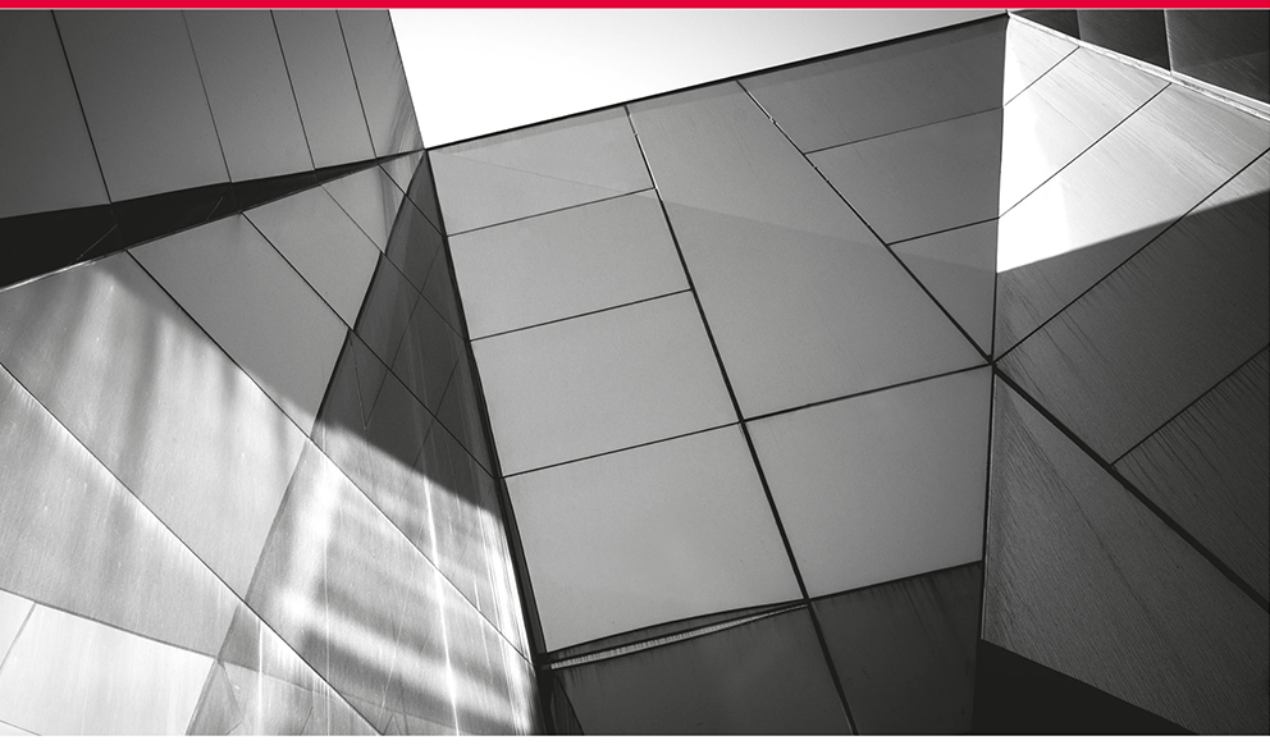
ORACLE



Java
Przewodnik dla początkujących

Wydanie VI

Herbert Schildt

Helion

Oracle Press™

Tytuł oryginału: Java™: A Beginner's Guide, Sixth Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-0808-4

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javpp6.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javpp6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ

8

Pakiety i interfejsy

W tym rozdziale poznasz:

- stosowanie pakietów,
- wpływ pakietów na poziom dostępu,
- modyfikator dostępu `protected`,
- importowanie pakietów,
- standardowe pakiety Javy,
- podstawy interfejsów,
- implementacje interfejsów,
- referencje interfejsów,
- zmienne interfejsów,
- dziedziczenie interfejsów,
- domyślne i statyczne metody interfejsów.

Ten rozdział poświęciłem na omówienie dwóch najbardziej innowacyjnych elementów Javy: pakietów i interfejsów. **Pakiety** są grupami powiązanych klas. Pakiety pomagają organizować kod i tworzą kolejną warstwę hermetyzacji. **Interfejs** definiuje zbiór metod, które zostaną zaimplementowane przez klasę. A zatem interfejs zapewnia możliwość określenia, co klasa będzie robić, lecz nie tego, jak to zrobi. Znajomość pakietów i interfejsów pozwoli Ci lepiej kontrolować organizację kodu.

Pakiety

W programowaniu pomocna jest możliwość grupowania powiązanych części programu. W Javie służą do tego pakiety. Pakiet spełnia dwa zadania. Po pierwsze, zapewnia mechanizm umożliwiający zorganizowanie powiązanych elementów programu. Dostęp do klas zdefiniowanych wewnątrz pakietu odbywa się z użyciem nazwy pakietu. W ten sposób pakiet dostarcza nazwy dla kolekcji klas. Po drugie, pakiet jest częścią mechanizmu kontroli dostępu stosowanego w Javie. Klasy zdefiniowane w pakiecie mogą być prywatne dla tego pakietu i tym samym niedostępne na zewnątrz pakietu. W ten sposób pakiet umożliwia hermetyzację klas. Przyjrzyjmy się bliżej każdej z wymienionych możliwości.

Tworząc klasę, przydzielamy jej nazwę należącą do **przestrzeni nazw**. Przestrzeń nazw definiuje region deklaracji. W Javie żadne dwie klasy nie mogą używać tej samej nazwy w tej samej przestrzeni nazw. Innymi słowy, nazwa klasy musi być unikatowa w danej przestrzeni nazw. Wszystkie dotychczasowe przykłady używały domyślnej (globalnej) przestrzeni nazw. Rozwiązanie takie sprawdza się w przypadku prostych programów, ale staje się problematyczne, gdy złożoność programu wzrasta

i w domyślnej przestrzeni nazw staje się tłoczno. W rozbudowanych programach nadanie klasie unikatowej nazwy może wtedy być problemem. Co gorsza, musisz również unikać kolizji nazw z kodem tworzonym przez innych programistów w ramach tego samego projektu, a także z bibliotekami Javy. Rozwiązanie tego problemu jest możliwe dzięki użyciu pakietu, gdyż umożliwia on podział przestrzeni nazw. Jeśli zdefiniujesz klasę w pakiecie, to do jej nazwy dołączasz nazwę pakietu i w ten sposób unikasz kolizji z innymi klasami o tej samej nazwie, ale należącymi do innych pakietów.

Ponieważ pakiety zawierają zwykle powiązane klasy, Java definiuje specjalne prawa dostępu do kodu umieszczonego w pakiecie. W pakiecie możesz zdefiniować kod, który jest dostępny dla pozostałego kodu w tym samym pakiecie, ale nie na zewnątrz pakietu. Pozwala to tworzyć samodzielne grupy powiązanych klas, których działanie pozostaje prywatne.

Definiowanie pakietu

Wszystkie klasy w Javie należą do jakiegoś pakietu. Jeśli nie użyjesz jawnie instrukcji `package`, to klasa należy do domyślnego (globalnego) pakietu. Pakiet domyślny nie ma nazwy i wobec tego jego użycie jest transparentne. Z tego powodu jak dotąd nie musiałeś się przejmować nazwami pakietów. Pakiet domyślny sprawdza się w przypadku prostych programów, ale nie w przypadku prawdziwych aplikacji. Tworząc te ostatnie, będziesz zwykle definiować jeden lub więcej pakietów.

Aby stworzyć pakiet, musisz umieścić instrukcję `package` na początku pliku źródłowego. Klasy definiowane w tym pliku będą wtedy należeć do tego pakietu. Ponieważ pakiet definiuje przestrzeń nazw, nazwy klas zdefiniowanych w tym pliku należą do przestrzeni nazw pakietu.

A oto ogólna postać instrukcji `package`:

```
package nazwa-pakietu;
```

Poniższa instrukcja tworzy pakiet o nazwie `mypack`:

```
package mypack;
```

Java używa systemu plików do zarządzania pakietami, umieszczając każdy pakiet we własnym katalogu. Na przykład pliki *.class* dla wszystkich klas zdefiniowanych w pakiecie `mypack` muszą zostać umieszczone w katalogu o nazwie `mypack`.

Podobnie jak dla innych elementów języka Java, również w przypadku nazw pakietów rozróżniane są małe i duże litery. Musisz zatem pamiętać, aby nazwa katalogu, w którym umieszczasz pliki klas, była dokładnie taka sama jak nazwa pakietu. Jeśli podczas uruchamiania przykładów zamieszczonych w tym rozdziale napotkasz problemy, sprawdź dokładnie nazwy katalogów i pakietów. Z tego powodu programiści zwykle używają jedynie małych liter w nazwach pakietów.

Tę samą instrukcję `package` możesz umieścić w więcej niż jednym pliku. Instrukcja `package` określa po prostu, do jakiego pakietu należą klasy zdefiniowane w pliku. Nie wyklucza zatem możliwości, aby inne klasy w innych plikach należały do tego samego pakietu. I rzeczywiście, w praktyce większość pakietów zwykle składa się z wielu plików.

Istnieje możliwość tworzenia hierarchii pakietów. W tym celu nazwy pakietów kolejnych poziomów oddzielamy od siebie kropką. Ogólna postać instrukcji `package` dotyczącej wielopoziomowej hierarchii pakietów jest następująca:

```
package pakiet1.pakiet2.pakiet3...pakietN;
```

W takim przypadku musisz zadbać o stworzenie odpowiedniej struktury katalogów. Na przykład klasy pakietu

```
package alpha.beta.gamma;
```

muszą zostać umieszczone w katalogu *.../alpha/beta/gamma*, gdzie ... oznacza ścieżkę dostępu do tego katalogu.

