ORACLE

Java
Kompendium programisty

Wydanie IX

Herbert Schildt

Java™

Helion

Oracle Press™

Tytuł oryginału: Java™ The Complete Reference, Ninth Edition

Tłumaczenie: Piotr Rajca
na podstawie „Java. Kompendium programisty. Wydanie VIII”
w tłumaczeniu Mikołaja Szczepaniaka

ISBN: 978-83-283-0812-1

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javkp9
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javkp9.zip

Printed in Poland.

Spis treści

CZĘŚĆ II
Biblioteka języka Java

CZĘŚĆ III
Wprowadzenie do programowania GUI przy użyciu pakietu Swing

ROZDZIAŁ

12

Wyliczenia, automatyczne opakowywanie typów prostych i adnotacje (metadane)

Ten rozdział dotyczy trzech stosunkowo nowych elementów języka programowania Java: typów wyliczeniowych, automatycznego opakowywania typów prostych i adnotacji (nazywanych także metadanymi). Każdy z nich rozszerza użyteczność języka, gdyż ułatwia wykonywanie typowych zadań programistycznych. W tym rozdziale zostaną dodatkowo omówione typy opakowujące (ang. *wrapper type*) i podstawy refleksji (ang. *reflection*).

Typy wyliczeniowe

Wersje Javy sprzed wydania JDK 5 nie zawierały pewnego elementu uważanego przez wielu programistów za bardzo przydatny: **typów wyliczeniowych**, tzw. **wyliczeń** (ang. *enumeration*). **Typ wyliczeniowy** w najprostszej postaci to lista nazwanych stałych. Choć Java oferuje inne elementy zapewniające podobne działanie, na przykład zmienne `final`, programiści oczekiwali raczej obsługi typów wyliczeniowych w tradycyjnej formie (w postaci, którą doskonale znali z wielu innych języków programowania). Począwszy od wydania JDK 5, typy wyliczeniowe są dostępne również dla programistów języka Java i stały się jego integralnym i często stosowanym elementem.

Wyliczenia w Javie w swojej podstawowej postaci przypominają wyliczenia znane z innych języków. Podobieństwo jest jednak tylko powierzchowne, gdyż w Javie wyliczenie definiuje typ klasy. Przekształcenie typów wyliczeniowych w pełnoprawne klasy pozwoliło znacznie wzbogacić ich możliwości. Na przykład wyliczenia w Javie mogą mieć konstruktory, metody i zmienne składowe. Choć trzeba było długo czekać na ich wprowadzenie, na pewno nikt nie czuje się zawiedziony ich możliwościami.

Podstawy wyliczeń

Wyliczenia tworzy się za pomocą nowego słowa kluczowego `enum`. Oto proste wyliczenie zawierające różne odmiany jabłek.

```
// Wyliczenie odmian jabłek.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Identyfikatory Jonathan, GoldenDel itd. nazywamy **stałymi wyliczeniowymi**. Każda taka stała jest niejawnie zadeklarowana jako publiczna i statyczna składowa wyliczenia Apple. Co więcej, typ stałych jest taki sam jak typ wyliczenia, w którym się znajdują (w tym przypadku Apple). Z tego powodu w języku Java stałe te nazywamy stałymi **własnego typu**, ponieważ są tego samego typu co wyliczenie, do którego należą.

Po zdefiniowaniu wyliczenia można utworzyć zmienną typu wyliczeniowego. Choć wyliczenie definiuje typ klasy, do utworzenia jego egzemplarza nie stosuje się operatora new. Deklaracja zmiennej typu wyliczeniowego przypomina więc deklaracje dowolnego typu prostego. Poniższy wiersz dokonuje deklaracji zmiennej ap typu wyliczeniowego Apple.

```
Apple ap;
```

Ponieważ zmienna ap jest egzemplarzem typu Apple, jedynymi wartościami, które można jej przypisać (a tym samym: które może przechowywać), są stałe wyliczeniowe. Oto sposób przypisania do ap wartości RedDel.

```
ap = Apple.RedDel;
```

Zauważ, że nazwa stałej została poprzedzona nazwą wyliczenia.

Można sprawdzić równość dwóch stałych wyliczeniowych, używając operatora relacji ==. Poniższy kod porównuje wartość zmiennej ap ze stałą GoldenDel.

```
if(ap == Apple.GoldenDel) //...
```

Wartości wyliczenia mogą być używane do sterowania instrukcją switch. Oczywiście wszystkie elementy case muszą używać stałych z tego samego wyliczenia przekazanego w instrukcji switch. Poniższy kod instrukcji switch jest w zupełności poprawny.

