. . .
lookAt(Collections.unmodifiableList(staff));
```

Metoda `Collections.unmodifiableList` zwraca obiekt klasy implementującej interfejs `List`. Jej metody dostępowe pobierają wartości z kolekcji `staff`. Oczywiście metoda `lookAt` może wywoływać wszystkie metody z interfejsu `List`, nie tylko dostępowe. Jednak wszystkie metody modyfikujące (takie jak `add`) zostały tak zmodyfikowane, aby zgłaszały wyjątek `UnsupportedOperationException`, zamiast przekazywać wywołanie do kolekcji.

Niemodyfikowalny widok nie czyni samej kolekcji niemodyfikowalną. Ją nadal można modyfikować przez oryginalną referencję (w tym przypadku `staff`). I nadal można wywoływać metody modyfikujące na jej elementach.

Widoki opakowują *interfejs*, nie rzeczywisty obiekt kolekcji, więc programista ma dostęp tylko do tych metod, które są zdefiniowane w interfejsie. Na przykład klasa `LinkedList` zawiera metody pomocnicze `addFirst` i `addLast`, które nie należą do interfejsu `List`. Nie ma więc do nich dostępu przez niemodyfikowalny widok.

Metoda `unmodifiableCollection` (jak również metody `synchronizedCollection` i `checkedCollection`, opisane dalej) zwraca kolekcję, której metoda `equals` nie wywołuje metody `equals` podstawowej kolekcji. Zamiast tego dziedziczy metodę `equals` klasy `Object`, sprawdzającą tylko, czy obiekty są identyczne. Jeśli zamienisz zbiór lub listę w kolekcję, stracisz możliwość porównywania treści. Widok działa w ten sposób, ponieważ sprawdzanie równości na tym poziomie hierarchii nie jest dobrze zdefiniowane. Widoki traktują metodę `hashCode` w taki sam sposób.

Preferuj metody opakowujące `unmodifiableSet` i `unmodifiableList`, których metody `equals` i `hashCode` są odpowiednie do zbiorów i list.

```
var names = Set.of("Piotr", "Paweł", "Maria");
Collections.unmodifiableCollection(names).equals(names) // fałsz
Collections.unmodifiableSet(names).equals(names) // prawda
```

9.5.3. Przedziały

Przedziały można tworzyć z kilku rodzajów kolekcji. Wyobraźmy sobie na przykład, że mamy listę o nazwie `staff` i chcemy z niej wyodrębnić elementy od 10 do 19. Można utworzyć widok tego przedziału za pomocą metody `subList`.

```
List<Employee> group2 = staff.subList(10, 20);
```

Pierwszy indeks jest wliczany, drugi nie — podobnie jak z parametrami metody `substring` z klasy `String`.

Na przedziale można wykonywać dowolne działania, a ich rezultat będzie automatycznie widoczny w całej liście. Można na przykład usunąć cały przedział:

```
group2.clear();    // redukcja personelu
```

Przedział `group2` jest teraz pusty, a z listy `staff` zostały usunięte te elementy, które znajdowały się w tym przedziale.

Przedziały uporządkowanych zbiorów i słowników tworzy się przy użyciu kolejności sortowania, a nie pozycji elementów w kolekcji. W interfejsie `SortedSet` znajdują się trzy metody:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

Zwracają one podzbiory wszystkich elementów, które są większe niż lub równe `from` i mniejsze od `to`. Podobne metody dla uporządkowanych słowników to:

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

Zwracają one widoki słowników zawierające pozycje, których *klucze* mieszczą się w określonych zakresach.

Interfejs `NavigableSet` daje większe możliwości kontroli działań na przedziałach. Można określić, czy wartości graniczne mają być wliczane (ang. *inclusive*):

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

9.5.4. Zbiory ze słowników wartości logicznych

Czasami API dostarcza przydatnych słowników o praktycznych cechach, jakie chcielibyśmy mieć w zbiorze. W punkcie 9.4.5 pokazałem, jak za pomocą klasy `LinkedHashMap` utworzyć pamięć podręczną usuwającą starsze elementy. Klasa `LinkedHashSet` nie ma takiej funkcjonalności.

Jeśli chcesz utworzyć zbiór 100 ostatnio wstawionych łańcuchów, to utwórz `LinkedHashMap` ➥`<String, Boolean>`, po czym wywołaj metodę `Collections.newSetFromMap`:

```
Set<String> cache = Collections.newSetFromMap(new LinkedHashMap<String, Boolean>()
    {
        protected boolean removeEldestEntry(Map.Entry<String, Boolean> eldest)
        {
            return size() > 100;
        }
});
```

Widok zbioru bazuje na słowniku. Kiedy dodajemy element `e`, to widok dodaje do słownika pozycję o kluczu `e` i wartości `Boolean.TRUE`. Jeśli w słowniku pojawi się więcej niż 100 pozycji, najstarsza zostanie usunięta.

Metodę `newSetFromMap` należy wywołać z pustym słownikiem. Potem nie należy modyfikować słownika, tylko pozwolić, aby wszystkie modyfikacje były wykonywane przez metody widoku zbioru. W tym celu najlepiej jest przekazać słownik do metody `newSetFromMap`, nie zachowując do niego referencji, tak jak pokazałem w poprzednim przykładzie.

Metoda `newSetFromMap` jest przydatna także podczas pracy ze strukturami `WeakHashMap` (punkt 9.4.4). W przypadku struktury `ConcurrentHashMap` użyj w zamian metody `newKeySet` (rozdział 10.).

9.5.5. Widoki odwrócone

Metody `reversed` interfejsów `SequencedCollection` i `SequencedSet` zwracają widoki, które pokazują elementy w odwróconej kolejności. Na przykład poniżej pokazuję, jak odwrotnie przeglądać listę lub zbiór drzewiasty:

```
for (String element : collection.reversed())
{
    działania na elemencie
}
```

W przypadku `SequencedMap` metoda `reversed` pokazuje słownik z kluczami w odwróconej kolejności.

9.5.6. Widoki kontrolowane

Widoki kontrolowane mają na celu wspieranie programisty w procesie usuwania błędów związanych z typami generycznymi. Wiemy już z rozdziału 8., że do generycznej kolekcji da się przemycić elementy niewłaściwego typu. Na przykład:

```
var strings = new ArrayList<String>();
ArrayList rawList = strings;    // Zostanie zgłoszone tylko ostrzeżenie (nie błąd) dotyczące
                                //  zgodności ze starszym kodem.
rawList.add(new Date());        // Teraz referencja strings wskazuje na obiekt typu Date!
```

Błąd w instrukcji `add` nie zostanie wykryty w czasie działania programu. W zamian wystąpi wyjątek rzutowania, kiedy w innej części kodu zostanie wywołana metoda `get` i zostanie wykonane rzutowanie jej wyniku na typ `String`.

Widok kontrolowany wykryje taki błąd. Zdefiniujmy następującą bezpieczną listę:

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

Metoda `add` widoku sprawdza, czy wstawiany obiekt należy do określonej klasy; jeśli nie, generuje wyjątek `ClassCastException`. Jest to korzystne, ponieważ błąd zostaje zgłoszony w odpowiednim miejscu:

```
ArrayList rawList = safeStrings;
rawList.add(new Date());        // Lista kontrolowana generuje wyjątek ClassCastException.
```

Widoki kontrolowane są ograniczone zestawem testów, które może przeprowadzać maszyna wirtualna w czasie działania programu. Na przykład struktury `Array`
`↪List<Pair<String>>` nie można ochronić przed wstawieniem do niej typu `Pair<Date>`, ponieważ maszyna wirtualna dysponuje tylko jedną surową klasą `Pair`.

9.5.7. Widoki synchronizowane

Jeśli do kolekcji mają dostęp procedury z kilku wątków, należy zadbać o to, aby nie została ona przypadkowo zniszczona. Na przykład sytuacja, w której jeden wątek próbuje dodać elementy do kolekcji `HashTable`, podczas gdy inny ją odświeża, mogłaby być fatalna w skutkach.

Projektanci języka zdecydowali, że zamiast implementować kolekcje bezpieczne dla wątków, uczynią zwykłe kolekcje bezpiecznymi dla wątków za pomocą mechanizmu widoków. Na przykład statyczna metoda `synchronizedMap` z klasy `Collections` zamienia zwykłe słowniki na słowniki z synchronizowanymi metodami dostępowymi:

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String,
➥Employee>());
```

Teraz można dostawać się do obiektu `map` z różnych wątków. Metody takie jak `get` i `put` są synchronizowane, to znaczy każde wywołanie metody musi zostać w pełni ukończone, zanim inny wątek będzie mógł wywołać kolejną metodę. Kwestię synchronizowanego dostępu do struktur danych szczegółowo omawiamy w rozdziale 10.