Wyszukiwanie pakietów i zmienna CLASSPATH

Jak już wyjaśniłem, pakietom odpowiadają katalogi. Powstaje zatem pytanie: skąd Java „wie", gdzie szukać utworzonych przez Ciebie pakietów? Odpowiedź na to pytanie składa się z trzech części. Po pierwsze, Java używa jako punktu wyjścia bieżącego katalogu. Zatem jeśli umieściłeś pakiet w pod-

katalogu bieżącego katalogu, zostanie on odnaleziony. Po drugie, możesz określić ścieżkę dostępu do katalogu za pomocą zmiennej środowiskowej CLASSPATH. Po trzecie, możesz użyć w tym samym celu opcji -classpath, wywołując programy javac i java.

Na przykład zakładając poniższą specyfikację pakietu:

```
package mypack;
```

aby pakiet mypack został odnaleziony, musi zostać spełniony jeden z następujących warunków: program musi być uruchamiany z katalogu znajdującego się bezpośrednio nad katalogiem *mypack*, zmienna CLASSPATH musi zawierać ścieżkę dostępu do katalogu *mypack* lub podczas uruchamiania interpretera java musisz określić ścieżkę dostępu do katalogu *mypack*, używając opcji -classpath.

Najprostszy sposób uruchomienia przykładów zamieszczonych w tej książce polega na utworzeniu podkatalogów dla pakietów poniżej bieżącego katalogu, umieszczeniu plików *.class* w odpowiednich podkatalogach i uruchamianiu programów z katalogu bieżącego. Rozwiązanie to zastosuję w kolejnych przykładach.

I jeszcze jedno: aby uniknąć problemów, najlepiej przechowywać wszystkie pliki *.java* i *.class* związane z danym pakietem w jego podkatalogu. A także kompilować każdy plik z poziomu katalogu bieżącego znajdującego się nad podkatalogiem pakietu.

Prosty przykład pakietu

Spróbujmy zastosować w praktyce informacje z poprzedniego podrozdziału. Przykład przedstawiony na listingu 8.1 tworzy prostą bazę danych zawierającą informacje o książkach, która należy do pakietu bookpack.

Listing 8.1. *BookDemo.java*

```
// Demonstracja prostego pakietu.
package bookpack; <-------- Plik należy do pakietu bookpack.

class Book { <-------- Zatem klasa Book również należy do bookpack.
  private String title;
  private String author;
  private int pubDate;

  Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
  }

  void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}

class BookDemo { <-------- Klasa BookDemo również należy do bookpack.
  public static void main(String args[]) {
    Book books[] = new Book[5];

    books[0] = new Book("Java. Przewodnik dla początkujących",
                        "Schildt", 2015);
    books[1] = new Book("Java. Kompendium programisty",
                        "Schildt", 2012);
    books[2] = new Book("Pan Tadeusz",
                        "Mickiewicz", 1974);
    books[3] = new Book("Stan zagrożenia",
                         "Clancy", 2003);
    books[4] = new Book("Ogniem i mieczem",
```

```
                        "Sienkiewicz", 1995);

    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

Nazwij plik źródłowy *BookDemo.java* i umieść go w katalogu *bookpack*.

Następnie skompiluj program. W tym celu możesz wywołać

```
javac bookpack/BookDemo.java
```

z poziomu katalogu znajdującego się bezpośrednio nad katalogiem *bookpack*. Spróbuj uruchomić program za pomocą poniższego polecenia:

```
java -Dfile.encoding=CP852 bookpack.BookDemo
```

Pamiętaj, że podczas uruchamiania programu katalogiem bieżącym powinien być katalog bezpośrednio nad katalogiem *bookpack* (lub użyj jednej z dwóch możliwości specyfikacji dostępu do katalogu *bookpack* omówionych w poprzednim podrozdziale).

Klasy `BookDemo` i `Book` należą teraz do pakietu `bookpack`. Oznacza to, że nie możesz teraz wykonać kodu klasy `BookDemo`. Poniższe polecenie nie powiedzie się:

```
java -Dfile.encoding=CP852 BookDemo
```

Odwołując się do klasy `BookDemo`, musisz podać jej pełną nazwę wraz z nazwą pakietu.

Pakiety i dostęp do składowych

We wcześniejszych rozdziałach omówiłem podstawy kontroli dostępu w Javie, w tym modyfikatory `private` i `public`. Nie wyczerpuje to jednak zagadnienia. Pakiety również uczestniczą w kontroli dostępu i dlatego kompletne omówienie zagadnienia musiało zaczekać, aż przedstawię pakiety.

Dostępność elementu jest określona przez specyfikację dostępu, domyślną, `private`, `public` lub `protected`. A także przez pakiet, do którego należy element. Zatem dostępność elementu jest określona przez jego dostępność wewnątrz klasy i dostępność wewnątrz pakietu. Takie wielowarstwowe rozwiązanie kontroli dostępu umożliwia wprowadzenie bogatego zbioru uprawnień dostępu. Różne poziomy dostępu przedstawiłem w tabeli 8.1. Przeanalizujmy każdą z opcji dostępu z osobna.

Jeśli składowa klasy nie ma jawnego modyfikatora dostępu, to jest dostępna wewnątrz swojego pakietu, ale nie na zewnątrz pakietu. Dlatego domyślny poziom dostępu stosujemy dla elementów, które mają być publicznie dostępne wewnątrz pakietu, ale pozostawać prywatne dla tego pakietu.

Składowe zadeklarowane jawnie jako `public` (nazywane także składowymi publicznymi) są dostępne wszędzie, w różnych klasach i różnych pakietach. Nie ma żadnych restrykcji związanych z dostępem do tych składowych i ich użyciem. Składowa zadeklarowana jako `private` (nazywana także składową prywatną) jest dostępna wyłącznie dla innych składowych tej samej klasy. Zatem przynależność do pakietu nie ma żadnego wpływu na dostęp do składowej prywatnej. Składowa zadeklarowana jako `protected` (nazywana także składową chronioną) jest dostępna wewnątrz swojego pakietu oraz dla wszystkich klas pochodnych, z klasami pochodnymi należącymi do innych pakietów włącznie.

Informacje podane w tabeli 8.1 dotyczą tylko składowych klas. W przypadku klasy istnieją tylko dwa poziomy dostępu: domyślny i `public`. Klasa zadeklarowana jako `public` jest dostępna dla dowolnego kodu. Jeśli natomiast ma domyślny poziom dostępu, jest dostępna jedynie dla kodu wewnątrz tego samego pakietu. Klasa zadeklarowana jako `public` musi rezydować w pliku o takiej samej nazwie jak nazwa klasy.

Przykład dostępu do pakietu

W poprzednim przykładzie klasy `Book` i `BookDemo` zostały umieszczone w tym samym pakiecie, a zatem klasa `BookDemo` mogła używać klasy `Book`, ponieważ domyślny poziom dostępu umożliwia swobodny dostęp wewnątrz pakietu. Sytuacja zmieniłaby się, gdyby klasy należały do różnych pakietów. W takim przypadku klasa `BookDemo` straciłaby dostęp do klasy `Book`. Aby udostępnić klasę `Book` innym pakietom,

Tabela 8.1. *Dostępność składowych klasy*

	Składowa prywatna	Składowa o domyślnym poziomie dostępu	Składowa chronina	Składowa publiczna
Dostępna w tej samej klasie	tak	tak	tak	tak
Dostępna w tym samym pakiecie dla klasy pochodnej	nie	tak	tak	tak
Dostępna w tym samym pakiecie dla klasy, która nie jest pochodną jej klasy	nie	tak	tak	tak
Dostępna w innym pakiecie dla klasy pochodnej	nie	nie	tak	tak
Dostępna w innym pakiecie dla klasy, która nie jest pochodną jej klasy	nie	nie	nie	tak

musisz wykonać trzy zmiany. Po pierwsze, zadeklarować klasę `Book` jako `public`. Dzięki temu klasa `Book` będzie dostępna na zewnątrz pakietu. Po drugie, jej konstruktor również musi być zadeklarowany jako `public`. I po trzecie, tak samo musisz postąpić z metodą `show()`. W ten sposób konstruktor i metoda `show()` będą dostępne również na zewnątrz pakietu. Zatem aby umożliwić wykorzystanie klasy `Book` przez inne pakiety, należy zdefiniować ją w sposób przedstawiony na listingu 8.2.

Listing 8.2. *Book.java*

```
// Klasa Book upubliczniona dla innych pakietów.
package bookpack;

public class Book {
  private String title;
  private String author;
  private int pubDate;

  // Teraz zadeklarowany jako public.
  public Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
  }

  // Teraz zadeklarowana jako public.
  public void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}
```

Klasa `Book` i jej składowe muszą być zadeklarowane jako `public`, aby mogły zostać użyte w innych pakietach.

Aby użyć klasy `Book` w innym pakiecie, musisz użyć instrukcji `import` omawianej w następnym rozdziale lub podać nazwę klasy wraz z pełną nazwą pakietu, do którego należy. Na listingu 8.3 przedstawiłem klasę `UseBook` należącą do pakietu `bookpackext`. Używa ona klasy `Book`, podając jej nazwę wraz z nazwą pakietu.

Listing 8.3. *Usebook.java*

```
// Ta klasa należy do pakietu bookpackext.
package bookpackext;

// Używa klasy Book należącej do pakietu bookpack.
```

```
class UseBook {
  public static void main(String args[]) {
    bookpack.Book books[] = new bookpack.Book[5];
    books[0] = new bookpack.Book("Java. Przewodnik dla początkujących",
                        "Schildt", 2015);
    books[1] = new bookpack.Book("Java. Kompendium programisty",
                        "Schildt", 2012);
    books[2] = new bookpack.Book("Pan Tadeusz",
                        "Mickiewicz", 1974);
    books[3] = new bookpack.Book("Stan zagrożenia",
                         "Clancy", 2003);
    books[4] = new bookpack.Book("Ogniem i mieczem",
                        "Sienkiewicz", 1995);

    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

Odwołanie do klasy Book musisz poprzedzić nazwą pakietu: bookpack.