```
// Wykorzystanie wyliczenia do sterowania instrukcją switch.
switch(ap) {
  case Jonathan:
    //...
  case Winesap:
    //...
```

Łatwo zauważyć, że stałe w klauzulach case nie są poprzedzane nazwą typu wyliczeniowego — zamiast zapisu Apple.Winesap zastosowano sam identyfikator Winesap. Wynika to z faktu, iż wyboru odpowiedniego typu wyliczeniowego dla klauzul case dokonano już w instrukcji switch. Próba użycia pełnej nazwy stałej wyliczeniowej spowoduje zgłoszenie błędu kompilacji.

Gdy wyświetla się stałą wyliczeniową, na przykład za pomocą instrukcji println(), zostanie przekazana jej nazwa. Wykonanie poniższej instrukcji:

```
System.out.println(Apple.Winesap);
```

spowoduje wyświetlenie na ekranie identyfikatora Winesap.

Następujący program łączy w sobie wszystkie wspomniane cechy wyliczeń i przedstawia zastosowanie wyliczenia Apple.

```
// Wyliczenie odmian jabłek.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
  public static void main(String args[])
  {
    Apple ap;

    ap = Apple.RedDel;

    // Wyświetlenie wartości wyliczenia.
    System.out.println("Wartość ap: " + ap);
    System.out.println();

    ap = Apple.GoldenDel;
```

```
    // Porównanie wartości dwóch wyliczeń.
    if(ap == Apple.GoldenDel)
      System.out.println("ap zawiera GoldenDel.\n");

    // Wykorzystanie wyliczenia do sterowania instrukcją switch.
    switch(ap) {
      case Jonathan:
        System.out.println("Jonathan jest czerwone.");
        break;
      case GoldenDel:
        System.out.println("Golden Delicious jest żółte.");
        break;
      case RedDel:
        System.out.println("Red Delicious jest czerwone.");
        break;
      case Winesap:
        System.out.println("Winesap jest czerwone.");
        break;
      case Cortland:
        System.out.println("Cortland jest czerwone.");
        break;
    }
  }
}
```

Wynik działania programu jest następujący.

```
Wartość ap: RedDel

ap zawiera GoldenDel

Golden Delicious jest żółte.
```

Metody values() i valueOf()

Wszystkie wyliczenia automatycznie zawierają dwie predefiniowane metody: `values()` i `valueOf()`. Ich ogólna postać jest następująca.

```
public static typ-wyliczeniowy[] values()
public static typ-wyliczeniowy valueOf(String tekst)
```

Metoda `values()` zwraca tablicę zawierającą listę stałych wyliczeniowych. Metoda `valueOf()` zwraca stałą wyliczeniową, której odpowiada tekst (obiekt `String`) przekazany jako argument *`tekst`*. Dla wcześniejszego przykładu z wyliczeniem `Apple` wykonanie operacji `Apple.valueOf("Winesap")` spowodowałoby zwrócenie stałej wyliczeniowej `Winesap`.

Poniższy program demonstruje praktyczne zastosowanie metod `values()` i `valueOf()`:

```
// Wykorzystanie wbudowanych metod typów wyliczeniowych.

// Wyliczenie odmian jabłek.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;

    System.out.println("Oto wszystkie stałe wyliczenia Apple:");

    // użycie values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);
```

```
    System.out.println();

    // użycie valueOf()
    ap = Apple.valueOf("Winesap");
    System.out.println("ap zawiera " + ap);

  }
}
```

Wynik działania programu jest następujący.

```
Oto wszystkie stałe wyliczenia Apple:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap zawiera Winesap
```

Zauważ, że program używa pętli `for` typu `for-each` do przejścia przez wszystkie elementy tablicy uzyskanej za pośrednictwem metody `values()`. W celu lepszej ilustracji działania kodu utworzyłem zmienną `allapples` i przypisałem jej referencję do tablicy stałych wyliczeniowych. Ten krok nie jest jednak potrzebny, ponieważ pętlę można napisać w sposób przedstawiony poniżej, eliminując potrzebę stosowania zmiennej `allapples`.

```
for(Apple a : Apple.values())
  System.out.println(a);
```

Zwróć uwagę na to, iż wartość odpowiadająca nazwie `Winesap` została pobrana za pomocą metody `valueOf()`.

```
ap = Apple.valueOf("Winesap");
```

Jak już wspomniano, metoda `valueOf()` zwraca stałą wyliczeniową skojarzoną z przekazaną (w formie łańcucha) nazwą.

Wyliczenia Javy jako typy klasowe

Jak już wspomniano, typy wyliczeniowe w Javie mają postać typów klasowych. Choć nie tworzy się egzemplarzy wyliczenia za pomocą operatora `new`, we wszystkich innych sytuacjach wyliczenie przypomina tradycyjną klasę. Zaimplementowanie wyliczeń w Javie jako pełnoprawnych klas, daje im ogromne możliwości. Na przykład można utworzyć konstruktor, dodać zmienne lub metody składowe, a nawet zaimplementować interfejs.

Trzeba zrozumieć, iż każda stała wyliczeniowa jest obiektem typu wyliczeniowego. Oznacza to, że konstruktor zdefiniowany w ramach typu wyliczeniowego będzie wywoływany przy okazji konstruowania każdej ze stałych wyliczeniowych. Co więcej, każda stała wyliczeniowa posiada własną kopię zmiennych składowych zdefiniowanych w wyliczeniu. Rozważmy następującą wersję wyliczenia `Apple`.

