O'REILLY®

Wydanie VI

Java w pigułce

POZNAJ NOWOŚCI JĘZYKA JAVA!

Benjamin J. Evans
David Flanagan

Helion

Tytuł oryginału: Java in a Nutshell, 6th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0623-3

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/javpi6.zip

Spis treści

Przedmowa **9**

Wstęp **11**

Część I. Wprowadzenie do języka Java 17

1. Wprowadzenie do środowiska Java **19**
Język, maszyna wirtualna i środowisko 19
Historia Javy i maszyny wirtualnej Javy w zarysie 23
Cykl istnienia programu Java 24
Bezpieczeństwo Javy 26
Porównanie Javy z innymi językami programowania 27
Krytyka Javy 28

2. Składnia Javy od podstaw **31**
Budowa programu w Javie 32
Struktura leksykalna 32
Podstawowe typy danych 35
Wyrażenia i operatory 42
Instrukcje 55
Metody 71
Podstawowe wiadomości o klasach i obiektach 77
Tablice 81
Typy referencyjne 87
Pakiety i przestrzenie nazw 91
Struktura plików Javy 95
Definiowanie i uruchamianie programów Java 96
Podsumowanie 97

3. Programowanie obiektowe w Javie 99
Podstawowe wiadomości o klasach 99
Pola i metody 101
Tworzenie i inicjowanie obiektów 107
Podklasy i dziedziczenie 111
Ukrywanie danych i hermetyzacja 119
Klasy i metody abstrakcyjne 125
Podsumowanie wiadomości o modyfikatorach 129

4. System typów Javy 131
Interfejsy 131
Typy ogólne 137
Wyliczenia i adnotacje 145
Typy zagnieżdżone 148
Wyrażenia lambda 162
Podsumowanie 165

5. Podstawy projektowania obiektowego w Javie 167
Wartości w języku Java 167
Ważne metody klasy java.lang.Object 168
Aspekty projektowania obiektowego 172
Wyjątki i ich obsługa 180
Bezpieczne programowanie w Javie 182

6. Zarządzanie pamięcią i współbieżność w Javie 185
Podstawowe pojęcia zarządzania pamięcią w Javie 185
Optymalizacja procesu usuwania nieużywanych obiektów w maszynie wirtualnej 188
Sterta maszyny wirtualnej HotSpot 191
Finalizacja 193
Mechanizmy współbieżności w Javie 195
Praca z wątkami 203
Podsumowanie 205

Część II. Praca na platformie Java 207

7. Zwyczaje programistyczne i tworzenie dokumentacji 209
Konwencje nazewnicze i dotyczące stosowania wielkich liter 209
Nadawanie nazw w praktyce 211
Komentarze dokumentacyjne 212
Porady na temat pisania programów przenośnych 219

8. Praca z kolekcjami i tablicami w Javie 223
Wprowadzenie do API Collections 223
Wyrażenia lambda w kolekcjach Javy 239
Podsumowanie 245

9. Obsługa najczęściej używanych formatów danych 247
Tekst 247
Liczby i matematyka 253
Data i godzina w Javie 8 258
Podsumowanie 263

10. Obsługa plików oraz wejścia i wyjścia 265
Klasyczny system wejścia i wyjścia Javy 265
Nowy system wejścia i wyjścia 270
Kanały i bufory NIO 273
Asynchroniczny system wejścia i wyjścia 275
Sieć 278

11. Ładowanie klas, refleksja oraz uchwyty do metod 283
Pliki klas, obiekty klas i metadane 283
Fazy ładowania klasy 285
Bezpieczne programowanie i ładowanie klas 287
Ładowanie klas w praktyce 289
Refleksja 292
Dynamiczne klasy pośredniczące 295
Uchwyty do metod 297

12. Nashorn 301
Wprowadzenie do Nashorna 301
Wykonywanie kodu JavaScript w Nashornie 302
Nashorn i pakiet javax.script 308
Nashorn dla zaawansowanych 310
Podsumowanie 315

13. Narzędzia platformy i profile 317
Narzędzia wiersza poleceń 317
Narzędzie VisualVM 329
Profile Java 8 334
Podsumowanie 339

Skorowidz 341

ROZDZIAŁ 4.

System typów Javy

W tym rozdziale przechodzimy od podstaw programowania obiektowego przy użyciu klas do dalszych pojęć, których znajomość również jest potrzebna każdemu, kto chce efektywnie wykorzystywać statyczny system typów języka Java.

Język programowania **o statycznej kontroli typów** to taki, w którym zmienne mają określone typy i przypisanie wartości do zmiennej nieodpowiedniego typu powoduje błąd kompilacji. Przykładem takiego języka jest Java. Języki, w których zgodność typów jest sprawdzana dopiero w czasie działania programu, nazywają się językami **o dynamicznej kontroli typów**. Przykładem takiego języka jest JavaScript.

System typów Javy składa się nie tylko z klas i typów podstawowych, ale również z innych rodzajów typów referencyjnych, które są związane z podstawową koncepcją klas, tylko różnią się od nich pod różnymi względami i zazwyczaj są traktowane przez kompilator `javac` lub maszynę wirtualną w specjalny sposób.

W poprzednich rozdziałach opisaliśmy już tablice i klasy, czyli dwa najczęściej używane rodzaje typów referencyjnych języka Java. W tym rozdziale zaczynamy opis kolejnego ważnego typu referencyjnego — **interfejsów**. Potem przejdziemy do typów ogólnych (generycznych), które również pełnią bardzo ważną funkcję w systemie typów Javy. Uzbrojeni w tę wiedzę przejdziemy do różnic między typami czasu kompilacji i czasu wykonywania programu.

Aby dopełnić obrazu typów referencyjnych języka Java, przyjrzymy się też dwóm specjalnym rodzajom klas i interfejsów — **wyliczeniom** i **adnotacjom**. Na końcu rozdziału dowiesz się, czym są **typy zagnieżdżone**, i poznasz wprowadzone w Javie 8 **wyrażenia lambda**.

Zaczniemy od interfejsów, które są chyba najważniejszym po klasach typem referencyjnym w Javie i stanowią kluczowy składnik budowy całego systemu typów tego języka.

Interfejsy

W rozdziale 3. opisaliśmy pojęcie dziedziczenia. Napisaliśmy też, że w Javie każda klasa może dziedziczyć bezpośrednio tylko po jednej klasie. Jest to dość poważne ograniczenie dla rodzajów programów obiektowych, jakie można pisać. Projektanci Javy zdawali sobie z tego sprawę, ale chcieli, aby mechanizmy obiektowości w ich języku były prostsze niż np. w języku C++.

Postanowili więc zastosować inne rozwiązanie — o nazwie interfejs. **Interfejs** podobnie jak klasa stanowi definicję nowego typu referencyjnego. Zgodnie z nazwą reprezentuje on tylko interfejs API — a więc dostarcza opisu typu i metod (oraz sygnatury), które klasy **implementujące** go powinny definiować.

Ogólnie rzecz biorąc, interfejs w Javie nie dostarcza żadnego kodu implementacyjnego dla opisywanych przez siebie metod. Ich implementacja jest **obowiązkowa** dla każdej klasy, która implementuje dany interfejs.

Ale interfejs może też zawierać metody oznaczone jako opcjonalne, które nie muszą zostać zdefiniowane w klasie implementującej ten interfejs. Do oznaczania takich metod służy słowo kluczowe `default`. Metody oznaczone tym modyfikatorem muszą mieć w interfejsie **domyślną** definicję, która będzie używana we wszystkich implementacjach, w których nie zostaną zdefiniowane.

Możliwość tworzenia metod opcjonalnych w interfejsach jest nowością wprowadzoną w Javie 8. Szerzej na temat metod opcjonalnych interfejsów piszemy w podrozdziale „Metody domyślne”.

Nie da się bezpośrednio utworzyć obiektu interfejsu ani składowej typu interfejsowego. Zamiast tego należy utworzyć klasę **implementującą** dany interfejs i w niej zdefiniować niezbędne metody.

Wszystkie egzemplarze takiej klasy są obiektami zarówno typu zdefiniowanego przez tę klasę, jak i typu zdefiniowanego przez interfejs. Obiekty nienależące do tej samej klasy lub nadklasy i tak mogą być tego samego typu dzięki implementacji tego samego interfejsu.

Definiowanie interfejsu

Definicja interfejsu wygląda bardzo podobnie do definicji klasy, w której wszystkie (niebędące domyślnymi) metody są abstrakcyjne, a słowo kluczowe `class` zastąpiono słowem `interface`. Na przykład poniżej znajduje się definicja interfejsu o nazwie `Centered`. Klasa `Shape`, np. taka jak przedstawiona w przykładach w rozdziale 3., może implementować ten interfejs, jeśli programista chce umożliwić ustawianie i sprawdzanie współrzędnych środka:

```
interface Centered {
  void setCenter(double x, double y);
  double getCenterX();
  double getCenterY();
}
```

Składowe interfejsu podlegają pewnym ograniczeniom:

- Wszystkie obowiązkowe metody interfejsu standardowo są abstrakcyjne i zamiast treści muszą mieć tylko średnik. Można stosować modyfikator `abstract`, ale zwyczajowo się go opuszcza.
- Interfejs definiuje publiczny interfejs API. Wszystkie składowe są więc publiczne i zwyczajowo opuszcza się modyfikator `public`. Definicja metody chronionej lub prywatnej w interfejsie jest błędem, który zostanie wychwycony w czasie kompilacji.

- Interfejs nie może zawierać definicji pól egzemplarzowych. Pola są szczegółem implementacyjnym, a interfejs jest specyfikacją, nie implementacją. W interfejsach dopuszczalne są jedynie stałe, zarazem statyczne, jak i finalne.
- Nie można tworzyć egzemplarzy interfejsów, więc interfejs nie może zawierać konstruktora.
- Interfejsy mogą zawierać typy zagnieżdżone, które domyślnie są statyczne i publiczne. Szczegółowy opis typów zagnieżdżonych znajduje się w podrozdziale „Typy zagnieżdżone".
- Od Javy 8 interfejs może zawierać metody statyczne. W poprzednich wersjach języka było to niedozwolone i powszechnie uważano to za wadę projektową tego języka programowania.

Rozszerzanie interfejsów

Interfejs może rozszerzać inny interfejs i podobnie jak klasa może w definicji zawierać klauzulę `extends`. Gdy jeden interfejs rozszerza inny, dziedziczy po nim wszystkie metody i stałe oraz może zawierać dodatkowe własne metody i stałe. Ale w odróżnieniu od klas klauzula `extends` interfejsu może zawierać więcej niż jedną nazwę nadinterfejsu. Poniżej znajduje się kilka przykładów interfejsów rozszerzających inne interfejsy:

```
interface Positionable extends Centered {
  void setUpperRightCorner(double x, double y);
  double getUpperRightX();
  double getUpperRightY();
}
interface Transformable extends Scalable, Translatable, Rotatable {}
interface SuperShape extends Positionable, Transformable {}
```

Interfejs rozszerzający kilka interfejsów dziedziczy wszystkie metody i stałe każdego z nich oraz może dodatkowo zawierać własne metody i stałe. Klasa implementująca taki interfejs musi implementować metody abstrakcyjne zdefiniowane bezpośrednio przez ten interfejs i wszystkie odziedziczone ze wszystkich nadinterfejsów.

Implementowanie interfejsu

Podobnie jak za pomocą słowa kluczowego `extends` oznacza się nadklasę klasy, za pomocą słowa kluczowego `implements` oznacza się interfejsy, które ona implementuje. Słowo kluczowe `implements` może się znajdować w definicji klasy za klauzulą `extends`. Po nim wypisuje się rozdzielaną przecinkami listę implementowanych przez klasę interfejsów.

Deklaracja interfejsu w klauzuli `implements` klasy oznacza, że dana klasa implementuje (tzn. zawiera treść główną definicji) każdą obowiązkową metodę tego interfejsu. Jeżeli klasa implementuje interfejs, ale nie dostarcza implementacji wszystkich jego obowiązkowych metod, to dziedziczy te metody abstrakcyjne i sama musi być zadeklarowana jako abstrakcyjna. Jeśli klasa implementuje więcej niż jeden interfejs, musi implementować wszystkie obowiązkowe metody każdego z tych interfejsów (albo być zadeklarowana jako abstrakcyjna).

Poniżej znajduje się definicja klasy `CenteredRectangle`, która rozszerza klasę `Rectangle` z rozdziału 3. i implementuje interfejs `Centered`:

```
public class CenteredRectangle extends Rectangle implements Centered {
  // nowe pola egzemplarzowe
  private double cx, cy;

  // konstruktor
```

```
  public CenteredRectangle(double cx, double cy, double w, double h) {
    super(w, h);
    this.cx = cx;
    this.cy = cy;
  }

  // Klasa dziedziczy wszystkie metody klasy Rectangle, ale musi zawierać implementacje wszystkich metod interfejsu Centered.
  public void setCenter(double x, double y) { cx = x; cy = y; }
  public double getCenterX() { return cx; }
  public double getCenterY() { return cy; }
}
```

Załóżmy, że w podobny sposób zaimplementowaliśmy też klasy `CenteredCircle` i `CenteredSquare`. Każda z nich rozszerza klasę `Shape`, więc egzemplarze tych klas można traktować jak egzemplarze klasy `Shape`. Jako że każda z nich implementuje interfejs `Centered`, to ich egzemplarze można też traktować jako obiekty tego typu. W poniższym przykładzie pokazano, że obiekty mogą być jednocześnie typu klasowego i interfejsowego:

```
Shape[] shapes = new Shape[3];  // tworzy tablicę do przechowywania kształtów

// tworzy kilka wycentrowanych kształtów i zapisuje je w tablicy Shape[]
// Nie trzeba stosować rzutowania: wszystko to są konwersje rozszerzające.
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);
// oblicza średnie pole powierzchni kształtów i
// średnią odległość od początku układu współrzędnych
double totalArea = 0;
double totalDistance = 0;
for( int i = 0; i < shapes.length; i++) {
  totalArea += shapes[i].area();  // oblicza pole powierzchni kształtów

  // Uwaga: zasadniczo użycie operatora instanceof w celu sprawdzenia
  // typu czasu wykonywania obiektu często jest oznaką problemów projektowych programu.
  if (shapes[i] instanceof Centered) { // Kształt jest typu Centered.
    // Zwróć uwagę na rzutowanie z typu Shape na Centered (nie byłoby potrzebne
    // do konwersji z typu CenteredSquare na Centered).
    Centered c = (Centered) shapes[i];

    double cx = c.getCenterX();  // pobranie współrzędnych środka
    double cy = c.getCenterY();  // obliczenie odległości od początku układu
    totalDistance += Math.sqrt(cx*cx + cy*cy);
  }
}
System.out.println("Średnie pole powierzchni: " + totalArea/shapes.length);
System.out.println("Średnia odległość: " + totalDistance/shapes.length);
```

W Javie interfejsy, podobnie jak klasy, są typami danych. Gdy klasa implementuje interfejs, to egzemplarze tej klasy można przypisywać do zmiennych typu tego interfejsu.

Nie należy interpretować tego przykładu w ten sposób, że konieczne jest przypisanie obiektu typu `CenteredRectangle` do zmiennej typu `Centered` przed wywołaniem metody `setCenter()` ani do zmiennej typu `Shape` przed wywołaniem metody `area()`. Klasa `CenteredRectangle` zawiera definicje tych metod odziedziczone po nadklasie `Rectangle`, więc można je wywoływać w dowolnym momencie.

Implementowanie kilku interfejsów

Załóżmy, że potrzebne są obiekty kształtów, które można pozycjonować nie tylko względem punktu środkowego, ale i prawego górnego rogu. Dodatkowo powiedzmy, że potrzebna jest możliwość zmniejszania i zwiększania tych kształtów. Przypomnijmy, że wprawdzie klasa może bezpośrednio dziedziczyć tylko po jednej klasie, ale może implementować dowolną liczbę interfejsów. Przy założeniu, że istnieją odpowiednie interfejsy `UpperRightCornered` i `Scalable`, klasę o żądanych właściwościach można zadeklarować w następujący sposób:

```
public class SuperDuperSquare extends Shape
  implements Centered, UpperRightCornered, Scalable {
  // Składowe klasy zostały pominięte.
}
```

Implementacja kilku interfejsów przez klasę oznacza tylko tyle, że klasa ta musi zawierać implementacje wszystkich abstrakcyjnych (obowiązkowych) metod ze wszystkich tych interfejsów.

Metody domyślne

W języku Java 8 pojawiła się możliwość definiowania w interfejsach metod zawierających implementację. Są to metody opcjonalne w reprezentowanym przez interfejs API i najczęściej nazywa się je **metodami domyślnymi**. Ich poznawanie zaczniemy od przeanalizowania powodów, dla których są one w ogóle potrzebne.

Zgodność wsteczna

Twórcy platformy Java od zawsze dużo uwagi poświęcają kwestiom zgodności nowych wersji ze starszymi. To sprawia, że kod napisany (a nawet skompilowany) dla starszej wersji platformy musi działać na nowszych wersjach. Zasada ta daje programistom pewność, że nowe wydanie pakietu JDK lub środowiska JRE nie spowoduje, iż aktualnie działający program nagle przestanie działać.

Zgodność wsteczna jest wielką zaletą platformy Java, ale jej utrzymanie wiąże się z pewnymi niedogodnościami. Jedną z nich jest to, że do interfejsów nie można dodawać nowych obowiązkowych metod.

Powiedzmy np., że chcemy wzbogacić interfejs `Positionable` o możliwość określania także pozycji lewego dolnego rogu:

```
public interface Positionable extends Centered {
  void setUpperRightCorner(double x, double y);
  double getUpperRightX();
  double getUpperRightY();
  void setLowerLeftCorner(double x, double y);
  double getLowerLeftX();
  double getLowerLeftY();
}
```

Jeśli po wprowadzeniu tych zmian spróbujemy użyć tego interfejsu z wcześniej napisanym kodem, to kod ten nie zadziała, ponieważ będzie w nim brakowało obowiązkowych metod `setLowerLeftCorner()`, `getLowerLeftX()` i `getLowerLeftY()`.

