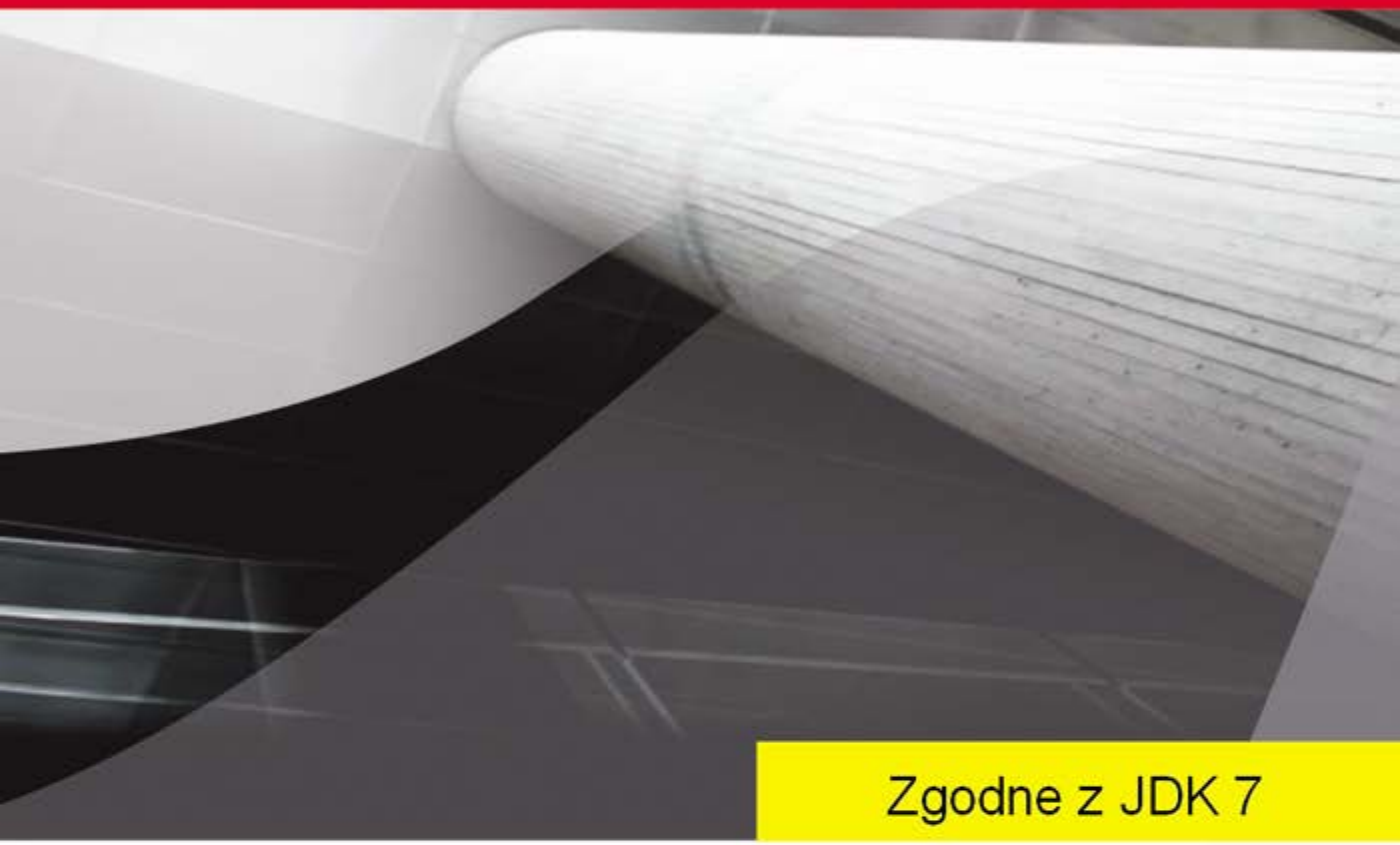
ORACLE



Zgodne z JDK 7

Java

Kompendium programisty

Wydanie VIII

Najbardziej aktualne źródło wiedzy o Javie!

ORIGINAL • AUTHENTIC
Oracle Press
ONLY FROM OSBORNE

Helion

Herbert Schildt

Tytuł oryginału: Java The Complete Reference, 8th Edition

Tłumaczenie: Mikołaj Szczepaniak

ISBN: 978-83-246-3767-6

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javkp8.zip

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javkp8
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 9.

Pakiety i interfejsy

Niniejszy rozdział omawia dwie najbardziej innowacyjne cechy języka Java: pakiety i interfejsy. **Pakiety** to pojemniki na klasy używane do odpowiedniego podziału przestrzeni nazw klas. Programista może utworzyć w swoim pakiecie na przykład klasę `List` bez ryzyka konfliktu nazw z klasą `List` należącą do innego pakietu. Pakiety zapewniają strukturę podobną do hierarchicznej i są jawnie importowane do nowych definicji klas.

W poprzednich rozdziałach przedstawiłem, w jaki sposób metody definiują interfejs dla danych klasy. Słowo kluczowe `interface` umożliwia całkowite oddzielenie interfejsu od jego implementacji. Interfejs w Javie określa zbiór metod, które będą implementowane przez jedną lub więcej klas. Sam interfejs nie definiuje żadnej implementacji. Interfejsy przypominają klasy abstrakcyjne, ale mają jedną bardzo ważną zaletę: klasa może implementować dowolną liczbę interfejsów. Przypomnę, iż klasa może dziedziczyć tylko po jednej klasie (abstrakcyjnej lub tradycyjnej).