```
// Użycie konstruktora, zmiennej składowej i metody dla wyliczenia.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

  private int price; // cena każdego z jabłek

  // konstruktor
  Apple(int p) { price = p; }

  int getPrice() { return price; }
}

class EnumDemo3 {
  public static void main(String args[])
```

```
  {
    Apple ap;

    // Wyświetl cenę odmiany Winesap.
    System.out.println("Winesap kosztuje " +
                       Apple.Winesap.getPrice() +
                       " złotych.\n");

    // Wyświetl wszystkie jabłka i ich ceny.
    System.out.println("Ceny wszystkich jabłek:");
    for(Apple a : Apple.values())
      System.out.println(a + " kosztuje " + a.getPrice() +
                         " złotych.");
  }
}
```

Wynik działania programu jest następujący.

```
Winesap kosztuje 15 złotych.

Ceny wszystkich jabłek:
Jonathan kosztuje 10 złotych.
GoldenDel kosztuje 9 złotych.
RedDel kosztuje 12 złotych.
Winesap kosztuje 15 złotych.
Cortland kosztuje 8 złotych.
```

Przedstawiona wersja programu zawiera trzy nowe elementy. Pierwszym jest zmienna składowa `price` przechowująca cenę danej odmiany jabłek. Po drugie, istnieje konstruktor `Apple()` przyjmujący jako parametr cenę jabłka. Trzeci element to metoda `getPrice()` zwracająca cenę jabłka.

W momencie deklarowania zmiennej `ap` w metodzie `main()` Java wywołuje konstruktor `Apple()` dla każdej stałej. Zauważ, że argumenty dla konstruktora przekazuje się, umieszczając je w nawiasie po nazwie stałej.

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Wartości te znajdą się w parametrze `p` konstruktora, który przypisze je do zmiennej składowej `price`. Pamiętaj, iż konstruktor zostanie wykonany dla każdej stałej.

Ponieważ każda stała wyliczeniowa posiada własną kopię zmiennej `price`, cenę danej odmiany jabłka pobiera się, wywołując metodę `getPrice()`. Cenę odmiany Winesap w metodzie `main()` pobieramy za pomocą poniższego wywołania.

```
Apple.Winesap.getPrice()
```

Ceny wszystkich odmian uzyskujemy, przetwarzając wyliczenie za pomocą `for`. Ponieważ każda stała wyliczeniowa dysponuje własną kopią zmiennej `price`, wartość skojarzona z jedną stałą jest przypisana tylko do tej stałej. Takie rozwiązanie jest niezwykle użyteczne, ale może być wprowadzone tylko wtedy, gdy wyliczenia zaimplementuje się jako klasy (jak w Javie).

Choć poprzedni przykład zawiera tylko jeden konstruktor, podobnie jak w tradycyjnych klasach można utworzyć kilka przeciążonych wersji konstruktora. Poniższa wersja wyliczenia `Apple` zawiera konstruktor domyślny ustawiający cenę na wartość –1, czyli brak ceny.

```
// Użycie domyślnego konstruktora wyliczenia.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

  private int price; // cena każdego z jabłek

  // konstruktor
  Apple(int p) { price = p; }

  // przeciążony konstruktor
  Apple() { price = -1; }

  int getPrice() { return price; }
}
```

W tej wersji odmiana RedDel nie przekazuje żadnego argumentu. Oznacza to, iż zostanie dla niej wywołany konstruktor domyślny powodujący ustawienie zmiennej price na wartość –1.

Ze stosowaniem typów wyliczeniowych wiążą się dwa ograniczenia. Po pierwsze, typ wyliczeniowy nie może dziedziczyć po innej klasie. Po drugie, wyliczenie nie może być klasą bazową. Innymi słowy, wyliczenia nie uczestniczą w hierarchii klas. Poza tymi ograniczeniami wyliczenia zachowują się podobnie do klas. Trzeba jednak pamiętać o tym, iż każda stała wyliczeniowa jest obiektem typu wyliczeniowego.

Wyliczenia dziedziczą po klasie Enum

Choć nie można korzystać z klasy bazowej przy definiowaniu wyliczenia, wszystkie wyliczenia automatycznie dziedziczą po klasie java.lang.Enum. Klasa ta definiuje kilka metod, z których można korzystać we wszystkich wyliczeniach. Klasa szczegółowo zostanie opisana w części II niniejszej książki, ale trzy jej metody warto opisać już teraz.

Można pobrać wartość liczbową przypisaną do stałej wyliczenia i wskazującą jej miejsce na liście. Wartość tę nazywa się **wartością kolejności** i pobiera się za pomocą metody ordinal().