9.5.8. Uwagi dotyczące operacji opcjonalnych

Widoki z reguły mają jakieś ograniczenia — mogą być tylko do odczytu, mogą nie mieć możliwości zmiany rozmiaru lub pozwalać na usuwanie elementów, ale nie na dodawanie, jak na przykład widok kluczy słownika. Próba wykonania niedozwolonej operacji na ograniczonym widoku kończy się zgłoszeniem wyjątku `UnsupportedOperationException`.

W dokumentacji API wiele metod klas i interfejsów kolekcyjnych oznaczono jako operacje opcjonalne (ang. *optional operations*). Wydaje się to sprzeczne z ideą interfejsu, który przecież — jak wiadomo — określa zestaw metod, które klasa *musi* implementować. Rzeczywiście ten stan rzeczy nie jest satysfakcjonujący z teoretycznego punktu widzenia. Lepszym rozwiązaniem byłoby utworzenie osobnych interfejsów dla widoków tylko do odczytu i widoków, które nie mogą zmienić rozmiaru kolekcji. Wtedy jednak liczba interfejsów potroiłaby się, a to dla projektantów biblioteki byłoby nie do zaakceptowania.

Czy powinno się rozszerzać technikę opcjonalnych operacji na własne projekty? Naszym zdaniem nie. Mimo iż kolekcje są używane często, styl programowania związany z ich implementacją nie jest typowy dla innych dziedzin. Projektanci biblioteki kolekcyjnej stoją przed niezwykle trudnym zadaniem zaspokojenia sprzecznych wymagań. Z jednej strony użytkownicy wymagają, aby biblioteka była łatwa do nauki, wygodna w użyciu, w pełni generyczna i niepodatna na błędy, a z drugiej strony ma być tak samo wydajna jak ręcznie optymalizowane algorytmy. Spełnienie tych wszystkich żądań naraz (czy choćby zbliżenie się do nich) jest najzwyczajniej niemożliwe. Jednak we własnej praktyce programistycznej nie będziesz często rozwiązywać tak ekstremalnych problemów. Powinno być możliwe znalezienie rozwiązań, które nie polegają na wykorzystaniu ostatecznego środka w postaci opcjonalnych operacji interfejsu.

java.util.List 1.2

- `static <E> List<E> of()` **9**
- `static <E> List<E> of(E e1)` **9**

  ...
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)` **9**
- `static <E> List<E> of(E... elements)` **9**

  Zwraca niezmienną listą danych elementów, które nie mogą być `null`.
- `static <E> List<E> copyOf(Collection<? extends E> coll)` **10**

  Zwraca niezmienną kopię danej kolekcji.

java.util.Set 1.2

- `static <E> Set<E> of()` **9**
- `static <E> Set<E> of(E e1)` **9**

  ...
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)` **9**
- `static <E> Set<E> of(E... elements)` **9**

  Zwraca niezmienny zbiór danych elementów, które nie mogą być `null`.
- `static <E> Set<E> copyOf(Collection<? extends E> coll)` **10**

  Zwraca niezmienną kopię danej kolekcji.

java.util.Map 1.2

- `static <K, V> Map<K, V> of()` **9**
- `static <K, V> Map<K, V> of(K k1, V v1)` **9**

  ...
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)` **9**

  Zwraca niezmienny słownik danych kluczy i wartości, które nie mogą być `null`.
- `static <K,V> Map.Entry<K,V> entry(K k, V v)` **9**

  Zwraca niezmienny element słownika o określonych kluczu i wartości, które nie mogą być `null`.
- `static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)` **9**

  Zwraca niezmienny słownik danych elementów.

- `static <K, V> Map<K,V> copyOf(Map<? extends K,? extends V> map)` **10**

  Zwraca niezmienną kopię danego słownika.

`java.util.Collections` **1.2**

- `static <E> Set<E> newSetFromMap(Map<E, Boolean> map)` **6**

  Zwraca zbiór bazujący na określonym słowniku, który początkowo powinien być pusty, a potem modyfikowany tylko przez zwrócony widok.

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
- `static <E> SequencedCollection<E> unmodifiableSequencedCollection(SequencedCollection<E> c)` **21**
- `static <E> List<E> unmodifiableList(List<E> c)`
- `static <E> Set<E> unmodifiableSet(Set<E> c)`
- `static <E> SequencedSet<E> unmodifiableSequencedSet(SequencedCollection<E> c)` **21**
- `static <E> SortedSet<E> unmodifiableSortedSet(SortedSet<E> c)`
- `static <E> SortedSet<E> unmodifiableNavigableSet(NavigableSet<E> c)` **8**
- `static <K, V> Map<K, V> unmodifiableMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> c)`
- `static <K, V> SequencedMap<K, V> unmodifiableSequencedMap(SequencedMap<K, V> c)` **21**
- `static <K, V> SortedMap unmodifiableNavigableMap(NavigableMap<K, V> c)` **8**

  Tworzy widoki kolekcji, których metody modyfikujące generują wyjątek `UnsupportedOperationException`.

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <E> NavigableSet synchronizedNavigableSet(NavigableSet<E> c)` **8**
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`
- `static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> c)` **8**

  Tworzy widoki kolekcji, których metody są zsynchronizowane.

- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`

- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <E> NavigableSet checkedNavigableSet(NavigableSet<E> c, Class<E> elementType)` **8**
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> NavigableMap checkedNavigableMap(NavigableMap<K, V> c, Class<K> keyType, Class<V> valueType)` **8**
- `static <E> Queue<E> checkedQueue(Queue<E> queue, Class<E> elementType)` **8**

  Tworzy widoki kolekcji, których metody zgłaszają wyjątek `ClassCastException`, jeśli wstawiany element jest złego typu.

- `static <E> List<E> nCopies(int n, E value)`

  Zwraca niemodyfikowalną listę zawierającą *n* identycznych kopii.

- `static <E> List<E> singletonList(E value)`
- `static <E> Set<E> singleton(E value)`
- `static <E> List<E> emptyList()`
- `static <T> Set<T> emptySet()`
- `static <E> SortedSet<E> emptySortedSet()`
- `static NavigableSet<E> emptyNavigableSet()`
- `static <K, V> Map<K, V> emptyMap()`
- `static <K, V> SortedMap<K, V> emptySortedMap()`
- `static <K, V> NavigableMap<K, V> emptyNavigableMap()`
- `static <T> Enumeration<T> emptyEnumeration()`
- `static <T> Iterator<T> emptyIterator()`
- `static <T> ListIterator<T> emptyListIterator()`

  Zwraca pustą kolekcję, słownik lub iterator.

`java.util.Arrays` **1.2**

- `static <E> List<E> asList(E... array)`

  Zwraca widok listowy elementów tablicy, który można modyfikować, ale nie można zmieniać jego rozmiaru.

`java.util.List<E>` **1.2**

- `List<E> subList(int firstIncluded, int firstExcluded)`

  Zwraca widok listowy elementów składający się z pozycji należących do określonego przedziału.

`java.util.SortedSet<E>` **1.2**

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

  Zwraca widok elementów z określonego przedziału.

`API java.util.NavigableSet<E>` **6**

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`

  Zwraca widok elementów z określonego przedziału. Parametr logiczny określa, czy element brzegowy ma być włączany do widoku.

`java.util.SortedMap<K, V>` **1.2**

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`

  Zwraca widok pozycji słownika, których klucze mieszczą się w określonym przedziale.

`java.util.NavigableMap<K, V>` **6**

- `NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)`
- `NavigableMap<K, V> headMap(K from, boolean fromIncluded)`
- `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`

  Zwraca widok pozycji słownika, których klucze mieszczą się w określonym przedziale. Parametr logiczny określa, czy element brzegowy ma być włączany do widoku.

`java.util.SequencedCollection<E>` **21**

- `SequencedCollection<E> reversed()`

  Zwraca widok sekwencyjnej kolekcji z elementami w odwróconej kolejności.

`java.util.SequencedSet<E>` **21**

- `SequencedSet<E> reversed()`

  Zwraca widok sekwencyjnego zbioru z elementami w odwróconej kolejności.

`java.util.SequencedMap<E>` **21**

- `SequencedMap<E> reversed()`

  Zwraca widok sekwencyjnego słownika z kluczami w odwróconej kolejności.

9.6. Algorytmy

System kolekcji Javy, oprócz implementacji klas kolekcyjnych, zawiera także kilka praktycznych algorytmów. W tym podrozdziale dowiesz się, jak korzystać z tych algorytmów i jak pisać własne, które będą dobrze współpracowały z systemem kolekcji.