Pakiety

W dotychczasowych przykładach nazwa każdej z przykładowych klas pochodziła z jednej, tej samej przestrzeni nazw. Oznacza to, że każda klasa musiała mieć unikatową nazwę, aby nie dochodziło do konfliktów nazw. Gdyby nie przestrzenie nazw, z pewnością w większych projektach szybko wyczerpałaby się lista opisowych nazw dla klas. Co więcej, trzeba zapewnić, by różni programiści nie tworzyli klas o takich samych nazwach. (Wyobraź sobie małą grupę programistów walczących o to, kto będzie mógł nadać swojej klasie nazwę `Foobar`, lub sytuację, gdy na wszystkich grupach dyskusyjnych toczy się spór o to, kto pierwszy nadał klasie nazwę `Espresso`). Projektanci Javy wymyślili bardzo elastyczny mechanizm dzielenia przestrzeni nazw na mniejsze kawałki. Mechanizm ten nosi nazwę pakietu. Pakiet to nie tylko przestrzeń nazw, ale też sposób kontroli widoczności i dostępu. Można zdefiniować klasy dostępne jedynie dla innych klas z tego samego pakietu. Co więcej, można zdefiniować metody dostępne dla innych klas tego samego pakietu, ale nieosiągalne dla pozostałych klas. W ten sposób klasy z pakietu mają bardzo dobrą wiedzę o sobie, ale nie udostępniają tych informacji całemu światu.

Definiowanie pakietu

Tworzenie pakietu nie jest trudne. Wystarczy zastosować polecenie `package` jako pierwsze wyrażenie w pliku źródłowym Javy. Wszystkie klasy zdefiniowane w tak rozpoczętym pliku zostaną przypisane do wskazanego pakietu. Pakiet określa przestrzeń nazw dla klas. Pominięcie instrukcji `package` powoduje przypisanie klasy do domyślnej przestrzeni nazw (przestrzeni bez nazwy). (Dzięki temu aż do tego rozdziału nie musieliśmy się zajmować pakietami). Pakiet domyślny nadaje się w zasadzie tylko i wyłącznie do pisania krótkich programów, więc nie jest stosowany w rzeczywistych aplikacjach. Na ogół stosuje się własne przestrzenie nazw, czyli definiuje się własne pakiety.

Oto ogólna postać instrukcji `package`.

```
package pakiet;
```

Element *pakiet* to nazwa pakietu. Poniższa instrukcja tworzy pakiet o nazwie `MyPackage`.

```
package MyPackage;
```

Java używa katalogów systemu plików do przechowywania pakietów. Oznacza to, że każdy plik *.class* utworzony dla pakietu `MyPackage` musi zostać umieszczony w katalogu o nazwie *MyPackage*. Pamiętaj, że wielkość liter ma znaczenie i nazwa katalogu musi być dokładnie taka sama jak nazwa pakietu.

Ta sama instrukcja `package` może pojawić się w więcej niż jednym pliku źródłowym. Po prostu określa ona, do jakiego pakietu należą klasy zdefiniowane w danym pliku. Inne klasy zdefiniowane w innym pliku źródłowym również mogą należeć do tego pakietu. W rzeczywistych aplikacjach pakiet składa się z wielu plików.

Można tworzyć hierarchię pakietów. W tym celu należy oddzielić nazwy poszczególnych pakietów znakiem kropki (`.`). Ogólna postać wielopoziomowego pakietu jest następująca.

```
package pakiet1[.pakiet2[.pakiet3]];
```

Hierarchia pakietów musi być odtworzona w systemie plików środowiska, w którym działa Java. Na przykład pakiet zadeklarowany za pomocą wyrażenia:

```
package java.awt.image;
```

w systemie Windows musi znaleźć się w katalogu *java\awt\images*. Warto ostrożnie dobierać nazwę pakietu. Nie można zmienić nazwy pakietu bez jednoczesnej zmiany nazwy katalogu.

Znajdowanie pakietów i ścieżka CLASSPATH

Jak już wspomniano, pakiety są reprezentowane w systemie operacyjnym przez katalogi. Powstaje więc pytanie: jak to się dzieje, że system wykonawczy Javy „wie", gdzie ma szukać utworzonych przez nas pakietów? Odpowiedź musi uwzględniać trzy aspekty. Po pierwsze, domyślnie system wykonawczy używa aktualnego katalogu jako początku ścieżki wyszukiwania. Jeśli pakiet znajduje się w aktualnym katalogu lub jego podkatalogu, zostanie odnaleziony.

Po drugie, zmienna środowiskowa CLASSPATH określa inne lokalizacje, w których trzeba szukać pakietów. Po trzecie, programista może wskazać ścieżkę do swoich klas za pomocą opcji -classpath poleceń *jva* i javac.

Rozważmy następującą specyfikację pakietu.

```
package MyPack;
```

Program odnajdzie pakiet MyPack, jeśli spełniony został jeden z trzech warunków. Program został uruchomiony z poziomu katalogu znajdującego się bezpośrednio nad katalogiem *MyPack* w hierarchii katalogów, zmienna środowiskowa CLASSPATH zawiera ścieżkę do katalogu *MyPack* lub opcja -classpath polecenia java wskazuje ścieżkę do katalogu *MyPack*.

W przypadku stosowania drugiego z wymienionych rozwiązań ścieżka do klas nie może obejmować samego katalogu *MyPack*. Powinna raczej wskazywać ścieżkę do katalogu nadrzędnego względem katalogu *MyPack*. Jeśli na przykład w środowisku Windows ścieżka do katalogu *MyPack* jest następująca:

C:\MyPrograms\Java\MyPack

prawidłowa ścieżka do klas powinna mieć następującą postać:

C:\MyPrograms\Java

Najprostszym sposobem sprawdzenia przykładów znajdujących się w niniejszym rozdziale jest utworzenie nowego katalogu o nazwie zgodnej z nazwą pakietu w aktualnym katalogu testowym i umieszczenie w nim wszystkich plików *.class*. Programy należy uruchamiać z katalogu testowego. W kolejnych przykładach zakładam takie właśnie podejście.