```
final int ordinal()
```

Metoda zwraca wartość kolejności stałej wyliczeniowej, dla której została wywołana. Wartości kolejności to liczby całkowite zaczynające się od 0. Z tego powodu w wyliczeniu Apple odmiana Jonathan posiada wartość kolejności 0, odmiana GoldenDel wartość 1, RedDel wartość 2 itd.

Do porównania wartości kolejności dwóch stałych wyliczeniowych służy metoda compareTo(). Jej ogólna postać jest następująca.

```
final int compareTo(typ-wyliczeniowy e)
```

Element *typ-wyliczeniowy* to typ wyliczenia; *e* to stała porównywana ze stałą, dla której została wywołana metoda. Pamiętaj, że stała, dla której jest wywoływana metoda, oraz stała przekazywana jako argument muszą być tego samego typu. Jeżeli wywoływana stała ma wartość kolejności mniejszą od *e*, wtedy metoda compareTo() zwraca wartość ujemną. Gdy wartości kolejności obu stałych są takie same, metoda zwraca 0. Jeżeli wywoływana stała ma wartość kolejności większą od *e*, wtedy metoda compareTo() zwraca wartość dodatnią.

Sprawdzanie równości stałej wyliczeniowej z dowolnym innym obiektem można wykonać za pomocą metody equals(), która przysłania domyślną metodę equals() klasy Object. Mimo że porównanie może dotyczyć stałej wyliczeniowej i dowolnego innego obiektu, metoda zwróci wartość true tylko wtedy, gdy przekazany obiekt będzie reprezentował tę samą stałą w ramach tego samego typu wyliczeniowego. Innymi słowy, metoda zwróci wartość false nawet wtedy, gdy przekaże się stałą innego wyliczenia, ale o takiej samej wartości kolejności.

Warto pamiętać, że dwie referencje do egzemplarzy typów wyliczeniowych zawsze można porównać za pomocą zwykłego operatora ==.

Poniższy program ilustruje użycie metod ordinal(), compareTo() i equals().

```
// Przykład użycia metod ordinal(), compareTo() i equals().

// Wyliczenie odmian jabłek.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Pobranie wartości przypisanych do stałych - metoda ordinal().
    System.out.println("Oto wszystkie stałe wyliczenia" +
                       " oraz ich wartości liczbowe: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());
```

```
    ap =  Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    System.out.println();

    // Przykład zastosowania metod compareTo() i equals().
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " znajduje się przed " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " znajduje się przed " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " jest równe " + ap3);

    System.out.println();

    if(ap.equals(ap2))
      System.out.println("Błąd!");

    if(ap.equals(ap3))
      System.out.println(ap + " jest równe " + ap3);

    if(ap == ap3)
      System.out.println(ap + " == " + ap3);

  }
}
```

Program po uruchomieniu wyświetla następujące wyniki.

```
Oto wszystkie stałe wyliczenia oraz ich wartości liczbowe:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel znajduje się przed RedDel
RedDel jest równe RedDel

RedDel jest równe RedDel
RedDel == RedDel
```

Inny przykład wyliczenia

Zanim przejdziemy do omawiania dalszych zagadnień, warto przyjrzeć się jeszcze jednemu przykładowi użycia typu wyliczeniowego. W rozdziale 9. pojawił się program automatycznego podejmowania decyzji. W tamtej wersji zmienne reprezentujące możliwe odpowiedzi `NO`, `YES`, `MAYBE`, `LATER`, `SOON` i `NEVER` zostały zadeklarowane w interfejsie. Choć tamto podejście jest całkowicie poprawne, zastosowanie wyliczenia wydaje się lepszym wyborem. Oto ulepszona wersja tamtego programu, używająca wyliczenia o nazwie `Answers` do udzielania odpowiedzi.

```
// Ulepszona wersja programu podejmowania decyzji
// z rozdziału 9. Ta wersja używa wyliczenia
// zamiast zmiennych interfejsu do reprezentacji
// możliwych odpowiedzi.

import java.util.Random;

// Wyliczenie ze wszystkimi możliwymi odpowiedziami.
enum Answers {
  NO, YES, MAYBE, LATER, SOON, NEVER
```

```
}

class Question {
  Random rand = new Random();
  Answers ask() {
    int prob = (int) (100 * rand.nextDouble());

    if (prob < 15)
      return Answers.MAYBE; // 15%
    else if (prob < 30)
      return Answers.NO;    // 15%
    else if (prob < 60)
      return Answers.YES;   // 30%
    else if (prob < 75)
      return Answers.LATER; // 15%
    else if (prob < 98)
      return Answers.SOON;  // 13%
    else
      return Answers.NEVER; // 2%
  }
}

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

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

Opakowania typów

Java używa typów prostych (nazywanych również typami podstawowymi), takich jak `int` lub `double`, do przechowywania podstawowych typów danych. Takie rozwiązanie podyktowane jest chęcią zwiększenia wydajności. Zastosowanie obiektów dla tych typów spowodowałoby ogromny narzut nawet w bardzo prostych obliczeniach. Oznacza to jednak, iż typy proste nie stanowią części hierarchii klasy i nie dziedziczą po klasie `Object`.