Opisywany problem można łatwo zaobserwować we własnym kodzie. Wystarczy skompilować plik klasy implementującej interfejs, a potem dodać do tego interfejsu nową obowiązkową metodę i spróbować uruchomić program z tą nową wersją interfejsu i starym plikiem klasy. Powinna nastąpić awaria programu z błędem `NoClassDefError`.

Ograniczenie to stanowiło problem dla projektantów Javy 8, ponieważ jednym z ich celów było uaktualnienie rdzennych bibliotek kolekcji i wprowadzenie metod używających wyrażeń lambda.

Rozwiązanie wymagało wprowadzenia nowego mechanizmu umożliwiającego modyfikowanie interfejsów o dodatek nowych opcjonalnych metod bez uniemożliwiania działania starym programom.

Implementacja metod domyślnych

Aby dodać nowe metody do interfejsu i nie spowodować niezgodności ze starszymi programami, konieczne było dostarczenie implementacji tych metod, która pozwoliłaby tym programom działać tak jak dotychczas. Zastosowano rozwiązanie w postaci metod domyślnych, które wprowadzono w Javie 8.

Metodę domyślną (czasami nazywaną metodą opcjonalną) można dodać do każdego interfejsu. Musi ona zawierać tzw. **implementację domyślną**, którą wpisuje się bezpośrednio w definicji interfejsu.

Podstawowe cechy metod domyślnych są następujące:

- Klasa implementująca może implementować metodę domyślną, ale nie musi tego robić.
- Jeżeli klasa implementuje metodę domyślną, to używana jest implementacja z tej klasy.
- Jeśli nie zostanie znaleziona inna implementacja metody, stosowana jest implementacja domyślna.

Przykładem metody domyślnej jest metoda `sort()`. Została ona dodana do interfejsu `java.util.List` w JDK 8, a jej definicja wygląda tak:

```
// Napis <E> to sposób zapisu typów ogólnych w Javie — szerzej na ich
// temat piszemy w następnym podrozdziale. Jeśli nie wiesz, czym są typy ogólne,
// na razie zignoruj ten szczegół.
interface List<E> {
  // pozostałe składowe pominięto

  public default void sort(Comparator<? super E> c) {
    Collections.<E>sort(this, c);
  }
}
```

W efekcie od Javy 8 każdy obiekt implementujący interfejs `List` ma metodę egzemplarzową `sort()`, za pomocą której można posortować listę przy użyciu odpowiedniego komparatora. Jako że typ zwrotny to `void`, można podejrzewać, iż metoda ta wykonuje sortowanie na miejscu, i jest to słuszne podejrzenie.

Interfejsy znacznikowe

Czasami potrzebny jest kompletnie pusty interfejs. Jego implementacja przez klasę polega tylko na wymienieniu jego nazwy w klauzuli `implements`, bez konieczności pisania definicji jakichkolwiek metod. Sprawia to, że egzemplarze tej klasy automatycznie stają się też egzemplarzami tego interfejsu. W Javie można sprawdzić, czy dany obiekt jest egzemplarzem wybranego interfejsu, za pomocą operatora `instanceof`, więc technika ta pozwala na dostarczenie pewnych dodatkowych informacji o obiekcie.

Przykładem interfejsu znacznikowego jest `java.io.Serializable`. Implementuje się go w klasie po to, by poinformować strumień `ObjectOutputStream`, że egzemplarze tej klasy można bezpiecznie serializować. Innym przykładem jest interfejs `java.util.RandomAccess`. Implementują go niektóre implementacje interfejsu `java.util.List`, aby informować, że dają szybki dostęp swobodny do elementów listy. Na przykład klasa `ArrayList` implementuje interfejs `RandomAccess`, a klasa `LinkedList` nie. W algorytmach, w których ważna jest szybkość operacji dostępu swobodnego, można sprawdzać implementację interfejsu `RandomAccess` w następujący sposób:

```
// Przed posortowaniem elementów dowolnie długiej tablicy warto
// sprawdzić, czy umożliwia ona szybki swobodny dostęp do elementów.
// Jeśli nie, to szybszym rozwiązaniem może być wykonanie kopii z szybkim
// dostępem losowym i potem wykonanie sortowania. Nie jest to jednak konieczne,
// gdy używa się metody java.util.Collections.sort().
List l = ...;  // jakaś lista
if (l.size() > 2 && !(l instanceof RandomAccess)) l = new ArrayList(l);
sortListInPlace(l);
```

Jak pokażemy później, system typów Javy jest bardzo ściśle powiązany z nazwami typów — rozwiązanie to nazywa się **typowaniem nominalnym** (ang. *nominal typing*). Dobrym przykładem tego jest interfejs znacznikowy, który nie ma nic **oprócz** nazwy.

Typy ogólne

Jedną z wielkich zalet platformy Java jest jej biblioteka standardowa. Zawiera ona wiele bardzo przydatnych elementów, a w szczególności solidne implementacje często używanych struktur danych. Ich obsługa jest w miarę łatwa, a poza tym istnieje dobra dokumentacja. Biblioteki te nazywają się kolekcjami Javy (ang. *Java Collections*) i poświęciliśmy im dużą część rozdziału 8. Jeśli jednak ktoś potrzebuje znacznie obszerniejszego materiału, to może zaopatrzyć się w książkę *Java Generics and Collections* Maurice'a Naftalina i Philipa Wadlera (O'Reilly).

Wprawdzie wczesne wersje kolekcji też były przydatne, ale miały pewną poważną wadę. Polegała ona na tym, że struktura danych (często nazywana **kontenerem**) w istocie ukrywała typ przechowywanych w niej danych.

Ukrywanie i hermetyzacja to fundamenty programowania obiektowego, ale w tym przypadku nieprzezroczystość kontenera sprawiała programistom wiele problemów.

Zaczniemy od naświetlenia tego problemu i pokażemy, jak wprowadzenie typów ogólnych pozwoliło go rozwiązać i ułatwić pracę rzeszom programistów Javy.

Wprowadzenie do typów ogólnych

Jeśli ktoś chce utworzyć kolekcję egzemplarzy klasy `Shape`, to do ich przechowywania może użyć struktury danych `List`:

```
List shapes = new ArrayList();   // utworzenie listy do przechowywania kształtów

// utworzenie paru wycentrowanych kształtów i zapisanie ich w liście
shapes. add( new CenteredCircle(1.0, 1.0, 1.0));
// Taki kod jest dozwolony, ale stanowi bardzo zły wybór.
shapes. add( new CenteredSquare(2.5, 2, 3));

// List::get() zwraca obiekt typu Object, więc aby otrzymać
// obiekt typu CenteredCircle, należy zastosować rzutowanie.
CenteredCircle c = (CentredCircle)shapes. get(0);

// Poniższy wiersz spowoduje błąd wykonywania programu.
CenteredCircle c = (CentredCircle)shapes. get(1);
```

Problem w tym kodzie dotyczy konieczności przeprowadzenia rzutowania, aby otrzymać obiekt w nadającej się do użytku postaci — struktura `List` „nie wie", jakiego typu obiekty zawiera. Ponadto możliwe jest zapisanie w jednym kontenerze obiektów różnych typów i wszystko będzie w porządku, dopóki ktoś nie wykona niedozwolonego rzutowania, które spowoduje awarię programu.

Potrzebna jest taka wersja listy, która rozpoznaje przechowywane w niej typy obiektów. Wówczas kompilator `javac` mógłby wykrywać niepoprawne argumenty przekazywane do metod listy i powodować błąd kompilacji, zamiast odkładać tę nieuchronną katastrofę do czasu wykonywania programu.

W Javie istnieje składnia pozwalająca rozwiązać ten problem. Aby zaznaczyć, że dany typ jest kontenerem przechowującym egzemplarze pewnego typu referencyjnego, należy umieścić nazwę typu **ładunku** tego kontenera w nawiasie trójkątnym:

```
// utworzenie listy obiektów typu CenteredCircle
List<CenteredCircle> shapes = new ArrayList<CenteredCircle>();

// utworzenie paru wycentrowanych kształtów i zapisanie ich w liście
shapes. add( new CenteredCircle(1.0, 1.0, 1.0));

// Poniższy wiersz spowoduje błąd kompilacji.
shapes. add( new CenteredSquare(2.5, 2, 3));

// List<CenteredCircle>::get() zwraca obiekt typu CenteredCircle, więc nie trzeba rzutowania.
CenteredCircle c = shapes. get(0);
```

Dzięki tej składni możliwe jest przechwytywanie już na etapie kompilacji dużej grupy błędów. Taki też jest oczywiście cel tworzenia statycznego systemu typów — eliminowanie na etapie kompilacji masy błędów, które inaczej wystąpiłyby w czasie działania programu.

Typy kontenerowe zazwyczaj nazywa się **typami ogólnymi** (**generycznymi**), a deklaruje się je następująco:

```
interface Box<T> {
  void box(T t);
  T unbox();
}
```

To oznacza, że interfejs `Box` jest konstrukcją ogólną, w której można przechowywać dane dowolnego typu. Sam w sobie interfejs nie jest kompletny — jest raczej ogólnym opisem całej rodziny interfejsów, po jednej dla każdego typu, którego nazwę można wpisać w miejsce parametru `T`.

Typy ogólne i parametry typów

Wiesz już, jak użyć typu ogólnego, aby zwiększyć poziom bezpieczeństwa programu przez wykorzystanie wiedzy dostępnej w czasie kompilacji w celu zapobieżenia powstaniu błędów podczas działania programu. W tym podrozdziale dokładniej poznasz właściwości typów ogólnych.

Składnia `<T>` ma specjalną nazwę — **parametr typu** — a inna nazwa typów ogólnych to **typy parametryzowane**. Odnosi się to do faktu, że typ kontenerowy (np. `List`) jest parametryzowany przez inny typ (przechowywanych w nim danych). Pisząc typ taki jak `Map<String, Integer>`, programista przypisuje parametrom typu konkretne wartości.

Przy tworzeniu typu parametryzowanego należy uważać, aby nie zakodować w nim żadnych założeń dotyczących parametrów typu. Zatem typ `List` w ogólnej wersji to `List<E>`. Parametr typu `E` służy jako symbol reprezentujący nazwę rzeczywistego typu, którego dane programista będzie przechowywał w tej strukturze.

Parametry typu zawsze reprezentują typy referencyjne. Nie można w ich miejsce wstawić typu prostego.

Parametru typu można używać w sygnaturach i treści metod, tak jakby był prawdziwym typem, np.:

```
interface List<E> extends Collection<E> {
  boolean add(E e);
  E get(int index);
  // pozostałe metody zostały pominięte
}
```

Należy zauważyć, że parametr typu `E` może być stosowany zarówno jako parametr typu zwrotnego, jak i argumentu metody. Programista nie przyjmuje żadnego założenia, że typ ładunku ma jakiekolwiek konkretne właściwości, a jedynie zakłada podstawową spójność — że typ wstawiony do kontenera zostanie też potem z niego pobrany.

Składnia diamentowa

W wyrażeniu tworzącym egzemplarz typu ogólnego prawa strona przypisania zawiera powtórzenie wartości parametru typu. Najczęściej jest to niepotrzebne, ponieważ kompilator może wydedukować wartości parametrów typu. W nowych wersjach Javy można opuścić powtarzające się wartości typów przy użyciu tzw. **składni diamentowej**.

Zobaczmy, jak zastosować tę składnię, przepisując jeden z wcześniejszych przykładów:

```
// utworzenie  listy obiektów typu CenteredCircle
List<CenteredCircle> shapes = new ArrayList<>();
```

Jest to drobna poprawa, jeśli chodzi o rozwlekłość instrukcji przypisania — udało się zaoszczędzić wpisywania paru znaków. Do tematu dedukcji typów wrócimy jeszcze przy okazji opisu wyrażeń lambda pod koniec tego rozdziału.

Wymazywanie typów

W podrozdziale „Metody domyślne" napisaliśmy, że właściciele Javy bardzo dbają o zgodność nowych wersji platformy ze starszymi. Dodatek typów ogólnych w Javie 5 jest kolejnym przykładem sytuacji, w której utrzymanie zgodności wstecznej sprawiało problemy.

Najważniejszą kwestią do rozwiązania było to, jak stworzyć system typów pozwalający na używanie starych, nieogólnych klas kolekcji razem z nowymi kolekcjami ogólnymi. Postanowiono zastosować rzutowanie:

```
List someThings = getSomeThings();
// Niebezpieczne rzutowanie, ale wiadomo, że zawartość listy someThings stanowią łańcuchy.
List<String> myStrings = (List<String>)someThings;
```

To oznacza, że typy `List` i `List<String>` są ze sobą zgodne przynajmniej na pewnym poziomie. Zgodność tę w Javie uzyskano dzięki **wymazywaniu typów**. Polega to na tym, że ogólne parametry typu są widoczne tylko w czasie kompilacji, a kompilator `javac` je usuwa, dzięki czemu w kodzie bajtowym są już nieobecne[1].

Nieogólny typ `List` nazywa się **typem surowym**. Typów surowych nadal można używać, nawet jeśli mają uogólnione odpowiedniki. Jednak ich obecność prawie zawsze świadczy o niskiej jakości kodu.

Sposób działania mechanizmu wymazywania typów sprawia, że kompilator `javac` działa na nieco innym systemie typów niż maszyna wirtualna — szerzej tematem tym zajęliśmy się w podrozdziale „Typy czasu kompilacji i wykonywania programu".

Ponadto wymazywanie typów uniemożliwia stosowanie pewnych definicji, które wydają się poprawne. W poniższym kodzie intencją programisty było policzenie zamówień znajdujących się w dwóch trochę różnych strukturach danych:

```
// nie da się skompilować
interface OrderCounter {
  // nazwa odnosi się do listy numerów zamówień
  int totalOrders(Map<String, List<String>> orders);

  // nazwa odnosi się do liczby wszystkich złożonych do tej pory zamówień
  int totalOrders(Map<String, Integer> orders);
}
```

Ten kod wygląda na poprawny, a jednak nie przejdzie kompilacji. Problem polega na tym, że mimo iż obie te metody wyglądają jak normalne przeciążone wersje, to po wymazywaniu typów ich sygnatury będą wyglądać tak:

```
int totalOrders(Map);
```

[1] Pewne ślady pozostają, co można sprawdzić przy użyciu refleksji.

Po wymazaniu typów pozostaje surowy typ kontenera, którym w tym przypadku jest `Map`. System wykonawczy nie będzie w stanie odróżnić tych dwóch metod po sygnaturach, przez co składnia ta jest niedozwolona w specyfikacji języka.

Symbole wieloznaczne

Typ parametryzowany, taki jak `ArrayList<T>`, **nie umożliwia tworzenia egzemplarzy**. Jest tak dlatego, że `<T>` to tylko parametr typu, symbol zastępczy rezerwujący miejsce dla nazwy prawdziwego typu. Dopiero po wstawieniu w jego miejsce konkretnej wartości (np. `ArrayList<String>`) typ staje się kompletny i można tworzyć jego egzemplarze.

To powoduje problem, jeśli typ jest nieznany w czasie kompilacji, ale na szczęście w systemie typów Javy istnieje na to sposób. Utworzono specjalną konstrukcję do jawnego oznaczania **nieznanego typu** — `<?>`. Jest to najprostszy przykład **typu wieloznacznego** (ang. *wildcard*) języka Java.

Oto przykład wyrażenia z użyciem typu nieznanego:

```
ArrayList<?> mysteryList = unknownList();
Object o = mysteryList.get(0);
```

Jest to prawidłowy kod w języku Java — `ArrayList<?>` to w odróżnieniu od formy `ArrayList<T>` kompletny typ zmiennej. Nic nie wiadomo o typie możliwej zawartości listy `mysteryList`, ale nie sprawia to problemu w tym kodzie. Z używaniem typu nieznanego wiążą się pewne ograniczenia. Na przykład poniższy kod nie przejdzie kompilacji:

```
// To nie przejdzie kompilacji.
mysteryList.add(new Object());
```

Powód jest prosty — nie wiadomo, jakiego typu dane będą przechowywane w `mysteryList`! Gdyby się np. okazało, że `mysteryList` jest egzemplarzem typu `ArrayList<String>`, to nie moglibyśmy się spodziewać możliwości zapisania w tej liście obiektów typu `Object`.

Jedyna wartość, którą zawsze można wstawić do kontenera, to `null` — jak wiadomo, może ona zastąpić każdy typ referencyjny. Nie jest to jednak zbyt przydatne, więc w specyfikacji Javy zabroniono tworzenia egzemplarzy kontenerów o nieznanym typie zawartości. Na przykład:

```
// To nie przejdzie kompilacji.
List<?> unknowns = new ArrayList<?>();
```

Bardzo ważne zastosowanie typów nieznanych wynika z pytania: „Czy `List<String>` jest podtypem typu `List<Object>`?”. Innymi słowy: czy można napisać taki kod?

```
// Czy to jest dozwolone?
List<Object> objects = new ArrayList<String>();
```

Na pierwszy rzut oka może się wydawać, że ma to sens — `String` jest podklasą klasy `Object`, więc wiadomo, że każdy element typu `String` jest jednocześnie poprawnym obiektem typu `Object`. Ale spójrz na poniższy kod:

```
// Czy to jest dozwolone?
List<Object> objects = new ArrayList<String>();

// Jeżeli tak, to co zrobić z tym?
objects.add( new Object());
```

Ze względu na to, że typ kontenera `objects` został zadeklarowany jako `List<Object>`, powinno dać się wstawić do niego egzemplarz typu `Object`. Ponieważ jednak rzeczywisty egzemplarz

tego kontenera przechowuje łańcuchy, próba dodania obiektu typu `Object` oznaczałaby niezgodność typów, więc kod ten spowodowałby błąd wykonywania programu.