9.6.1. Dlaczego algorytmy generyczne?

Generyczne interfejsy kolekcyjne mają pewną dużą zaletę — algorytmy wystarczy zaimplementować tylko jeden raz. Weźmy na przykład prosty algorytm znajdujący największy element w kolekcji. Standardowo jego implementacja polegałaby na użyciu pętli. Poniższa procedura znajduje największy element tablicy:

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
  if (largest.compareTo(a[i]) < 0)
    largest = a[i];
```

Oczywiście w przypadku listy tablicowej kod wyglądałby nieco inaczej:

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
  if (largest.compareTo(v.get(i)) < 0)
    largest = v.get(i);
```

Jak wygląda sytuacja w listach powiązanych? Nie istnieje w nich szybki mechanizm dostępu swobodnego, ale można użyć iteratora:

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
  T next = iter.next();
  if (largest.compareTo(next) < 0)
    largest = next;
}
```

Pisanie tych pętli jest uciążliwe, a ponadto są one podatne na błędy. Łatwo popełnić błąd pomyłki o jeden (ang. *off-by-one error*), nie wiadomo, czy pętla zadziała prawidłowo w przypadku pustego kontenera lub zawierającego tylko jeden element. Perspektywa ciągłego testowania i debugowania całego tego kodu nie jest zachęcająca, ale implementacja całej masy metod także nie napawa radością, na przykład:

```
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

W takiej sytuacji z pomocą przychodzą interfejsy kolekcyjne. Pomyślmy, jak powinien wyglądać *minimalny* interfejs potrzebny do realizacji tego algorytmu. Dostęp swobodny za pomocą metod `get` i `set` jest operacją bardziej złożoną niż zwykły dostęp za pomocą iteratora. Jak przekonaliśmy się przy okazji szukania największego elementu listy powiązanej, do wykonania tego zadania nie jest potrzebny dostęp swobodny. Największy element można znaleźć za pomocą prostej iteracji po elementach. Dlatego metodę `max` można zaimplementować w taki sposób, aby przyjmowała *każdy* obiekt implementujący interfejs `Collection`.

```
public static <T extends Comparable> T max(Collection<T> elements)
{
  if (elements.isEmpty()) throw new NoSuchElementException();
  Iterator<T> iter = elements.iterator();
  T largest = iter.next();
  while (iter.hasNext())
  {
    T next = iter.next();
    if (largest.compareTo(next) < 0)
      largest = next;
  }
  return largest;
}
```

Oczywiście, jako że tablice nie są kolekcjami, musimy napisać dla nich osobną metodę:

```
static <T extends Comparable> T max(T[] a)
```

Algorytmy generyczne dają bardzo duże możliwości. W bibliotece standardowej C++ znajduje się mnóstwo przydatnych algorytmów, z których każdy operuje na kolekcjach generycznych. Biblioteka Java Collections Framework nie jest tak bogata, ale klasy `Collections` i `Arrays` zapewniają podstawowe funkcje: sortowanie, wyszukiwanie binarne i kilka prostych algorytmów użytkowych.

W Java Collections Framework brakuje niektórych przydatnych algorytmów, ale można je znaleźć w bibliotece strumieni opisanej w tomie drugim. Aby na przykład znaleźć wartość największą lub najmniejszą, zamień kolekcję lub tablicę w strumień:

```
largest = coll.stream().max(Comparator.naturalOrder()).get();
```

9.6.2. Sortowanie i tasowanie

Weterani komputerowi pamiętają jeszcze czasy, kiedy algorytmy sortujące programowali ręcznie za pomocą kart dziurkowanych. Obecnie algorytmy sortujące wchodzą w skład biblioteki standardowej większości języków programowania. Java nie jest tu wyjątkiem.

Metoda `sort` z klasy `Collections` sortuje kolekcje implementujące interfejs `List`.

```
var staff = new ArrayList<String>();
Wstawienie elementów do kolekcji
Collections.sort(staff);
```

Ta metoda zakłada, że elementy listy implementują interfejs `Comparable`. Aby posortować listę w inny sposób, można użyć metody `sort` z interfejsu `List` i przekazać obiekt typu `Comparator`. Oto sposób posortowania pracowników według wysokości wynagrodzenia:

```
Collections.sort(staff, sort(Comparator.comparingDouble(Employee::getSalary));
```

Aby posortować listę w kolejności *malejącej*, należy użyć metody `Comparator.reverseOrder()`. Tworzy ona komparator, który zwraca wynik operacji `b.compareTo(a)`. Na przykład poniższa instrukcja sortuje elementy listy `staff` w kolejności malejącej według porządku określonego przez metodę `compareTo` należącą do typu elementu.

```
Collections.sort(staff, sort(Comparator.reverseOrder())
```

Natomiast poniższe wyrażenie sortuje pracowników w kolejności od najwyższego wynagrodzenia:

```
Collections.sort(staff, sort(Comparator.comparingDouble(Employee::getSalary).reversed())
```

W tym miejscu może się rodzić pytanie, jak metoda `sort` sortuje listy. Typowe algorytmy sortujące prezentowane w książkach o algorytmach działają na tablicach i korzystają z dostępu swobodnego. W listach ten typ dostępu może być bardzo wolny. Można je jednak szybko sortować przy użyciu algorytmu sortowania przez scalanie. W Javie jednak nie wykorzystano tej metody. W zamian wszystkie elementy listy są wrzucane do tablicy, tam sortowane przy użyciu innej wersji algorytmu sortowania przez scalanie, a następnie kopiowane w uporządkowanej kolejności z powrotem do listy.

Algorytm sortowania stosowany w Java Collections Framework ustępuje nieco prędkością algorytmowi **quick sort**, który jest standardowo używany do implementacji algorytmów sortujących ogólnego przeznaczenia. Ma on jednak pewną ważną zaletę — jest **stabilny**, to znaczy nie zamienia miejscami takich samych elementów. Po co przejmować się kolejnością identycznych elementów? Oto typowa sytuacja. Mamy listę pracowników posortowaną według nazwisk. Teraz sortujemy według wysokości zarobków. Stabilny algorytm sortujący zachowa kolejność nazwisk. Mówiąc inaczej, lista będzie posortowana najpierw według zarobków, a potem według nazwisk.

Ponieważ kolekcje nie muszą implementować wszystkich swoich metod opcjonalnych, wszystkie metody przyjmujące parametry kolekcyjne muszą informować, kiedy dana kolekcja może być bezpiecznie przekazana do algorytmu. Na przykład z pewnością nie można przekazać listy `unmodifiableList` do algorytmu `sort`. Jakiego rodzaju listę *można* przekazać? Zgodnie z dokumentacją lista musi być modyfikowalna, ale nie musi zezwalać na zmianę rozmiaru.

Właściwości list można przedstawić następująco:

- Lista jest **modyfikowalna** (ang. *modifiable*), jeśli obsługuje metodę `set`.
- Lista **zezwala na zmianę jej rozmiaru** (ang. *resizable*), jeśli obsługuje metody `add` i `remove`.

W rozdziale 6. pokazałem metodę `Arrays.sort`. Są dwie wersje – jedna do sortowania tablic obiektów implementujących interfejs `Comparable`, a druga z komparatorem:

```
String[] staff = . . .;
Arrays.sort(staff);
Arrays.sort(staff, Comparator.comparingDouble(Employee::getSalary));
```

W klasie `Collections` dostępna jest też metoda `shuffle`, która działa odwrotnie do sortowania — tasuje elementy listy. Na przykład:

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards, RandomGenerator.getDefault());
```

Jeśli metodzie `shuffle` zostanie przekazana lista nieimplementująca interfejsu `RandomAccess`, jej zawartość zostanie skopiowana do tablicy, przetasowana i wstawiona z powrotem w odmiennym porządku.

Program przedstawiony na listingu 9.7 wstawia do listy tablicowej 49 obiektów typu `Integer` zawierających liczby od 1 do 49. Następnie tasuje zawartość tej listy i wybiera sześć pierwszych elementów. Na końcu sortuje pobrane wartości i drukuje je.

Listing 9.7. shuffle/ShuffleTest.java

```
package shuffle;

import java.util.*;
import java.util.random.*;