Mimo wszystkich zalet typów prostych (w wymiarze wydajności) w pewnych sytuacjach lepszym rozwiązaniem jest operowanie na obiektowej reprezentacji tych typów. Na przykład nie można przekazać typu prostego jako referencji, której wymaga metoda. Wiele standardowych struktur danych Javy opiera się na obiektach, co oznacza, iż do tych struktur nie można przekazać typów prostych. W celu zniwelowania tego rodzaju problemów Java zawiera **opakowania typów** — klasy hermetyzujące pojedynczą wartość typu prostego. Wszystkie te klasy zostaną szczegółowo omówione w części II książki; ponieważ jednak są bezpośrednio powiązane z funkcją automatycznego opakowywania i rozpakowywania typów prostych, warto wprowadzić je już teraz.

W Javie istnieją następujące opakowania typów prostych: `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` i `Boolean`. Klasy te oferują wiele metod ułatwiających pełną integrację typów prostych z hierarchią klas Javy. Oto krótkie omówienie poszczególnych typów.

Klasa Character

Klasa `Character` to opakowanie dla typu `char`. Konstruktor klasy ma następującą postać.

```
Character(char zn)
```

Element *zn* to znak, który zostanie opakowany przez obiekt klasy `Character`.

Metoda `charValue()` służy do pobrania wartości `char` przechowywanej w obiekcie.

Aby uzyskać wartość typu `char` zawartą w obiekcie klasy `Character`, należy wywołać metodę `charValue()`:

```
char charValue()
```

Metoda zwraca znak reprezentowany przez obiekt opakowania.

Klasa Boolean

Klasa `Boolean` to opakowanie dla typu `boolean`. Konstruktory klasy mają następującą postać.

```
Boolean(boolean wartośćLogiczna)
Boolean(String wartośćŁańcuchowa)
```

W pierwszej wersji parametr *wartośćLogiczna* musi mieć albo wartość `true`, albo wartość `false`. W drugiej wersji, jeśli przekazany parametr *wartośćŁańcuchowa* zawiera łańcuch `"true"` (pisany albo małymi, albo wielkimi literami), nowy obiekt klasy `Boolean` będzie reprezentował wartość `true`. W przeciwnym razie będzie reprezentował wartość `false`.

Metoda `booleanValue()` służy do pobrania wartości `boolean` przechowywanej w obiekcie.

```
boolean booleanValue()
```

Metoda zwraca wartość `true` lub `false` reprezentowaną przez obiekt.

Opakowania typów numerycznych

Typy opakowujące najczęściej stosuje się dla wartości numerycznych. Oto typy związane z liczbami: `Byte`, `Short`, `Integer`, `Long`, `Float` i `Double`. Wszystkie te typy dziedziczą po klasie abstrakcyjnej `Number`. Klasa ta deklaruje metody zwracające wartość obiektu w kilku różnych formatach liczbowych. Metody te zostały wymienione poniżej.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Metoda `doubleValue()` zwraca wartość obiektu jako typ `double`, metoda `floatValue()` jako typ `float` itd. Wymienione metody są implementowane przez wszystkie liczbowe typy opakowujące.

Wszystkie opakowania dla liczb definiują dwa konstruktory, które umożliwiają utworzenie obiektu na podstawie podanej wartości lub jej reprezentacji łańcuchowej. Na przykład klasa `Integer` ma następujące konstruktory.

```
Integer(int wartość)
Integer(String łańcuch)
```

Jeżeli *łańcuch* nie zawiera poprawnej reprezentacji tekstowej liczby, konstruktor zgłasza wyjątek `NumberFormatException`.

Wszystkie typy opakowujące przykrywają metodę `toString()`, która zwraca czytelną, tekstową (łańcuchową) reprezentację wartości przechowywanej w danym obiekcie. Umożliwia to między innymi bezpośrednie przekazywanie obiektu jako argumentu metody `println()`. Nie trzeba dokonywać wcześniejszej konwersji do typu prostego.

Poniższy przykładowy program ilustruje wykorzystanie typu opakowującego liczby całkowitej do hermetyzacji wartości, a następnie jej wydobycia.

```
// Przykład użycia opakowania.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = new Integer(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // wyświetla 100 100
  }
}
```

Kod opakowuje wartość całkowitą 100 obiektem klasy `Integer` nazwanym `iOb`. Następnie pobiera tę wartość z obiektu i przypisuje ją do zmiennej `i`.

Proces umieszczania wartości wewnątrz nazywa się **opakowywaniem**. Następujący wiersz opakowuje więc wartość 100 w obiekcie typu `Integer`.

```
Integer iOb = new Integer(100);
```

Proces wydobywania wartości z typu opakowującego nazywa się **rozpakowywaniem** lub **wydobywaniem**. Następujący wiersz wydobywa wartość z obiektu `iOb`.

```
int i = iOb.intValue();
```

Powyższy sposób opakowywania i wydobywania wartości był stosowany od samego początku w języku Java. Wraz z wydaniem JDK 5 wprowadzono jednak rozwiązanie, które znacznie uprościło stosowanie opakowań — automatyczne opakowywanie i wydobywanie typów prostych.