Prosty przykład pakietu

Pamiętając o wcześniejszej wskazówce, warto skompilować i uruchomić poniższy przykład.

```
// prosty pakiet
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": " + bal+ " zł");
  }
}

class AccountBalance {
  public static void main(String args[]) {
```

```
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Willhelm Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

Nadaj plikowi źródłowemu nazwę *AccountBalance.java* i umieść go w katalogu *MyPack*.

Następnie skompiluj kod. Umieść wynikowy plik *.class* w katalogu *MyPack*. Uruchom program, używając poniższego polecenia.

```
java MyPack.AccountBalance
```

Należy pamiętać, że powyższe polecenie powinno być wykonane z poziomu katalogu powyżej katalogu *MyPack*. (Alternatywnym rozwiązaniem jest użycie jednej z dwóch pozostałych technik opisanych w poprzednim punkcie i umożliwiających wskazanie ścieżki do katalogu *MyPack*).

Klasa `AccountBalance` stanowi część pakietu `MyPack`. Oznacza to, iż nie można jej wykonać bezpośrednio, czyli zastosować poniższego polecenia.

```
java AccountBalance
```

Nazwę klasy trzeba poprzedzić nazwą pakietu.

Ochrona dostępu

W poprzednich rozdziałach wspomniałem o wielu aspektach mechanizmu sterowania dostępem i modyfikatorach dostępu. Na przykład teraz każdy wie, iż składowa prywatna klasy może być odczytywana tylko i wyłącznie przez inne składowe tej klasy. Pakiety dodają nowy wymiar do sterowania dostępem. Java zapewnia wiele poziomów ochrony, aby programista mógł szczegółowo określić widoczność zmiennych i metod w klasach, podklasach i pakietach.

Klasy i pakiety to rozwiązania pozwalające hermetyzować zmienne oraz metody, a także określać przestrzenie nazw. Pakiety działają jak pojemniki na klasy i inne pakiety. Klasy to pojemniki przechowujące dane i kod. Klasa stanowi najmniejszą jednostkę abstrakcji w języku Java. Dzięki wzajemnemu współdziałaniu klas i pakietów możliwe są cztery kategorie widoczności składowych klas:

- podklasy z tego samego pakietu,
- inne klasy z tego samego pakietu,
- podklasy z innych pakietów,
- inne klasy z innych pakietów.

Trzy modyfikatory dostępu (`private`, `public` i `protected`) pozwalają tworzyć wiele poziomów dostępu dla wymienionych kategorii. Tabela 9.1 podsumowuje wszystkie możliwe kombinacje.

Tabela 9.1. *Dostęp do składowych klasy*

	private	Brak modyfikatora	protected	public
Ta sama klasa	Tak	Tak	Tak	Tak
Ten sam pakiet, podklasa	Nie	Tak	Tak	Tak
Ten sam pakiet, inna klasa	Nie	Tak	Tak	Tak
Inny pakiet, podklasa	Nie	Nie	Tak	Tak
Inny pakiet, inna klasa	Nie	Nie	Nie	Tak

Choć mechanizm sterowania dostępem w Javie wydaje się złożony, bardzo łatwo go zrozumieć. Cokolwiek zadeklarowane z modyfikatorem `public` jest dostępne z dowolnego miejsca. Cokolwiek zadeklarowane z modyfikatorem `private` nie jest dostępne poza własną klasą. Gdy element nie ma jawnie określonego modyfikatora, jest widoczny w podklasach oraz w innych klasach tego samego pakietu. Jest to dostęp domyślny. Jeżeli element ma być widoczny poza pakietem, ale tylko dla klas dziedziczących po danej klasie, należy użyć modyfikatora `protected`.

Tabela 9.1 dotyczy tylko składowych klasy. Sama klasa może posiadać tylko dwa poziomy dostępu: domyślny i publiczny. Gdy klasa zostanie zadeklarowana z modyfikatorem `public`, jest dostępna dla dowolnego innego kodu. Zastosowanie domyślnego dostępu powoduje, iż klasa dostępna jest tylko dla innych klas z tego samego pakietu.

Przykład dostępu

Kolejny przykład ilustruje wykorzystanie wszystkich kombinacji modyfikatorów dostępu. Przykład składa się z pięciu klas i dwóch pakietów. Należy pamiętać, że klasy znajdujące się w pakietach muszą zostać umieszczone w katalogach nazwanych tak samo jak odpowiednie pakiety — w tym przypadku w katalogach *p1* i *p2*.

Plik źródłowy pierwszego pakietu definiuje trzy klasy: `Protection`, `Derived` i `SamePackage`. Pierwsza klasa definiuje cztery zmienne typu `int` o różnym dostępie. Zmienna `n` ma dostęp domyślny, `n_pri` dostęp prywatny, `n_pro` dostęp chroniony, a `n_pub` dostęp publiczny.

Pozostałe klasy z przykładu będą próbowały uzyskać dostęp do zmiennych egzemplarza klasy `Protection`. Wiersze, które spowodowałyby błąd kompilacji w związku z ograniczeniami dostępu, zostały opatrzone komentarzem jednowierszowym (znaki `//`). Przed każdym z tych wierszy dodatkowo umieszczono komentarz informujący o poziomie ochrony i ograniczeniach dostępu.

Druga klasa, `Derived`, jest podklasą klasy `Protection` z tego samego pakietu `p1`. Powoduje to, iż klasa `Derived` ma dostęp do wszystkich zmiennych klasy bazowej poza `n_pri`, zmienną prywatną. Trzecia klasa, `SamePackage`, nie jest podklasą klasy `Protection`, ale znajduje się w tym samym pakiecie, więc również ma dostęp do wszystkich zmiennych poza `n_pri`.

Kod z pliku *Protection.java*.

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("konstruktor klasy bazowej");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Kod z pliku *Derived.java*.

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("konstruktor podklasy");
    System.out.println("n = " + n);

    // tylko ta sama klasa
    //  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Kod z pliku *SamePackage.java*.