Zwróć uwagę, że wszystkie odwołania do klasy `Book` zostały poprzedzone nazwą pakietu `bookpack`. Jeśli tego nie zrobisz, klasa `Book` nie zostanie odnaleziona podczas próby kompilacji klasy `UseBook`.

Składowe chronione

Rozpoczynającym przygodę z językiem Java zrozumienie znaczenia i zastosowania modyfikatora `protected` często sprawia kłopoty. Jak już wyjaśniłem, składowa chroniona, czyli składowa zadeklarowana jako `protected`, jest dostępna w swoim pakiecie oraz we wszystkich klasach pochodnych należących do innych pakietów. Zatem składowa chroniona jest dostępna dla wszystkich klas pochodnych, ale nie jest dostępna dla dowolnego kodu poza pakietem.

Aby lepiej wyjaśnić efekt zastosowanie modyfikatora `protected`, posłużę się przykładem. Najpierw zmodyfikujemy klasę `Book` w taki sposób, aby jej zmienne składowe były zadeklarowane jako chronione. Zmodyfikowaną wersję klasy `Book` przedstawiłem na listingu 8.4.

Listing 8.4. *Book.java*

```
// Zmienne składowe klasy Book zadeklarowane jako chronione.
package bookpack;

public class Book {
  // teraz zadeklarowane jako protected
  protected String title;
  protected String author;
  protected int pubDate;

  public Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
  }

   public void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}
```

Te składowe są teraz zadeklarowane jako chronione.

Następnie utwórz klasę pochodną klasy `Book` o nazwie `ExtBook` i klasę o nazwie `ProtectDemo`, która używa klasy `ExtBook`. Klasa `ExtBook` wprowadza dodatkową składową przechowującą nazwę wydawcy

i kilka metod dostępowych. Obie klasy umieścisz we własnym pakiecie o nazwie `bookpackext`. Przedstawiłem je na listingu 8.5.

Listing 8.5. *ProtectDemo.java*

```
// Demonstruje użycie modyfikatora protected.
package bookpackext;

class ExtBook extends bookpack.Book {
  private String publisher;

  public ExtBook(String t, String a, int d, String p) {
    super(t, a, d);
    publisher = p;
  }

  public void show() {
    super.show();
    System.out.println(publisher);
    System.out.println();
  }

  public String getPublisher() { return publisher; }
  public void setPublisher(String p) { publisher = p; }

  /* Poniższe metody są poprawne, gdyż klasa
     pochodna ma dostęp do składowych chronionych. */
  public String getTitle() { return title; }
  public void setTitle(String t) { title = t; }
  public String getAuthor() { return author; }
  public void setAuthor(String a) { author = a; }
  public int getPubDate() { return pubDate; }
  public void setPubDate(int d) { pubDate = d; }
}

class ProtectDemo {
  public static void main(String args[]) {
    ExtBook books[] = new ExtBook[5];

    books[0] = new ExtBook("Java. Przewodnik dla początkujących",
                           "Schildt", 2015, "Helion");
    books[1] = new ExtBook("Java. Kompendium programisty",
                           "Schildt", 2012, "Helion");
    books[2] = new ExtBook("Pan Tadeusz",
                           "Mickiewicz", 1974, "PIW");
    books[3] = new ExtBook("Stan zagrożenia",
                            "Clancy", 2003, "Amber");
    books[4] = new ExtBook("Ogniem i mieczem",
                           "Sienkiewicz", 1995, "PIW");

    for(int i=0; i < books.length; i++) books[i].show();

    // Wyszukuje książki konkretnego autora
    System.out.println("Wszystkie tytuły, których autorem jest Schildt.");
    for(int i=0; i < books.length; i++)
      if(books[i].getAuthor() == "Schildt")
        System.out.println(books[i].getTitle());

//    books[0].title = "test title"; // Błąd -- składowa nie jest tutaj dostępna
  }
}
```

Dostęp do składowych klasy `Book` jest dozwolony w klasach pochodnych.

Dostęp do składowej `protected` nie jest możliwy w klasie, która nie jest pochodną.

Najpierw przyjrzyj się klasie `ExtBook`. Ponieważ jest ona klasą pochodną klasy `Book`, ma dostęp do chronionych składowych klasy `Book`, mimo że należy do innego pakietu. Może zatem bezpośrednio używać składowych `title`, `author` i `pubDate`, implementując metody dostępowe dla tych zmiennych. Jednak w klasie `ProtectDemo` dostęp do tych zmiennych nie jest możliwy, ponieważ `ProtectDemo` nie jest klasą pochodną klasy `Book`. Jeśli usuniesz znak komentarza rozpoczynający poniższy wiersz, to kompilacja programu zakończy się błędem.

```
//     books[0].title = "test title"; // Błąd -- składowa nie jest tutaj dostępna
```

Import pakietów

W poprzednich przykładach pokazałem, że używając klas z innego pakietu, możesz poprzedzać ich nazwy pełną nazwą pakietu. Jednak takie rozwiązanie szybko staje się męczące i mało eleganckie, zwłaszcza gdy klasy należą do pakietu umieszczonego głęboko w hierarchii. Skoro Javę wymyślili programiści dla programistów, a programiści nie lubią pracochłonnych konstrukcji, nie będzie niespodzianką, że istnieje wygodniejszy sposób korzystania z zawartości pakietów: instrukcja `import`. Stosując tę instrukcję, możesz udostępnić jeden lub więcej elementów pakietu. Wtedy możesz ich używać bez podawania pełnych nazw pakietów.

Ekspert odpowiada

Pytanie: Również w języku C++ istnieje słowo kluczowe protected. Czy jego zastosowanie jest podobne jak w Javie?

Odpowiedź: Podobne, ale nie takie samo. W języku C++ składowa chroniona jest dostępna w klasach pochodnych, ale poza tym jest prywatna. W języku Java składowa chroniona jest dostępna dla dowolnego kodu w tym samym pakiecie, ale tylko dla klas pochodnych na zewnątrz pakietu. Zatem przepisując kod z języka C++ na Javę, musisz zwrócić uwagę na tę różnicę.

Ogólna postać instrukcji `import` jest następująca:

```
import pakiet.nazwaklasy
```

W tym przypadku *`pakiet`* jest pełną nazwą pakietu, a *`nazwaklasy`* dotyczy importowanej klasy. Jeśli chcesz importować całą zawartość pakietu, zamiast nazwy klasy używasz gwiazdki. Oto przykłady obu postaci instrukcji `import`:

```
import mypack.MyClass
import mypack.*;
```

W pierwszym przykładzie klasa `MyClass` zostaje importowana z pakietu `mypack`. Drugi przykład oznacza import wszystkich klas należących do pakietu `mypack`. W pliku źródłowym Java instrukcja `import` musi znaleźć się zaraz po instrukcji `package` (jeśli taka istnieje) i przed definicją jakiejkolwiek klasy.

Instrukcji `import` możemy użyć, aby udostępnić pakiet `bookpack` i używać klasy `Book` bez poprzedzania jej nazwą pakietu. Wystarczy w tym celu dodać poniższą instrukcję `import` na początku każdego pliku, który używa klasy `Book`.

```
import bookpack.*;
```

Na listingu 8.6 przedstawiłem wersję klasy `UseBook` wykorzystującą instrukcję `import`.