Należy sobie uświadomić, że mimo iż dozwolone (bo typ `String` dziedziczy po typie `Object`) jest napisanie takiego kodu:

```
Object o = new String("X");
```

nie znaczy to, że analogiczna instrukcja dla kontenera ogólnego również jest dozwolona:

```
// To nie przejdzie kompilacji.
List<Object> objects = new ArrayList<String>();
```

Inaczej można powiedzieć w ten sposób, że `List<String>` **nie** jest podtypem typu `List<Object>`. Jeśli potrzebna jest taka relacja między kontenerami, należy użyć typu nieznanego:

```
// To jest dozwolone.
List<?> objects = new ArrayList<String>();
```

To oznacza, że `List<String>` **jest** podtypem typu `List<?>`, mimo iż w takim przypisaniu jak powyższe zostaną utracone pewne informacje o typie. Na przykład typem zwrotnym metody `get()` jest teraz `Object`. Ponadto należy podkreślić, że `List<?>` nie jest podtypem żadnego typu `List<T>` dla jakiejkolwiek wartości `T`.

Typ nieznany sprawia niektórym programistom kłopoty, wywołując pytania w rodzaju: „Czemu zamiast typu nieznanego nie można po prostu użyć typu `Object`?”. Jak jednak pokazaliśmy, możliwość tworzenia relacji podtypowych między typami ogólnymi wymaga istnienia pojęcia typu nieznanego.

Symbole wieloznaczne z ograniczeniami

Tak naprawdę typy wieloznaczne w języku Java nie kończą się na typie nieznanym. Istnieją jeszcze **typy wieloznaczne z ograniczeniami**, zwane też **ograniczeniami parametrów typu**. Umożliwiają one określenie, jakie typy mogą zostać wstawione w miejsce parametru typu.

Używa się ich do opisywania hierarchii dziedziczenia głównie typów nieznanych i tworzenia stwierdzeń w rodzaju: „Nic nie wiem o tym typie oprócz tego, że musi implementować interfejs `List`”. Stwierdzenie to należałoby wyrazić jako `? extends List` w parametrze typu. Może to być przydatna pomoc dla programisty, który zamiast używać kompletnie nieznanego typu, wie przynajmniej, jakie są jego niektóre właściwości.

Słowa kluczowego `extends` używa się zawsze, niezależnie od tego, czy typ ograniczający jest typem klasowym, czy interfejsowym.

Jest to przykład koncepcji zwanej **wariancją typów**, która jest ogólną teorią opisującą relacje między typami kontenerów i dziedziczeniem typów przechowywanych w nich danych.

Kowariancja typów
: Oznacza, że typy kontenerów łączy taka sama relacja jak typy przechowywanych w nich danych. Wyraża się to za pomocą słowa kluczowego `extends`.

Kontrawariancja typów
: Oznacza, że typy kontenerów łączy odwrotna relacja niż typy przechowywanych w nich danych. Wyraża się to za pomocą słowa kluczowego `super`.

Zasady te są wspominane w opisach typów kontenerów pełniących funkcję producentów lub konsumentów typów. Na przykład jeżeli `Cat` rozszerza `Pet`, to `List<Cat>` jest podtypem typu `List<? extends Pet>`. `List` odgrywa rolę **producenta** obiektów typu `Cat` i odpowiednim słowem kluczowym jest `extends`.

W przypadku kontenera będącego jedynie **konsumentem** egzemplarzy jakiegoś typu należałoby użyć słowa kluczowego `super`.

Opisane zasady zostały sformułowane przez Joshuę Blocha i występują pod nazwą PECS (ang. *Producer Extends, Consumer Super*).

W rozdziale 8. przekonasz się, że kowariancja i kontrawariancja spotykane są w całej bibliotece kolekcji Javy. Ich najważniejszym zadaniem jest zapewnienie poprawnego i niezaskakującego działania typów ogólnych.

Kowariancja tablic

W początkowych wersjach Javy, zanim w ogóle wprowadzono biblioteki kolekcji, problem dotyczący wariancji typów kontenerowych występował w odniesieniu do tablic. Bez wariancji typów trudno było poprawnie napisać nawet tak prostą metodę jak `sort()`:

```
Arrays. sort(Object[] a);
```

Dlatego tablice w Javie są kowariantne — w tamtych czasach uznano to za zło konieczne, które niestety powodowało powstanie luki w statycznym systemie typów:

```
// Ten kod jest jak najbardziej poprawny.
String[] words = {"Witaj, świecie!" };
Object[] objects = words;

// A niech to, błąd wykonawczy.
objects[0] = new Integer(42);
```

Badania przeprowadzone na nowoczesnych otwartych bazach kodu sugerują, że kowariancja tablic jest bardzo rzadko spotykana i prawie zawsze stanowi usterkę w projekcie języka programowania[2]. Powinno się jej unikać przy pisaniu nowego kodu.

Metody ogólne

Metoda ogólna to taka metoda, która przyjmuje jako argumenty obiekty dowolnego typu referencyjnego.

Na przykład poniższa metoda imituje działanie operatora `,` (przecinek) z języka C, który służy do łączenia wyrażeń mających skutki uboczne:

```
// Ta klasa nie jest ogólna.
public class Utils
  public static <T> T comma(T a, T b) {
    return a;
  }
}
```

[2] Raoul-Gabriel Urma, Janina Voigt, *Using the OpenJDK to Investigate Covariance in Java*, „Java Magazine", maj – czerwiec 2012, s. 44 – 47.

Mimo że w definicji metody użyto parametru typu, zawierająca ją klasa nie jest ogólna. W tym przypadku specjalną składnię zastosowano tylko po to, by zaznaczyć, że metody można używać w dowolny sposób oraz że jej typ zwrotny jest taki sam jak typ zwrotny argumentu.

Używanie i projektowanie typów ogólnych

Przy pracy z typami ogólnymi w Javie czasami dobrze jest myśleć w kategoriach dwóch poziomów wtajemniczenia:

Praktyk
: Praktyk używa istniejących bibliotek ogólnych i tworzy stosunkowo proste własne klasy ogólne. Programista taki powinien także posiadać podstawową wiedzę o wymazywaniu typów, ponieważ niektóre cechy składni Javy trudno jest zrozumieć bez przynajmniej elementarnej wiedzy o tym, jak system wykonawczy obsługuje typy ogólne.

Projektant
: Projektant nowych bibliotek zawierających typy ogólne musi dysponować o wiele szerszą wiedzą na ich temat. W specyfikacji występują pewne trudniejsze części, jak np. pełny opis typów wieloznacznych, i są opisane zaawansowane zagadnienia, takie jak wiadomości o błędach „capture of".

Typy ogólne to jedna z najbardziej skomplikowanych części specyfikacji języka Java. Zawiera wiele pułapek, o których nie każdy programista musi wiedzieć, przynajmniej nie przy pierwszym zetknięciem z tą częścią systemu.

Typy czasu kompilacji i wykonywania programu

Spójrz na poniższy fragment kodu:

```
List<String> l = new ArrayList<>();
System.out.println(l);
```

Można zadać pytanie: jakiego typu jest `l`? Odpowiedź zależy od tego, czy interesuje nas `l` w czasie kompilacji (tzn. typ widziany przez kompilator `javac`), czy `l` w czasie działania programu (typ widziany przez maszynę wirtualną).

Dla kompilatora `javac l` będzie listą łańcuchów i te informacje wykorzysta do sprawdzenia, czy nie ma błędów składniowych, np. próby dodania niedozwolonego typu.

Natomiast dla maszyny wirtualnej `l` będzie obiektem typu `ArrayList`, jak widać w wyniku instrukcji `println()`. W czasie wykonywania programu zmienna `l` ma typ surowy dzięki wymazywaniu typów.

Oznacza to, że typy czasu wykonywania i kompilacji nieco się między sobą różnią. Najdziwniejsze jest jednak to, że pod pewnymi względami typ czasu wykonywania jest jednocześnie bardziej **i** mniej konkretny niż typ czasu kompilacji.

Typ czasu wykonywania jest mniej specyficzny niż typ czasu kompilacji, ponieważ utracona została informacja o typie przechowywanych danych — zniknęła w wyniku wymazywania typów i pozostał tylko typ surowy.

Typ czasu kompilacji jest mniej specyficzny niż typ czasu wykonywania, gdyż nie wiadomo dokładnie, jaki konkretny typ będzie mieć `l` — wiadomo tylko, że będzie to jakiś typ zgodny z typem `List`.

Wyliczenia i adnotacje

W Javie istnieją specjalne rodzaje klas i interfejsów odgrywające specjalne role w systemie typów. Są to **typy wyliczeniowe** i **typy adnotacyjne**, które zazwyczaj krócej nazywa się po prostu **wyliczeniami** (ang. *enum*) i **adnotacjami** (ang. *annotation*).

Wyliczenia

Wyliczenia są rodzajem klas o ograniczonej funkcjonalności i tylko niewielkiej liczbie możliwych wartości.

Załóżmy np., że trzeba zdefiniować typ do reprezentowania kolorów czerwonego, zielonego i niebieskiego oraz że są to jedyne dopuszczalne wartości tego typu. Taki typ można zdefiniować przy użyciu słowa kluczowego `enum`:

```
public enum PrimaryColor {
  // Na końcu listy egzemplarzy nie musi być znaku ;.
  CZERWONY, ZIELONY, NIEBIESKI
}
```

Do pól wyliczenia `PrimaryColor` można się teraz odwoływać tak, jakby były polami statycznymi: `PrimaryColor.CZERWONY`, `PrimaryColor.ZIELONY` i `PrimaryColor.NIEBIESKI`.

W innych językach programowania, np. C++, do tego celu najczęściej używa się stałych wartości całkowitoliczbowych, ale technika zastosowana w Javie zapewnia lepsze bezpieczeństwo typów i większą elastyczność. Na przykład jako że wyliczenia są specjalnym rodzajem klas, mogą zawierać pola i metody składowe. Jeśli mają treść główną (składającą się z pól i metod), to na końcu listy składowych wymagany jest średnik.

Załóżmy np., że potrzebne jest wyliczenie zawierające kilka wielokątów foremnych (figur geometrycznych, w których wszystkie boki i kąty są równe), na których można wykonywać pewne działania (za pomocą metod). Można w tym celu utworzyć wyliczenie przyjmujące wartość jako parametr:

```
public enum RegularPolygon {
  // W wyliczeniach z parametrami średnik jest potrzebny.
  TRIANGLE(3), SQUARE(4), PENTAGON(5), HEXAGON(6);

  private Shape shape;

  public Shape getShape() {
    return shape;
  }

  private RegularPolygon( int sides) {
    switch (sides) {
      case 3:
        // Zakładamy, że mamy kilka ogólnych konstruktorów
        // kształtów, które pobierają jako parametry
        // długość boku i miarę kąta w stopniach.
        shape = new Triangle(1, 1, 1, 60, 60, 60);
```

```
            break;
          case 4:
            shape = new Rectangle(1, 1);
            break;
          case 5:
            shape = new Pentagon(1, 1, 1, 1, 1, 108, 108, 108, 108, 108);
            break;
          case 6:
            shape = new Hexagon(1, 1, 1, 1, 1, 1, 120, 120, 120, 120, 120, 120);
            break;
        }
      }
    }
```

Parametry te (w tym przykładzie tylko jeden) są przekazywane do konstruktora w celu utworzenia poszczególnych egzemplarzy wyliczenia. Jako że egzemplarze wyliczenia są tworzone przez system wykonawczy Javy i nie mogą być tworzone z zewnątrz, konstruktor został zadeklarowany jako prywatny.

Wyliczenia mają pewne specjalne właściwości:

- Wszystkie niejawnie rozszerzają klasę `java.lang.Enum`.
- Nie mogą być ogólne.
- Mogą implementować interfejsy.
- Nie mogą być rozszerzane.
- Mogą zawierać metody abstrakcyjne tylko wtedy, gdy wszystkie wartości wyliczenia są zaimplementowane.
- Mogą mieć tylko prywatny konstruktor.

Adnotacje

Adnotacje to specjalny rodzaj interfejsu, który jak sama nazwa wskazuje, opatruje przypisem pewną część programu.

Weźmy np. adnotację `@Override`. Była już używana w niektórych wcześniejszych przykładach i może zastanawiałeś się, co to jest.

Najkrótsza i dość zaskakująca odpowiedź jest taka, że adnotacja ta nic nie robi.

Nieco dłuższa (i mniej poważna) odpowiedź jest taka, że `@Override`, jak wszystkie inne adnotacje, nie wywołuje bezpośrednich efektów, tylko służy jako dodatkowa informacja na temat metody, której dotyczy — w tym przypadku oznacza, że metoda przesłania metodę z nadklasy.

Jest to przydatna wskazówka dla kompilatorów i zintegrowanych środowisk programistycznych — jeśli programista pomyli się przy wpisywaniu nazwy przesłanianej metody, to obecność adnotacji `@Override` przy metodzie, która nie przesłania żadnej innej metody, zaalarmuje kompilator, że coś jest nie tak.

Adnotacje nie mogą zmieniać semantyki programu, a ich jedynym zadaniem jest dostarczanie dodatkowych metainformacji. Ściśle rzecz biorąc, oznacza to, że adnotacje nie powinny wpływać na sposób wykonywania programu, a tylko przekazywać informacje dla kompilatora i innych narzędzi działających przed rozpoczęciem wykonywania programu.

Kilka podstawowych adnotacji platformy jest zdefiniowanych w pakiecie `java.lang`. Początkowo istniały tylko adnotacje `@Override`, `@Deprecated` i `@SuppressWarnings`, oznaczające odpowiednio przesłonięcie metody, metodę, której używania się nie zaleca, oraz to, że dana metoda generuje pewne ostrzeżenia, które należy stłumić.

W Javie 7 dodano do tego zestawu adnotację `@SafeVarargs` (rozszerzone tłumienie ostrzeżeń dla metod o zmiennej liczbie argumentów), a w Javie 8 — `@FunctionalInterface`. Adnotacja `@FunctionalInterface` oznacza, że dany interfejs może być używany jako cel dla wyrażenia lambda — jest przydatnym znacznikiem, chociaż nie ma obowiązku jego stosowania.

W porównaniu ze zwykłymi interfejsami adnotacje mają pewne specjalne właściwości:

- Wszystkie niejawnie rozszerzają interfejs `java.lang.annotation.Annotation`.
- Nie mogą być ogólne.
- Nie mogą rozszerzać żadnego innego interfejsu.
- Mogą definiować tylko metody nieprzyjmujące argumentów.
- Nie mogą definiować metod zgłaszających wyjątki.
- Mają ograniczony zestaw typów zwrotnych dla metod.
- Mogą mieć domyślną wartość zwrotną dla metod.

Definiowanie własnych adnotacji

Definiowanie własnych adnotacji do użytku w swoich programach nie jest trudne. Służy do tego słowo kluczowe `@interface`, którego używa się bardzo podobnie jak słów kluczowych `class` i `interface`.

Kluczem do pisania własnych adnotacji jest wykorzystanie „metaadnotacji”. Jest to specjalny rodzaj adnotacji występujących w definicjach nowych typów adnotacji.

Metaadnotacje są zdefiniowane w pakiecie `java.lang.annotation` i umożliwiają zdefiniowanie zasady dotyczącej tego, gdzie dany typ adnotacji może być używany oraz jak powinien być traktowany przez kompilator i system wykonawczy.

Istnieją dwie podstawowe metaadnotacje, które są w zasadzie niezbędne przy tworzeniu nowego typu adnotacji: `@Target` i `@Retention`. Obie przyjmują wartości reprezentowane jako wyliczenia.

Metaadnotacja `@Target` określa, gdzie dana adnotacja może być używana w kodzie źródłowym. Wyliczenie `ElementType` ma następujący zestaw dopuszczalnych wartości: `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, `LOCAL_VARIABLE`, `ANNOTATION_TYPE`, `PACKAGE`, `TYPE_PARAMETER` oraz `TYPE_USE`.

Druga metaadnotacja to `@Retention`. Określa ona, w jaki sposób kompilator `javac` i system wykonawczy Javy mają przetwarzać daną adnotację. Może mieć jedną z trzech wartości reprezentowanych przez wyliczenie `RetentionPolicy`:

`SOURCE`
: Adnotacje z tą zasadą zatrzymania są odrzucane przez kompilator `javac` w czasie kompilacji.

`CLASS`
: To oznacza, że adnotacja będzie obecna w pliku klasy, ale niekoniecznie będzie dostępna w czasie wykonywania programu przez maszynę wirtualną. Ustawienia tego używa się rzadko, ale czasami można je spotkać w narzędziach do statycznej analizy kodu bajtowego.

`RUNTIME`
: To oznacza, że adnotacja będzie dostępna w kodzie użytkownika w czasie działania programu (przy użyciu refleksji).