```
package p1;

class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("konstruktor z tego samego pakietu");
    System.out.println("n = " + p.n);

    // tylko ta sama klasa
    //  System.out.println("n_pri = " + p.n_pri);

    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Teraz zajmijmy się kodem źródłowym drugiego pakietu, `p2`. Dwie zdefiniowane w nim klasy pozwalają sprawdzić dwa pozostałe przypadki związane ze sterowaniem dostępem. Pierwsza klasa, `Protection2`, jest podklasą klasy `p1.Protection`. Ma ona dostęp do wszystkich zmien-

nych poza zmiennymi `n_pri` (ponieważ jest to zmienna prywatna) i `n` (gdyż stosuje domyślny poziom dostępu). Pamiętaj, że dostęp domyślny zapewnia jedynie dostęp z poziomu tej samej klasy lub pakietu. Klasy potomne należące do innych pakietów nie mają dostępu do takiej zmiennej. Klasa `OtherPackage` ma dostęp jedynie do jednej zmiennej, `n_pub`, zadeklarowanej z modyfikatorem `public`.

Kod z pliku *Protection2.java*.

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("konstruktor podklasy z innego pakietu");

    // tylko ta sama klasa lub pakiet
    //  System.out.println("n = " + n);

    // tylko ta sama klasa
    //  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Kod z pliku *OtherPackage.java*.

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("konstruktor z innego pakietu");

    // tylko ta sama klasa lub pakiet
    //  System.out.println("n = " + p.n);

    // tylko ta sama klasa
    //  System.out.println("n_pri = " + p.n_pri);

    // tylko ta sama klasa, pakiet lub podklasa
    //  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Poniżej znajdują się kody źródłowe dwóch testów służących do sprawdzenia działania obu pakietów. Oto kod testujący pakiet `p1`.

```
// Test pakietu p1.
package p1;

// Tworzenie egzemplarzy klas z p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
```

```
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

Kod testujący pakiet `p2` jest następujący.

```
// Test pakietu p2.
package p2;

// Tworzenie egzemplarzy klas z p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Import pakietów

Istnienie pakietów pozwala łatwo segregować klasy i tworzyć różne grupy. Właśnie z tego powodu wszystkie wbudowane klasy Javy znajdują się w pakietach. Żadna z klas dostarczanych z Javą nie znajduje się w domyślnym pakiecie. Ponieważ nazwy klas trzeba poprzedzać nazwą pakietu, takie rozwiązanie staje się uciążliwe, jeżeli z danej klasy korzysta się bardzo często. Z tego powodu Java wykorzystuje instrukcję `import`, która umożliwia uwidocznienie konkretnych klas lub całych pakietów. Po dokonaniu importu wystarczy podać tylko i wyłącznie nazwę klasy. Instrukcja `import` jest tak naprawdę pomocą programisty. Z punktu widzenia samego języka nie trzeba jej stosować w żadnym z programów. Gdy jednak wielokrotnie w programie korzysta się z tych samych klas, importowanie pozwala zaoszczędzić mnóstwa pisania.

W plikach źródłowych Javy instrukcja importu znajduje się tuż po instrukcji `package` (o ile istnieje), a przed definicjami klas. Oto ogólna postać instrukcji `import`.

```
import pakiet1[.pakiet2].(nazwaklasy | *);
```

Element *pakiet1* to nazwa pakietu na najwyższym poziomie, natomiast element *pakiet2* to nazwa pakietu znajdującego się w głównym pakiecie. W zasadzie nie istnieje ograniczenie co do liczby poziomów pakietów, wszystko zależy od możliwości systemu plików systemu operacyjnego. Na końcu pojawia się konkretna *nazwaklasy* lub znak gwiazdki (*), który oznacza, iż kompilator powinien zaimportować cały pakiet. Poniższy fragment kodu przedstawia oba rozwiązania.

```
import java.util.Date;
import java.io.*;
```

Wszystkie standardowe klasy języka Java znajdują się w pakiecie `java`. Podstawowe metody związane z obsługą języka zostały umieszczone w pakiecie `java.lang`. Choć wszystkie inne pakiety trzeba importować w sposób jawny, Java automatycznie w sposób niejawny importuje wszystkie klasy tego pakietu. Jest to równoznaczne pisaniu poniższego wiersza na początku każdego pliku źródłowego.

```
import java.lang.*;
```

Jeśli klasa o tej samej nazwie istnieje w dwóch różnych pakietach importowanych z wykorzystaniem gwiazdki, kompilator nie zgłosi błędu. Kompilacja nie powiedzie się dopiero wtedy, gdy spróbuje się użyć jednej z tych klas. Wtedy trzeba podać pełną nazwę klasy, czyli nazwę samej klasy poprzedzoną nazwą pakietu.

Warto podkreślić, że wyrażenie `import` jest opcjonalne. Do każdej klasy można odwołać się bez importowania odpowiednich pakietów — stosując **pełne nazwy** (obejmujące wszystkie pakiety i podpakiety hierarchii). Następujący fragment kodu używa instrukcji importu.

```
import java.util.*;
class MyDate extends Date {
}
```

Poniżej znajduje się ten sam przykład bez instrukcji `import`.

```
class MyDate extends java.util.Date {
}
```

W tej wersji zastosowano pełną nazwę klasy `Date`.

Zgodnie z tabelą 9.1, jeśli nie tworzy się podklasy, w momencie importu pakietu dostępne będą tylko te składowe, które zostały zadeklarowane jako `public`. Jeżeli więc klasa `Balance` pakietu `MyPack` ma być dostępna poza swoim pakietem, trzeba ją zadeklarować jako klasę publiczną i umieścić w osobnym pliku źródłowym.

```
package MyPack;

/* Teraz klasa Balance, jej konstruktor oraz metoda
    show() są publiczne. Oznacza to, że mogą być używane
    przez klasy z innych pakietów.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": " + bal+ " zł");
  }
}
```

Klasa `Balance` została teraz zadeklarowana z modyfikatorem `public`. Poza tym metoda `show()` i konstruktor także zostały upublicznione. Oznacza to, iż klasy z innych pakietów mają pełny dostęp do tych elementów. Klasa `TestBalance` importuje pakiet `MyPack` i może skorzystać z klasy `Balance`.