Listing 8.6. *UseBook2.java*

```
// Demonstruje instrukcję import.
package bookpackext;
import bookpack.*; <-- Importuje cały pakiet bookpack.

// Używa klasy Book należącej do pakietu bookpack.
class UseBook2 {
  public static void main(String args[]) {
    Book books[] = new Book[5]; <-- Teraz możesz odwoływać się do klasy Book bezpośrednio,
                                    bez nazwy pakietu.
```

```
    books[0] = new Book("Java. Przewodnik dla początkujących",
                        "Schildt", 2015);
    books[1] = new Book("Java. Kompendium programisty",
                        "Schildt", 2012);
    books[2] = new Book("Pan Tadeusz",
                        "Mickiewicz", 1974);
    books[3] = new Book("Stan zagrożenia",
                         "Clancy", 2003);
    books[4] = new Book("Ogniem i mieczem",
                        "Sienkiewicz", 1995);
    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

Zwróć uwagę, że nazwa pakietu nie jest już używana dla klasy `Book`.

Biblioteka klas Java używa pakietów

Jak już wspomniałem w tej książce, Java definiuje wiele klas standardowych dostępnych dla wszystkich programów. Tę bibliotekę klas często określa się mianem Java API (*Application Programming Interface*). Klasy Java API zostały umieszczone w pakietach. Na szczycie ich hierarchii umieszczono pakiet `java`. Poniżej znajduje się wiele innych pakietów; kilka z nich przedstawiłem w tabeli 8.2.

Tabela 8.2. *Pakiety Java API*

Pakiet	Opis
`java.lang`	Zawiera wiele klas ogólnego przeznaczenia
`java.io`	Zawiera klasy obsługi wejścia i wyjścia
`java.net`	Zawiera klasy obsługi sieci
`java.applet`	Zawiera klasy umożliwiające tworzenie apletów
`java.awt`	Zawiera klasy tzw. Abstract Window Toolkit

Od początku tej książki korzystasz już z pakietu `java.lang`. Zawiera on między innymi klasę `System`, której używałeś, wyświetlając informacje za pomocą metody `println()`. Pakiet `java.lang` jest unikatowy pod tym względem, że jest automatycznie importowany do każdego programu w języku Java. Dlatego w dotychczasowych przykładach nie musiałeś używać instrukcji `import java.lang`. Pozostałe pakiety wymagają jednak jawnego użycia instrukcji `import`. Kilka z nich poznasz bliżej w kolejnych rozdziałach.

Interfejsy

W programowaniu obiektowym często pomocna jest możliwość zdefiniowania operacji, które musi wykonywać klasa, jednak bez określania szczegółów ich implementacji. Poznałeś już jeden przykład takiego rozwiązania: metodę abstrakcyjną. Metoda abstrakcyjna definiuje sygnaturę metody, ale nie dostarcza jej implementacji. Klasy pochodne muszą dostarczyć implementacji każdej metody abstrakcyjnej zdefiniowanej w klasie bazowej. Innymi słowy, metoda abstrakcyjna definiuje *interfejs*, ale nie *implementację*. Mimo że klasy abstrakcyjne i metody abstrakcyjne są przydatne, w Javie zdecydowano rozwinąć tę koncepcję o krok dalej. Udostępniono możliwość zupełnego oddzielenia interfejsu klasy od jej implementacji poprzez zastosowanie słowa kluczowego `interface`.

Pod względem składni interfejsy przypominają nieco klasy abstrakcyjne, gdyż także w nich można podać jedną lub więcej klas, które nie mają ciała. Aby czynności wykonywane przez te metody zostały zdefiniowane, wszystkie metody interfejsu należy zaimplementować. A zatem interfejsy określają, co musi być zrobione, ale nie określają, jak te czynności należy wykonać. Po zdefiniowaniu interfejs może być implementowany przez dowolną liczbę klas. Jedna klasa może natomiast implementować dowolnie wiele interfejsów.

Implementacja interfejsu oznacza, że klasa musi dostarczyć ciał (implementacji) dla metod opisanych przez interfejs. Szczegóły implementacji są wewnętrzną sprawą klasy. Dwie klasy mogą implementować ten sam interfejs w różny sposób, ale każda z nich musi obsługiwać ten sam zbiór metod. Dzięki temu kod może za pomocą tego samego interfejsu używać obiektów obydwu klas. Udostępniając słowo kluczowe `interface`, Java umożliwia pełne wykorzystanie polimorfizmu rozumianego jako „jeden interfejs, wiele metod".

Zanim przejdziemy do dalszego poznawania interfejsów, muszę zwrócić uwagę na jedną ważną informację. Otóż w JKD 8 wprowadzono zmianę w interfejsach, która znacząco wpływa na ich możliwości. We wcześniejszych wersjach języka interfejsy nie mogły definiować żadnych implementacji. A zatem do momentu wprowadzenia JDK 8 interfejsy mogły określać jedynie to, co klasa mogła robić, lecz nie określały, jak to robiła. Jednak w JDK 8 ta sytuacja uległa zmianie. Obecnie istnieje już możliwość dodawania do interfejsu metody **implementacji domyślnej**. Oznacza to, że teraz interfejsy mogą już określać pewne zachowania. Niemniej jednak te domyślne metody stanowią w zasadzie możliwość specjalnego przeznaczenia i bynajmniej nie przekreślają dotychczasowego charakteru i znaczenia interfejsów. Dlatego jako zasadę ogólną należy przyjąć, że tak jak było dotychczas, przeważnie będziemy tworzyć i stosować interfejsy pozbawione metod domyślnych. Dlatego też zacznę od przedstawienia interfejsów w ich tradycyjnej postaci, a metodę domyślną interfejsów opiszę pod koniec rozdziału.

Poniżej przedstawiłem uproszczoną ogólną postać tradycyjnej deklaracji interfejsu:

```
dostęp interface nazwa{
typ-zwracany nazwa-metody1(lista-parametrów);
typ-zwracany nazwa-metody2(lista-parametrów);
typ zmienna1 = wartość;
typ zmienna2 = wartość;
//...
typ-zwracany nazwa-metodyN(lista-parametrów);
typ zmiennaN = wartość;
}
```

W tym przypadku *dostęp* oznacza modyfikator `public` albo nie jest używany. W tym drugim przypadku zostaje zastosowany domyślny poziom dostępu i interfejs jest dostępny jedynie wewnątrz pakietu. Jeśli zadeklarujemy interfejs jako `public`, może być on używany przez dowolny kod (pamiętaj jednak, że interfejs zadeklarowany jako `public` musi znajdować się w pliku o nazwie takiej samej jak nazwa interfejsu). *Nazwa-interfejsu* może być dowolnym, poprawnym identyfikatorem.

W przypadku interfejsów o tradycyjnej postaci w metodach deklarujemy jedynie typ zwracany i sygnaturę. Metody interfejsu są metodami abstrakcyjnymi. Każda klasa implementująca interfejs musi implementować wszystkie jego metody. Metody interfejsu są niejawnie zadeklarowane jako `public`.

Zmienne interfejsu nie są zmiennymi instancji. Niejawnie są zadeklarowane jako `public`, `final` i `static` i wymagają inicjalizacji. W efekcie spełniają jedynie rolę stałych. Na listingu 8.7 przedstawiłem przykład interfejsu. Specyfikuje on interfejs klasy generującej sekwencję liczb.

Listing 8.7. *Series.java*

```
public interface Series {
  int getNext(); // zwraca następny element sekwencji
  void reset(); // restart
  void setStart(int x); // określa wartość początkową
}
```

Interfejs ten został zadeklarowany jako `public` i wobec tego może zostać zaimplementowany w dowolnym pakiecie.

Implementacje interfejsów

Po zdefiniowaniu interfejsu mogą go implementować różne klasy. Aby zaimplementować interfejs, umieszczamy w definicji klasy klauzulę `implements`, a następnie tworzymy implementację metod wymaganych przez interfejs. Ogólna postać deklaracji klasy zawierającej klauzulę `implements` przedstawia się następująco:

```
class nazwaklasy extends klasabazowa implements interfejs {
  // ciało-klasy
}
```

Jeśli klasa ma stanowić implementację więcej niż jednego interfejsu, to nazwy interfejsów rozdziela się przecinkami. Oczywiście klauzula `extends` jest opcjonalna.

Metody implementujące interfejs muszą zostać zadeklarowane jako `public`. Ich sygnatury muszą dokładnie odpowiadać sygnaturom metod, które zawiera definicja interfejsu.

Na listingu 8.8 przedstawiłem przykład klasy implementującej przedstawiony wcześniej interfejs `Series`. Klasa `ByTwos` generuje serię liczb, w której kolejna liczba jest większa o 2 od poprzedniej.

Listing 8.8. *ByTwos.java*

```
// Implementuje interfejs Series
class ByTwos implements Series {
  int start;
  int val;

  ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

Implementuje interfejs `Series`.

Zwróć uwagę, że metody `getNext()`, `reset()` i `setStart()` zostały zadeklarowane jako `public`. Jest to obowiązkowe. Gdy implementujesz metodę interfejsu, musisz zadeklarować ją jako `public`, gdyż wszystkie składowe interfejsu są niejawnie zadeklarowane jako `public`.

Na listingu 8.9 przedstawiłem klasę demonstrującą działanie klasy `ByTwos`.

Listing 8.9. *SeriesDemo.java*

```
class SeriesDemo {
  public static void main(String args[]) {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      System.out.println("Następna wartość: " +
                         ob.getNext());

    System.out.println("\nRestart");
    ob.reset();
```

```
    for(int i=0; i < 5; i++)
      System.out.println("Następna wartość: " +
                         ob.getNext());

    System.out.println("\nRozpoczynam od 100");
    ob.setStart(100);
    for(int i=0; i < 5; i++)
      System.out.println("Następna wartość: " +
                         ob.getNext());
  }
}
```

A oto wynik działania tego programu:

```
Następna wartość: 2
Następna wartość: 4
Następna wartość: 6
Następna wartość: 8
Następna wartość: 10

Restart
Następna wartość: 2
Następna wartość: 4
Następna wartość: 6
Następna wartość: 8
Następna wartość: 10

Rozpoczynam od 100
Następna wartość: 102
Następna wartość: 104
Następna wartość: 106
Następna wartość: 108
Następna wartość: 110
```

Klasy implementujące interfejsy mogą również definiować dodatkowe, własne składowe. Na przykład wersja klasy `ByTwos` przedstawiona na listingu 8.10 dodaje własną metodę `getPrevious()` zwracającą poprzednią wartość.

Listing 8.10. *ByTwos.java*

```
// Implementuje interfejs Series i dodaje metodę getPrevious().
class ByTwos implements Series {
  int start;
  int val;
  int prev;
  ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
    prev = start-2;
  }

  public void setStart(int x) {
    start = x;
```

```
    val = x;
    prev = x - 2;
  }

  int getPrevious() {  <-- Dodaje metodę, której nie definiuje interfejs Series.
    return prev;
  }
}
```

Zwróć uwagę, że wprowadzenie metody `getPrevious()` wymagało zmian w implementacji metod zdefiniowanych przez interfejs `Series`. Ponieważ jednak interfejs metod pozostał taki sam, zmiany nie mają żadnego wpływu na działanie programu demonstrującego klasę `ByTwos`. Jest to jedna z zalet stosowania interfejsów.

Jak już wspomniałem, interfejs może implementować dowolna liczba klas. Na listingu 8.11 przedstawiłem klasę `ByThrees` generującą sekwencję wielokrotności liczby 3.