/**
 * Program demonstrujący algorytmy tasowania i sortowania
 * @version 1.13 2023-09-30
 * @author Cay Horstmann
 */
public class ShuffleTest
{
   public static void main(String[] args)
   {
      var numbers = new ArrayList<Integer>();
      for (int i = 1; i <= 49; i++)
         numbers.add(i);
      Collections.shuffle(numbers, RandomGenerator.getDef());
      List<Integer> winningCombination = numbers.subList(0, 6);
      Collections.sort(winningCombination);
      System.out.println(winningCombination);
   }
}
```

`java.util.Collections` **1.2**

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`

  Sortuje elementy listy przy użyciu stabilnego algorytmu sortującego. Gwarantowana wydajność tego algorytmu wynosi $O(n \log n)$, gdzie `n` oznacza długość listy.

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
- `static void shuffle(List<?> elements, RandomGenerator r)` **21**

  Tasuje element listy. Wydajność tego algorytmu to $O(n\ a(n))$, gdzie `n` to długość listy, a `a(n)` to średni czas dostępu do elementu.

`java.util.List<E>` **1.2**

- `default void sort(Comparator<? super T> comparator)` **8**

  Sortuje listę przy użyciu podanego komparatora.

`java.util.Comparator<E>` **1.2**

- `static <T extends Comparable<? super T>> Comparator<T> reverseOrder()` **8**

  Zwraca komparator odwracający kolejność określoną przez interfejs `Comparable`.

- `default Comparator<T> reversed()` **8**

  Zwraca komparator odwracający kolejność określoną przez ten komparator.

9.6.3. Wyszukiwanie binarne

Typowy proces poszukiwania obiektu w tablicy polega na odwiedzaniu po kolei wszystkich elementów aż do natrafienia na ten właściwy. Jeśli tablica jest posortowana, można poszukiwany element porównać ze środkowym elementem tej tablicy. Jeśli szukany element jest mniejszy, szukanie kontynuuje się w pierwszej połowie elementów, w przeciwnym przypadku w drugiej. Dzięki temu skala zadania zmniejsza się o połowę. Ten sam proces jest powtarzany na wybranej połowie itd. Jeśli na przykład tablica zawiera 1024 elementy, szukany element można znaleźć (lub stwierdzić, że go nie ma) w dziesięciu krokach, podczas gdy wyszukiwanie liniowe wymagałoby średnio 512 operacji, aby znaleźć element, i 1024, aby potwierdzić, że go nie ma.

Algorytm ten implementuje metoda `binarySearch` z klasy `Collections`. Należy pamiętać, że warunkiem do poprawnego działania tego algorytmu jest wcześniejsze posortowanie kolekcji. Na wejściu należy podać kolekcję, która ma zostać przeszukana (musi implementować interfejs `List`), i element, który ma zostać znaleziony. Jeśli kolekcja nie jest posortowana przez metodę `compareTo` z interfejsu `Comparable`, konieczne jest dostarczenie dodatkowo komparatora.

```
i = Collections.binarySearch(elements, target);
i = Collections.binarySearch(elements, target, comparator);
```

Wartość zwrotna metody binarySearch większa bądź równa zeru określa indeks pasującego obiektu. To znaczy elements.get(i) równa się target w porządku porównywania. Wartość ujemna oznacza, że element nie został znaleziony. Można jednak wykorzystać ją do określenia miejsca, w którym *należałoby* target wstawić do kolekcji, aby zachować porządek sortowania. Miejsce to jest następujące:

```
insertionPoint = -i - 1;
```

Nie można jednak napisać po prostu -i, ponieważ wartość zero nie byłaby jednoznaczna. Innymi słowy, element w odpowiednim miejscu umieszcza poniższa procedura:

```
if (i < 0)
  elements.add(-i - 1, target);
```

Aby było warte uwagi, wyszukiwanie binarne musi obsługiwać dostęp swobodny. Gdyby trzeba było w poszukiwaniu środkowego elementu przemierzyć połowę listy powiązanej, to stracilibyśmy wszystkie jego zalety. Jeśli algorytmowi binarySearch przekażesz listę, która nie implementuje interfejsu RandomAccess, to przestawia się on na algorytm iteracyjny o wydajności wyszukiwania liniowego.

Klasa Arrays ma implementacje wyszukiwania binarnego dla tablic typów podstawowych i obiektów.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

  Szuka klucza w liście posortowanej przy użyciu algorytmu wyszukiwania binarnego, jeśli elementy implementują interfejs RandomAccess, albo wyszukiwania liniowego we wszystkich pozostałych przypadkach. Długość czasu działania tych metod to O(a(n) log n), gdzie n określa średni czas dostępu do elementu. Zwracają indeks klucza w liście lub wartość ujemną i, jeśli takiego klucza nie ma w liście. W takim przypadku taki klucz powinien zostać wstawiony w miejscu obliczanym za pomocą wzoru -i - 1, aby zachować porządek sortowania.

java.util.Arrays 1.2

- `static int binarySearch(T[] a, T key)`
- `static int binarySearch(T[] a, int start, int end, T key)` **6**

  Używa algorytmu wyszukiwania binarnego w celu znalezienia klucza key w posortowanej tablicy a. Jeśli znajdzie klucz, zwraca jego indeks. W przeciwnym razie zwraca ujemną wartość r; -r - 1 to miejsce, w którym należy wstawić klucz, aby tablica a pozostała posortowana. Typem komponentu T tablicy może być Object, int, long, short, char, byte, boolean, float lub double.

- `static <T> T binarySearch(T[] a, T key, Comparator<? super T> c)`
- `static <T> T binarySearch(T[] a, int start, int end, T key, Comparator<? super T> c)` **6**

  Używa algorytmu wyszukiwania binarnego w celu znalezienia klucza `key` w tablicy `a`, która została posortowana przy użyciu określonego komparatora.

9.6.4. Proste algorytmy

W klasie `Collections` znajduje się jeszcze kilka prostych, ale przydatnych algorytmów. Wśród nich można znaleźć prezentowany na początku tego rozdziału algorytm wyszukiwania największej wartości w kolekcji. Z pozostałych można wymienić algorytm kopiujący elementy z jednej listy do innej, zapełniający kontener stałą wartością czy odwracający listę.

Po co w bibliotece umieszczać takie proste algorytmy? Przecież każdy programista mógłby je zaimplementować za pomocą pojedynczych pętli. Lubimy te algorytmy, ponieważ ułatwiają życie programistom *czytającym* cudzy kod. Czytając kod napisany przez kogoś innego, zawsze musimy rozszyfrować intencje innego programisty. Spójrz na przykład na poniższą pętlę:

```
for (int i = 0; i < words.size(); i++)
  if (words.get(i).equals("C++")) words.set(i, "Java");
```

Teraz porównaj ją z poniższym wywołaniem metody:

```
Collections.replaceAll(words, "C++", "Java");
```

Kiedy widzimy wywołanie metody, od razu wiemy, jakie jest przeznaczenie danego fragmentu kodu.

Znajdujące się niżej wyciągi z API opisują proste algorytmy dostępne w klasie `Collections`.

Domyślne metody `Collection.removeIf` i `List.replaceAl` są tylko odrobinę bardziej skomplikowane. Do testowania lub przekształcania elementów należy przekazać wyrażenie lambda. Poniżej na przykład usuwamy wszystkie krótkie słowa, a w pozostałych wszystkie litery zmieniamy na małe:

```
words.removeIf(w -> w.length() <= 3);
words.replaceAll(String::toLowerCase);
```

Predykat `removeIf` powinien sprawdzać tylko ten element, którego usunięcie jest rozważane, a nie kolekcję. Jeśli wykonasz tę drugą czynność, to jej efekt będzie zależny od implementacji.

Wersja `ArrayList` metody `removeIf` przegląda elementy dwa razy. Najpierw wyszukuje elementy, które należy usunąć, a następnie wszystkie je usuwa. W innych kolekcjach znalezione elementy są usuwane od razu.

To ma znaczenie, gdy predykat odczytuje kolekcję. Spójrz na poniższy przykład, w którym `words` jest kolekcją typu `ArrayList` o długości 3:

```
var words = new ArrayList<String>(List.of("Ada", "C++", "Java"))
words.removeIf(w -> w.length() == words.size()); // Teraz words zawiera ["Java"]
```

W pierwszym przebiegu do usunięcia zostały oznaczone wszystkie elementy o długości 3, a w drugim przebiegu zostały one usunięte.

Gdyby jednak zmienna `words` była kolekcją typu `LinkedList` albo `HashSet`, to takie samo wywołanie metody `removeIf` spowodowałoby usunięcie tylko jednego z trzyliterowych słów. Po pierwszym usunięciu predykat sprawdza, czy kolekcja ma rozmiar 2.

Skąd ta różnica? Naiwna, jednoprzebiegowa implementacja byłaby nieefektywna na kolekcjach bazujących na tablicy. Programiści nie od razu się zorientowali, że istnieje efektywny jednoprzebiegowy algorytm. A kiedy już się dowiedzieli, to postanowili go nie używać ze względu na zgodność — zobacz *https://bugs.openjdk.org/browse/JDK-8143577*.

`java.util.Collections` **1.2**

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`

  Zwraca najmniejszy lub największy element kolekcji (wartości brzegowe parametrów uproszczone dla zachowania przejrzystości).

- `static <T> void copy(List<? super T> to, List<T> from)`

  Kopiuje wszystkie elementy z listy źródłowej do tych samych miejsc w liście docelowej. Lista docelowa musi mieć przynajmniej taką samą długość jak lista źródłowa.

- `static <T> void fill(List<? super T> l, T value)`

  Wstawia na wszystkich pozycjach w liście tę samą wartość.

- `static <T> boolean addAll(Collection<? super T> c, T... values)` **5**

  Dodaje wszystkie wartości do określonej kolekcji i zwraca wartość `true`, jeśli kolekcja w wyniku tego działania zmieniła się.

- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)` **1.4**

  Zastępuje wszystkie elementy identyczne z `oldValue` wartościami `newValue`.

- `static int indexOfSubList(List<?> l, List<?> s)` **1.4**
- `static int lastIndexOfSubList(List<?> l, List<?> s)` **1.4**

  Zwraca indeks pierwszej lub ostatniej podlisty listy `l` równej `s` lub `-1`, jeśli żadna podlista `l` nie jest równa `s`. Jeśli na przykład lista `l` to `[s, t, a, r]`, a `s` to `[t, a, r]`, obie metody zwrócą indeks 1.

- `static void swap(List<?> l, int i, int j)` **1.4**

  Zamienia miejscami elementy znajdujące się w określonych miejscach.

- `static void reverse(List<?> l)`

  Odwraca kolejność elementów listy. Na przykład lista `[t, a, r]` po odwróceniu to `[r, a, t]`. Długość czasu działania tej metody to `O(n)`, gdzie `n` określa długość listy.

- `static void rotate(List<?> l, int d)` **1.4**

  Przeprowadza rotację elementów listy, przenosząc obiekt z indeksem `i` do miejsca `(i + d) % l.size()`. Na przykład rotacja listy `[t, a, r]` o dwa pola daje w wyniku listę `[a, r, t]`. Długość czasu działania tej metody to `O(n)`, gdzie `n` określa długość listy

- `static int frequency(Collection<?> c, Object o)` **5.0**

  Zwraca liczbę elementów w `c`, które są równe obiektowi `o`.

- `boolean disjoint(Collection<?> c1, Collection<?> c2)` **5.0**

  Zwraca wartość `true`, jeśli kolekcje nie mają więcej elementów wspólnych.

`java.util.Collection<T>` **1.2**

- `default boolean removeIf(Predicate<? super E> filter)` **8**

  Usuwa wszystkie elementy spełniające warunek.

`java.util.List<E>` **1.2**

- `default void replaceAll(UnaryOperator<E> op)` **8**

  Wykonuje operację na wszystkich elementach listy.

9.6.5. Operacje zbiorowe

Istnieje kilka metod do zbiorowego kopiowania i usuwania elementów. Na przykład poniższe wyrażenie usuwa z `coll1` wszystkie elementy obecne w `coll2`.

```
coll1.removeAll(coll2);
```

Natomiast poniższe wyrażenie usuwa z `coll1` wszystkie elementy, których *nie* ma w `coll2`.

```
coll1.retainAll(coll2);
```

Zobaczmy typowy przykład zastosowania tej operacji.

Przyjmijmy, że chcemy znaleźć *część wspólną* dwóch zbiorów. Zaczniemy od utworzenia nowego zbioru do przechowywania znalezionego wyniku.

```
var result = new HashSet<String>(firstSet);
```

Wykorzystujemy tutaj fakt, że każda kolekcja posiada konstruktor, którego parametr jest inną kolekcją przechowującą wartości początkowe.

Teraz użyjemy metody `retainAll`:

```
result.retainAll(secondSet);
```

Zachowuje ona wszystkie elementy, które znajdują się w obu zbiorach. W ten sposób znaleźliśmy część wspólną dwóch zbiorów bez tworzenia pętli.

Można pójść nawet dalej i zastosować operację zbiorczą do *widoku*. Wyobraźmy sobie, że dysponujemy słownikiem przechowującym obiekty typu `Employee` z kluczami w postaci identyfikatorów `ID` oraz zbiorem identyfikatorów pracowników, którzy mają zostać zwolnieni.

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

Wystarczy utworzyć zbiór kluczy i usunąć wszystkie identyfikatory zwolnionych pracowników.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Ponieważ zbiór kluczy jest widokiem słownika, klucze i skojarzeni z nimi pracownicy są automatycznie usuwani z tego słownika.

Przy użyciu widoku podprzedziału można ograniczyć operacje zbiorcze do podlist i podzbiorów. Załóżmy, że chcemy dodać 10 elementów z listy do innego kontenera. Pobieramy dziesięć pierwszych elementów i umieszczamy je w podliście:

```
relocated.addAll(staff.subList(0, 10));
```

Na podprzedziale tym można także wykonywać operacje modyfikujące.

```
staff.subList(0, 10).clear();
```

9.6.6. Konwersja pomiędzy kolekcjami a tablicami

Ponieważ znaczna część API Javy powstała przed pojawieniem się systemu Java Collections Framework, czasami konieczna jest konwersja tradycyjnych tablic na bardziej nowoczesne kolekcje.

Jeśli dysponujemy tablicą, musimy przekonwertować ją na kolekcję. Można do tego celu użyć metody `List.Of`. Na przykład:

```
String[] names = . . .;
List<String> staff = List.of(names);
```

Utworzenie tablicy z kolekcji nie jest już takie łatwe. Można oczywiście użyć do tego metody `toArray`:

```
Object[] names = staff.toArray();
```

Jednak wynik będzie tablicą obiektów typu `Object`. Nie można zastosować rzutowania, nawet jeśli wiadomo, że kolekcja zawierała obiekty określonego typu:

```
String[] names = (String[]) staff.toArray();   // Błąd!
```

Metoda `toArray` zwraca tablicę typu `Object[]`, którego nie można zmienić. Zamiast tego należy do metody `toArray` przekazać wyrażenie konstrukcyjne. Zostanie użyty konstruktor, który utworzy tablicę odpowiedniego typu:

```
String[] values = staff.toArray(String[]::new);
```

Zanim pojawił się pakiet JDK 11, programista musiał używać innej formy metody toArray, przekazując tablicę odpowiedniego typu:

```
String[] values = staff.toArray(new String[0]);
```

Ta metoda `toArray` tworzy inną tablicę tego samego typu. Albo, jeśli tablica ma wystarczającą długość, zostaje wykorzystana ta tablica:

```
staff.toArray(new String[staff.size()]);
```

W tym przypadku nie zostaje utworzona żadna nowa tablica.

9.6.7. Pisanie własnych algorytmów

Pisząc własny algorytm (a raczej każdą metodę przyjmującą jako parametr kolekcję), należy, kiedy to tylko możliwe, wykorzystywać *interfejsy*, a nie konkretne implementacje. Wyobraźmy sobie na przykład, że chcemy przetworzyć pewne elementy. Oczywiście możemy zaimplementować taką metodę:

```
public void processItems(ArrayList<Item> items)
{
  for (Item item : items)
    operacje na elementach
}
```

Jednak w ten sposób ograniczamy zakres ruchu wywołującego metodę — elementy muszą być podane w postaci obiektu `ArrayList`. Jeśli zdarzy się, że elementy te będą w jakiejś innej kolekcji, konieczne będzie ich przepakowanie. Znacznie lepiej byłoby wykorzystać bardziej ogólną kolekcję.

Należy sobie zadać następujące pytanie: jaka jest najbardziej ogólna metoda, która przyda się w tym zastosowaniu? Czy zależy nam na kolejności? W takim razie należy użyć listy. Jeśli natomiast kolejność jest nieistotna, można użyć dowolnej kolekcji:

```
public void processItems(Collection<Item> items)
{
  for (Item item : items)
    operacje na elementach
}
```

Teraz metodę tę można wywołać przy użyciu struktury `ArrayList`, `LinkedList` bądź tablicy opakowanej w wywołanie metody `List.of`.

W tym przypadku lepszym rozwiązaniem jest akceptowanie `Iterable<Item>`. Interfejs `Iterable` zawiera jedną abstrakcyjną metodę `iterator`, której ulepszona pętla `for` używa w swoich wewnętrznych mechanizmach. Interfejs `Collection` rozszerza `Iterable`.

Z drugiej strony, jeśli metoda zwraca wiele elementów, nie należy ograniczać swoich możliwości w odniesieniu do przyszłych ulepszeń. Spójrz na poniższy przykład kodu:

```
public ArrayList<Item> lookupItems(. . .)
{
  var result = new ArrayList<Item>();
  . . .
  return result;
}
```

Ta metoda informuje, że zwraca `ArrayList`, mimo że dla wywołującego rodzaj listy prawie na pewno będzie bez znaczenia. Jeśli zamiast tego zwrócisz `List`, zawsze możesz dodać gałąź zwracającą pustą lub singletonową listę za pomocą wywołania metody `List.of`.

Skoro używanie interfejsów kolekcyjnych jako parametrów i typów zwrotnych jest takim dobrym rozwiązaniem, czemu nie jest ono stosowane konsekwentnie w API Javy? Na przykład klasa `JComboBox` ma dwa konstruktory:

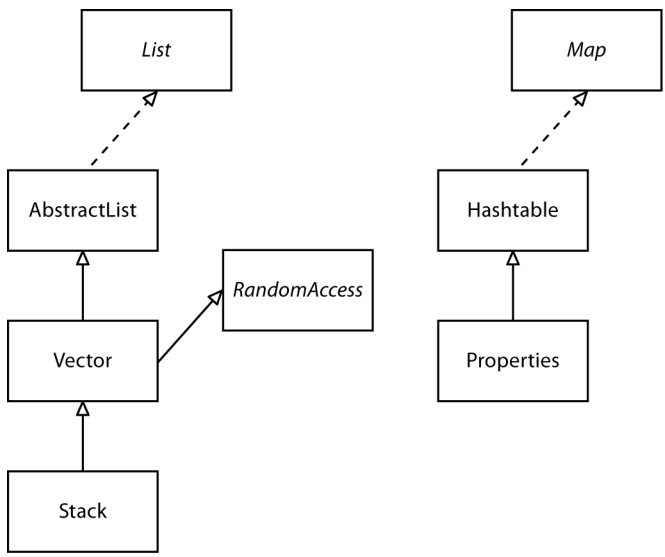
```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

Powodem jest po prostu czas. API `Swing` powstał przed Java Collections Framework.

9.7. Stare kolekcje

Od początku istnienia Javy w języku istnieje kilka „przestarzałych" klas kontenerowych, które zostały dodane jeszcze przed powstaniem Java Collections Framework.

Wszystkie je wcielono do tego systemu (zobacz rysunek 9.12), więc poniżej przedstawiamy ich zwięzły opis.



Rysunek 9.12. Stare klasy w ramowym systemie kolekcji

9.7.1. Klasa Hashtable

Standardowa metoda `Hashtable` pełni takie samo zadanie co `HashMap` i ma zasadniczo taki sam jak ona interfejs. Metody tej klasy, podobnie jak klasy `Vector`, są synchronizowane. Jeśli nie potrzebujesz zgodności ze starym kodem, używaj klasy `HashMap`. Jeśli potrzebna jest możliwość dostępu współbieżnego, należy skorzystać ze struktury `ConcurrentHashMap` — zobacz rozdział 10.

9.7.2. Wyliczenia

Stare kolekcje do przemierzania elementów ułożonych liniowo wykorzystują interfejs `Enume↪ration`. Znajdują się w nim dwie metody: `hasMoreElements` i `nextElement`. Są one wiernymi odpowiednikami metod `hasNext` i `next` z interfejsu `Iterator`.

Jeśli znajdziesz ten interfejs ze starymi klasami, to przy użyciu metody `Collections.list` możesz zbierać elementy w kolekcji `ArrayList`. Na przykład klasa `LogManager` ukazuje nazwy loggerów tylko w postaci obiektów typu `Enumeration`. Tak można pobrać je wszystkie:

```
ArrayList<String> loggerNames = Collections.list(LogManager.getLoggerNames());
```

Ewentualnie wyliczenie można zamienić w iterator:

```
LogManager.getLoggerNames().asIterator().forEachRemaining(n -> { . . . });
```

Czasami spotyka się stare metody wymagające wyliczenia jako parametru. Statyczna metoda `Collections.enumeration` tworzy obiekt wyliczeniowy zapełniony elementami pobranymi z kolekcji. Na przykład:

```
List<InputStream> streams = . . .;
var in = new SequenceInputStream(Collections.enumeration(streams));
// Konstruktor SequenceInputStream wymaga na wejściu typu wyliczeniowego.
```

W języku C++ jako parametry często stosuje się iteratory. Na szczęście w Javie niewielu programistów stosuje tę technikę. O wiele lepiej jest zamiast iteratora przekazać kolekcję. Kolekcja jest o wiele bardziej użyteczna. Odbiorca może zawsze w razie potrzeby wydobyć iterator z obiektu kolekcji, a poza tym ma do dyspozycji wszystkie metody tej kolekcji. W starszym kodzie można jednak spotkać wyliczenia, ponieważ były one jedynym sposobem uzyskania kolekcji generycznych przed wprowadzeniem Java Collections Framework w Javie 1.2.

`java.util.Enumeration<E>` **1.0**

- `boolean hasMoreElements()`

  Zwraca wartość `true`, jeśli są jeszcze jakieś elementy.

- `E nextElement()`

  Zwraca kolejny element do odwiedzenia. Nie należy wywoływać tej metody, jeśli metoda `hasMoreElements()` zwróci wartość `false`.

- `default Iterator<E> asIterator()` **9**

  Zwraca iterator przeglądający elementy wyliczenia.

`java.util.Collections` **1.2**

- `static <T> Enumeration<T> enumeration(Collection<T> c)`

  Zwraca wyliczenie zawierające elementy `c`.

- `public static <T> ArrayList<T> list(Enumeration<T> e)`

  Zwraca listę tablicową zawierającą elementy znajdujące się w `e`.

9.7.3. Słowniki własności

Słownik własności (ang. *property map*) to struktura danych o bardzo szczególnych cechach. Ma trzy wyróżniające ją właściwości:

- Klucze i wartości są łańcuchami.
- Słownik ten można łatwo zapisać w pliku i załadować z pliku.
- Istnieje dodatkowa tablica z wartościami domyślnymi.

Klasa Javy implementująca słownik własności nosi nazwę `Properties`. Słowniki własności są powszechnie wykorzystywane do określania opcji konfiguracyjnych programów, na przykład:

```
var settings = new Properties();
settings.setProperty("width", "600.0");
settings.setProperty("filename", "/home/cay/books/corejava/code/v1ch09/raven.html");
```

Za pomocą metody `getProperty` można wyszukać wartość odpowiadającą kluczowi.

```
String filename = settings.getProperty("filename");
```

Z powodów historycznych klasa `Properties` implementuje `Map<Object, Object>`. Dzięki temu można używać metod `get` i `put` interfejsu `Map`. Tylko że metoda `get` zwraca typ `Object`, a metoda `put` pozwala na wstawienie dowolnego obiektu. Lepiej jest trzymać się metod `getProperty` i `setProperty`, które działają na łańcuchach, nie obiektach.

Za pomocą metody `store` można zapisać własności w pliku. Drugi argument to komentarz, który zostanie dodany do pliku.

```
var out = new FileWriter("program.properties", StandardCharsets.UTF_8);
settings.store(out, "Właściwości programu");
```

Przykładowy zbiór daje następujący wynik:

```
#Program Properties
#Sun Dec 31 12:54:19 PST 2023
top=227.0
left=1286.0
```

```
width=423.0
height=547.0
filename=/home/cay/books/corejava/code/v1ch09/raven.html
```

Aby załadować własności z pliku, należy użyć następującej instrukcji:

```
var in = new FileReader("program.properties");
settings.load(in);
```

Gdy metody `load` i `store` są używane ze strumieniami wejścia i wyjścia, to stosowane jest archaiczne kodowanie znaków ISO-8859-1 i znaki > U+00FF są zarezerwowane jako symbole zastępcze Unicode. Aby włączyć kodowanie UTF-8, należy używać operacji odczytu i zapisu, jak w powyższych przykładach. Przed Javą 18 należy ręcznie ustawić kodowanie na `StandardCharsets.UTF_8`.

Klasa `Properties` ma dwa mechanizmy dostarczania wartości domyślnych. Po pierwsze, przy każdym wyszukiwaniu wartości łańcucha można określić wartość domyślną, która ma zostać użyta, jeśli dany klucz nie zostanie znaleziony.

```
String filename = settings.getProperty("filename", "");
```

Jeśli w słowniku własności znajduje się własność `"filename"`, zostanie ona ustawiona na określony łańcuch. W przeciwnym przypadku zostanie ustawiona na pusty łańcuch.