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {
```

```
        /* Ponieważ klasa Balance jest klasą publiczną, można
           jej użyć oraz wywołać jej konstruktor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // wywołanie metody show() także nie sprawi problemów
      }
    }
```

Jako eksperyment warto usunąć modyfikator `public` sprzed klasy `Balance` i spróbować ponownie skompilować kod. Kompilator zgłosi błąd.

Interfejsy

Za pomocą słowa kluczowego `interface` można w pełni oddzielić interfejs klasy od jej implementacji. Interfejs określa, co klasa musi robić, ale nie wskazuje, w jaki sposób ma te zadania wykonać. Interfejsy są składniowo podobne do klas, ale nie mogą się w nich pojawić zmienne składowe, a metody deklaruje się bez podawania jakiegokolwiek kodu. W ten sposób określa się metody, które klasa musi zaimplementować, ale nie wskazuje się w żaden sposób, jak tego dokonać. Dowolna liczba klas może implementować dany interfejs. Co więcej, jedna klasa może implementować wiele różnych interfejsów.

Aby zaimplementować interfejs, klasa musi zawierać wszystkie metody wskazane w interfejsie. Klasa może je implementować w dowolny sposób. Właśnie dzięki słowu kluczowemu `interface` język Java może w pełni realizować założenie „jeden interfejs, wiele metod", czyli jeden z aspektów polimorfizmu.

Interfejsy zostały tak zaprojektowane, by zapewniać dynamiczne dostosowywanie się do warunków w trakcie działania programów. W tradycyjnym rozwiązaniu wywołanie metody jednej klasy przez inną klasę wymaga sprawdzenia przez kompilator, czy sygnatura wywoływanej metody jest poprawna. Gdyby nie istniało nic innego poza tym wymogiem, Java zawierałaby statyczne i nierozszerzalne środowisko klas. W takim systemie programiści dążyliby do umieszczania właściwych implementacji coraz wyżej w hierarchii klas, tak aby odpowiednie rozwiązania były dostępne dla coraz większej liczby podklas. Interfejs pozwala rozwiązać ten problem, gdyż oddziela definicję metod od hierarchii dziedziczenia. Ponieważ interfejsy tworzą inną hierarchię niż klasy, klasy niepowiązane ze sobą w hierarchii dziedziczenia mogą implementować te same interfejsy. Właśnie w tym tkwi ogromny potencjał interfejsów Javy.


Uwaga

Interfejsy Javy oferują większość rozwiązań, które w innych językach programowania (na przykład C++) można osiągnąć wyłącznie przy użyciu mechanizmu dziedziczenia wielobazowego (wielodziedziczenia).

Definiowanie interfejsu

Interfejs definiuje się podobnie jak klasę. Oto ogólna postać takiej definicji.

```
dostęp interface nazwa {
  zwracany-typ nazwa-metody1(lista-parametrów);
  zwracany-typ nazwa-metody2(lista-parametrów);
  typ zmienna-finalna1 = wartość;
  typ zmienna-finalna2 = wartość;
  //...
  zwracany-typ nazwa-metodyN(lista-parametrów);
  typ zmienna-finalnaN = wartość;
}
```

W przypadku braku modyfikatora dostępu (elementu *dostęp*) stosuje się domyślny poziom dostępu, zatem interfejs jest dostępny tylko dla klas należących do tego samego pakietu. Zastosowanie modyfikatora `public` umożliwia stosowanie tego interfejsu w dowolnym kodzie. W takim przypadku jeden plik może zawierać tylko jeden interfejs publiczny, a nazwa tego pliku musi odpowiadać nazwie interfejsu. Element *nazwa* to nazwa interfejsu będąca dowolnym, poprawnym identyfikatorem. Zauważ, że definiowane metody nie zawierają żadnego kodu. Po liście parametrów pojawia się znak średnika. W zasadzie można powiedzieć, że są to metody abstrakcyjne, ponieważ w interfejsie nie może pojawić się implementacja żadnej z metod. Każda klasa związana z danym interfejsem musi implementować wszystkie jego metody.

Wewnątrz deklaracji interfejsu mogą pojawić się zmienne. Są one jednak niejawnie ustawiane na zmienne typu `final` i `static`, co oznacza, że nie mogą zostać zmienione przez implementującą je klasę. Co więcej, w trakcie ich deklarowania trzeba je zainicjalizować stałą. Dla wszystkich metod i zmiennych niejawnie jest stosowany modyfikator dostępu `public`.

Poniżej znajduje się przykład definicji interfejsu. Deklaruje on prosty interfejs, który zawiera jedną metodę o nazwie `callback()` przyjmującą jeden parametr typu `int`.

```
interface Callback {
  void callback(int param);
}
```

Implementacja interfejsu

Raz zdefiniowany interfejs może być implementowany przez wiele klas. Aby zaimplementować interfejs, trzeba dodać do klasy klauzulę `implements` oraz dodać wszystkie metody zdefiniowane w interfejsie. Ogólna postać klasy wykorzystującej interfejsy jest następująca.

```
dostęp class nazwaklasy [extends klasa-bazowa]
       [implements interfejs[, interfejs...]] {
  // ciało klasy
}
```

Jeśli klasa implementuje wiele interfejsów, ich nazwy oddziela się przecinkami. Jeżeli dwa implementowane interfejsy deklarują tę samą metodę, metoda ta będzie stosowana przez klientów obu tych interfejsów. Metody implementujące interfejs muszą być metodami publicznymi. Co więcej, sygnatura typów implementowanej metody i jej deklaracji z interfejsu musi być taka sama.

Oto krótki przykład klasy implementującej zdefiniowany wcześniej interfejs `Callback`.