Listing 8.11. *ByThrees.java*

```
// Implementuje interfejs Series
class ByThrees implements Series {  <-- Implementuje interfejs Series w inny sposób.
  int start;
  int val;

  ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

I jeszcze jedno: jeśli zadeklarowałeś, że klasa implementuje interfejs, ale nie dostarczyłeś pełnej implementacji wszystkich metod zdefiniowanych przez interfejs, to musisz zadeklarować tę klasę jako `abstract`. Oczywiście nie możesz tworzyć obiektów takiej klasy, ale możesz użyć jej jako abstrakcyjnej klasy bazowej, pozostawiając jej klasom pochodnym dostarczenie kompletnej implementacji.

Referencje interfejsu

Możesz być nieco zaskoczony możliwością zadeklarowania zmiennej, której typem jest interfejs. Innymi słowy, tworzysz w ten sposób zmienną referencyjną interfejsu. Możesz jej użyć, aby odwołać się do dowolnego obiektu, który implementuje ten interfejs. Gdy wywołujesz metodę dla obiektu, do którego odwołujesz się za pomocą referencji interfejsu, wykonana zostanie wersja metody implementowana przez ten konkretny obiekt. Proces ten jest podobny do stosowania referencji klasy bazowej do obiektów klas pochodnych, co omówiłem w rozdziale 7.

Przykład użycia referencji interfejsu przedstawiłem na listingu 8.12. Używa on tej samej zmiennej referencyjnej interfejsu do wywoływania metod obiektów klas `ByTwos` i `ByThrees`.

Listing 8.12. *SeriesDemo2.java*

```
// Demonstruje użycie referencji interfejsu.

class ByTwos implements Series {
  int start;
  int val;

  ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}

class ByThrees implements Series {
  int start;
  int val;

  ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}

class SeriesDemo2 {
  public static void main(String args[]) {
    ByTwos twoOb = new ByTwos();
    ByThrees threeOb = new ByThrees();
    Series ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      System.out.println("Kolejna wartość sekwencji ByTwos:" +
                         ob.getNext());
      ob = threeOb;
      System.out.println("Kolejna wartość sekwencji ByThrees: " +
                         ob.getNext());
```

Dostęp do obiektu za pomocą referencji interfejsu.

```
    }
  }
}
```

W metodzie `main()` zmienna `ob` została zadeklarowana jako zmienna referencyjna, której typem jest interfejs `Series`. Oznacza to, że możesz jej przypisać referencję dowolnego obiektu, który implementuje interfejs `Series`. W tym przypadku zmienna jest używana dla obiektów `twoOb` i `threeOb`, które należą odpowiednio do klas `ByTwos` i `ByThrees`. Obie wymienione klasy implementują interfejs `Series`. Zmienna referencyjna interfejsu zna jedynie metody zdefiniowane przez interfejs. Zatem za pomocą zmiennej `ob` nie uzyskasz dostępu do dodatkowych składowych lub metod, które może obsługiwać konkretny obiekt.

Przykład 8.1. Tworzymy interfejs Queue

ICharQ.java
IQDemo.java

Działanie interfejsów zademonstruję na praktycznym przykładzie. We wcześniejszych rozdziałach stworzyliśmy klasę `Queue` implementującą kolejkę znaków o stałym rozmiarze. Istnieje jednak wiele innych sposobów implementacji kolejek. Kolejki mogą mieć stały rozmiar lub „rosnąć". Mogą być liniowe, co szybko może doprowadzić do ich przepełnienia, albo cykliczne, co pozwala zawsze umieszczać w nich kolejne elementy, jeśli tylko inne są pobierane z kolejki. Elementy kolejki mogą być przechowywane w tablicach, listach, drzewie binarnym i innych strukturach danych. Niezależnie od sposobu implementacji kolejki jej interfejs pozostaje taki sam. Definiują go metody `put()` i `get()`. Ponieważ możemy oddzielić interfejs kolejki od jej implementacji, to łatwo możemy zdefiniować interfejs kolejki, pozostawiając szczegóły różnym implementacjom.

W tym przykładzie stworzysz interfejs kolejki przechowującej znaki i trzy jego implementacje. Wszystkie trzy implementacje będą przechowywać znaki w tablicy. Pierwsza implementacja będzie liniową kolejką o stałym rozmiarze, którą stworzyliśmy już w poprzednich rozdziałach. Druga implementacja będzie kolejką cykliczną. Gdy implementacja takiej kolejki dotrze do końca tablicy, indeksy operacji `put` i `get` automatycznie powracają na początek tablicy. Dzięki temu do kolejki może trafiać dowolnie wiele elementów, pod warunkiem że równocześnie są z niej pobierane inne elementy. Ostatnia z implementacji będzie tworzyć dynamiczną kolejkę, zwiększającą swój rozmiar, gdy to konieczne.

1. Utwórz plik *ICharQ.java* i umieść w nim poniższą definicję interfejsu:

```
// Interfejs kolejki przechowującej znaki.
public interface ICharQ {
  // Umieszcza znak w kolejce.
  void put(char ch);

  // Pobiera znak z kolejki.
  char get();
}
```

   Interfejs ten jest bardzo prosty, składa się tylko z dwóch metod. Każda klasa implementująca interfejs `ICharQ` będzie musiała zaimplementować te metody.

2. Utwórz plik o nazwie *IQDemo.java*.

3. Na początek umieść w nim poniższą definicję klasy `FixedQueue`:

```
// Kolejka znaków o stałym rozmiarze.
class FixedQueue implements ICharQ {
  private char q[]; // tablica przechowująca elementy kolejki
  private int putloc, getloc; // indeksy operacji put i get

  // Tworzy pustą kolejkę o podanym rozmiarze.
  public FixedQueue(int size) {
    q = new char[size]; // przydziela pamięć kolejce
    putloc = getloc = 0;
  }
```

```
  // Umieszcza znak w kolejce.
  public void put(char ch) {
    if(putloc==q.length) {
      System.out.println(" -- Kolejka pełna.");
      return;
    }

    q[putloc++] = ch;
  }

  // Pobiera znak z kolejki.
  public char get() {
    if(getloc == putloc) {
      System.out.println(" -- Kolejka pusta.");
      return (char) 0;
    }

    return q[getloc++];
  }
}
```

Tę implementację interfejsu `ICharQ` zaadaptowaliśmy z klasy `Queue` z rozdziału 5. i jej działanie powinno być dla Ciebie zrozumiałe.

4. Dodaj w pliku *IQDemo.java* klasę `CircularQueue` przedstawioną poniżej. Implementuje ona cykliczną kolejkę znaków.

```
// Kolejka cykliczna.
class CircularQueue implements ICharQ {
  private char q[]; // tablica przechowująca elementy kolejki
  private int putloc, getloc; // indeksy operacji put i get

  // Tworzy pustą kolejkę o podanym rozmiarze.
  public CircularQueue(int size) {
    q = new char[size+1]; // przydziela pamięć kolejce
    putloc = getloc = 0;
  }

  // Umieszcza znak w kolejce.
  public void put(char ch) {
    /* Kolejka jest pełna, jeśli putloc ma wartość mniejszą
       o jeden niż getloc lub gdy putloc wskazuje koniec tablicy,
       a getloc początek. */
    if(putloc+1==getloc |
       ((putloc==q.length-1) & (getloc==0))) {
      System.out.println(" -- Kolejka pełna.");
      return;
    }

    q[putloc++] = ch;
    if(putloc==q.length) putloc = 0; // powrót na początek tablicy
  }

  // Pobiera znak z kolejki.
  public char get() {
    if(getloc == putloc) {
      System.out.println(" -- Kolejka pusta.");
      return (char) 0;
    }
    char ch = q[getloc++];
    if(getloc==q.length) getloc = 0; // powrót na początek tablicy
    return ch;
  }
}
```

Kolejka cykliczna umożliwia ponowne wykorzystanie miejsca w tabeli zwolnionego przez elementy pobrane z kolejki. Dzięki temu w kolejce cyklicznej można umieszczać dowolną liczbę elementów, pod warunkiem że inne elementy są z niej pobierane. Chociaż implementacja działania takiej kolejki jest prosta i sprowadza się do wyzerowania odpowiedniego indeksu, gdy kolejka osiągnie koniec tablicy, to warunki brzegowe są na pierwszy rzut oka nieco bardziej skomplikowane. Kolejka cykliczna nie jest pełna, gdy osiągnie koniec tablicy, ale gdy umieszczenie kolejnego elementu spowodowałoby nadpisanie elementu, który nie został jeszcze pobrany z kolejki. Zatem metoda `put()` musi sprawdzić kilka warunków, aby ustalić, czy kolejka jest pełna. Jak sugerują to komentarze umieszczone w kodzie, kolejka jest pełna, gdy wartość indeksu `putloc` jest o jeden mniejsza niż indeksu `getloc`, albo gdy `putloc` znajduje się na końcu tablicy, a `getloc` na początku. Kolejka jest pusta, gdy wartości indeksów `getloc` i `putloc` są równe. Aby ułatwić sprawdzanie tych warunków, tablica, w której jest przechowywana kolejka, jest o jeden większa od długości kolejki.

5. Umieść w pliku *IQDemo.jav*a klasę `DynQueue` przedstawioną poniżej. Implementuje ona kolejkę zwiększającą swój rozmiar, gdy kolejka zapełni się.

```
// Kolejka dynamiczna.
class DynQueue implements ICharQ {
  private char q[]; // tablica przechowująca elementy kolejki
  private int putloc, getloc; // indeksy operacji put i get

  // Tworzy pustą kolejkę o podanym rozmiarze.
  public DynQueue(int size) {
    q = new char[size]; // przydziela pamięć kolejce
    putloc = getloc = 0;
  }

  // Umieszcza znak w kolejce.
  public void put(char ch) {
    if(putloc==q.length) {
      // zwiększa rozmiar kolejki
      char t[] = new char[q.length * 2];

      // kopiuje elementy do nowej kolejki
      for(int i=0; i < q.length; i++)
        t[i] = q[i];

      q = t;
    }

    q[putloc++] = ch;
  }

  // Pobiera znak z kolejki.
  public char get() {
    if(getloc == putloc) {
      System.out.println(" -- Kolejka pusta.");
      return (char) 0;
    }

    return q[getloc++];
  }
}
```

W tej implementacji próba umieszczenia znaku w pełnej kolejce powoduje utworzenie dwa razy większej tablicy, do której zostaje skopiowana zawartość kolejki. Referencja nowej tablicy zostaje przypisana składowej `q`.

6. Aby zademonstrować działanie trzech implementacji interfejsu `ICharQ`, umieść w pliku *IQDemo.java* poniższą definicję klasy. Używa ona referencji interfejsu `ICharQ`, odwołując się do wszystkich trzech kolejek.