Jeśli określanie wartości domyślnej w każdym wywołaniu metody `getProperty` jest zbyt żmudne, wszystkie wartości domyślne można umieścić w dodatkowym słowniku własności i przekazywać go do konstruktora pierwszego słownika własności.

```
var defaultSettings = new Properties();
defaultSettings.setProperty("width", "600");
defaultSettings.setProperty("height", "400");
defaultSettings.setProperty("filename", "");
. . .
var settings = new Properties(defaultSettings);
```

Można nawet zdefiniować domyślne wartości dla domyślnych wartości, jeśli przekaże się inny argument słownika własności do konstruktora `defaultSettings`, ale zwykle nikt z tej możliwości nie korzysta.

W plikach z przykładowym kodem znajduje się program przedstawiający sposób użycia własności do przechowywania i ładowania stanu programu. Wykorzystuje on program ImageViewer z rozdziału 2. oraz zapamiętuje położenie ramki, rozmiar ramki i ostatnio załadowany plik. Uruchom program, załaduj plik, a następnie przesuń okno i zmień jego rozmiar. Potem zamknij program i otwórz go ponownie, aby przekonać się, że zapamiętał otwarty wcześniej plik oraz ustawienia okna. Można też ręcznie edytować plik *.corejava/ImageViewer.properties* znajdujący się w katalogu głównym.

Własności to proste tabele bez hierarchicznej struktury. Często imituje się w nich hierarchię poprzez wprowadzenie nazw kluczy, na przykład `window.main.color`, `window.main.title` itd. Tylko że klasa `Properties` nie zawiera żadnych metod, które mogłyby być pomocne w organizacji takich hierarchii. Dlatego do przechowywania skomplikowanych konfiguracji preferencji lepiej używać klasy `Preferences` — patrz rozdział 10 w tomie drugim.

`java.util.Properties` **1.0**

- `Properties()`

  Tworzy pustą mapę właściwości.

- `Properties(Properties defaults)`

  Tworzy pustą mapę właściwości z określonymi wartościami domyślnymi.

- `String getProperty(String key)`

  Pobiera właściwość. Zwraca łańcuch powiązany z kluczem (`key`) lub – jeśli go nie znajdzie w tabeli – łańcuch powiązany z kluczem w domyślnej tabeli albo `null`, jeśli klucza nie było także w domyślnej tabeli.

- `String getProperty(String key, String defaultValue)`

  Pobiera właściwość o domyślnej wartości, jeśli klucz (`key`) nie zostanie znaleziony. Zwraca łańcuch powiązany z kluczem lub domyślny łańcuch, jeśli nie było go w tabeli.

- `Object setProperty(String key, String value)`

  Ustawia właściwość. Zwraca poprzednio ustawioną wartość danego klucza.

- `Set<String> stringPropertyNames()` **6**

  Zwraca zbiór wszystkich kluczy, włącznie z kluczami z domyślnego słownika.

- `void load(Reader in) throws IOException` **6**

  Wczytuje słownik właściwości z czytnika.

- `void store(Writer out, String header)` **6**

  Zapisuje słownik właściwości w writerze. `header` znajduje się w pierwszym wierszu zapisanego pliku.

9.7.4. Właściwości systemowe

Metoda `System.getProperties` zwraca obiekt typu `Properties` zawierający informacje dotyczące systemu. Na przykład katalog główny ma klucz `"user.home"`.

Można go odczytać za pomocą metody `getProperties` zwracającej klucz jako łańcuch:

```
String userDir = System.getProperty("user.home");
```

Aby sprawdzić wersję Javy w maszynie wirtualnej, należy znaleźć właściwość `"java.version"`. W odpowiedzi zostanie zwrócony łańcuch w rodzaju `"21.0.1"` (albo `"1.8.0"` do Javy 8).

Jak widać, w Javie 9 zmieniono sposób numeracji wersji. Ta pozornie drobna zmiana narobiła kłopotów wielu narzędziom, które korzystały ze starego formatu. Dlatego, jeśli przetwarzasz łańcuch wersji, koniecznie przeczytaj dokument JEP 322 na stronie *http://openjdk.java.net/jeps/322*, aby dowiedzieć się, jak będą formatowane łańcuchy wersji w przyszłości — albo przynajmniej do następnej zmiany formatu numerowania.

Aby wydrukować wszystkie właściwości systemowe, wykonaj poniższe polecenie:

```
java -XshowSettings:properties
```

Właściwość systemowa `java.version` jest ustawiana przez maszynę wirtualną i nie należy jej zmieniać. Inne właściwości systemowe można zmieniać za pomocą opcji wiersza poleceń `-D`:

```
java -Duser.language=fr -Duser.country=CA MyProg
```

Aby uczynić właściwości dostępnymi w `jshell`, użyj opcji `-R` w następujący sposób:

```
jshell -R-Duser.language=fr -R-Duser.country=CA
```

Aby uzyskać liczbową lub logiczną wartość właściwości systemowej, użyj metod statycznych `Boolean.getBoolean()`, `Integer.getInteger` i `Long.getLong`:

```
Integer version = Integer.getInteger("java.specification.version");
```

`java.lang.System 1.0`

- `Properties getProperties()`

  Pobiera wszystkie własności systemowe. Aplikacja musi mieć uprawnienia do pobierania wszystkich własności albo zostanie zgłoszony wyjątek bezpieczeństwa.

- `String getProperty(String key)`

  Pobiera własność systemową o określonym kluczu. Poniższe własności zawsze są obecne:

```
java.version
java.version.date
java.vendor
java.vendor.url
java.home
java.class.path
java.library.path
java.class.version
java.specification.version
java.specification.vendor
java.specification.name
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.io.tmpdir
user.name
user.home
user.dir
```

```
native.encoding
stdout.encoding
stderr.encoding
```

`java.lang.Boolean` **1.0**

- `static boolean getBoolean(String name)`

  Zwraca `true`, jeśli właściwość systemowa o określonej nazwie ma wartość `"true"` (ignorując wielkość liter), i `false` w przeciwnym przypadku.

`java.lang.Integer` **1.0**

- `static Integer getInteger(String nm)`
- `static Integer getInteger(String nm, int val)`
- `static Integer getInteger(String nm, Integer val)`

  Zwraca wynik wywołania metody `Integer.decode` na właściwości systemowej o określonej nazwie. Jeśli nie ma właściwości systemowej o takiej nazwie lub jeśli jej wartości nie można rozkodować, metoda ta zwraca wartość domyślną albo – jeśli żadna nie jest podana – `null`.

`java.lang.Long` **1.0**

- `static Long getLong(String nm)`
- `static Long getLong(String nm, int val)`
- `static Long getLong(String nm, Long val)`

  Zwraca wynik wywołania metody `Long.decode` na właściwości systemowej o określonej nazwie. Jeśli nie ma właściwości systemowej o takiej nazwie lub jeśli jej wartości nie można rozkodować, metoda ta zwraca wartość domyślną albo – jeśli żadna nie jest podana – `null`.

9.7.5. Stosy

API Javy od wersji 1.0 zawiera klasę `Stack` udostępniającą powszechnie znane metody, takie jak `push` i `pop`. Klasa ta dziedziczy jednak po klasie `Vector`, która z teoretycznego punktu widzenia nie jest zadowalająca — pozwala na stosowanie takich niewłaściwych stosom metod jak `insert` i `remove`, wstawiających i usuwających wartości w dowolnym miejscu, nie tylko na wierzchu stosu.

`java.util.Stack<E>` **1.0**

- `E push(E item)`

  Wstawia element na stos i go zwraca.

- `E pop()`

  Usuwa i zwraca element znajdujący się na wierzchu stosu. Nie należy używać tej metody na pustym stosie.

- `E peek()`

  Zwraca element z wierzchu stosu, nie usuwając go. Nie należy wywoływać tej metody na pustym stosie.

9.7.6. Zbiory bitów

Klasa `BitSet` umożliwia przechowywanie szeregów bitowych zapakowanych w „słowa" zależne od implementacji, które obecnie są wartościami typu `long`. To o wiele efektywniejsze rozwiązanie niż użycie tablicy lub wartości logicznej.

Klasa `BitSet` udostępnia wygodny interfejs do odczytu, ustawiania oraz resetowania poszczególnych bitów. Interfejs ten pozwala uniknąć tworzenia masek bitowych i innych operacji na bitach, które są konieczne przy przechowywaniu bitów w zmiennych typu `int` lub `long`.