```
class Client implements Callback {
  // implementuje interfejs Callback
  public void callback(int p) {
    System.out.println("wywołanie callback() z wartością " + p);
  }
}
```

Zauważ, że metoda `callback()` została zadeklarowana z modyfikatorem `public`.

> Implementowane metody publiczne muszą być deklarowane jako publiczne (z modyfikatorem `public`).

W pełni poprawne jest umieszczanie w klasie implementującej interfejs dodatkowych składowych, na przykład metod. Poniższa wersja klasy `Client` nie tylko implementuje metodę `callback()`, ale też dodaje własną metodę `nonIfaceMeth()`.

```
class Client implements Callback {
  // implementuje interfejs Callback
  public void callback(int p) {
    System.out.println("wywołanie callback() z wartością " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Klasa implementująca interfejs " +
                       "może zawierać także własne metody.");
  }
}
```

Dostęp do implementacji za pośrednictwem referencji do interfejsu

Zmienne reprezentujące referencje do obiektów można deklarować przy użyciu interfejsów (zamiast odpowiednich typów klasowych). Takiej zmiennej można przypisać egzemplarz dowolnej klasy implementującej zadeklarowany interfejs. Wywołanie metody dla takiej referencji spowoduje wywołanie odpowiedniej wersji metody należącej do właściwego egzemplarza klasy. Jest to bardzo ważny aspekt interfejsów. Metoda do wykonania jest wyszukiwana dynamicznie w trakcie działania programu, co oznacza, iż klasy można tworzyć później niż wywołujący je kod. Kod korzystający z metod interfejsu nie musi dysponować żadną wiedzą o obiekcie, którego będzie dotyczyło to wywołanie. Cała idea wywołań metod interfejsu jest bardzo podobna do wywołań metod przesłoniętych w podklasach klasy bazowej (patrz poprzedni rozdział).

> Ponieważ dynamiczne wyszukiwanie metody do wykonania wiąże się ze znacznymi kosztami w porównaniu z tradycyjnymi wywołaniami metody w Javie, należy unikać stosowania interfejsów w kodzie, który ma zapewniać przede wszystkim wysoką wydajność.

Następujący kod wywołuje metodę `callback()` przy użyciu zmiennej referencyjnej interfejsu.

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Wynik działania programu jest następujący.

```
wywołanie callback() z wartością 42
```

Zauważ, że zmienna `c` jest typu `Callback`, a mimo to został jej przypisany obiekt typu `Client`. Za pośrednictwem zmiennej `c` można wywołać tylko metodę `callback()` — nie można jej użyć do uzyskania dostępu do pozostałych składowych klasy `Client`. Innymi słowy, zmienna referencyjna interfejsu „wie" tylko o metodach, które pojawiły się w deklaracji interfejsu. Z tego względu zmiennej `c` nie uda się wywołać metody `nonIfaceMeth()`, gdyż została ona zdefiniowana w klasie `Client`, ale nie w interfejsie `Callback`.

Powyższy przykład ilustruje co prawda składnię korzystania ze zmiennej referencyjnej, której typ wskazuje interfejs (nie klasę), ale nie demonstruje pełnego potencjału tego rodzaju referencji w obszarze polimorfizmu. Aby lepiej zobrazować możliwości tego mechanizmu, warto opracować drugą implementację interfejsu `Callback`.

```
// Inna implementacja interfejsu Callback.
class AnotherClient implements Callback {
  // implementuje interfejs Callback
  public void callback(int p) {
    System.out.println("Inna wersja metody callback()");
    System.out.println("p podniesione do kwadratu to " + (p*p));
  }
}
```

Wykonaj kod poniższej klasy.

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // teraz c odnosi się do obiektu AnotherClient
    c.callback(42);
  }
}
```

Program wykonujący powyższy kod wygeneruje następujące wyniki.

```
wywołanie callback() z wartością 42
Inna wersja metody callback()
p podniesione do kwadratu to 1764
```

Łatwo zauważyć, że wywoływana wersja metody `callback()` jest identyfikowana dynamicznie w zależności od typu obiektu wskazywanego przez zmienną `c` w czasie wykonywania. Choć jest to bardzo prosty przykład, wkrótce przedstawię bardziej praktyczne zastosowanie.

Implementacja częściowa

Klasa, która dołącza pewien interfejs, ale nie implementuje wszystkich zdefiniowanych w nim metod, musi zostać zadeklarowana jako klasa abstrakcyjna. Oto przykład.

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  //...
}
```

Ponieważ klasa `Incomplete` nie implementuje metody `callback()`, musi zostać zadeklarowana jako klasa abstrakcyjna (modyfikator `abstract`). Klasy dziedziczące po `Incomplete` muszą albo zaimplementować metodę `callback()`, albo same zostać zadeklarowane jako abstrakcyjne.

Interfejsy zagnieżdżone

Interfejs można zadeklarować w formie składowej klasy lub innego interfejsu. Taki interfejs określa się mianem **interfejsu składowego** lub **interfejsu zagnieżdżonego**. Interfejs zagnieżdżony można zadeklarować z modyfikatorem dostępu `public`, `private` lub `protected`. W tym aspekcie interfejsy zagnieżdżone różnią się od interfejsów najwyższego poziomu, które — jak już wspomniano — należy deklarować albo z modyfikatorem `public`, albo bez modyfikatora dostępu (wówczas jest stosowany domyślny poziom dostępu). Interfejs zagnieżdżony stosowany poza zasięgiem, w którym został zadeklarowany, musi być identyfikowany przez pełną nazwę, tj. poprzedzony nazwą klasy (lub interfejsu), której jest członkiem. Oznacza to, że poza klasą lub interfejsem, w którym zadeklarowano interfejs zagnieżdżony, należy stosować pełną nazwę tego interfejsu.

Oto prosty przykład ilustrujący interfejs zagnieżdżony:

```
// Przykład interfejsu zagnieżdżonego.