```
// Demonstruje interfejs ICharQ.
class IQDemo {
  public static void main(String args[]) {
    FixedQueue q1 = new FixedQueue(10);
    DynQueue q2 = new DynQueue(5);
    CircularQueue q3 = new CircularQueue(10);

    ICharQ iQ;

    char ch;
    int i;

    iQ = q1;
    // Umieszcza znaki w kolejce o stałym rozmiarze.
    for(i=0; i < 10; i++)
      iQ.put((char) ('A' + i));

    // Wyświetla zawartość kolejki.
    System.out.print("Zawartość kolejki o stałym rozmiarze: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }
    System.out.println();

    iQ = q2;
    // Umieszcza znaki w kolejce dynamicznej.
    for(i=0; i < 10; i++)
      iQ.put((char) ('Z' - i));

    // Wyświetla zawartość kolejki.
    System.out.print("Zawartość kolejki dynamicznej: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }

    System.out.println();

    iQ = q3;
    // Umieszcza znaki w kolejce cyklicznej.
    for(i=0; i < 10; i++)
      iQ.put((char) ('A' + i));
    // Wyświetla zawartość kolejki.
    System.out.print("Zawartość kolejki cyklicznej: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }

    System.out.println();

    // Umieszcza więcej znaków w kolejce cyklicznej.
    for(i=10; i < 20; i++)
      iQ.put((char) ('A' + i));

    // Wyświetla zawartość kolejki.
    System.out.print("Zawartość kolejki cyklicznej: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }
```

```
    System.out.println("\nUmieszczam i pobieram znak" +
                       " z kolejki cyklicznej.");

    // Umieszcza i pobiera znak.
    for(i=0; i < 20; i++) {
      iQ.put((char) ('A' + i));
      ch = iQ.get();
      System.out.print(ch);
    }

  }
}
```

7. Poniżej przedstawiłem wynik działanie tego programu:

```
Zawartość kolejki o stałym rozmiarze: ABCDEFGHIJ
Zawartość kolejki dynamicznej: ZYXWVUTSRQ
Zawartość kolejki cyklicznej: ABCDEFGHIJ
Zawartość kolejki cyklicznej: KLMNOPQRST
Umieszczam i pobieram znak z kolejki cyklicznej.
ABCDEFGHIJKLMNOPQRST
```

8. A oto kilka zadań dla Ciebie. Utwórz cykliczną wersję implementacji `DynQueue`. Rozszerz interfejs `ICharQ` o metodę `reset()` przywracającą stan wyjściowy kolejki. Utwórz metodę `static` kopiującą zawartość kolejki jednego typu do kolejki innego typu.

Zmienne w interfejsach

Jak już wspomniałem, interfejs może deklarować zmienne, ale niejawnie przyjmuje się, że zmienne te są zadeklarowane jako `public`, `static` i `final`. Może więc wydawać Ci się, że ich przydatność jest bardzo ograniczona, ale nie jest to prawdą. Skomplikowane programy wykorzystują wiele stałych opisujących rozmiary tablic, różne ograniczenia, wartości specjalne i tym podobne. Ponieważ kod takich programów zwykle znajduje się w wielu plikach źródłowych, to musi istnieć wygodny sposób udostępniania stałych w każdym pliku. W języku Java można użyć w tym celu interfejsu.

Aby zdefiniować zbiór stałych współdzielonych przez różne pliki źródłowe, utwórz interfejs, który zawiera tylko te stałe i nie definiuje żadnych metod. Każda klasa, która używa tych stałych, będzie „implementować" ten interfejs, dzięki czemu uzyska do nich dostęp. Przykład takiego zastosowania interfejsu przedstawiłem na listingu 8.13.

Listing 8.13. *IConstD.java*

```
// Interfejs zawierający stałe.
interface IConst {
  int MIN = 0;
  int MAX = 10;
  String ERRORMSG = "Błąd zakresu";
}

class IConstD implements IConst {
  public static void main(String args[]) {
    int nums[] = new int[MAX];

    for(int i=MIN; i < 11; i++) {
      if(i >= MAX) System.out.println(ERRORMSG);
      else {
        nums[i] = i;
        System.out.print(nums[i] + " ");
      }
    }
  }
}
```

Technika stosowania interfejsów w celu definiowania często używanych stałych jest nieco kontrowersyjna. Opisałem ją tu wyłącznie w celu wyczerpującej prezentacji możliwości i zastosowań interfejsów.

Interfejsy mogą dziedziczyć

Interfejs może dziedziczyć po innym interfejsie poprzez zastosowanie słowa kluczowego `extends`. Składnia jest taka sama jak w przypadku dziedziczenia klas. Jeśli klasa implementuje interfejs, który dziedziczy po innym interfejsie, musi dostarczyć implementacji dla wszystkich metod wymaganych przez łańcuch dziedziczenia interfejsów. Ilustruje to przykład przedstawiony na listingu 8.14.

Listing 8.14. *IFExtend.java*

```
// Dziedziczenie interfejsów.
interface A {
  void meth1();
  void meth2();
}

// B dziedziczy meth1() i meth2() i dodaje meth3().
interface B extends A {
  void meth3();
}

// Ta klasa musi implementować wszystkie metody interfejsów A i B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implementacja meth1().");
  }

  public void meth2() {
    System.out.println("Implementacja meth2().");
  }

  public void meth3() {
    System.out.println("Implementacja meth3().");
  }
}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

B dziedziczy po A.

W ramach eksperymentu możesz spróbować usunąć implementację metody `meth1()` z klasy `MyClass`. Przekonasz się, że spowoduje to błąd kompilacji. Jak wspomniałem wcześniej, każda klasa implementująca interfejs musi implementować wszystkie metody tego interfejsu, włączając w to metody dziedziczone po innych interfejsach.

Domyślne metody interfejsów

Jak już pisałem wcześniej, przed udostępnieniem JDK 8 interfejsy nie mogły zawierać żadnych implementacji. Oznacza to, że we wszystkich wcześniejszych wersjach języka Java wszystkie metody interfejsów musiały być metodami abstrakcyjnymi, czyli pozbawionymi ciała. To tradycyjna forma interfejsów,

której dotyczyły wszystkie informacje podane we wcześniejszej części rozdziału. Jednak sytuacja zmieniła się po wprowadzeniu JDK 8, gdyż w tej wersji język Java został rozszerzony o możliwość dodawania do interfejsów tak zwanej **metody domyślnej**. Metoda domyślna pozwala na zdefiniowanie domyślnej implementacji metody interfejsu. Innymi słowy, dzięki wprowadzeniu metody domyślnej metoda interfejsu może już posiadać ciało — nie musi być metodą abstrakcyjną. W trakcie prac nad wprowadzeniem tej możliwości metody domyślne nazywano także **metodami rozszerzeń**, dlatego obecnie można się spotkać z oboma tymi terminami.

Głównym powodem wprowadzenia metod domyślnych była chęć zapewnienia sposobu rozszerzania interfejsu bez wywoływania problemów w już istniejącym kodzie. Pamiętasz zapewne, że wszystkie metody zdefiniowane w interfejsie muszą zostać zaimplementowane. W przeszłości, kiedy do popularnych i często używanych interfejsów dodawano nowe metody, mogło to doprowadzić do problemów w już istniejącym kodzie, spowodowanych brakiem implementacji tych nowych metod. Wprowadzenie metody domyślnej rozwiązuje ten problem, gdyż zapewnia możliwość dostarczenia implementacji, która będzie używana, jeśli nie zostanie jawnie podana żadna inna implementacja. A zatem dodanie metody domyślnej nie wywoła żadnych problemów w już istniejącym kodzie.

Kolejnym powodem wprowadzenia metod domyślnych była chęć zapewnienia możliwości określania w interfejsach metod, które zależnie do sposobu używania danego interfejsu byłyby w zasadzie opcjonalne. Na przykład interfejs może definiować grupę metod służących do operowania na sekwencji elementów. Jedna z nich może nosić nazwę `remove()` i służyć do usuwania elementu z sekwencji. Niemniej jednak jeśli ten sam interfejs ma służyć do obsługi zarówno sekwencji o zmiennej zawartości, jak i takich, których zawartość nie może ulegać zmianie, to metoda `remove()` właściwie będzie metodą opcjonalną, gdyż sekwencje o niezmiennej zawartości nie będą z niej korzystać. W przeszłości klasa stanowiąca implementację sekwencji o niezmiennej zawartości musiała definiować pustą implementację metody `remove()`, niezależnie od tego, że nigdy by jej nie używała. Obecnie w takim interfejsie można by umieścić domyślną implementację metody `remove()`, która niczego nie zrobi lub zgłosi błąd. Udostępnienie tej metody domyślnej sprawi, że klasa reprezentująca sekwencję o niezmiennej zawartości nie będzie już musiała definiować swojej własnej, pustej wersji metody `remove()`. A zatem dostarczenie metody domyślnej sprawia, że implementacja metody `remove()` staje się czynnością opcjonalną, a nie wymaganą.

Koniecznie należy zaznaczyć, że wprowadzenie metod domyślnych w żaden sposób nie zmienia kluczowego aspektu interfejsów: wciąż nie można w nich definiować zmiennych instancyjnych. Czyli kluczowa różnica pomiędzy interfejsami i klasami polega na tym, że klasy mogą zachowywać stan, a interfejsy nie. Co więcej, wciąż nie można utworzyć instancji interfejsu, dysponując wyłącznie nim samym. Konieczna jest do tego klasa, która będzie go implementować. A zatem mimo że zaczynając od JDK 8, interfejsy mogą definiować metody domyślne, to jednak aby utworzyć instancję interfejsu, musi on zostać zaimplementowany w jakiejś klasie.