Automatyczne opakowywanie typów prostych

Począwszy od wydania JDK 5, język Java oferuje dwa ważne mechanizmy: **automatyczne opakowywanie** i **automatyczne wydobywanie typów prostych**. Automatyczne opakowywanie oznacza, iż typ prosty jest automatycznie umieszczany w odpowiadającym mu typie opakowującym, jeśli sytuacja wymaga użycia obiektu. Nie trzeba jawnie tworzyć obiektu. Automatyczne wydobywanie oznacza, iż typ opakowujący jest automatycznie rozpakowywany w razie konieczności użycia typu prostego. Nie trzeba jawnie wywoływać metody `intValue()` lub `doubleValue()`.

Nowa funkcja języka Java ułatwia pisanie niektórych algorytmów, ponieważ eliminuje konieczność żmudnego, ręcznego opakowywania i wydobywania wartości. Co więcej, takie rozwiązanie zapobiega popełnianiu błędów i stanowi jeden z ważnych elementów typów sparametryzowanych, które działają jedynie dla obiektów. Automatyczne opakowywanie ułatwia też korzystanie z kolekcji frameworku Collections, które zostaną omówione w części II książki.

Automatyczne opakowywanie eliminuje konieczność ręcznego tworzenia obiektów opakowujących wartości typów prostych — wystarczy przypisać typ prosty zmiennej referencyjnej typu opakowującego. Java automatycznie utworzy odpowiedni obiekt. Oto nowoczesny sposób tworzenia obiektu `Integer` przechowującego wartość 100.

```
Integer iOb = 100; // automatyczne opakowanie typu int
```

Zauważ, że nie pojawia się jawne tworzenie obiektu za pomocą operatora `new`. Java wykonuje to zadanie za programistę.

W celu rozpakowania obiektu wystarczy przypisać zmiennej odpowiedniego typu prostego referencję do obiektu. Oto przykład rozpakowania obiektu `iOb`.

```
int i = iOb; // automatyczne rozpakowanie
```

Java sama zajmuje się wszystkimi szczegółami.

Poniżej znajduje się wcześniejszy program napisany z wykorzystaniem automatycznego opakowywania i rozpakowywania.

```
// Przykład automatycznego opakowywania.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // automatyczne opakowanie typu int

    int i = iOb; // automatyczne rozpakowanie

    System.out.println(i + " " + iOb);  // wyświetla 100 100
  }
}
```

Automatyczne opakowywanie i metody

Działanie tego mechanizmu nie ogranicza się tylko do operacji przypisania. Automatyczne opakowywanie występuje zawsze wtedy, gdy typ prosty wymaga konwersji na obiekt. Automatyczne rozpakowanie występuje zawsze wtedy, gdy obiekt musi zostać zamieniony na typ prosty. Oznacza to, iż automatyczne opakowanie może mieć miejsce na przykład w momencie przekazania argumentu do metody lub zwrócenia wartości przez metodę. Oto przykład wart przeanalizowania.

```
// Automatyczne opakowanie i rozpakowanie dotyczy
// także argumentów metod i zwracanych wartości.

class AutoBox2 {
  // Pobierz parametr Integer
  // i zwróć wartość typu int.
  static int m(Integer v) {
    return v ; // automatyczne rozpakowanie do int
  }

  public static void main(String args[]) {
    // Przekazanie wartości typu int do m() i przypisanie zwróconej wartości
    // obiektowi typu Integer. Argument 100 jest automatycznie opakowany typem
    // Integer. Zwracana wartość jest ponownie automatycznie opakowywana
    // typem Integer.
    Integer iOb = m(100);

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

Program wyświetli na ekranie następującą wartość.

```
100
```

Zauważ, że metoda `m()` wymaga argumentu typu `Integer` i zwraca jako wynik typ `int`. Wewnątrz metody `main()` metodzie `m()` jest przekazywana wartość 100. Ponieważ metoda `m()` wymaga typu `Integer`, przekazana wartość jest automatycznie opakowywana. Następnie metoda zwraca typ `int` odpowiadający wartości z obiektu `Integer`. Powoduje to automatyczne rozpakowanie `v`. Ponieważ uzyskana wartość jest przypisywana do zmiennej `iOb`, następuje ponowne automatyczne opakowanie wartości typu `int`.

Automatyczne opakowywanie i rozpakowywanie w wyrażeniach

Ogólnie automatyczne opakowywanie lub rozpakowywanie ma miejsce zawsze wtedy, gdy wymagana jest konwersja na obiekt lub z obiektu. Dotyczy to również wyrażeń. W wyrażeniu obiekt numeryczny jest automatycznie rozpakowywany. W razie konieczności wynik wyrażenia jest opakowywany obiektem. Oto przykład takiego zachowania.