Spójrz na prosty przykład definicji adnotacji o nazwie `@Nickname`, która umożliwia programiście zdefiniowanie dodatkowej nazwy dla metody, aby ułatwić jej znalezienie w czasie działania programu za pomocą refleksji:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Nickname {
    String[] value() default {};
}
```

To cały kod definicji adnotacji. Określono w nim element składni, przy którym adnotacja ta może występować, zasadę zatrzymywania oraz nazwę elementu. Jako że celem jest umożliwienie dodania alternatywnej nazwy dla metody, konieczne było też zdefiniowane metody w tej adnotacji. Mimo to definicje nowych własnych adnotacji są bardzo zwięzłe.

Oprócz dwóch opisanych podstawowych metaadnotacji istnieją jeszcze metaadnotacje `@Inherited` i `@Documented`. Są one znacznie rzadziej używane, a ich szczegółowy opis można znaleźć w dokumentacji platformy.

Adnotacje typów

W Javie 8 do wyliczenia `ElementType` dodano dwie nowe wartości — `TYPE_PARAMETER` i `TYPE_USE`. Umożliwiają one stosowanie adnotacji w nowych, wcześniej niedostępnych miejscach, a konkretnie wszędzie tam, gdzie użyty jest jakiś typ. To pozwala programiście na pisanie takiego kodu jak poniższy:

```
@NotNull String safeString = getMyString();
```

Dodatkowe informacje o typie przekazywane przez adnotację `@NotNull` mogą zostać wykorzystane przez specjalny system sprawdzania typów do wykrywania nieprawidłowości (w tym przypadku potencjalnego wyjątku `NullPointerException`) i przeprowadzania dodatkowej analizy statycznej. Podstawowa dystrybucja Javy 8 zawiera kilka podstawowych narzędzi do sprawdzania typów w formie wtyczek, ale dodatkowo udostępnia też infrastrukturę umożliwiającą programistom tworzenie własnych takich narzędzi.

W tym podrozdziale opisaliśmy typy wyliczeniowe i adnotacyjne języka Java. Teraz przechodzimy do opisu następnej ważnej części systemu typów Javy: typów zagnieżdżonych.

Typy zagnieżdżone

Wszystkie klasy, interfejsy i wyliczenia pokazane do tej pory to **typy najwyższego poziomu**. To znaczy, że są bezpośrednimi składnikami pakietów, a ich definicje są niezależne od innych typów. Ale definicje typów można też zagnieżdżać w innych definicjach typów. W efekcie powstają tzw. **typy zagnieżdżone**, często nazywane „klasami wewnętrznymi”, które stanowią bardzo ważny element języka programowania Java.

Typów zagnieżdżonych używa się do dwóch różnych celów, z których oba wiążą się z hermetyzacją:

- Typ może zostać zagnieżdżony, gdy potrzebuje dostępu na szczególnych zasadach do wnętrza innego typu — będąc typem składowym, ma takie same prawa jak zmienne i metody składowe, więc może naginać zasady hermetyzacji.
- Dany typ może być potrzebny tylko do ściśle określonego celu i w bardzo ograniczonej części kodu. Ta ścisła izolacja jest potrzebna dlatego, że typ ten stanowi szczegół implementacyjny i powinien być oddzielony od reszty systemu.

Typy zagnieżdżone można też sobie wyobrażać jako konstrukcje, które są w jakiś sposób związane z innym typem — nie mogą istnieć całkiem niezależnie od pewnej jednostki. Typ można zagnieździć w innym typie na cztery sposoby:

Statyczne typy składowe
: Statyczny typ składowy to typ zdefiniowany jako składowa statyczna innego typu. Statycznie są zawsze zagnieżdżone interfejsy, wyliczenia i adnotacje (nawet jeśli programista nie użyje słowa kluczowego `static`).

Niestatyczne klasy składowe
: Niestatyczny typ składowy to po prostu typ składowy niezadeklarowany jako statyczny. W ten sposób można deklarować tylko klasy.

Klasy lokalne
: Klasa lokalna to taka, która jest zdefiniowana i widoczna tylko w bloku kodu Java. Lokalnie nie można definiować interfejsów, wyliczeń i adnotacji.

Klasy anonimowe
: Klasa anonimowa to rodzaj klasy lokalnej pozbawionej nazwy mającej jakieś znaczenie w języku Java. Interfejsy, wyliczenia i adnotacje nie mogą być anonimowe.

Termin „typy zagnieżdżone” jest poprawny i precyzyjny, ale programiści używają go rzadko. Zamiast tego większość użytkowników Javy stosuje znacznie mniej dokładne określenie: „klasa wewnętrzna”. W zależności od sytuacji może ono oznaczać niestatyczną klasę składową, klasę lokalną lub klasę anonimową, ale nie statyczny typ składowy, i nie ma możliwości ich rozróżnienia.

Na szczęście mimo niedoskonałości terminologii dotyczącej opisu typów zagnieżdżonych składnia jest bardzo precyzyjna i z kontekstu zazwyczaj jasno wynika, o jakim typie zagnieżdżonym jest mowa.

Poniżej znajduje się bardziej szczegółowy opis każdego z czterech rodzajów typów zagnieżdżonych. W każdej sekcji opisaliśmy właściwości danego typu, ograniczenia dotyczące jego zastosowania oraz specjalne wymagania składniowe. Na zakończenie tej części przedstawiliśmy uwagę implementacyjną wyjaśniającą, jak typy zagnieżdżone działają od wewnątrz.

Statyczne typy składowe

Statyczny typ składowy jest bardzo podobny do zwykłego typu najwyższego poziomu, ale dla wygody został zagnieżdżony w przestrzeni nazw innego typu. Oto podstawowe właściwości statycznych typów składowych:

- Statyczny typ składowy jest jak pozostałe składowe statyczne klasy: pola i metody statyczne.
- Statyczny typ składowy nie jest związany z jakimkolwiek egzemplarzem zawierającej go klasy (tzn. nie ma obiektu `this`).
- Statyczny typ składowy ma dostęp tylko do statycznych składowych zawierającej go klasy.
- Statyczny typ składowy ma dostęp do wszystkich składowych statycznych (wliczając inne statyczne typy składowe) zawierającego go typu.
- Zagnieżdżone interfejsy, wyliczenia i adnotacje są domyślnie statyczne, niezależnie od tego, czy użyto słowa kluczowego `static`, czy nie.
- Każdy typ zagnieżdżony w interfejsie lub adnotacji jest domyślnie statyczny.
- Statyczny typ składowy można definiować zarówno w typach najwyższego poziomu, jak i w dowolnie głęboko zagnieżdżonych innych składowych typach statycznych.
- Statycznego typu składowego nie można zdefiniować w innym rodzaju typu zagnieżdżonego.

Przeanalizujemy krótki przykład składni definicji statycznych typów składowych. Na listingu 4.1 pokazano interfejs pomocniczy zdefiniowany jako statyczna składowa klasy. Ponadto w kodzie tym pokazano sposób użycia tego interfejsu zarówno w zawierającej go klasie, jak i w klasach zewnętrznych. Zwróć uwagę na zastosowanie hierarchicznej nazwy w klasie zewnętrznej.

Listing 4.1. Definiowanie i używanie statycznego interfejsu składowego

```
// klasa implementująca stos jako listę powiązaną
public class LinkedStack {

    // Ten statyczny interfejs składowy określa sposób powiązania obiektów.
    // Słowo kluczowe static jest opcjonalne: wszystkie zagnieżdżone interfejsy są statyczne.
    static interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // Początek listy jest obiektem Linkable.
    Linkable head;

    // Treść metod została pominięta.
    public void push(Linkable node) { ... }

    public Object pop() { ... }
}

// Ta klasa implementuje statyczny interfejs składowy.
class LinkableInteger implements LinkedStack.Linkable {
    // Poniżej znajdują się dane węzła i jego konstruktor.
    int i;
    public LinkableInteger(int i) { this.i = i; }

    // To są dane i metody potrzebne do implementacji interfejsu.
    LinkedStack.Linkable next;

    public LinkedStack.Linkable getNext() { return next; }

    public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

Właściwości statycznych typów składowych

Statyczny typ składowy ma dostęp do wszystkich statycznych składowych zawierającego go typu, wliczając składowe prywatne. Działa to też w drugą stronę, tzn. metody typu nadrzędnego mają dostęp do wszystkich składowych statycznego typu składowego, wliczając składowe prywatne. Statyczny typ składowy ma nawet dostęp do wszystkich składowych każdego innego statycznego typu składowego, wliczając składowe prywatne. Statyczny typ składowy może używać innych statycznych typów składowych bez poprzedzania ich nazwy nazwą typu nadrzędnego.

Statyczny typ składowy nie może mieć takiej samej nazwy jak którakolwiek z zawierających go klas. Ponadto statyczne typy składowe można definiować tylko w typach najwyższego poziomu i innych statycznych typach składowych. Tak naprawdę zasada ta jest elementem szerszej reguły, zabraniającej definiowania jakichkolwiek statycznych składowych w klasach składowych, lokalnych i anonimowych.

Typy najwyższego poziomu mogą być publiczne lub prywatne w odniesieniu do pakietu (jeśli w ich deklaracji nie użyto słowa kluczowego `public`). Deklarowanie ich jako prywatnych lub chronionych nie miałoby sensu. Widoczność chroniona byłaby równoważna z prywatną pakietową, a prywatna klasa najwyższego poziomu nie mogłaby być stosowana przez jakikolwiek inny typ.

Natomiast statyczne typy składowe są składowymi, więc mogą mieć taką samą widoczność jak każda inna składowa zawierającego je typu. Modyfikatory widoczności w odniesieniu do nich mają takie samo znaczenie jak w odniesieniu do pozostałych składowych. Przypomnijmy, że wszystkie składowe interfejsów (i adnotacji) są domyślnie publiczne, więc statyczne typy składowe zagnieżdżone w interfejsach i adnotacjach nie mogą być chronione ani prywatne.

Na przykład na listingu 4.1 interfejs `Linkable` jest zadeklarowany jako publiczny, toteż może go implementować każda klasa, której obiekty powinno dać się przechowywać w strukturze `LinkedStack`.

Poza klasą nadrzędną statyczny typ składowy ma nazwę składającą się z nazwy typu zewnętrznego i typu wewnętrznego (np. `LinkedStack.Linkable`).

W większości przypadków składnia ta stanowi cenne przypomnienie, że klasa wewnętrzna jest powiązana z zawierającym ją typem. Ale w języku Java można też za pomocą dyrektywy `import` bezpośrednio zaimportować statyczny typ składowy:

```
import pkg.LinkedStack.Linkable;  // importuje jeden wybrany typ zagnieżdżony
// importuje wszystkie typy zagnieżdżone z klasy LinkedStack
import pkg.LinkedStack.*;
```

Dzięki tej operacji typu zagnieżdżonego można używać bez dodawania nazwy zawierającego go typu (np. wystarczy napisać `Linkable`).

Statyczny typ składowy można też zaimportować za pomocą dyrektywy `import static`. Szerzej na temat dyrektyw `import` i `import static` pisaliśmy w rozdziale 2., w podrozdziale „Pakiety i przestrzenie nazw”.

Ponieważ jednak import typu zagnieżdżonego powoduje ukrycie faktu, że typ ten jest ściśle związany z typem go zawierającym — co zazwyczaj jest bardzo ważną informacją — rzadko się to robi.

Niestatyczne klasy składowe

Niestatyczna klasa składowa to klasa zadeklarowana jako składowa innej klasy lub wyliczenia bez użycia słowa kluczowego `static`:

- Jeżeli statyczny typ składowy jest analogiczny do pola klasy i metody klasy, to niestatyczna klasa składowa jest analogiczna do pola egzemplarza i metody egzemplarza.
- Tylko klasy mogą być niestatycznymi typami składowymi.
- Egzemplarz niestatycznej klasy składowej jest zawsze związany z egzemplarzem typu nadrzędnego.
- Kod niestatycznej klasy składowej ma dostęp do wszystkich pól i metod (statycznych i niestatycznych) typu nadrzędnego.
- Kilka cech składni Javy istnieje tylko po to, by możliwa była praca z egzemplarzem nadrzędnym niestatycznej klasy składowej.

Na listingu 4.2 pokazano sposób definicji i użycia klasy składowej. Jest to rozszerzenie poprzedniego przykładu klasy `LinkedStack` o możliwość wyliczenia elementów stosu za pomocą metody `iterator()` zwracającej implementację interfejsu `java.util.Iterator`. Implementacja tego interfejsu jest zdefiniowana jako klasa składowa.

Listing 4.2. Iterator zaimplementowany jako klasa składowa

```
import java.util.Iterator;

public class LinkedStack {

    // nasz statyczny interfejs składowy
    public interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // początek listy
    private Linkable head;

    // treść metod pominięto
    public void push(Linkable node) { ... }
    public Linkable pop() { ... }

    // Ta metoda zwraca obiekt typu Iterator dla tej klasy LinkedStack.
    public Iterator<Linkable> iterator() { return new LinkedIterator(); }

    // To jest implementacja interfejsu Iterator,
    // zdefiniowana jako niestatyczna klasa składowa.
    protected class LinkedIterator implements Iterator<Linkable> {
        Linkable current;

        // W konstruktorze użyto prywatnego pola klasy nadrzędnej.
        public LinkedIterator() { current = head; }

        // Trzy poniższe metody są zdefiniowane przez interfejs Iterator.
        public boolean hasNext() { return current != null; }

        public Linkable next() {
            if (current == null)
                throw new java.util.NoSuchElementException();
            Linkable value = current;
```

```
            current = current.getNext();
            return value;
        }

        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```

Zwróć uwagę na sposób zagnieżdżenia klasy `LinkedIterator` w klasie `LinkedStack`. Jako że `Linked`
`↪Iterator` jest klasą pomocniczą wykorzystywaną tylko w klasie `LinkedStack`, umieszczenie jej definicji bardzo blisko miejsca użycia sprawia, że projekt klasy jest bardziej przejrzysty.

Właściwości klas składowych

Tak jak pola i metody egzemplarzowe, tak każdy egzemplarz niestatycznej klasy składowej jest związany z egzemplarzem klasy, w której ją zdefiniowano. Oznacza to, że kod klasy składowej ma dostęp do wszystkich pól i metod egzemplarzowych (a także składowych statycznych) egzemplarza nadrzędnego, wliczając składowe prywatne.

Zostało to już zilustrowane w listingu 4.2. Poniżej jeszcze raz przedstawiamy konstruktor `LinkedStack.LinkedIterator()`:

```
public LinkedIterator() { current = head; }
```

Kod ten ustawia pole `current` klasy wewnętrznej na wartość pola `head` klasy nadrzędnej. Wszystko działa poprawnie, mimo że `head` jest polem prywatnym klasy nadrzędnej.

Niestatyczna klasa składowa, jak każda składowa klasy, może mieć przypisany jeden ze standardowych modyfikatorów dostępu. Na listingu 4.2 klasa `LinkedIterator` jest zadeklarowana jako chroniona, więc jest niedostępna w kodzie (w innym pakiecie) używającym klasy `LinkedStack`, ale jest dostępna we wszystkich podklasach tej klasy.

Ograniczenia dotyczące klas składowych

Klasy składowe mają dwa poważne ograniczenia:

- Niestatyczna klasa składowa nie może mieć takiej samej nazwy jak którakolwiek z klas nadrzędnych lub pakiet. Jest to bardzo ważna zasada, która **nie** dotyczy pól i metod.
- Niestatyczne klasy składowe nie mogą zawierać pól statycznych, metod ani typów, z wyjątkiem pól stałych zadeklarowanych jednocześnie jako statyczne i finalne.

Składowe statyczne są konstrukcjami najwyższego poziomu, niezwiązanymi z jakimkolwiek obiektem, natomiast każda niestatyczna klasa składowa jest związana z egzemplarzem zawierającej ją klasy. Definicja statycznej składowej najwyższego poziomu w klasie składowej niebędącej na najwyższym poziomie powodowałaby niejasności, więc zostało to zabronione.

Składnia klas składowych

Najważniejszą właściwością klas składowych jest to, że mają dostęp do pól i metod egzemplarzowych zawierających je obiektów. Przykład tego można było zaobserwować w konstruktorze `LinkedStack.LinkedIterator()` na listingu 4.2:

```
public LinkedIterator() { current = head; }
```

W tym przykładzie `head` jest polem nadrzędnej klasy `LinkedStack` i zostaje ono przypisane do pola `current` klasy `LinkedIterator` (będącego składową niestatycznej klasy składowej).

Gdybyśmy chcieli użyć jawnych referencji i słowa kluczowego `this`, to musielibyśmy się posłużyć specjalną składnią do jawnego odnoszenia się do nadrzędnego egzemplarza obiektu `this`. Na przykład w konstruktorze wyglądałoby to tak:

```
public LinkedIterator() { this.current = LinkedStack.this.head; }
```

Ogólna postać tej składni to *`nazwaklasy`*`.this`, gdzie *`nazwaklasy`* to nazwa klasy nadrzędnej. Zauważ, że klasy składowe same mogą zawierać klasy składowe do dowolnego poziomu głębokości zagnieżdżenia. Ponieważ jednak żadna klasa składowa nie może mieć takiej samej nazwy jak którakolwiek z klas nadrzędnych, użycie nazwy klasy nadrzędnej i słowa kluczowego `this` jest idealnym ogólnym sposobem na odnoszenie się do dowolnego egzemplarza nadrzędnego.

Ta specjalna składnia jest potrzebna tylko wtedy, gdy trzeba się odnieść do składowej klasy nadrzędnej, która jest ukryta przez składową o takiej samej nazwie w klasie składowej.

Zakres dostępności a dziedziczenie

Zwróć uwagę, że klasa najwyższego poziomu może rozszerzać klasę składową. Po wprowadzeniu niestatycznych klas składowych dla każdej klasy należy rozważyć dwie osobne hierarchie. Pierwsza to **hierarchia dziedziczenia**, od nadklasy do podklasy, która definiuje dziedziczone przez klasę składową pola i metody. Druga to **hierarchia nadrzędności**, od klasy nadrzędnej do podrzędnej, która definiuje zestaw pól i metod znajdujących się w zakresie widoczności (dostępności) klasy składowej.

Znajomość właściwości i podstawowych zasad dotyczących tych dwóch hierarchii jest bardzo ważna:

- Hierarchie te są od siebie całkowicie niezależne i nie można ich mylić.
- Należy powstrzymać się od stwarzania kolizji nazw, tzn. sytuacji, w których pole lub metoda w nadklasie ma taką samą nazwę jak pole lub metoda w klasie nadrzędnej.
- Jeżeli konflikt nazw wystąpi, odziedziczone pole lub metoda ma pierwszeństwo przed polem bądź metodą o takiej samej nazwie z klasy nadrzędnej.
- Odziedziczone pola i metody znajdują się w zakresie klasy je dziedziczącej i mają pierwszeństwo przed polami i metodami o takiej samej nazwie z zakresów nadrzędnych.
- Aby zapobiec myleniu hierarchii klas z hierarchią nadrzędności, unikaj tworzenia głębokich hierarchii nadrzędności.
- Jeżeli klasa jest zagnieżdżona głębiej niż na dwa poziomy, to prawdopodobnie spowoduje więcej zamieszania, niż jest warta.
- Jeśli klasa ma głęboką hierarchię klas (tzn. ma wielu przodków), zastanów się, czy nie lepiej ją zdefiniować jako klasę najwyższego poziomu zamiast niestatyczną klasę składową.

Klasy lokalne

Klasa lokalna jest zadeklarowana lokalnie w bloku kodu Java zamiast jako składowa klasy. Lokalnie można definiować tylko klasy. Interfejsy, wyliczenia i adnotacje jako typy składowe muszą być na najwyższym poziomie lub statyczne. Najczęściej lokalne klasy definiuje się w metodach, ale można je też definiować w statycznych inicjatorach i inicjatorach egzemplarzy klas.

Tak jak wszystkie bloki kodu w Javie muszą się znajdować w definicjach klas, tak wszystkie klasy lokalne muszą się znajdować w blokach. Z tego powodu klasy lokalne mają wiele wspólnych cech z klasami składowymi. Ale z reguły lepiej jest je traktować jak całkiem osobny rodzaj typu zagnieżdżonego.

W rozdziale 5. szczegółowo opisaliśmy, kiedy lepiej wybrać klasę lokalną, a kiedy wyrażenie lambda.

Cechą wyróżniającą klasy lokalnej jest to, że jest ograniczona do zakresu bloku kodu. Podobnie jak zmienna lokalna klasa lokalna jest dostępna tylko w tym bloku kodu, w którym została zdefiniowana. Na listingu 4.3 pokazano zmodyfikowaną metodę `iterator()` z klasy `LinkedStack` w taki sposób, że klasa `LinkedIterator` jest w niej klasą lokalną, a nie klasą składową.

Dzięki temu definicja klasy jeszcze bardziej przybliżyła się do miejsca użycia, co powinno dodatkowo zwiększyć czytelność kodu. Dla uproszczenia na listingu 4.3 pokazano tylko kod źródłowy metody `iterator()`, bez całej otaczającej ją klasy `LinkedStack`.

Listing 4.3. Definiowanie i używanie klasy lokalnej

```
// Ta metoda zwraca obiekt typu Iterator dla tej klasy LinkedStack.
public Iterator<Linkable> Iterator() {
    // Poniżej znajduje się definicja LinkedIterator jako klasy lokalnej.
    class LinkedIterator implements Iterator<Linkable> {
        Linkable current;

        // W konstruktorze użyto prywatnego pola klasy nadrzędnej.
        public LinkedIterator() { current = head; }

        // Trzy poniższe metody są zdefiniowane przez interfejs Iterator.
        public boolean hasNext() { return current != null; }

        public Linkable next() {
            if (current == null)
                throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }

        public void remove() { throw new UnsupportedOperationException(); }
    }

    // tworzy i zwraca egzemplarz właśnie zdefiniowanej klasy
    return new LinkedIterator();
}
```

Właściwości klas lokalnych

Klasy lokalne mają następujące ciekawe właściwości:

- Tak jak klasy składowe klasy lokalne są związane z egzemplarzem nadrzędnym i mają dostęp do wszystkich składowych, także prywatnych, klasy nadrzędnej.
- Oprócz pól zdefiniowanych przez klasę nadrzędną klasy lokalne mają dostęp do wszystkich zmiennych lokalnych, parametrów metod i parametrów wyjątków znajdujących się w zasięgu lokalnej definicji metody i zadeklarowanych jako finalne.

Ograniczenia dotyczące klas lokalnych

Klasy lokalne podlegają następującym ograniczeniom:

- Nazwa klasy lokalnej istnieje tylko w bloku zawierającym jej definicję. Nie można jej używać poza tym blokiem. (Ale należy zauważyć, że egzemplarze klasy lokalnej utworzone w zakresie tej klasy mogą dalej istnieć poza tym zakresem. Sytuacja ta jest bardziej szczegółowo opisana w dalszej części tej sekcji).
- Klas lokalnych nie można deklarować jako publicznych, chronionych, prywatnych ani statycznych.
- Podobnie jak klasy składowe, i z tych samych powodów, klasy lokalne nie mogą zawierać statycznych pól, metod ani klas. Jedynym wyjątkiem od tej reguły są stałe jednocześnie statyczne i finalne.
- Typów interfejsowych, wyliczeniowych i adnotacyjnych nie można definiować lokalnie.
- Klasa lokalna, tak jak klasa składowa, nie może mieć takiej samej nazwy jak którakolwiek z jej klas nadrzędnych.
- Jak napisaliśmy wcześniej, klasa lokalna może używać lokalnych zmiennych, parametrów metod, a nawet parametrów wyjątków dostępnych w jej zakresie, ale tylko pod warunkiem, że te zmienne lub parametry są finalne. Wiąże się to z tym, że czas istnienia egzemplarza klasy lokalnej może być znacznie dłuższy niż czas wykonywania metody, w którym klasa ta jest zdefiniowana.

Klasa lokalna zawiera prywatną wewnętrzną kopię wszystkich używanych przez siebie zmiennych (są one automatycznie generowane przez kompilator `javac`). Jedynym sposobem na zapewnienie zgodności zmiennej lokalnej z prywatną kopią jest wymuszenie, aby lokalna zmienna była finalna.

Zakres dostępności klasy lokalnej

W części poświęconej niestatycznym klasom składowym napisaliśmy, że klasa składowa ma dostęp do wszystkich składowych odziedziczonych po nadklasie i wszystkich składowych zdefiniowanych przez klasę ją zawierającą. To samo dotyczy klas lokalnych, chociaż te mają też dostęp do finalnych lokalnych zmiennych i parametrów. Na listingu 4.4 przedstawiono różne rodzaje pól i zmiennych, które mogą być dostępne dla klasy lokalnej.

Listing 4.4. Pola i zmienne dostępne dla klasy lokalnej

```
class A { protected char a = 'a' ; }
class B { protected char b = 'b' ; }
```

```
public class C extends A {
  private char c = 'c' ;  // prywatne pola widoczne w klasie lokalnej
  public static char d = 'd' ;
  public void createLocalObject(final char e)
  {
    final char f = 'f' ;
    int i = 0;  // zmienna i nie jest finalna, więc jest bezużyteczna dla klasy lokalnej
    class Local extends B
    {
      char g = 'g' ;
      public void printVars()
      {
        // Wszystkie te pola i zmienne są dostępne dla tej klasy.
        System.out.println(g);  // (this.g) g jest polem tej klasy
        System.out.println(f);  // f jest finalną zmienną lokalną
        System.out.println(e);  // e jest finalnym lokalnym parametrem
        System.out.println(d);  // (C.this.d) pole d klasy zawierającej
        System.out.println(c);  // (C.this.c) pole c klasy zawierającej
        System.out.println(b);  // b jest dziedziczone przez tę klasę
        System.out.println(a);  // a jest dziedziczone przez klasę zawierającą
      }
    }
    Local l = new Local();  // utworzenie egzemplarza klasy lokalnej
    l.printVars();  // i wywołanie jego metody printVars().
  }
}
```

Zakres leksykalny a zmienne lokalne

Zmienna lokalna jest zdefiniowana w bloku kodu określającym jej **zakres dostępności**. Poza tym blokiem zmienna lokalna jest niedostępna, ponieważ przestaje istnieć. W obrębie klamry zawierającej definicję zmiennej lokalnej zmienna ta jest dostępna w każdym miejscu.

Ten rodzaj wyznaczania zakresu dostępności, zwany **zakresem leksykalnym**, definiuje tylko sekcję kodu źródłowego, w której można używać danej zmiennej. Programiści często traktują to jak zakres **tymczasowy**, tzn. zmienna lokalna istnieje od momentu początku do końca wykonywania przez maszynę wirtualną danego bloku kodu. W większości przypadków taki sposób myślenia o zmiennych lokalnych i zakresie ich dostępności ma sens.

Ale wprowadzenie klas lokalnych nieco zaburzyło tę sielankę. Aby zrozumieć dlaczego, należy sobie uświadomić, że egzemplarze klasy lokalnej mogą istnieć dłużej, niż trwa wykonywanie bloku kodu zawierającego definicję tej klasy lokalnej.

Innymi słowy, jeżeli programista utworzy egzemplarz klasy lokalnej w bloku kodu, to egzemplarz ten nie znika automatycznie po zakończeniu wykonywania przez maszynę wirtualną tego bloku kodu. W efekcie, mimo że definicja klasy była lokalna, egzemplarze tej klasy mogą wyjść poza obszar tej definicji.

Skutki tego bywają zaskakujące dla niektórych programistów. Wiąże się to z tym, że klasy lokalne mogą używać zmiennych lokalnych, a więc mogą też zawierać kopie wartości z już nieistniejących zakresów leksykalnych. Można to zaobserwować w poniższym przykładzie:

```
public class Weird {
  // statyczny interfejs składowy używany poniżej
  public static interface IntHolder { public int getValue(); }
```

```
  public static void main(String[] args) {
    IntHolder[] holders = new IntHolder[10];
    for(int i = 0; i < 10; i++) {
      final int fi = i;

      // klasa lokalna
      class MyIntHolder implements IntHolder {
        // użycie zmiennej finalnej
        public int getValue() { return fi; }
      }
      holders[i] = new MyIntHolder();
    }

    // Klasa lokalna jest już poza zasięgiem, więc nie można jej używać.
    // Ale w tablicy mamy 10 prawidłowych egzemplarzy tej klasy. Lokalna
    // zmienna fi jest tu poza zasięgiem, ale nadal jest w zasięgu metody
    // getValue() każdego z tych 10 obiektów. Można więc wywołać metodę
    // getValue() dla każdego obiektu, aby go wydrukować. Spowoduje to
    // wydrukowanie cyfr od 0 do 9.
    for(int i = 0; i < 10; i++) {
      System.out.println(holders[i]. getValue());
    }
  }
}
```

Aby zrozumieć ten kod, należy sobie uświadomić, że zakres leksykalny metod klasy lokalnej nie ma nic wspólnego z tym, kiedy interpreter rozpoczyna i kończy wykonywanie bloku kodu zawierającego definicję tej klasy.

Każdy egzemplarz klasy lokalnej zawiera automatycznie utworzoną prywatną kopię każdej używanej w nim finalnej zmiennej lokalnej, dzięki czemu ma własną prywatną kopię zakresu, który istniał w czasie jego tworzenia.

Lokalna klasa `MyIntHolder` mogłaby zostać nazwana **zamknięciem**. Ogólnie rzecz biorąc, zamknięcie to obiekt zawierający zapisany stan zakresu i udostępniający ten stan w późniejszym czasie.

Zamknięcia są wykorzystywane w niektórych stylach programowania i mają rozmaite implementacje w różnych językach programowania. Język Java implementuje zamknięcia jako klasy lokalne, klasy anonimowe i wyrażenia lambda.

Klasy anonimowe

Klasa anonimowa to lokalna klasa niemająca nazwy. Jej definicja i wyrażenie tworzące egzemplarz mieszczą się w jednym zwięzłym wyrażeniu zawierającym operator `new`. Podczas gdy definicja klasy lokalnej jest instrukcją w bloku kodu, definicja klasy anonimowej jest wyrażeniem, co znaczy, że może występować jako część większego wyrażenia, np. wywołania metody.

Klasy anonimowe opisujemy, aby dostarczyć kompletnych informacji o języku Java, ale należy pamiętać, że w Javie 8 klasy te w większości zastosowań zostały wyparte przez wyrażenia lambda (patrz „Podsumowanie”).

Spójrz na listing 4.5, na którym znajduje się implementacja `LinkedIterator` jako klasy anonimowej w metodzie `iterator()` klasy `LinkedStack`. Porównaj to z listingiem 4.4, na którym widać tę samą klasę zaimplementowaną jako klasa lokalna.

Listing 4.5. Wyliczenie zaimplementowane przy użyciu klasy anonimowej

```
public Iterator<Linkable> iterator() {
    // Klasa anonimowa jest zdefiniowana jako część instrukcji zwrotnej.
    return new Iterator<Linkable>() {
        Linkable current;
        // zamiana konstruktora na inicjator egzemplarza
        { current = head; }

        // Trzy poniższe metody są zdefiniowane przez interfejs Iterator.
        public boolean hasNext() { return current != null; }
        public Linkable next() {
            if (current == null)
              throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    };  // Zwróć uwagę na średnik, który jest w tym miejscu niezbędny i oznacza koniec instrukcji return.
}
```

Jak widać, składnia definicji klasy anonimowej i tworzenia jej egzemplarza składa się ze słowa kluczowego `new`, nazwy klasy i treści klasy w klamrze. Jeśli po słowie kluczowym `new` znajduje się nazwa klasy, to zostaje utworzona klasa anonimowa będąca podklasą klasy o tej nazwie. Jeżeli nazwa za słowem kluczowym `new` należy do interfejsu, jak w dwóch poprzednich przykładach, to klasa anonimowa implementuje ten interfejs i rozszerza klasę `Object`.

Składnia klas anonimowych nie przewiduje możliwości użycia klauzuli `extends` i `implements` ani określenia nazwy dla tej klasy.

Jako że klasa anonimowa nie ma nazwy, w jej treści nie da się zdefiniować konstruktora. Jest to jedno z podstawowych ograniczeń tego rodzaju klas. Wszystkie argumenty podane w nawiasie za nazwą nadklasy są niejawnie przekazywane do konstruktora tej nadklasy. A ponieważ klasy anonimowe często tworzy się jako rozszerzenia prostych klas, których konstruktory nie przyjmują żadnych argumentów, nawiasy w definicjach klas anonimowych często pozostają puste. W każdym przedstawionym przykładzie klasa anonimowa implementowała interfejs i rozszerzała klasę `Object`. Jako że konstruktor `Object()` nie przyjmuje żadnych argumentów, nawias był zawsze pusty.

Ograniczenia dotyczące klas anonimowych

Ponieważ klasa anonimowa jest rodzajem klasy lokalnej, klasy anonimowe mają podobne ograniczenia jak klasy lokalne. Klasa anonimowa nie może zawierać żadnych statycznych pól, metod ani klas, z wyjątkiem stałych `static final`. Interfejsów, wyliczeń i adnotacji nie można definiować anonimowo. Ponadto klasy anonimowe, podobnie jak lokalne, nie mogą być publiczne, prywatne, chronione ani statyczne.

Składnia definicji klasy anonimowej łączy definicję z tworzeniem egzemplarza. Nie należy używać takiej klasy zamiast klasy lokalnej, jeśli potrzeba więcej niż jednego egzemplarza przy każdym wykonywaniu danego bloku kodu.

Ponieważ klasa anonimowa nie ma nazwy, nie da się w niej zdefiniować konstruktora. Jeżeli konstruktor jest potrzebny, to należy utworzyć klasę lokalną, chociaż zamiast konstruktora często można użyć inicjatora egzemplarza.

Wprawdzie zakres zastosowań inicjatorów egzemplarzy (opisanych w podrozdziale „Domyślne wartości i inicjatory pól") nie ogranicza się do klas anonimowych, ale zostały one wprowadzone do języka właśnie ze względu na nie. Klasa anonimowa nie może zawierać definicji konstruktora, więc ma tylko konstruktor domyślny. Przy użyciu inicjatora egzemplarza można jednak to ograniczenie obejść.

Jak działają typy zagnieżdżone

W kilku poprzednich podrozdziałach opisaliśmy właściwości czterech rodzajów typów zagnieżdżonych. Wiedza ta powinna być wystarczająca, szczególnie dla tych programistów, którzy planują tylko używać tych typów. Ale niektórym w ich zrozumieniu może pomóc też znajomość sposobu ich implementacji.

Wprowadzenie typów zagnieżdżonych nie spowodowało zmian w maszynie wirtualnej Javy ani formacie plików klas tego języka. Natomiast dla interpretera nie istnieje coś takiego jak typ zagnieżdżony — wszystkie klasy są najwyższego poziomu.

Aby typ zagnieżdżony rzeczywiście zachowywał się jak typ zdefiniowany w innej klasie, kompilator Javy wstawia do generowanych klas specjalne ukryte pola, metody i argumenty konstruktora. Te ukryte pola i metody często określa się jako **syntetyczne**.

Aby zobaczyć, jakie sztuczki kompilator stosuje w celu zapewnienia działania typów zagnieżdżonych, można za pomocą narzędzia `javap` zdekompilować zawierające je pliki klas (szerzej na temat narzędzia `javap` piszemy w rozdziale 13.).

Implementacja typów zagnieżdżonych opiera się na tym, że kompilator `javac` tworzy dla każdego z nich osobny plik klasy reprezentujący klasę najwyższego poziomu. Te skompilowane pliki klas mają specjalne nazwy, których nie dałoby się utworzyć w normalnym kodzie użytkownika.

Przypomnijmy pierwszy przykład klasy `LinkedStack` (listing 4.1) zawierającej statyczny interfejs składowy o nazwie `Linkable`. W wyniku kompilacji tej klasy powstaną dwa pliki klas. Pierwszy z nich zgodnie z oczekiwaniami będzie mieć nazwę *LinkedStack.class*.

Ale drugiemu zostanie nadana nazwa *LinkedStack$Linkable.class*. Znak *$* jest automatycznie wstawiany przez kompilator. W pliku tym zostanie zapisana implementacja statycznego interfejsu składowego, o którym była mowa.

Jako że typy zagnieżdżone w czasie kompilacji są zamieniane na zwykłe klasy najwyższego poziomu, tracą one przywileje dostępu do składowych swojego kontenera. Jeśli zatem statyczny typ składowy używa składowej prywatnej (lub innej wymagającej zwiększonych uprawnień dostępu) typu go zawierającego, kompilator generuje syntetyczne metody dostępowe (z domyślnym dostępem pakietowym) i konwertuje wyrażenia używające prywatnych składowych w wyrażenia wywołujące te specjalnie wygenerowane metody.

Oto zasady nadawania nazw każdemu z czterech rodzajów typów zagnieżdżonych:

Typy składowe (statyczne i niestatyczne)
: Typom składowym nazwy są nadawane zgodnie ze wzorem *TypZawierający$TypSkładowy.class*.

Klasy anonimowe
: Jako że klasy anonimowe nie mają nazw, nazwy reprezentujących je plików są szczegółem implementacyjnym. Kompilator `javac` Oracle/OpenJDK oznacza je numerami (np. *TypZawierający$1.class*).

Klasy lokalne
: Klasom lokalnym nazwy nadawane są zgodnie z kombinacją (np. *TypZawierający$1TypSkładowy.class*).

Zobaczmy też, jak kompilator `javac` udostępnia syntetyczny dostęp w niektórych specyficznych przypadkach potrzebnych typom zagnieżdżonym.

Implementacja niestatycznej klasy składowej

Każdy egzemplarz niestatycznej klasy składowej jest związany z egzemplarzem klasy zawierającej. Kompilator egzekwuje tę zależność przez zdefiniowanie w każdej klasie składowej syntetycznego pola o nazwie `this$0`. W polu tym przechowywana jest referencja do egzemplarza zawierającego.

Konstruktor każdej niestatycznej klasy składowej otrzymuje dodatkowy parametr inicjujący to pole. Przy każdym wywołaniu konstruktora klasy składowej kompilator automatycznie przekazuje referencję do klasy zawierającej dla tego dodatkowego parametru.

Implementacja klasy lokalnej i anonimowej

Klasa lokalna może się odnosić do pól i metod zawierającej ją klasy z dokładnie tych samych powodów co niestatyczna klasa składowa. Klasa ta otrzymuje ukrytą referencję do klasy zawierającej w konstruktorze i zapisuje ją w prywatnym polu syntetycznym dodanym przez kompilator. Podobnie jak niestatyczne klasy składowe klasy lokalne mogą używać pól i metod prywatnych klasy zawierającej, ponieważ kompilator dodaje wszystkie niezbędne do tego metody dostępowe.

Klasy lokalne od klas składowych odróżnia to, że mogą odwoływać się do zmiennych lokalnych w zakresie, w którym są zdefiniowane. Najważniejszym ograniczeniem w tym przypadku jest jednak to, że klasy lokalne mogą się odwoływać tylko do zmiennych lokalnych i parametrów finalnych. Powód tego ograniczenia jest oczywisty w implementacji.

Klasa lokalna może wykorzystywać zmienne lokalne, gdyż kompilator `javac` automatycznie dodaje do niej prywatne pola egzemplarzowe do przechowywania kopii każdej zmiennej lokalnej używanej przez tę klasę.

Ponadto kompilator dodaje ukryte parametry do każdego konstruktora klasy lokalnej służące do inicjacji tych automatycznie utworzonych prywatnych pól. Klasa lokalna w rzeczywistości nie używa zmiennych lokalnych, a jedynie własnych prywatnych kopii tych zmiennych. Powodowałoby to niespójność, gdyby zmienne lokalne można było zmieniać poza klasą lokalną[3].

[3] Szerzej tym tematem zajmujemy się w opisie pamięci i stanu zmiennego w rozdziale 6.

Wyrażenia lambda

Wyrażenia lambda są jedną z najbardziej oczekiwanych nowości wprowadzonych w Javie 8. Umożliwiają one wpisywanie niewielkich porcji kodu w linii jako literałów i stosowanie bardziej funkcyjnego stylu programowania.

W rzeczywistości wiele z tych technik dało się stosować już od dawna za pomocą typów zagnieżdżonych, poprzez wywołania zwrotne i procedury obsługowe, ale problem stanowiła niezgrabna składnia. W szczególności, aby np. zdefiniować pojedynczy wiersz kodu w wywołaniu zwrotnym, trzeba było zdefiniować kompletnie nowy typ.

Jak napisaliśmy w rozdziale 2., definicja wyrażenia lambda składa się z listy parametrów (których typy są z reguły dedukowane) powiązanych z treścią metody:

```
(p, q) -> { /* treść metody */ }
```

Jest to bardzo zwięzły sposób reprezentowania prostych metod, który może całkowicie wyprzeć z użycia klasy anonimowe.

Wyrażenie lambda ma prawie wszystkie części metody, oczywiście z wyjątkiem nazwy. Z tego powodu wielu programistów lubi traktować te wyrażenia tak, jakby były „metodami anonimowymi".

Weźmy np. metodę `list()` z klasy `java.io.File`, która zwraca listę plików znajdujących się w katalogu. Ale przed zwróceniem tej listy przekazuje nazwę każdego pliku do obiektu `FilenameFilter`, który musi dostarczyć programista. Obiekt ten akceptuje lub odrzuca każdy z tych plików.

Poniżej znajduje się przykładowa definicja klasy `FilenameFilter`, przepuszczająca tylko pliki o nazwach zakończonych napisem *.java*. Jest to klasa anonimowa:

```
File dir = new File("/src" );  // katalog, którego pliki mają zostać zwrócone

// wywołanie metody list() z pojedynczą anonimową implementacją
// klasy FilenameFilter jako argumentem
String[] filelist = dir.list( new FilenameFilter() {
  public boolean accept(File f, String s) {
    return s.endsWith(".java" );
  }
});
```

Przy użyciu wyrażeń lambda kod ten można uprościć:

```
File dir = new File("/src" );  // katalog, którego pliki mają zostać zwrócone

String[] filelist = dir.list((f, s) -> { return s.endsWith(".java" ); });
```

Dla każdego pliku na liście zostaje wykonane wyrażenie lambda. Jeżeli metoda zwraca `true` (co ma miejsce w przypadku, gdy nazwa pliku kończy się napisem *.java*), to plik zostaje dodany do wyjściowej tablicy `filelist`.

Technika, w której blok kodu sprawdza, czy element kontenera spełnia pewien warunek, i zwraca tylko te elementy, które ten warunek spełniają, nazywa się **idiomem filtrowania** (ang. *filter idiom*) — jest to jedna ze standardowych technik programowania funkcyjnego, którą warto poznać trochę dokładniej.

Konwersja wyrażeń lambda

Gdy kompilator `javac` napotyka wyrażenie lambda, interpretuje je jako treść metody o specyficznej sygnaturze — tylko której metody?

W celu znalezienia odpowiedzi na to pytanie kompilator przegląda pobliski kod. Aby być prawidłowym kodem Java, wyrażenie lambda musi spełniać następujące warunki:

- Musi występować tam, gdzie jest oczekiwany egzemplarz typu interfejsowego.
- Oczekiwany typ interfejsowy powinien mieć dokładnie jedną metodę obowiązkową.
- Oczekiwana metoda interfejsu powinna mieć sygnaturę dokładnie odpowiadającą sygnaturze wyrażenia lambda.

Jeśli warunki te są spełnione, następuje utworzenie egzemplarza typu implementującego ten oczekiwany interfejs i treść lambdy zostaje użyta jako implementacja obowiązkowej metody.

Te nieco skomplikowane zasady wynikają z decyzji, aby zachować czystą nominatywność systemu typów Javy (tzn. by pozostał oparty na nazwach). Mówi się, że wyrażenie lambda jest **konwertowane** na egzemplarz odpowiedniego typu interfejsowego.

Niektórzy programiści lubią też nazywać typ, na który konwertowana jest lambda, **interfejsem zawierającym jedną metodę abstrakcyjną** (ang. *single abstract method type* — SAM). Należy zwrócić uwagę, że dla mechanizmu wyrażeń lambda przydatny jest jedynie interfejs zawierający tylko jedną metodę niedomyślną.

Mimo podobieństwa wyrażeń lambda do klas anonimowych lambdy **nie** są tylko cukrem syntaktycznym. Lambdy są zaimplementowane przy użyciu uchwytów do metod (opisanych w rozdziale 11.) i specjalnej nowej instrukcji kodu bajtowego maszyny wirtualnej `invokedynamic`.

Jak widać, wprowadzone w Javie 8 wyrażenia lambda zostały dopasowane do istniejącego systemu typów tego języka programowania, w którym główną rolę odgrywa typowanie nominalne.

Referencje do metod

Przypomnijmy, że wyrażenia lambda można traktować jak metody pozbawione nazw. Spójrz więc na poniższe wyrażenie lambda:

```
// W prawdziwym programie kod ten byłby pewnie krótszy dzięki inferencji typów.
(MyObject myObj) -> myObj.toString()
```

Wyrażenie to zostanie automatycznie przekonwertowane na implementację interfejsu `@FunctionalInterface` zawierającego jedną niedomyślną metodę przyjmującą jeden obiekt klasy `MyObject` i zwracającą obiekt typu `String`. Ale można odnieść wrażenie, że to przerost formy nad treścią, i dlatego w Javie 8 dodano składnię ułatwiającą odczyt i zapis takich rzeczy:

```
MyObject::toString
```

Jest to skrót o nazwie **referencja do metody** (ang. *method reference*), który polega na użyciu istniejącej metody jako wyrażenia lambda. Można to traktować jak zastosowanie istniejącej metody przy jednoczesnym zignorowaniu jej nazwy, co pozwala na wykonanie normalnej automatycznej konwersji lambdy.

Programowanie funkcyjne

Java to obiektowy język programowania. Ale od chwili dodania do niego wyrażeń lambda stał się znacznie bliższy także językom funkcyjnym.

Nie istnieje ogólnie przyjęta definicja **funkcyjnego języka programowania**, ale przynajmniej wszyscy się zgadzają, że język taki powinien umożliwiać reprezentowanie funkcji jako wartości, którą można zapisać w zmiennej.

W Javie od zawsze (od wersji 1.1) istnieje możliwość reprezentowania funkcji przez klasy wewnętrzne, ale składnia tego jest skomplikowana i niejasna. Sytuację tę znacznie poprawiły wyrażenia lambda, w związku z czym wielu programistów będzie się z pewnością starać wykorzystywać elementy programowania funkcyjnego w Javie, ponieważ stało się to o wiele łatwiejsze.

Pierwszą próbką programowania funkcyjnego, z którą prawdopodobnie zetkną się programiści Javy, będą trzy podstawowe i niezwykle przydatne techniki:

`map()`
: Technika mapy jest stosowana z listami i innymi kontenerami o podobnej budowie. Polega ona na przekazaniu funkcji, która zostaje użyta, do każdego elementu kolekcji i utworzeniu nowej kolekcji zawierającej wyniki zastosowania tej funkcji do każdego z tych elementów po kolei. W efekcie może nastąpić przekształcenie kolekcji jednego typu w kolekcję innego typu.

`filter()`
: Przykład zastosowania techniki filtrowania już przedstawiliśmy w opisie sposobu zamiany anonimowej implementacji klasy `FilenameFilter` na wyrażenie lambda. Filtr służy do utworzenia nowego podzbioru kolekcji na podstawie pewnych kryteriów. Zwróć uwagę, że w programowaniu funkcyjnym normą jest tworzenie nowej kolekcji zamiast modyfikowania istniejącej.

`reduce()`
: Technika redukcji występuje w kilku postaciach. Jest to operacja agregacyjna mogąca występować pod nazwami `fold`, `accumulate`, `aggregate` lub `reduce`. Jej działanie polega na tym, że bierze się wartość początkową i funkcję agregacyjną (bądź redukcyjną) i wykonuje się tę funkcję po kolei na każdym elemencie kolekcji, tworząc ostateczny wynik przez wykonanie serii wyników pośrednich — coś podobnego do sumy bieżącej.

W Javie te i kilka innych podstawowych technik mają pełne wsparcie. Bardziej szczegółowe objaśnienie implementacji znajduje się w rozdziale 8., poświęconym strukturom danych i kolekcjom Javy oraz przede wszystkim abstrakcji **strumienia**, dzięki której to wszystko jest możliwe.

Na koniec chcielibyśmy dodać kilka słów ostrzeżenia. Javę należy traktować jako język „trochę wspomagający programowanie funkcyjne". Nie jest to typowy język funkcyjny, nigdy też nie było takich aspiracji. Oto kilka cech Javy, które świadczą o tym, że z pewnością nie jest ona funkcyjnym językiem programowania:

- W Javie nie ma typów strukturalnych, co oznacza brak „prawdziwych" typów funkcyjnych. Każde wyrażenie lambda jest automatycznie konwertowane na odpowiedni typ nominalny.

- Wymazywanie typów stanowi utrudnienie dla programowania funkcyjnego — funkcje wyższego rzędu mogą tracić bezpieczeństwo typowe.
- Java jest inherentnie zmienna (patrz rozdział 6.), a zmienność jest często postrzegana jako wysoce niepożądana cecha funkcyjnych języków programowania.

Mimo tych niedogodności dostępność podstawowych narzędzi programowania funkcyjnego — a w szczególności takich idiomów, jak mapa, filtr i redukcja — jest wielkim krokiem naprzód dla całej społeczności skupionej wokół Javy. Te podstawowe idiomy są tak przydatne, że większość programistów Javy nigdy nie będzie potrzebować żadnych bardziej zaawansowanych udogodnień dostępnych w typowo funkcyjnych językach programowania.

Podsumowanie

Przeanalizowaliśmy system typów Javy i otrzymaliśmy wyraźny obraz jego podstawowych właściwości. Poniżej znajduje się zwięzły opis najważniejszych cech systemu typów języka Java:

Nominalny
: Nazwa typu w Javie jest bardzo ważna. W języku tym nie można używać typów strukturalnych, które są dostępne w niektórych innych językach programowania.

Statyczny
: Typy wszystkich zmiennych w Javie są znane w czasie kompilacji.

Obiektowy/imperatywny
: Kod źródłowy w Javie jest obiektowy i w całości musi się mieścić w metodach, które z kolei muszą się znajdować w klasach. Jedynie typy podstawowe uniemożliwiają przyjęcie zasady, że „wszystko jest obiektem".

Odrobinę funkcyjny
: W Javie możliwe jest stosowanie niektórych podstawowych technik programowania funkcyjnego, ale jest to głównie udogodnienie dla programistów, a nie coś poważniejszego.

Czasami z możliwością dedukcji typów
: Kod Java ma być maksymalnie czytelny (nawet dla początkujących programistów) i preferowane są w nim bezpośrednie instrukcje, nawet jeśli oznacza to konieczność stosowania powtórzeń informacji.

O wysokim poziomie zgodności wstecznej
: Java to głównie język biznesowy, więc zgodność wsteczna i ochrona istniejących baz kodu są traktowane priorytetowo.

Z wymazywaniem typów
: W Javie można używać typów parametryzowanych, ale informacje te są niedostępne w czasie wykonywania programu.

System typów Javy zmieniał się w czasie (chociaż była ta powolna i ostrożna ewolucja) i dzięki dodatkowi wyrażeń lambda stał się równorzędny z systemami typów innych popularnych języków programowania. Lambdy i metody domyślne stanowią największą zmianę od Javy 5 i wprowadzenia typów ogólnych, adnotacji oraz związanych z nimi rozwiązań.

Metody domyślne oznaczają poważną zmianę w metodyce programowania obiektowego w Javie — możliwe, że największą od czasu powstania tego języka. Od Javy 8 interfejsy mogą zawierać kod implementacyjny. To całkowicie zmienia naturę Javy. Wcześniej był to język z pojedynczym dziedziczeniem, a teraz obsługuje wielodziedziczenie (chociaż tylko zachowań — nadal nie da się dziedziczyć w ten sposób stanów).

Mimo tych innowacji system typów Javy nie dorównuje (i nie ma takich planów) pod względem możliwości systemom typów takich języków, jak Scala czy Haskell. System typów Javy z założenia ma być prosty, czytelny i łatwy do opanowania przez początkujących.

Ponadto Java wiele skorzystała na rozwoju technik typowania rozwijanych w innych językach w ciągu ostatnich 10 lat. Na przykład Scala jest statycznie typowanym językiem, w którym mimo to osiągnięto wiele cech funkcyjnego języka programowania dzięki użyciu inferencji typów. Twórcy Javy skorzystali z wielu rozwiązań zastosowanych w tym języku, mimo że języki te znacznie się między sobą różnią pod względem projektu.

Długo oczekiwane wyrażenia lambda zostały wreszcie dodane, dzięki czemu Java stała się jeszcze lepszym językiem programowania. To, czy większość zwykłych programistów Javy będzie potrzebować dodatkowych możliwości — z którymi nierozerwalnie wiąże się zwiększona złożoność — zaawansowanego (i znacznie mniej nominalnego) systemu typów, takiego jak w języku Scala, czy wystarczą im „odrobinę funkcjonalne narzędzia programistyczne" dodane w Javie 8 (np. mapa, filtr, redukcja itp.), okaże się w przyszłości. Powinno być ciekawie.

Skorowidz

.NET, 19

A

abstrakcja, 223
  Stream, 242
  wejścia i wyjścia, 273
adnotacja, 88, 131, 145, 146
  @Deprecated, 147, 284
  @override, 168
  @Override, 146, 147
  @SuppressWarnings, 147
  nazwa, 210
  tworzenie, 147
  typu, 148
adres URL, 278, 279
algorytm
  oznaczanie i usuwanie, 185, 186, 192
  usuwania śmieci, 185
alokacja wątkowa, 190
annotation, *Patrz:* adnotacja
aplikacja, 21
Apple, 20
array, *Patrz:* tablica
arytmetyka
  zmiennoprzecinkowa, 255
asercja, 70, 71
autoboxing, *Patrz:* opakowywanie automatyczne
autounboxing, *Patrz:* rozpakowywanie automatyczne

B

backtick, 305
benign data race, *Patrz:* dane łagodny wyścig
bezpieczeństwo, 120, 182, 220, 335
  współbieżności, *Patrz:* współbieżność bezpieczeństwo
bezpieczny moment, 187
biblioteka standardowa, 137
Bloch Joshua, 143
blocking queue, *Patrz:* kolejka blokująca
blokada, *Patrz:* monitor
błąd, 67
  składniowy, 144
bound method reference, *Patrz:* referencja wiązana do metody
bufor
  bajtów mapowany, 275
  MappedByteBuffer, 275
  NIO, 273

C

catch, *Patrz:* wyjątek przechwycenie
CMS, 192
codepoint, *Patrz:* jednostka kodowa
Collection, *Patrz:* kolekcja
Collection view, *Patrz:* kolekcja widok
concurrent mark and sweep, *Patrz:* CMS
czas, 258, 262, 263
  zapytanie, 261
  znacznik, 259

D

dane
  kontrola dostępu, 124
  łagodny wyścig, 250
  typ, *Patrz:* typ
data, 258, 262, 263
deklaracja
  import, 32, 93
  package, 32, 92
delegacja, 176
demon, 202, 304
doc comment, *Patrz:* komentarz dokumentacyjny
dokument
  miejscowy, *Patrz:* dokument śródliniowy
  śródliniowy, 306
dokumentacja, 212, 213
konstruktor, 114
dostępność pakietowa, 121
dyrektywa
  import, 95
  import static, 95
  package, 95
dziedziczenie, 100, 111, 128, 133, 177
  hierarchia, 154
  kontrola dostępu, 123, 124

E

eager evaluation, *Patrz:* wartościowanie gorliwe
egzemplarz, 143
encapsulation, *Patrz:* hermetyzacja
enum, *Patrz:* wyliczenie
evacuating collector, *Patrz:* śmieciarka ewakuacyjna
evacuation, *Patrz:* ewakuacja
ewakuacja, 189

F

filtr, 240, 241, *Patrz:* technika filtrowania
finalizacja, 193, 194
Fujitsu, 20
funkcja
  agregacyjna, 164
  bez nazwy, 80
  obsługi wejścia i wyjścia, 265, 270
    asynchroniczna, 275, 276, 277
    wady, 269
  redukcyjna, 164
  trygonometryczna, 257

G

G1, 193
garbage collector, *Patrz:* śmieciarka
Garbage First, *Patrz:* G1
GC, *Patrz:* śmieciarka
GC root, *Patrz:* korzeń GC
generator liczb pseudolosowych, 258
generic method, *Patrz:* metoda ogólna
graceful completion, *Patrz:* wzorzec eleganckiego zakończenia

H

heredoc, *Patrz:* dokument śródliniowy
hermetyzacja, 100, 119, 120, 137
HP, 20

I

IBM, 20
identyfikator, 34
implementacja, 12, 219, 220
  Stack, 231
  Vector, 231
inicjator
  egzemplarza, 110
  statyczny, 110
instrukcja, 32, 55, 56
  assert, 70
  break, 61
  break, 65
  continue, 65
  dekrementacji, 56
  do, 62
  for, 63, 64
  foreach, 64
  if, 58, 59
  if-else, 58, 59, 60
    klauzula else if, 59
  inkrementacji, 56
  pętli, *Patrz:* pętla
  przypisania, 56
  pusta, 57
  return, 61, 66
  switch, 60, 61
  synchronized, 66
  throw, 61, 67, 73
  try, 70
  try z zasobami, 70, 269
  try-catch-finally, 67, 68, 69
  tworzenia obiektu, 56
  while, 62
  wyrażeniowa, 56
  wywołania metody, 56
  z etykietą, 57
  złożona, 57
interfejs, 100, 131, 132, 134, 173
  AutoCloseable, 269
  BlockingQueue, 235, 236
  Callable, 312
  Cloneable, 82, 85, 225
  Closeable, 269
  Collection, 223, 224, 225, 226, 227, 231, 235
  Comparable, 168
  CompletionHandler, 276
  definicja, 132
  Future, 276
  implementowanie, 133, 135
  Iterable, 223
  Iterator, 223, 230
  java.io.ObjectStreamConstants, 172
  java.io.Serializable, 137
  java.lang.Comparable, 171
  java.lang.Iterable, 229
  java.util.function.Function, 174
  java.util.List, 93, 137
  javax.script, 310
  komentarz dokumentacyjny, *Patrz:* komentarz dokumentacyjny interfejsu
  List, 223, 224, 227, 228, 231, 245
  Map, 223, 225, 231, 232, 233, 234
  Method Handles, 297
  nazwa, 210
  Path, 271
  Predicate, 240
  pusty, 137
  Queue, 224, 235, 236
  refleksji, 292, 294
  rozszerzanie, 133
  Serializable, 82, 225
  Set, 223, 224, 226
  SortedMap, 223, 233
  SortedSet, 223, 227
  strumienia, 242
  TemporalQuery, 261
  użytkownika graficzny, 317, 329
  zawierający jedną metodę abstrakcyjną, *Patrz:* SAM
  znacznikowy, 137
interpreter, 21, 26
iterator, 64

J

Java, 20, 27
  bezpieczeństwo, 26, 30
  ekosystem, 22
  historia, 23
  implementacja, *Patrz:* implementacja

kod przenośny, 219, 220
kod źródłowy, 24
kolekcja, *Patrz:* kolekcja
maszyna wirtualna, *Patrz:* JVM
OpenJDK, *Patrz:* OpenJDK
składnia, *Patrz:* składnia
środowisko wykonawcze, 19
wersja, 23, 29
wydajność, 29
zgodność wsteczna, 135
Java EE, 20
Java Enterprise Edition, *Patrz:* Java EE
Java ME, 20
Java Mobile Edition, *Patrz:* Java ME
Java SE, 20
Java Standard Edition, *Patrz:* Java SE
Java Virtual Machine, *Patrz: JVM*
pakiet, 223
interfejs, 137, 173
javac, *Patrz:* program javac
javadoc, 317
program, 213
jednostka
kodowa, 37
kompilacji, 32
język
bezpieczny pod względem typów, 182
C, 27
C++, 27
funkcyjny, 80, 164
Haskell, 80
interpretowany, 26
Java, *Patrz:* Java
JavaScript, 28, 301, 302, 309
Nashorn, 313, 314
rozszerzenie, 313
wyrażenie lambda, 312
Lisp, 80
obiektowy, 27, 164
OCaml, 80
PHP, 27
proceduralny, 27
typowany
dynamicznie, 28
statycznie, 28
języka
kontrola typów
dynamiczna, 131
statyczna, 131
Jigsaw, 334
JIT, 22, 25, 29
Just-In-Time compilation, *Patrz:* JIT
JVisualVM, 329
JVM, 19, 20, 21, 328, 332
HotSpot, 22, 29
proces aktywny, 324

K

kanał, 274
klasa, 77, 99
abstrakcyjna, 101, 126, 133, 173
anonimowa, 149, 158, 159, 161
ograniczenia, 159
ArrayList, 137
AsynchronousFileChannel, 275
AsynchronousServerSocket ↪Channel, 275
AsynchronousSocket ↪Channel, 275
bez dokumentacji, 220
Boolean, 90
BufferedReader, 267
BufferedWriter, 267
Byte, 39, 90
Channel, 275, 276, 277
Character, 37, 90
Class, 80
Classloader, 288
ConcurrentHashMap, 233
ConcurrentSkipListMap, 233
CopyOnWriteArraySet, 226
DateTimeFormatter, 259
definiowanie, 78, 101
dezasembler, 328
Double, 40, 91
efektywnie niezmienna, 250
Error, 76
Exception, 76, 181
Externalizable, 216
File, 265, 266, 275
wady, 269
FileInputStream, 267
FileOutputStream, 267
FileReader, 267
Files, 270
FileWriter, 267
finalna, 112
Float, 40, 91
hierarchia, 113, 128
identyfikator, 34
implementacja interfejsu, 100, 101, 132
inicjacja, 287
InputStream, 267
InputStreamReader, 267
Integer, 39, 91
java.io.InterruptedIOException, 181
java.lang.Character, 34
java.lang.ClassLoader, 289
java.lang.Error, 180
java.lang.Exception, 180
java.lang.invoke.MethodHandle, 298
java.lang.Math, 115, 257
java.lang.Object, 82, 112, 168, 286
java.lang.reflect.Proxy, 295
java.lang.String, 112
java.lang.System, 115
java.math.BigDecimal, 256
java.time.Duration, 259
java.util.AbstractList, 173
java.util.Arrays, 86
java.util.Collections, 237
java.util.Date, 258, 263
java.util.Formatter, 221
komentarz dokumentacyjny, 219
konkretna, 126
konsolidacja, 285
kontrola dostępu, 121, 123, 124
LinkedList, 236
lokalna, 149, 155, 156, 161
zakres, 156
Long, 39, 91
ładowanie, 24, 285, 287, 288, 289

klasa
hierarchia programów, 290
Object, 172
ObjectInputStream, 172
ObjectOutputStream, 172, 173
otokowa, 90
OutputStream, 267
Paths, 272
Pattern, 251
pośrednicząca dynamiczna, 295, 296, 297
PrintWriter, 267
przygotowywanie do użycia, 286
publiczna, 96
Reader, 267
rozszerzanie, 100, 101, 111
RuntimeException, 181
serializacja, 215, 216
ServerSocket, 280
Short, 39, 90
singletonowa, 180
składowa, 149, 152, 153, 161
Socket, 280
Stream, 243
String, 37, 79, 128, 247
StringBuffer, 249
StringBuilder, 249
sun.misc.Unsafe, 220
sygnatura, 100
Thread, 197, 201
Throwable, 180, 181
ukrywanie danych, *Patrz:* hermetyzacja
URL, 278
URLClassLoader, 289
URLConnection, 278
usunięta, 331
weryfikacja, 286
wewnętrzna, 149
Writer, 267
załadowana, 331
klucz, 231
kod
aplikacji, 21
bajtowy, 25, 26
invokedynamic, 297
weryfikacja, 26
instrukcji, 25
źródłowy, 32
kompilator, 318
kolejka, 234, 236
blokująca, 235
FIFO, 234
LIFO, 234
priorytetowa, 234
kolekcja, 137, 223, 239
konwersja na tablicę, 238
opakowaniowa, 236
przekształcenie, 164
pusta, 237
widok, 231
komentarz, 33
dokumentacyjny, 33, 212, 213, 215
interfejsu, 219
klasy, 219
konstruktora, 219
metody, 219
odniesienie, 217
pakietu, 219
jednowierszowy, 33
skryptowy, 305
uniksowy, 305
wielowierszowy, 33, 212
kompilacja na czas, *Patrz:* JIT
kompilator
JIT, 320
kliencki, 320
serwerowy, 320
kompozycja, 175
konstruktor, 32, 73, 107, 133
definiowanie, 107, 108
domyślny, 114, 115
komentarz dokumentacyjny, 219
łańcuch, 114
podklasy, 113
wywoływanie, 108
kontener, 137, 138, 164
ByteBuffer, 273
konwersja, 128
rozszerzająca, 40, 127
rzutowania, 55
tablicy, *Patrz:* tablica konwersja
zawężająca, 40, 127, 128
korzeń GC, 187

L

lazy evalutation, *Patrz:* wartościowanie leniwe
List, *Patrz:* lista
lista, 138, 164, 223, 230
oparta na tablicach, 223
literał, 35, 42
całkowitoliczbowy, 41
binarny, 38
notacja, 38
ósemkowy, 38
liczbowy, 35
łańcuchowy, 33, 34, 35, 79
typowy, 80
zmiennoprzecinkowy, 39
znakowy, 35

Ł

łańcuch, 33, 79, 247
interpolacja, 305
kod skrótu, 250
konkatenacja, 248
konwersja na wartości liczbowe, 39
niezmienność, 249
tekstowy, *Patrz:* łańcuch
wielowierszowy, 306
wyszukanie, *Patrz:* wyrażenie regularne

M

Map, *Patrz:* słownik
mapa, *Patrz:* technika mapy
mark and sweep, *Patrz:* algorytm oznaczanie i usuwanie
maszyna wirtualna
HotSpot, 185, 187, 190, 320, 334
sterta, 191
Javy, *Patrz:* JVM
mechanizm
finalizacji, 29
ładowania klas, 24
metaadnotacja, 147
@Retention, 147
@Target, 147

metaznak, 251, 252
method reference, *Patrz:* referencja do metody
metoda, 32, 71
  abs, 257
  abstrakcyjna, 72, 74, 126
  accept, 280
  add, 225, 226, 235
  addAll, 226, 227
  allocateDirect, 273
  anonimowa, 73
  argument, 71, 73
    zmienna liczba, 76
  arraycopy, 86, 238
  Arrays.toString, 86
  AsynchronousFileChannel.↪open, 277
  binarySearch, 86
  brakująca, 245
  Byte.parseByte, 39
  call, 312
  Class.forName, 289