I ostatnia sprawa: jako ogólną zasadę należy przyjąć fakt, że metody domyślne stanowią rozwiązanie specjalnego przeznaczenia. Interfejsy, które będziemy tworzyć w przyszłości, wciąż będą przede wszystkim określać, co należy zrobić, a nie jak te operacje wykonywać. Niemniej jednak dodanie metod domyślnych zapewnia nam dodatkową elastyczność.

Podstawowe informacje o metodach domyślnych

Metoda domyślna interfejsu jest definiowana bardzo podobnie do zwyczajnych metod w klasach. Jedyna różnica polega na tym, że deklaracja metody domyślnej jest poprzedzana słowem kluczowym **`default`**. W ramach przykładu przeanalizuj poniższy prosty interfejs (listing 8.15):

Listing 8.15. *MyIF.java*

```
public interface MyIF {
  // To jest "normalna" deklaracja metody interfejsu.
  // NIE definiuje ona domyślnej implementacji.
  int getUserID();
```

```
  // A to jest metoda domyślna. Zauważ, że podaje ona
  // domyślną implementację.
  default int getAdminID() {
    return 1;
  }
}
```

Interfejs MyIF deklaruje dwie metody. Pierwsza z nich, `getUserID()`, jest standardową deklaracją metody interfejsu. Nie definiuje ona żadnej implementacji. Z kolei druga metoda, `getAdminID()`, jest metodą domyślną i zawiera domyślną implementację. W powyższym przykładzie metoda domyślna jedynie zwraca wartość 1. Zwróć szczególną uwagę na postać deklaracji metody `getAdminID()`. Została ona poprzedzona słowem kluczowym `default`. Tę składnię można opisać bardziej ogólnie: w celu zdefiniowania metody domyślnej jej deklarację należy poprzedzić słowem kluczowym `default`.

Ponieważ metoda `getAdminID()` zawiera domyślną implementację, zatem nie trzeba jej przesłaniać w klasie implementującej interfejs `MyIF`. Innymi słowy, jeśli klasa implementująca interfejs nie poda własnej implementacji tej metody, to zostanie użyta implementacja domyślna. Na przykład prosta klasa MyIFImp przedstawiona na listingu 8.16 jest całkowicie prawidłowa:

Listing 8.16. *MyIFImp.java*

```
// Klasa implementująca interfejs MyIF.
class MyIFImp implements MyIF {
  // Tylko metoda getUserID() zdefiniowana w interfejsie MyIF musi zostać
  // zaimplementowana. W przypadku metody getAdminID() możemy
  // skorzystać z jej domyślnej implementacji.
  public int getUserID() {
    return 100;
  }
}
```

Przykład przedstawiony na listingu 8.17 tworzy instancję klasy `MyIFImp` i używa jej w celu wywołania metod `getUserID()` i `getAdminID()`.

Listing 8.17. *DefaultMethodDemo.java*

```
// Zastosowanie metody domyślnej.
class DefaultMethodDemo {
  public static void main(String args[]) {

    MyIFImp obj = new MyIFImp();

    // Metodę getUserID() możemy wywołać, gdyż została jawnie
    // zaimplementowana w klasie MyIFImp:
    System.out.println("Identyfikator użytkownika to: " + obj.getUserID());

    // Możemy także wywołać metodę getAdminID(), gdyż jest ona
    // metodą domyślną i także posiada implementację:
    System.out.println("Identyfikator administratora to: " + obj.getAdminID());
  }
}
```

Powyższy program zwróci następujące wyniki:

```
Identyfikator użytkownika to 100
Identyfikator administratora to 1
```

Jak widać, automatycznie została użyta domyślna implementacja metody `getAdminID()`. Definiowanie jej w klasie `MyIFImp` nie było konieczne. A zatem w przypadku metody `getAdminID()` implementowanie jej w klasie jest opcjonalne. (Oczywiście gdyby klasa wymagała zwrócenia innej wartości identyfikatora administratora, to zaimplementowanie metody `getAdminID()` byłoby *konieczne*).

Klasa implementująca interfejs oczywiście może podać własną implementację jego metody domyślnej, co więcej, takie rozwiązanie jest bardzo często stosowane. Na przykład klasa `MyIFImp2` przedstawiona na listingu 8.18 przesłania metodę `getAdminID()`:

Listing 8.18. *MyIFImp2.java*

```
class MyIFImp2 implements MyIF {
  // W tej wersji klasy podaliśmy implementacje obu metod:
  // getUserID() oraz getAdminID().
  public int getUserID() {
    return 100;
  }

  public int getAdminID() {
    return 42;
  }
}
```

W tym przypadku wywołanie metody `getAdminID()` zwróci wartość inną niż domyślna.

Praktyczny przykład metody domyślnej

Chociaż poprzednie przykłady zaprezentowały mechanikę stosowania metody domyślnej, to jednak nie wykazały jej przydatności w zastosowaniach praktycznych. W tym celu wrócimy do interfejsu `Series` przedstawionego we wcześniejszej części rozdziału. Na potrzeby rozważeń przyjmijmy, że interfejs ten jest bardzo popularny i korzysta z niego bardzo wielu programistów. Co więcej, załóżmy, że na podstawie analizy wzorców zastosowania udało się ustalić, że do bardzo wielu klas implementujących ten interfejs dodawana była metoda zwracająca tablicę zawierającą *n* kolejnych elementów serii. Z tego względu podjąłeś decyzję o rozszerzeniu interfejsu i dodaniu do niego takiej metody; ma to być metoda `getNextArray()` o następującej postaci:

```
int[] getNextArray(int n)
```

Gdzie `n` określa liczbę pobieranych elementów serii. Przed wprowadzeniem metod domyślnych taka modyfikacja interfejsu `Series` spowodowałaby awarię istniejącej bazy kodu, gdyż jego istniejące implementacje nie definiowałyby metody `getNextArrray()`. Niemniej jednak dzięki podaniu domyślnej implementacji tej nowej metody można ją dodać do interfejsu bez wywoływania żadnych problemów.

W niektórych przypadkach, kiedy do istniejącego interfejsu zostanie dodana metoda domyślna, w razie próby jej użycia klasa, która go implementuje, zgłosi błąd. Takie rozwiązanie jest konieczne, gdy nie można znaleźć implementacji metod domyślnych, których można użyć we wszystkich możliwych sytuacjach. Takie typy metod domyślnych definiują kod, który właściwie można uznać za opcjonalny. Jednak w niektórych przypadkach można zdefiniować metodę domyślną, która będzie działać we wszystkich sytuacjach. Dotyczy to także naszej przykładowej metody `getNextArray()`. Ponieważ interfejs `Series` wymaga implementacji metody `getNext()`, więc domyślna implementacja metody `getNextArray()` może z niej skorzystać. A zatem poniższy listing 8.19 przedstawia interfejs `Series`, stanowiący nową wersję interfejsu `Series` i zawierający metodę domyślną o nazwie `getNextArray()`:

Listing 8.19. *Series2.java*

```
// Rozbudowana wersja interfejsu Series, zawierająca
// metodę domyślną o nazwie getNextArray().
public interface Series {
  int getNext(); // Zwraca następną liczbę w serii.

  // Zwraca tablicę zawierającą n następnych elementów
  // serii.
  default int[] getNextArray(int n) {
```

```
    int[] vals = new int[n];

    for(int i=0; i < n; i++) vals[i] = getNext();
    return vals;
  }

  void reset(); // restart
  void setStart(int x); // określa wartość początkową
}
```

Zwróć szczególną uwagę na sposób, w jaki została zaimplementowana metoda domyślna `getNextArray()`. Ponieważ metoda `getNext()` należała do początkowej specyfikacji interfejsu `Series`, zatem będzie ona dostępna w każdej klasie, która go implementuje. Dlatego można jej użyć w metodzie `getNextArray()`, by pobrać *n* następnych elementów serii. W efekcie każda klasa implementująca rozszerzoną wersję interfejsu `Series` będzie mogła używać metody `getNextArray()` bez konieczności jej przesłaniania czy wprowadzania jakichkolwiek dodatkowych modyfikacji. Oznacza to także, że ta zmiana interfejsu `Series` nie doprowadzi do problemów w żadnym już istniejącym kodzie. Oczywiście nic nie stoi na przeszkodzie, by klasa implementująca ten interfejs dostarczyła własną implementację metody `getNextArray()`.

Jak pokazał powyższy przykład, metody domyślne zapewniają dwie podstawowe korzyści:

- Dają możliwość łagodnego, stopniowego przekształcania interfejsu bez wywoływania problemów w istniejącym kodzie.
- Pozwalają dostarczać opcjonalne możliwości funkcjonalne bez konieczności pisania w klasach implementujących pustych metod zastępczych, gdy funkcjonalności te nie są potrzebne.

W przypadku przedstawionej powyżej metody `getNextArray()` szczególne znaczenie ma ta druga zaleta. Jeśli klasa implementująca interfejs `Series` nie wymaga możliwości, jakie daje metoda `getNextArray()`, to nie będzie musiała zawierać jej pustej implementacji. A to oznacza, że kod tej klasy będzie bardziej przejrzysty.

Problemy wielokrotnego dziedziczenia

Zgodnie z informacjami podanymi we wcześniejszej części książki Java nie udostępnia wielokrotnego dziedziczenia klas. Jednak teraz, kiedy interfejsy mogą zawierać metody domyślne, mógłbyś się zastanawiać, czy nie pozwalają one ominąć tego ograniczenia. Otóż w gruncie rzeczy nie — nie pozwalają. Przypomnij sobie, że wciąż istnieje jedna, podstawowa różnica pomiędzy interfejsami i klasami: klasy mogą przechowywać stan (dzięki zmiennym instancyjnym), a interfejsy nie.