// Klasa A zawiera interfejs składowy.
class A {
  // to jest interfejs zagnieżdżony
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// Klasa B implementuje interfejs zagnieżdżony.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}

class NestedIFDemo {
  public static void main(String args[]) {

    // użycie referencji typu interfejsu zagnieżdżonego
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("liczba 10 nie jest ujemna ");
    if(nif.isNotNegative(-12))
```

```
        System.out.println("to nie zostanie wyświetlone ");
    }
}
```

Jak widać, klasa `A` definiuje interfejs składowy nazwany `NestedIF` i zadeklarowany jako interfejs publiczny. Poniższy zapis oznacza, że klasa `B` implementuje ten interfejs zagnieżdżony:

```
implements A.NestedIF
```

Warto zwrócić uwagę na pełną nazwę tego interfejsu (obejmującą nazwę klasy, do której należy). W ciele metody `main()` utworzono zmienną referencyjną typu `A.NestedIF` nazwaną `nif`, po czym przypisano tej zmiennej referencję obiektu klasy `B`. Takie rozwiązanie jest dopuszczalne, ponieważ klasa `B` implementuje interfejs `A.NestedIF`.

Stosowanie interfejsów

Aby lepiej zrozumieć siłę drzemiącą w interfejsach, warto przyjrzeć się bardziej praktycznemu przykładowi. We wcześniejszych rozdziałach pojawiła się klasa o nazwie `Stack`, która implementowała stos o określonym rozmiarze. Istnieje wiele różnych sposobów implementacji stosu. Na przykład stos może mieć stały rozmiar lub rozszerzać się. Co więcej, wartości ze stosu można przechowywać w tablicy, liście, drzewie binarnym itp. Niezależnie od implementacji interfejs stosu zawsze pozostaje taki sam. Innymi słowy, metody `pop()` i `push()` działają tak samo niezależnie od sposobu przechowywania danych na stosie. Z tego względu warto wydzielić interfejs stosu, a implementację pozostawić wyspecjalizowanym klasom. Przyjrzyjmy się dwóm przykładom.

Pierwszy kod to interfejs definiujący stos liczb całkowitych. Kod należy umieścić w pliku źródłowym *IntStack.java*. Interfejs zostanie wykorzystany przez dwie różne implementacje stosu.

```
// Definicja interfejsu dla stosu liczb całkowitych.
interface IntStack {
  void push(int item); // zapamiętanie elementu
  int pop(); // pobranie elementu
}
```

Następujący program tworzy klasę o nazwie `FixedStack`, która implementuje stos o stałym rozmiarze.

```
// Implementacja IntStack używająca tablicy o stałym rozmiarze.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // alokacja i inicjalizacja stosu
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // umieszczenie elementu na szczycie stosu
  public void push(int item) {
    if(tos==stck.length-1) // użycie zmiennej składowej length
```

```
      System.out.println("Stos jest pełny.");
    else
      stck[++tos] = item;
  }

  // zdjęcie elementu ze szczytu stosu
  public int pop() {
    if(tos < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // umieszczenie pewnych liczb na stosach
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // zdjęcie liczb ze stosów
    System.out.println("Stos w mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stos w mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```

Poniżej znajduje się druga implementacja interfejsu `IntStack`, która tworzy stos o dynamicznie zmieniającym się rozmiarze. W tej implementacji tworzony jest stos z pewnym rozmiarem początkowym. Jeśli zostanie on przekroczony, stos jest rozszerzany — jego rozmiar wzrasta dwukrotnie.

```
// Implementacja rozszerzającego się stosu.
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // alokacja i inicjalizacja stosu
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // umieszczenie elementu na szczycie stosu
  public void push(int item) {
    // jeśli stos jest pełny, alokacja nowej tablicy o większym rozmiarze
    if(tos==stck.length-1) {
```

```
      int temp[] = new int[stck.length * 2]; // podwojenie rozmiaru
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // zdjęcie elementu ze szczytu stosu
  public int pop() {
    if(tos < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // pętle powodują zwiększenie rozmiaru stosu
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Stos w mystack1:");
    for(int i=0; i<12; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stos w mystack2:");
    for(int i=0; i<20; i++)
       System.out.println(mystack2.pop());
  }
}
```

Poniższy kod programu korzysta z obu implementacji: `FixedStack` i `DynStack`. Korzysta przy tym ze zmiennej referencyjnej interfejsu. Oznacza to, że wywoływane metody `pop()` i `push()` są identyfikowane dynamicznie (w czasie wykonywania programu) zamiast w trakcie kompilacji.

```
/* Utworzenie zmiennej referencyjnej interfejsu i
   użycie jej do dostępu do stosu.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // utworzenie zmiennej referencyjnej interfejsu
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // załadowanie stosu dynamicznego
    // umieszczenie wartości na stosie
    for(int i=0; i<12; i++) mystack.push(i);
```

```
    mystack = fs; // załadowanie stosu o stałym rozmiarze
    for(int i=0; i<8; i++) mystack.push(i);

    mystack = ds;
    System.out.println("Wartości na stosie dynamicznym:");
    for(int i=0; i<12; i++)
      System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Wartości na stosie o stałym rozmiarze:");
    for(int i=0; i<8; i++)
      System.out.println(mystack.pop());
  }
}
```

W tym programie zmienna `mystack` jest typu `IntStack`. Jeśli więc zawiera taką samą wartość jak `ds`, korzysta z metod `push()` i `pop()` zdefiniowanych dla klasy `DynStack`. Jeśli zawiera taką samą wartość jak `fs`, korzysta z metod `push()` i `pop()` zdefiniowanych dla klasy `FixedStack`. Jak już wspomniałem, wybór wersji metody dokonywany jest w sposób dynamiczny w trakcie wykonywania programu. Dostęp do wielu implementacji interfejsu przez zmienne referencyjne to najbardziej użyteczny sposób wykorzystania polimorfizmu w języku Java.