  Class::defineClass, 297
  Classloader::defineClass, 285
  clear, 225
  clone, 82, 85, 172
  collect, 240
  Collection.remove, 235
  Collection::parallelStream, 245
  Collection::removeIf, 245
  Collection::spliterator, 245
  Collection::stream, 245
  compareTo, 171, 236
  completed, 276
  contains, 226
  countStackFrames, 203
  deepEquals, 86
  deepHashCode, 86
  deepToString, 86
  defineClass, 289
  DELETE, 279
  destroy, 203
  domyślna, 135, 239
    implementacja, 136
  dostępowa, 177
  drainTo, 235
  drukująca tekst sformatowany, 76
egzemplarza, 105, 152
egzemplarzowa, 102, 119, 174, 175
element, 236
equals, 86, 90, 170
  przesłanianie, 171
exp, 257
fabryczna, 273
failed, 276
filter, 240, 241
finalize, 193, 194
finalna, 74, 126
findConstructor, 298
findGetter, 298
findSetter, 298
findStatic, 298
findVirtual, 298
firstKey, 233
flatMap, 244
floor, 257
format, 76, 221
generyczna, *Patrz:* metoda ogólna
get, 227, 276
GET, 279
getDeclaredMethod, 294
getId, 201
getInstance, 180
getMethod, 294
getName, 201
getPriority, 201
getState, 202
hashCode, 170, 171, 250
hasNext, 230
HEAD, 279
headMap, 233
identyfikator, 34
inicjacyjna, 109, 110
Integer.parseInt, 39
interrupt, 202
invoke, 299
invokeExact, 299
isAlive, 202
isDone, 276
isJavaIdentifierPart, 34
isJavaIdentifierStart, 34
iterator, 230, 231
java.util.Arrays.equals, 90
jjs, 302, 303, 304
join, 202
jrunscript, 302, 303
klasowa, 74, 102, 150, 174, 175
klasy, 104
klauzula throws, 73
komentarz dokumentacyjny, *Patrz:* komentarz dokumentacyjny metody
lastKey, 233
List::sort, 245
loadClass, 289
log, 257
log10, 257
macierzysta, 219
main, 96, 319
map, 241
Map::compute, 245
Map::computeIfAbsent, 245
Map::computeIfPresent, 245
Map::forEach, 245
Map::getOrDefault, 245
Map::merge, 245
Map::putIfAbsent, 245
Map::remove, 245
Map::replace, 245
Math.ceil, 41
Math.floor, 41
Math.round, 41
Math.sqrt, 46
max, 257
MethodHandles.lookup, 298
min, 257
modyfikator, *Patrz:* modyfikator
nazwa, 72, 73, 210
next, 230
niepubliczna, 294
notify, 204, 205
offer, 235, 236
ogólna, 72, 143
opakowująca, 236
OPTIONS, 279
parametr, 210
peek, 236
poll, 235, 236
pop, 204
POST, 279
pow, 257

metoda
printf, 221
println, 221
prywatna, 126
przeciążanie, 73, 95
przesłanianie, *Patrz:* przesłanianie
put, 235, 236
PUT, 279
queryFrom, 261
reduce, 241, 244
referencja wiązana, *Patrz:* referencja wiązana do metody
remove, 225, 226, 235
removeAll, 227
resume, 203
retainAll, 225, 227
Runtime.exec, 219
set, 227
setAccessible, 294, 298
setDaemon, 202
setName, 201
setPriority, 201
setUncaughtException ↪Handler, 202
singleton, 237
singletonList, 237
singletonMap, 237
skutki uboczne, 72
sleep, 202
sort, 86
specyfikacja, 72
start, 202
statyczna, 39, 126
stop, 203
stream, 242
String::hashCode, 250
subList, 228
subMap, 233
suspend, 203
sygnatura, 72, 100, 297
typ, 73
synchronizowana, 199
System.arraycopy, 85
System.getenv, 219
System.out.printf, 76
System.out.println, 73, 175, 248
tailMap, 233
take, 235, 236
toString, 170, 248
TRACE, 279
uchwyt, 297, 298
uogólniona, *Patrz:* metoda ogólna
valueOf, 248
varargs, 76
wait, 202, 204, 205
writeExternal, 216
wyjątek, 75
wyjątek kontrolowany, 72, 73, 76
wyszukiwanie, 297
wirtualne, 118
wywołanie, 55, 56
żądania, 279
Microsoft .NET, 19
modyfikator, 73, 74, 101, 129
abstract, 74, 101, 129, 132
default, 129, 132
dostępu, 100, 103, 120
final, 74, 101, 103, 112, 129
native, 74, 130
private, 74, 100, 120, 130
protected, 74, 100, 120, 130
public, 74, 100, 120, 130
static, 74, 103, 104, 130
strictfp, 75, 101, 130
synchronized, 75, 130
transient, 103, 130
volatile, 103, 130
monitor, 200, 203, 331

N

nadinterfejs, 133
nadklasa, 100, 112, 154
pola ukrywanie, 115
nadrzędność hierarchia, 154
Nashorn, 301, 302
dowiązanie symboliczne, 308
funkcja pomocnicza, 307
wywoływanie Javy, 310
zmienna specjalna, 305
Nashorna, 309
polecenie powłoki, 303
nominal typing, *Patrz:* typowanie nominalne

O

obiekt, 77, 78, 79, 100
alokacja, *Patrz:* alokacja
awansowanie, 190
domyślnie widoczny, 198
grupa, *Patrz:* kolekcja
java.lang.Throwable, 180
klasy, 283, 284
Method, 293
nieużywany, 190, 191, 192
osiągalny, 187
pokolenie, *Patrz:* pokolenie
porównywanie, 90
producent, 143
przejściowy, 188
składowa, 54
tworzenie, 55, 56, 78
próbkowanie, 332
zmienność, 198
żywy, *Patrz:* obiekt osiągalny
odśmiecanie z podziałem na pokolenia, 189
ograniczenie parametrów typu, *Patrz:* typ wieloznaczny z ograniczeniami
opakowywanie automatyczne, 91
OpenJDK, 20
operator, 35, 42, 44
!, 50
!=, 49
%, 47
&, 50, 51
&&, 46, 49, 50
(), 55
(), 55
*, 47
., 54
/, 47
?:, 46, 53
[], 55
^, 51, 52
|, 50, 51
||, 46, 50
~, 51
+, 46, 47
++, 46, 48

+=, 47, 53
<, 49
<<, 52
<=, 49
=, 53
==, 40, 48, 49, 90, 170
>, 49
->, 55
>=, 49
>>, 52
>>>, 52
alternatywa bitowa, 51, 52
alternatywa logiczna, 50
argument, 45
arytmetyczny, 46
binarny, 45
bitowa alternatywa wykluczająca, 52
bitowy, 51
definicji wyrażenia lambda, 55
dekrementacji, 48
dodawania, 46
dopełnienie bitowe, 51
dostępu do elementów tablicy, 55
dostępu do składowych obiektu, 54
dwuargumentowy, *Patrz:* operator binarny
dzielenia, 47
dzielenia modulo, 47
iloczyn bitowy, 51
iloczyn logiczny, 50
instanceof, 54
jednoargumentowy, 45
  minus, 47
konwersji rzutowania, 55
logiczna alternatywa wykluczająca, 51
logiczny, 49
łączność, 42, 44
mniejszości, 49
mniejszy lub równy, 49
mnożenia, 47
negacja logiczna, 50
new, 55, 107
nierówności, 49
niskopoziomowy, 51
odejmowania, 47
  postinkrementacji, 48
  preinkrementacji, 48
  priorytet, 42
  przeciążanie, 27
  przesunięcia, 51
    w lewo, 52
    w prawo bez znaku, 52
    w prawo ze znakiem, 52
  przypisania, 52, 53
  równości, 40, 48, 90
  skutki uboczne, 46
  trójargumentowy, 45
  tworzenia nowego obiektu, 55
  warunkowa alternatywa logiczna, 50
  warunkowy, 45, 53
  warunkowy iloczyn logiczny, 49
  większy lub równy, 49
  wywołania metody, 46, 55
Oracle Corporation, 20

P

pakiet, 91, 218, 335
  deklaracja, 92
  java.awt.peer, 220
  java.lang, 92, 93
  java.lang.annotation, 147
  java.lang.concurrent, 235
  java.lang.reflect, 91
  java.nio.channels, 274
  java.time.chrono, 259
  java.time.format, 259
  java.time.temporal, 259
  java.time.zone, 259
  java.util.concurrent, 233, 331
  javax.net, 278
  javax.script, 308, 309
  komentarz dokumentacyjny, *Patrz:* komentarz dokumentacyjny pakietu
  kontrola dostępu, 120
  nazwa, 92, 209
  org.apache.commons.net, 92
  org.w3c, 91
  p, 93
pamięć
  alokacja, 327
  współdzielenie, 195
  wyciek, 186, 269
  zarządzanie, 185
para zastępcza, 37
PECS, 143
pętla, 62
  do, 62
  for, 63, 64, 229
    część inicjacyjna, 58
  foreach, 64, 229
  while, 62
planista, 195
plik, 95, 266
  .java, 96
  JAR, 209
  klasy, 283
  nazwa, 96, 221
  overview.html, 219
  package.html, 219
podklasa, 100, 111, 112, 154
  konstruktor, *Patrz:* konstruktor podklasy
pokolenie, 188, 190
  długość życia przewidywana, 188
  młode, 191
    Eden, 191
  stare, 191
pole, 32
  deklaracja, 102
  egzemplarza, 105, 152
  egzemplarzowe, 102, 133
  klasowe, 102, 103, 150
    wartość domyślna, 109
  nazwa, 210
  statyczne, 103, 104
    publiczne, 104
polecenie
  curl, 305
  jar, 322
polimorfizm, 27
Postel Jon, 281
powłoka
  jjs, 304
  Nashorna, 303
prawo Postela, 281
primitive specialization, *Patrz:* klasa Stream specjalizacja podstawowa

primitive type, *Patrz:* typ prosty
priority queue, *Patrz:* kolejka priorytetowa
profil, 335
  kompaktowy, 335, 337, 338
program, 96
  autor, 214
  do ładowania klas, 290, 291
  jar, 317, 321, 322
  java, 317, 319, 320, 321
  javac, 24, 25, 36, 218, 317, 318, 319, 323
    przełącznik, 318
  javadoc, 34, 212, 213, 219, 322
  javap, 317, 328, 329
  jconsole, 329
  jdeps, 317, 323, 324
  jinfo, 317, 326
  jmap, 317, 327, 332
  jps, 317, 324, 325
  jstack, 317, 327, 331
  jstat, 317, 325
  jstatd, 317, 324, 325, 326, 329
  jvisualvm, 317, 329
  tworzenie, 96
  uruchamianie, 96
  zbiór roboczy, 188
programowanie funkcyjne, 164
projektowanie obiektowe, 172
protokół
  HTTP, 278
  IP, 282
  IPv6, 282
  sieciowy, 278
  TCP, 280
przepełnienie, 38
przesłanianie, 100, 116, 117, 147, 171
  wywoływanie, 118
przestrzeń
  nazw, 28, 32, 91, 92
  obiektów ocalałych, 190

Q

queue, *Patrz:* kolejka

R

Red Hat, 20
redukcja, 241, *Patrz:* technika redukcji
referencja, 167, 186
  do metody, 163
  kopia, 89
  this, 106
  wiązana do metody, 241
refleksja, 292, 294, 295
regex, *Patrz:* wyrażenie regularne
regexp, *Patrz:* wyrażenie regularne
regulator, 262
rozpakowywanie automatyczne, 91
run until shutdown, *Patrz:* wzorzec działaj do zamknięcia
rzutowanie, 41
  konwersja, 55

S

safepoint, *Patrz:* bezpieczny moment
SAM, 163
SAP, 20
scheduler, *Patrz:* planista
sekcja krytyczna, 199
sekwencja specjalna, 36, *Patrz też:* znak \u
separator, 35
Set, *Patrz:* zbiór
single abstract method type, *Patrz:* SAM
składnia, 31, 34
  diamentowa, 139
składowa, 32, 132, 218
  klasowa, *Patrz:* składowa statyczna
  kontrola dostępu, 121, 123, 124
  niestatyczna, 100
  statyczna, 100, 101, 150
    importowanie, 94, 95
  widoczność, 179
słaba hipoteza pokoleń, *Patrz:* WGH
słownik, 223, 231, 240, 241
słowo
  kluczowe, 33, 34
    @interface, 147
    assert, 70
    break, 61, 65
    class, 101
    default, 100, 132
    extends, 101, 142
    implements, 101, 133
    import static, 94
    interface, 132
    new, 83
    null, 80
    package, 92
    protected, 179
    public, 73
    static, 73, 110
    super, 119, 142
    synchronized, 199, 200
    this, 106, 108
    throws, 74
    transient, 216
    unsigned, 38
    void, 73
    volatile, 200
  zarezerwowane, 34
    default, 34
    false, 35, 36
    final, 34
    null, 35
    true, 35, 36
stała, 172
  finalna, 210
  MAX_VALUE, 39, 40
  MIN_VALUE, 39, 40
  NaN, 40
  NEGATIVE_INFINITY, 40
  POSITIVE_INFINITY, 40
  statyczna, 210
stan, 101
standard IEEE 754-1985, 39
statement, *Patrz:* instrukcja
stop-the-world pause, *Patrz:* STW
strumień, 164
  wejścia i wyjścia, 267
STW, 187

Sun, 20
superclass, *Patrz:* nadklasa
symbol wieloznaczny, 141, *Patrz też:* typ wieloznaczny

Ś

śmieciarka, 185, 186
  ewakuacyjna, 189, 190
  implementacja, 185

T

tablica, 81
  alokacji, 186
  długość, 81, 83
  element, 55, 81, 84
    numeracja, 81
  indeks, 81
  inicjowanie, 83
  iterowanie, 85
  konwersja, 128
    na kolekcję, 238
  kopiowanie, 85
  kowariancja, 82, 143
  przeszukiwanie, 86
  skrótów, 223
  sortowanie, 86
  typ, 81, 82
  wielowymiarowa, 86
technika
  filtrowania, 164, 165, 240, 241
  mapy, 164, 165
  redukcji, 164, 165, 241
  refleksji, 292, 294, 295
  słownika, 240, 241
  użycia buforów bezpośrednich, 273
thread, *Patrz:* wątek
thread-local allocation buffer, *Patrz:* alokacja bufor wątkowy
throw, *Patrz:* wyjątek zgłoszenie
token, 34, 35
  leksykalny, 32
TWR, *Patrz:* instrukcja try z zasobami
typ, 218
  boolean, 36, 40
  byte, 38, 41
  ByteBuffer, 273
  całkowitoliczbowy, 35, 36, 38, 40, 41
    dzielenie, 39
    modulo przez zero, 39
    reprezentacja, 254
  char, 36, 37, 41, 79
  Class, 294
  czasu kompilacji, 144, 145
  czasu wykonywania, 144
  double, 39, 40
    nieskończoność, 39
    zero, 40
  egzemplarz, 54
  float, 39, 40
  generyczny, *Patrz:* typ ogólny
  inferencja, 81
  int, 37, 38, 41, 51
  java.lang.Throwable, 180
  kontrawariancja, 142
  kowariancja, 142, 143
  logiczny, *Patrz:* typ boolean
  long, 38, 41, 51
  MethodType, 297
  obiektowy, 36
  Object, 294
  ogólny, 91, 137, 138, 144
  parametr, 139, 141
    ograniczenie, *Patrz:* typ wieloznaczny z ograniczeniami
  parametryzowany, *Patrz:* typ ogólny
  Path, 271, 272
  podstawowy, 167
  prosty, 35, 88, 167
    konwersja, 40, 41, *Patrz też:* konwersja
    rzutowanie, rzutowanie
  referencyjny, 32, 36, 87, 88, 100, 131
    nadklasa, 128
    nazwa, 209
  short, 38, 41
  składowy
    niestatyczny, 149, 152, 153, 161
    statyczny, 149, 151, 161
  string, 37
  wariancja, 142
  wartościowy, 167
  wieloznaczny, 141, 142
  wyliczeniowy, 210
  wymazywanie, 140
  zagnieżdżony, 148, 149, 160
    nazwa, 161
  zaimportowany, 93
  zbiorczy, 88
  zmiennoprzecinkowy, 35, 39, 40, 75
    reprezentacja, 255
  zwrotny, 45
    kowariantny, 116
type safe language, *Patrz:* język bezpieczny pod względem typów
typowanie nominalne, 137

V

VisualGC, 334
VisualVM, 329

W

wartościowanie
  gorliwe, 244
  leniwe, 244
wątek, 75, 195, 203
  cykl życia, 196
  licznik programu, 195
  sterta, 197
  stos, 195, 197
  uśpiony, 197
weak generational hypothesis, *Patrz:* WGH
WGH, 188, 189
wielodziedziczenie, 27
wielowątkowość, *Patrz:* współbieżność
wiersz poleceń, 21, 317
  uruchamianie skryptów, 303
  -version, 214

wildcard, *Patrz:* typ wieloznaczny
wrapper class, *Patrz:* klasa otokowa
wrapper method, *Patrz:* metoda opakowująca
współbieżność, 185, 195, 203
  bezpieczeństwo, 198
wstrzykiwanie zależności, 180
wydra, 34
wyjątek, 67, 181
  ArithmeticException, 39
  ArrayStoreException, 82
  ClassCastException, 225
  CloneNotSupportedException, 82, 85
  ClosedByInterruptException, 202
  FileNotFoungException, 75
  IllegalAccessException, 297
  java.io.EOFException, 181
  java.io.FileNotFoundException, 181
  java.lang.ArrayIndexOutOfBoundsException, 181
  kontrolowany, 72, 73, 75, 76, 181
  MalformedURLException, 76
  niekontrolowany, 72, 75, 181, 225
  NullPointerException, 75, 148, 225
  obsługa, 67, 68
  projektowanie, 180
  przechwycenie, 67
  UnsupportedOperationException, 174
  zgłoszenie, 67
wykluczanie, 199
wyliczenie, 88, 131, 145
  ElementType, 147
  RetentionPolicy, 147
  Thread.State, 196
wyrażenie, 42
  inicjacyjne, *Patrz:* inicjator
  lambda, 55, 80, 162, 163, 239, 312
    konwersja, 163
    składnia, 80
  podstawowe, 42
  regularne, 250, 252
    metaznak, 251, 252
wzorzec
  Dekorator, 176
  działaj do zamknięcia, 201
  eleganckiego zakończenia, 201
  projektowy, 167
  Singleton, 179, 180
  wyszukania w tekście, *Patrz:* wyrażenie regularne

Z

zakres
  dostępności zmiennej, 58
  leksykalny, 157
  tymczasowy, 157
zbiór, 223, 225
zmienna, 42
  globalna, 104
  identyfikator, 34
  inicjator, 57
  lokalna, 57, 186
    nazwa, 210
    zakres, 157, *Patrz też:* zakres
  zakres dostępności, 58
zmienność zawartości obiektowej, 167
znacznik
  dokumentacyjny, 213
    @author, 214
    @deprecated, 215
    @exception, 215
    @link, 213, 219
    @param, 214
    @return, 214
    @see, 215, 217, 219
    @serial, 215
    @serialData, 216
    @serialField, 216
    @since, 215
    @throws, 215
    @version, 214
  HTML, 213
  śródliniowy, 216
    @code, 217
    @docRoot, 217
    @inheritDoc, 217
    @link, 216, 217
    @linkplain, 216, 217
    @literal, 217
    @value, 217
znak
  \', 37
  !, 50
  !=, 49
  ", 37
  #, 305
  #!, 308
  $, 34
  %, 47
  &, 50, 51
  &&, 50
  (), 55
  *, 47, 51, 251
  */, 33, 212
  ., 54
  /, 47
  /*, 33, 212
  /**, 33, 212
  //, 33, 305
  ?:, 53
  [], 55
  \\, 36, 37
  ^, 51, 52
  _, *Patrz:* znak podkreślenia
  |, 51
  +, 46, 47
  ++, 48
  +=, 47
  <<, 52
  =, 53
  ==, 40, 48, 90, 170
  >, 49
  ->, 55
  >=, 49
  >>, 52
  >>>, 52
  \000, 36
  \b, 37
  backtick, 305
  biały, 33
  chiński, 37
  cudzysłowu, 33, 37
  dolara, 34
  \f, 37

Han, 37
interpunkcyjny, 34, 35
kanji, 34
Latin-1, 37
\n, 37, 221
nowego wiersza, 33, 37
podkreślenia, 34
podwójnego cudzysłowu, 34, 37, 79
\r, 37, 221
\r\n, 221
separatora, 221
spacji, 33
\t, 36, 37
tabulatora, 33, 37
\u, 36, 37
u05D0, 36
Unicode, 36, 37, 73
UTF-8, 36
waluty, 34
\ xxx, 37
獺, *Patrz:* znak kanji

O'REILLY®

Java w pigułce

Java to język programowania wybierany wszędzie tam, gdzie są wymagane najwyższe bezpieczeństwo i wydajność. Znajduje zastosowanie przy tworzeniu systemów bankowych oraz zaawansowanych aplikacji do zarządzania przedsiębiorstwami. Są to obszary, w których pomyłki bywają niezwykle drogie. To niejedyne zastosowania Javy! Ten język sprawdza się również wtedy, gdy trzeba szybko stworzyć aplikację internetową (niezależnie od jej wielkości) albo narzędzia różnego przeznaczenia. Java przyda się wszędzie!

W ostatnim czasie na rynku pojawiły się dwie kolejne wersje tego języka, oznaczone numerami 7 i 8. Zawierają one wiele nowości i ulepszeń, dzięki którym życie programisty staje się prostsze, a tworzone oprogramowanie — lepsze. Najnowsze wydanie tej cenionej książki zostało uzupełnione o informacje na temat tych właśnie wersji. Dzięki niej błyskawicznie poznasz i wykorzystasz nowe techniki w codziennej pracy. Sięgnij po ten podręcznik i poznaj najlepsze techniki programowania współbieżnego, zasady podejścia obiektowego oraz możliwości asynchronicznego wykonywania operacji wejścia-wyjścia. Ta książka jest obowiązkową lekturą dla wszystkich programistów języka Java!

Przekonaj się, jak:

- wykorzystać najnowsze elementy języka Java
- zwiększyć wydajność dzięki narzędziom pakietu OpenJDK
- wykonywać asynchroniczne operacje wejścia-wyjścia
- używać narzędzi pakietu OpenJDK

Poznaj najskrytsze tajemnice języka Java!

Benjamin J. Evans — Java Champion, JavaOne Rockstar, współzałożyciel firmy jClarity, specjalista w zakresie oceny wydajności dla zespołów programistycznych i operacyjnych, prelegent zajmujący się tematyką platformy Java, wydajności i współbieżności.

David Flanagan — specjalista ds. programowania interfejsów użytkownika, autor książek poświęconych językowi Ruby oraz bibliotece jQuery.

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

ISBN 978-83-283-0623-3