```
// Automatyczne opakowywanie w wyrażeniach.

class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Oryginalna wartość iOb: " + iOb);

    // Poniższy kod powoduje automatyczne rozpakowanie iOb,
    // inkrementowanie wartości i ponownie opakowanie
    // wyniku w obiekcie iOb.
    ++iOb;
    System.out.println("Po ++iOb: " + iOb);

    // Tutaj iOb jest rozpakowywane, obliczana jest wartość
    // wyrażenia i wynik jest ponownie pakowany do iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 po wyrażeniu: " + iOb2);

    // Obliczenie tego samego wyrażenia,
    // ale bez automatycznego opakowywania.
    i = iOb + (iOb / 3);
    System.out.println("i po wyrażeniu: " + i);

  }
}
```

Program generuje następujące wyniki.

```
Oryginalna wartość iOb: 100
Po ++iOb: 101
iOb2 po wyrażeniu: 134
i po wyrażeniu: 134
```

Warto zwrócić szczególną uwagę na poniższy wiersz.

```
++iOb;
```

Wyrażenie w tej formie oznacza inkrementację wartości przechowywanej w obiekcie `iOb`: najpierw dochodzi do rozpakowania wartości, następnie jest ona inkrementowana, a na końcu ponownie opakowywana obiektem.

Automatyczne rozpakowywanie umożliwia mieszanie w jednym wyrażeniu różnych typów obiektów numerycznych. Po rozpakowaniu zachodzi standardowa promocja i konwersja typów. Poniższy program jest całkowicie poprawny.

```
class AutoBox4 {
  public static void main(String args[]) {

    Integer iOb = 100;
    Double dOb = 98.6;


    dOb = dOb + iOb;
    System.out.println("dOb po wyrażeniu: " + dOb);
  }
}
```

Wynik działania programu jest następujący.

```
dOb po wyrażeniu: 198.6
```

Zarówno obiekt `dOb` typu `Double`, jak i obiekt `iOb` typu `Integer` uczestniczył w dodawaniu. Wynik został opakowany i umieszczony w obiekcie `dOb`.

Dzięki automatycznemu rozpakowywaniu można używać obiektów liczb całkowitych do sterowania instrukcją `switch`. Oto przykład.

```
Integer iOb = 2;

switch(iOb) {
  case 1: System.out.println("jeden");
    break;
  case 2: System.out.println("dwa");
    break;
  default: System.out.println("błąd");
}
```

W momencie obliczania wyrażenia z instrukcji `switch` obiekt `iOb` jest automatycznie rozpakowywany.

Jak łatwo zauważyć, automatyczne opakowywanie i rozpakowywanie ułatwia pisanie wyrażeń korzystających z obiektów numerycznych. Dawniej, aby osiągnąć ten sam efekt, trzeba było dokonywać rzutowania i wywoływania metod takich jak `intValue()`.

Automatyczne opakowywanie typów znakowych i logicznych

Jak już wspomniano, Java udostępnia także klasy opakowań dla typów `boolean` i `char`: odpowiednio `Boolean` i `Character`. Automatyczne opakowywanie i rozpakowywanie dotyczy również tych typów. Oto przykład.

```
// Automatyczne opakowywanie i rozpakowywanie obiektów klas Boolean i Character.

class AutoBox5 {
  public static void main(String args[]) {

    // automatyczne opakowanie/rozpakowanie wartości typu boolean
    Boolean b = true;

    // Zastosowanie b w wyrażeniu warunkowym powoduje
    // jego automatyczne rozpakowanie.
    if(b) System.out.println("b wynosi true");

    // automatyczne opakowanie/rozpakowanie wartości typu char
    Character ch = 'x'; // opakowanie typu char
    char ch2 = ch; // rozpakowanie char

    System.out.println("ch2 wynosi " + ch2);
  }
}
```

Program generuje następujące wyniki.

```
b wynosi true
ch2 wynosi x
```

W tym programie warto przede wszystkim zauważyć, iż zachodzi automatyczne rozpakowanie `b` w wyrażeniu warunkowym instrukcji `if`. Każde wyrażenie instrukcji `if` musi dać w wyniku wartość typu `boolean`. Ponieważ `b` zawiera wartość logiczną, dochodzi do automatycznego rozpakowania i użycia tej wartości jako wyniku wyrażenia. Z tego względu wspomniana instrukcja jest dla kompilatora w pełni poprawna.

Dzięki automatycznemu rozpakowywaniu można stosować obiekt `Boolean` w strukturach sterujących. Wewnątrz pętli `while`, `for` i `do-while` dochodzi do automatycznego rozpakowania obiektu do typu `boolean`. Poniższy kod jest w pełni poprawny.

```
Boolean b;
// ...
while(b) { // ...
```

Automatyczne opakowywanie pomaga zapobiegać błędom

Automatyczne opakowywanie, poza samą wygodą, pomaga zapobiegać błędom programistów. Rozważmy następujący program.