Na przykład dla zbioru `BitSet` o nazwie `bucketOfBits` poniższa instrukcja zwróci wartość `true`, jeśli *i*-ty bit jest włączony, lub `false` w przeciwnym przypadku.

```
bucketOfBits.get(i)
```

Podobnie poniższa instrukcja włącza *i*-ty bit.

```
bucketOfBits.set(i)
```

Z kolei poniższa instrukcja wyłącza *i*-ty bit.

```
bucketOfBits.clear(i)
```

Za pomocą obiektu typu `BitSet` można reprezentować zbiór nieujemnych liczb całkowitych o ograniczonej wartości maksymalnej. Ustaw i-ty bit, aby pokazać, że liczba `i` znajduje się w zbiorze. Jest to o wiele wydajniejsze od użycia `Set<Integer>`, jeżeli górna granica nie jest zbyt duża i zbiór ma wiele elementów.


Szablon `bitset` w C++ ma taką samą funkcjonalność jak klasa `BitSet` w Javie.

`java.util.BitSet` **1.0**

- `BitSet(int initialCapacity)`

  Tworzy zbiór bitów.

- `int cardinality()` **1.4**

  Zwraca liczbę ustawionych bitów lub, w przypadku traktowania ich jako zbioru liczb całkowitych, liczbę elementów

- `int length()` **1.2**

  Zwraca „logiczną długość" (1 plus indeks najwyższego ustawionego). Ta operacja może być używana do iteracji przez elementy.

- `int size()`

  Zwraca liczbę bitów obecnie dostępnych w wewnętrznej strukturze danych, nie liczbę elementów zbioru.

- `boolean get(int bit)`

  Pobiera bit.

- `void set(int bit)`

  Ustawia bit.

- `void clear(int bit)`

  Usuwa bit.

- `void and(BitSet set)`

  Tworzy sumę logiczną dwóch zbiorów bitów.

- `void or(BitSet set)`

  Tworzy alternatywę logiczną dwóch zbiorów.

- `void xor(BitSet set)`

  Wykonuje działanie logiczne XOR (lub wykluczające) na dwóch zbiorach bitów.

- `void andNot(BitSet set)`

  Usuwa wszystkie bity ze zbioru bitów, które są ustawione w innym zbiorze bitów.

- `IntStream stream()` **8**

  Zwraca strumień ustawionych wartości indeksów bitów, lub, w przypadku traktowania ich jako zbioru liczb całkowitych, strumień elementów

Zastosowanie zbiorów bitów zademonstrujemy za pomocą algorytmu do znajdowania liczb pierwszych, czyli sita Eratostenesa (liczba pierwsza to taka, która dzieli się bez reszty tylko przez siebie samą i jeden, a sito Eratostenesa jest pierwszą odkrytą metodą obliczania tych liczb). Nie jest to najlepszy algorytm do znajdowania liczb pierwszych, ale z niewyjaśnionych powodów zyskał dużą popularność jako test wydajności kompilatorów (nie jest to też dobry test wydajności, ponieważ testuje przede wszystkim operacje bitowe).

Jest to jednak ukłon w stronę tradycji — przedstawiamy implementację tego algorytmu. Prezentowany program zlicza wszystkie liczby pierwsze w zakresie od 2 do 2 000 000 (jest ich 148 933, a więc pewnie lepiej ich wszystkich nie drukować).

Nie zagłębiając się zbytnio w szczegóły implementacyjne tego programu, jego zadaniem jest przejście przez zbiór bitów zawierający dwa miliony elementów. Najpierw włączamy wszystkie bity. Następnie wyłączamy bity będące wielokrotnościami liczb, które wiadomo, że są pierwsze. Liczby odpowiadające bitom pozostałym po tym procesie są pierwsze. Listing 9.8 przedstawia implementację tego programu w języku Java, a listing 9.9 — w języku C++.

Mimo że sito nie jest dobrym testem wydajności, nie mogliśmy się oprzeć pokusie zmierzenia czasu obu implementacji algorytmu. Oto wyniki uzyskane na komputerze z procesorem i7-1165G7, 32 GB pamięci RAM i systemem Ubuntu 22.04:

- C++ (g++ 11.4.0): 70 milisekund,
- Java (Java 21): 20 milisekund.

Test ten przeprowadziłem w trzynastu wydaniach tej książki i w ostatnich dziewięciu Java bije C++ na głowę. Trzeba jednak ujawnić, że jeśli zoptymalizujemy kompilator C++, pobije on Javę, uzyskując czas 16 milisekund. Taki wynik w Javie można by było uzyskać tylko wtedy, gdyby program działał na tyle długo, aby włączył się kompilator JIT Hotspot.

Listing 9.8. sieve/Sieve.java

```
package sieve;

import java.util.*;

/**
 * Program wykonujący test porównawczy na bazie algorytmu sita Eratostenesa. Oblicza wszystkie
 * liczby pierwsze do 2 000 000.
 * @version 1.22 2021-06-17
 * @author Cay Horstmann
 */
public class Sieve
{
    public static void main(String[] s)
    {
        int n = 2000000;
        long start = System.nanoTime();
        var bitSet = new BitSet(n + 1);
        int i;
        for (i = 2; i <= n; i++)
            bitSet.set(i);
        i = 2;
        while (i * i <= n)
        {
            if (bitSet.get(i))
            {
                int k = i * i;
                while (k <= n)
                {
                    bitSet.clear(k);
                    k += i;
                }
            }
            i++;
        }
        long end = System.nanoTime();
        System.out.println(bitSet.cardinality() + " liczb pierwszych");
        System.out.println((end - start) / 1000 + " milisekund");
    }
}
```

Listing 9.9. Sieve.cpp

```
/**
  @version 1.22 2021-06-17
  @author Cay Horstmann
*/
#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    const int N = 2000000;
    clock_t cstart = clock();
    bitset<N + 1> b;
    int i;
    for (i = 2; i <= N; i++)
        b.set(i);
    i = 2;
    while (i * i <= N)
    {
    if (b.test(i))
    {
        int k = i * i;
        while (k <= N)
        {
            b.reset(k);
            k += i;
        }
    }
    i++;
}

clock_t cend = clock();
double millis = 1000.0 * (cend - cstart) / CLOCKS_PER_SEC;

cout << b.count() << " liczb pierwszych\n" << millis << " milisekund\n";

return 0;
}
```

Na tym kończy się nasza podróż po architekturze kolekcji w języku Java. Jak widać udostępnia ona bogaty zestaw klas kolekcyjnych, mających na celu zaspokajanie potrzeb programisty.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

JAVA:

doskonałość kodu zaczyna się od solidnych podstaw!

W 1995 roku rozpoczęła się nowa epoka programowania: świat poznał Javę. Dziś jest to potężny, dojrzały i wszechstronny język służący do pisania dużych systemów, małych programów, aplikacji mobilnych i internetowych. Java nieustannie się rozwija, a każda nowa wersja przynosi profesjonalnym programistom przemyślane funkcjonalności.

Ta książka jest kolejnym, zaktualizowanym i uzupełnionym wydaniem klasycznego podręcznika dla programistów, którzy szukają dokładnego opisu języka Java i jego platformy. Zawiera szczegółowe omówienie wszystkich jego składników, w tym najnowszych ulepszeń dodanych w wersji 21. W poszczególnych rozdziałach znajdują się przykłady kodu, które ilustrują najnowsze składniki obszernej biblioteki Javy — przystępne i praktyczne, stanowią świetny punkt wyjścia do pisania własnego kodu. W pierwszym tomie podręcznika znalazły się podstawowe zagadnienia związane z programowaniem w Javie, od programowania obiektowego, przez techniki refleksji i obiektów pośrednich, po wyrażenia lambda, adnotacje i system modułów platformy Java.

W książce między innymi:

- techniki, idiomy i najlepsze praktyki pisania kodu w języku Java
- interfejsy, klasy wewnętrzne i wyrażenia lambda
- zapieczętowane hierarchie klas i przetwarzanie danych strukturalnych
- obsługa wyjątków i skuteczne techniki debugowania
- korzystanie z typów generycznych i standardowych kolekcji Javy
- stosowanie modelu współbieżności Javy

Cay S. Horstmann jest profesorem informatyki na Uniwersytecie Stanowym w San José i autorem najpopularniejszych w Polsce podręczników do nauki Javy. Został wyróżniony tytułem Java Champion. Regularnie występuje na konferencjach programistycznych. Urodził się w północnej części Niemiec, obecnie mieszka w USA.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3023-0

9 788328 930230

Cena: 149,00 zł

Java™

Pearson