Pomimo to metody domyślne częściowo oferują możliwości, które normalnie można by powiązać z pojęciem wielokrotnego dziedziczenia. Na przykład można stworzyć klasę implementującą dwa interfejsy. Jeśli każdy z nich będzie definiował metody domyślne, to klasa odziedziczy po nich jakieś zachowania. A zatem metody domyślne w pewnym stopniu faktycznie udostępniają możliwość wielokrotnego dziedziczenia zachowań. Jak się można spodziewać, w takim przypadku pojawia się także możliwość występowania konfliktów nazw.

W ramach przykładu załóżmy, że klasa `MyClass` implementuje dwa interfejsy: `Alpha` i `Beta`. Co się stanie, jeśli w obu tych interfejsach będzie dostępna metoda `reset()` posiadająca domyślną implementację? Czy klasa `MyClass` użyje wersji metody pochodzącej z interfejsu `Alpha`, czy z interfejsu `Beta`? A co by się stało w sytuacji, gdyby interfejs `Beta` rozszerzał interfejs `Alfa`? Która z wersji metody domyślnej zostałaby użyta? A co by się stało, gdyby klasa `MyClass` udostępniała własną implementację metody `reset()`? Aby rozwiązywać zarówno te, jak i inne podobne problematyczne sytuacje, Java udostępnia zestaw reguł pozwalających na likwidowanie takich konfliktów.

Przede wszystkim w każdym przypadku metoda zaimplementowana w klasie będzie mieć pierwszeństwo przed metodami domyślnymi zdefiniowanymi w interfejsach. A zatem jeśli klasa `MyClass` udostępni własną wersję metody `reset()`, przesłaniającą metody domyślne zdefiniowane w interfejsach, to właśnie ona będzie używana. Stanie się tak, nawet jeśli klasa `MyClass` będzie implementować oba interfejsy — `Alpha` i `Beta`. W takim przypadku obie implementacje metod domyślnych zostaną przesłonięte przez metodę klasy `MyClass`.

Dodatkowo zostanie zgłoszony błąd, jeśli klasa implementuje dwa interfejsy, z których każdy udostępnia tę samą metodę domyślną, oraz jeśli metoda ta nie została zaimplementowana w samej klasie. Wracając do naszego przykładu: jeśli klasa `MyClass` będzie implementować interfejsy `Alpha` i `Beta`, lecz nie udostępni implementacji metody `reset()`, to zostanie zgłoszony błąd.

W przypadkach, gdy jeden interfejs dziedziczy po drugim i oba udostępniają tę samą metodę domyślną, pierwszeństwo będzie miała wersja metody zdefiniowana w interfejsie dziedziczącym. A zatem — wracając do naszego przykładu — gdyby interfejs `Beta` rozszerzał interfejs `Alpha`, to zostałaby użyta metoda `reset()` zdefiniowana w interfejsie `Beta`.

Do metody domyślnej można odwołać się jawnie, używając nowej wersji słowa kluczowego `super`. Poniżej przedstawiłem jego ogólną postać:

```
NazwaInterfejsu.super.nazwaMetody()
```

Na przykład aby w interfejsie `Beta` odwołać się do domyślnej implementacji metody `reset()` zdefiniowanej w interfejsie `Alpha`, należałoby użyć następującej instrukcji:

```
Alpha.super.reset();
```

Stosowanie metod statycznych w interfejsach

W JDK 8 interfejsy udostępniają jeszcze jedną, nową możliwość: definiowanie jednej lub kilku metod **statycznych**. Podobnie jak metody statyczne definiowane w klasach, także metody statyczne definiowane w interfejsach można wywoływać niezależnie od jakichkolwiek obiektów. Oznacza to, że możliwość wywoływania takich metod nie wymaga ani zaimplementowania interfejsu, ani utworzenia jakichkolwiek jego instancji. Zamiast tego takie metody można wywoływać, podając nazwę interfejsu, kropkę oraz nazwę metody. Poniżej przedstawiłem ogólną postać takiego wywołania:

```
NazwaInterfejsu.nazwaMetodyStatycznej
```

Warto zwrócić uwagę, że odpowiada to sposobowi wywoływania metod statycznych zdefiniowanych w klasach.

Poniższy przykład (listing 8.20) przedstawia interfejs `MyIF2`, zmodyfikowaną wersję interfejsu `MyIF`, do którego dodano metodę statyczną o nazwie `getUniversalID()`. W naszym przykładzie metoda ta zwraca wartość `0`.

Listing 8.20. *MyIF2.java*

```
public interface MyIF {
  // To jest "normalna" deklaracja metody interfejsu.
  // NIE definiuje ona domyślnej implementacji.
  int getUserID();

  // A to jest metoda domyślna. Zauważ, że podaje ona
  // domyślną implementację.
  default int getAdminID() {
    return 1;
  }

  // To jest metoda statyczna.
  static int getUniversalID() {
    return 0;
  }
}
```

Metodę `getUniversalID()` można wywoływać w następujący sposób:

```
int uID = MyIF2.getUniversalID();
```

Ponieważ metoda `getUniversalID()` jest metodą statyczną, więc jak już wspominałem, możliwość korzystania z niej nie wymaga wcześniejszego implementowania interfejsu `MyIF2` ani tworzenia jego instancji.

I jeszcze ostatnia uwaga: statyczne metody zdefiniowane w interfejsach nie są dziedziczone, i to ani przez klasy implementujące dany interfejs, ani przez interfejsy, które go rozszerzają.

Ostatnie uwagi o pakietach i interfejsach

Chociaż przykłady w tej książce nie będą często używać pakietów ani interfejsów, oba te narzędzia odgrywają istotną rolę w programowaniu w języku Java. Praktycznie każda prawdziwa aplikacja w języku Java korzysta z pakietów, a znaczna część wykorzystuje również interfejsy. Dlatego ważne jest, żebyś dobrze je opanował.

Test sprawdzający

1. Wykorzystując kod z podrozdziału „Przykład 8.1. Tworzymy interfejs Queue", umieść interfejs `ICharQ` i jego trzy implementacje w pakiecie o nazwie `qpack`. Klasę demonstrującą działanie kolejki, `IQDemo`, pozostaw w pakiecie domyślnym i pokaż, w jaki sposób może ona importować klasy pakietu `qpack` i ich używać.
2. Czym jest przestrzeń nazw? Dlaczego ważna jest możliwość podziału przestrzeni nazw w Javie?
3. Pakiety przechowywane są w _____.
4. Wyjaśnij różnicę pomiędzy modyfikatorem dostępu `protected` a domyślnym poziomem dostępu.
5. Omów oba sposoby użycia klas pakietu przez inne pakiety.
6. „Jeden interfejs, wiele metod" to podstawowa zasada polimorfizmu w Javie. Który z mechanizmów języka stanowi najlepszy przykład jej zastosowania?
7. Ile klas może implementować interfejs? Ile interfejsów może implementować klasa?
8. Czy interfejsy mogą dziedziczyć?
9. Utwórz interfejs klasy `Vehicle` z rozdziału 7. Nazwij go `IVehicle`.
10. Zmienne interfejsu są niejawnie zadeklarowane jako `static` i `final`. Czy można je udostępniać innym częściom programu?
11. Pakiet jest w zasadzie kontenerem klas. Prawda czy fałsz?
12. Który ze standardowych pakietów Javy jest automatycznie importowany do każdego programu?
13. Jakiego słowa kluczowego używa się podczas deklarowania metod domyślnych?
14. Czy zaczynając od JDK 8, można definiować metody statyczne w interfejsach?
15. Załóżmy, że interfejs `ICharQ` przedstawiony w przykładzie 8.1 stał się bardzo popularny i już od kilku lat jest powszechnie używany. Chciałbyś jednak do niego dodać nową metodę, `reset()`, która ma służyć do przywrócenia początkowego, pustego stanu kolejki. W jaki sposób mógłbyś to zrobić bez wywoływania problemów w już istniejącym kodzie, zakładając, że używasz JDK 8?
16. W jaki sposób są wywoływane statyczne metody interfejsów?

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Poznaj i wykorzystaj możliwości języka Java!

Java to najczęściej wybierany język programowania do zadań specjalnych. Jeżeli Twój projekt wymaga najwyższego poziomu bezpieczeństwa, ogromnej wydajności oraz sprawdzonych rozwiązań, to wybór może być tylko jeden! Właśnie Java jest najczęściej używana do tworzenia zaawansowanych systemów bankowych oraz aplikacji do zarządzania przedsiębiorstwami i finansami. Ale czy sprawdzi się w typowym projekcie? Oczywiście!

Rozpocznij własną przygodę z tym popularnym językiem programowania korzystając z tej książki. Kolejne wydanie zostało poprawione i zaktualizowane o nowości ze świata Javy. Każda strona zawiera bezcenną wiedzę na temat składni języka, stosowanych w nim konstrukcji, programowania obiektowego i nie tylko. Sprawdzisz tu, jak obsługiwać wyjątki, korzystać ze strumieni oraz z wątków. Jeżeli masz ambicję stworzyć atrakcyjny interfejs użytkownika z użyciem JavaFX, również będziesz usatysfakcjonowany znalezionymi w tej książce informacjami. Jest to doskonała lektura dla wszystkich osób chcących poznać Javę — jeden z najbardziej cenionych języków programowania.

Dzięki tej książce:

- **poznasz składnię oraz typowe konstrukcje języka Java**
- **zdobędziesz wiedzę na temat programowania obiektowego**
- **obsłużysz sytuacje wyjątkowe w Twojej aplikacji**
- **wykorzystasz możliwości JavaFX**
- **opanujesz ceniony język programowania**

Herbert Schildt — autorytet w świecie programistów Javy, C, C++ oraz C#. Jest autorem bestsellerów poświęconych programowaniu w tych językach. Swoją uwagę koncentruje na językach programowania, interpreterach, kompilatorach oraz sterowaniu robotami. Bierze aktywny udział w procesie standaryzacji języków programowania.

Helion

34430 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

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

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

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-0808-4

9 788328 308084

cena: 89,00 zł

Oracle Press