Zmienne w interfejsach

Można użyć interfejsów do wprowadzenia w wielu klasach tych samych stałych. Wystarczy w interfejsie zadeklarować i zainicjalizować zmienne. Klasa implementująca taki interfejs będzie w swoim zasięgu „widziała" wszystkie te zmienne jako stałe. (Przypomina to tworzenie stałych dyrektywą `#define` lub deklaracją `const` w językach C lub C++). Jeśli interfejs nie zawiera żadnych metod, wtedy implementująca go klasa nie musi deklarować żadnych dodatkowych metod. Zmienne występujące w interfejsie są widziane przez klasę korzystającą z interfejsu jako zmienne typu `final`. Kolejny przykład używa interfejsu do zaimplementowania automatu podejmującego decyzje.

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
      return NO;           // 30%
    else if (prob < 60)
      return YES;          // 30%
    else if (prob < 70)
      return MAYBE;        // 10%
```

```
    else if (prob < 85)
      return LATER;         // 15%
    else if (prob < 98)
      return SOON;          // 13%
    else
      return NEVER;         // 2%
  }
}

class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("Nie");
        break;
      case YES:
        System.out.println("Tak");
        break;
      case MAYBE:
        System.out.println("Może");
        break;
      case LATER:
        System.out.println("Później");
        break;
      case SOON:
        System.out.println("Wkrótce");
        break;
      case NEVER:
        System.out.println("Nigdy");
        break;
    }
  }

  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Zauważ, że program korzysta z jednej ze standardowych klas Javy — klasy `Random`. Klasa generuje liczby pseudolosowe. Zawiera kilka metod zwracających wylosowane liczby w formacie najbardziej odpowiednim dla danego programu. W tym przykładzie stosujemy metodę `nextDouble()`, która zwraca losową wartość zmiennoprzecinkową w przedziale od `0.0` do `1.0`.

W przykładowym programie dwie klasy, `Question` i `AskMe`, implementują interfejs `Shared` ➥`Constants` definiujący stałe `YES`, `NO`, `MAYBE`, `SOON`, `LATER` i `NEVER`. Kod wewnątrz klas korzysta z tych zmiennych w taki sposób, jakby były one zadeklarowane w tej klasie lub odziedziczone po innej klasie. Zauważ, że wyniki działania programu są różne po każdym jego uruchomieniu.

```
Wkrótce
Później
Tak
Nie
```

Interfejsy można rozszerzać

Jeden interfejs może dziedziczyć po innym interfejsie. Wystarczy w tym celu użyć słowa kluczowego `extends`. Składnia jest dokładnie taka sama jak w przypadku dziedziczenia po klasie. Gdy klasa implementuje interfejs dziedziczący po innym interfejsie, musi zapewnić wszystkie metody zdefiniowane w całym łańcuchu dziedziczenia. Oto przykład.

```
// Jeden interfejs rozszerza inny.
interface A {
  void meth1();
  void meth2();
}

// teraz B zawiera meth1() i meth2() oraz dodaje własne meth3()
interface B extends A {
  void meth3();
}

// klasa musi zaimplementować wszystkie metody zdefiniowane w interfejsach 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();
  }
}
```

Jako eksperyment można spróbować usunąć implementację metody `meth1()` z klasy `MyClass`. Spowoduje to zgłoszenie błędu kompilacji. Dowolna klasa implementująca dany interfejs musi zdefiniować kod wszystkich metod interfejsu, także tych, które interfejs odziedziczył po innym interfejsie.

Choć przykłady przedstawione w książce niezbyt często korzystają z pakietów i interfejsów, w rzeczywistych programach Javy oba elementy są niezwykle ważne. W zasadzie wszystkie poważne programy Javy umieszcza się w pakietach. Większość z tych programów definiuje własne interfejsy. Warto więc dobrze zapoznać się z tymi kwestiami.

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

Z

Ź

Ż

ORACLE

Java

Wydanie VIII

Kompendium programisty

Podobnie jak wcześniejsze popularne języki programowania komputerów, Java to mieszanka najlepszych elementów swoich poprzedników. Jej dodatkową zaletą jest innowacyjna koncepcja samego języka, wynikająca z unikatowej misji. Obecnie Java jest bardzo chętnie wykorzystywana we wszelkich projektach informatycznych. Silne typowanie, jasno określona, przejrzysta składnia oraz ogromna liczba bibliotek rozwiązujących najbardziej wymyślne problemy zjednują jej wielu zwolenników.

Książka, którą trzymasz w ręku, jest unikalną pozycją, gdyż jako jedna z pierwszych omawia nowości z ostatniego wydania języka Java, oznaczonego numerem 7. Znajdziesz tu informacje na temat nowego API do obsługi operacji wejścia-wyjścia, wykorzystania łańcuchów znaków w wyrażeniach *switch* oraz wychwytywania wielu wyjątków w ramach jednego bloku *catch*. Poza nowościami autor omawia konstrukcje obecne w języku od lat. To idealna propozycja dla każdego programisty Javy, który treści zawarte w książce może traktować wybiórczo i sięgać po nią, gdy będzie miał wątpliwości co do sposobu rozwiązania konkretnego problemu. Natomiast dla osób, które chcą poznać język Java i wkroczyć w świat zaawansowanych technologii, będzie ona pasjonującym przewodnikiem.

W trakcie lektury:

- poznasz nowości w najnowszym wydaniu Javy
- zaznajomisz się z historią tego języka
- nauczysz się podstaw składni
- stworzysz aplikację internetową
- poznasz mocne i słabe strony Javy

Kompletne źródło informacji na temat języka Java!

ISBN 978-83-246-3767-6

Cena 179,00 zł

Informatyka w najlepszym wydaniu