```
// Błąd przy ręcznym rozpakowywaniu.
class UnboxingError {
  public static void main(String args[]) {

    Integer iOb = 1000; // automatyczne opakowanie wartości 1000

    int i = iOb.byteValue(); // ręczne wydobycie wartości!!!

    System.out.println(i);  // nie spowoduje wyświetlenia wartości 1000!
  }
}
```

Program nie wyświetli oczekiwanej wartości 1000, ale –24! Wynika to z faktu, iż rozpakowanie zostało wykonane metodą `byteValue()`, która spowodowała obcięcie wartości do zakresu dostępnego dla typu `byte`. Automatyczne rozpakowywanie zabezpiecza przed tego rodzaju błędami, ponieważ Java automatycznie zastosuje metodę zgodną z typem `int`.

Ponieważ automatyczne opakowywanie zawsze tworzy odpowiedni obiekt, a automatyczne wydobywanie zawsze pobiera odpowiednią wartość, nie można doprowadzić do błędnego doboru typu i wartości. W tych rzadkich przypadkach, w których programista świadomie chce dokonać zmiany typu, zawsze może zastosować ręczne opakowywanie lub rozpakowywanie. Oczywiście traci się wtedy zalety automatyzacji. Ogólnie: nowo tworzony kod języka Java powinien stosować automatyczne opakowywanie.

Słowo ostrzeżenia

Ponieważ język Java dokonuje automatycznego opakowywania i rozpakowywania, niektórzy programiści mogą ulec pokusie całkowitej rezygnacji z tradycyjnych typów prostych na rzecz obiektów klasy `Integer` czy `Double`. Można na przykład napisać następujący kod.

```
// Złe użycie automatycznego opakowywania!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Przeciwprostokątna ma wartość " + c);
```

Obiekty typu `Double` przechowują wartości używane do obliczenia przeciwprostokątnej trójkąta prostokątnego. Choć kod jest poprawny od strony technicznej i wykonuje swoje zadanie bez problemów, stanowi bardzo zły przykład użycia automatycznego opakowywania. Dużo wydajniejsze obliczeniowo będzie w tym przypadku zastosowanie typu prostego `double`. Różnica w wydajności obu rozwiązań wynika z tego, że automatyczne opakowywanie wiąże się z kosztami, które nie występują w przypadku typów prostych.

Ogólnie: należy stosować typy opakowujące tylko wtedy, gdy potrzebna jest obiektowa reprezentacja typów prostych. Automatyczne opakowywanie nie zostało dodane do Javy, by po cichu wyeliminować typy proste.

Adnotacje (metadane)

Począwszy od wydania JDK 5, język Javy obsługuje nowy sposób umieszczania dodatkowych informacji w plikach z kodem źródłowym. Nowy mechanizm, który nazwano **adnotacjami**, w żaden sposób nie wpływa na działanie programu. Adnotacje pozostawiają niezmienioną semantykę programu. Okazuje się jednak, że informacje zawarte w adnotacjach mogą być używane przez rozmaite narzędzia na etapie wytwarzania i wdrażania oprogramowania. Adnotacja może zostać przetworzona na przykład przez generator kodu źródłowego. Opisywany mechanizm bywa też określany mianem **metadanych**, jednak nazwa **adnotacje** lepiej oddaje istotę tego rozwiązania i cieszy się większą popularnością.

Podstawy tworzenia adnotacji

Adnotacje można tworzyć przy użyciu mechanizmu na bazie interfejsu. Zacznijmy od przykładu, który deklaruje adnotację `MyAnno`.

```
// Przykład prostego typu adnotacji.
@interface MyAnno {
  String str();
  int val();
}
```

Zauważ znak `@` poprzedzający słowo kluczowe `interface`. Za pomocą tego znaku można zasygnalizować kompilatorowi, że ma do czynienia z deklaracją typu adnotacji. Wewnątrz adnotacji zadeklarowano dwie metody składowe: `str()` i `val()`. Wszystkie adnotacje składają się tylko z deklaracji metod. Nie podaje się kodu deklarowanych metod, Java sama je implementuje. Co więcej, metody te zachowują się jak zmienne.

Adnotacja nie może zawierać klauzuli `extends`. Okazuje się jednak, że wszystkie adnotacje automatycznie rozszerzają interfejs `Annotation`. Deklaracja tego głównego interfejsu znajduje się w pakiecie `java.lang.annotation`. Przesłania on metody `hashCode()`, `equals()` i `toString()` zdefiniowane w klasie `Object`. Dodatkowo zawiera metodę `annotationType()`, która zwraca obiekt `Class` reprezentujący wywołaną adnotację.

Po zadeklarowaniu adnotacji można zacząć jej używać do oznaczania innych elementów kodu. Przed wprowadzeniem JDK 8 adnotacje można było dodawać wyłącznie do deklaracji i właśnie ten sposób ich stosowania przedstawimy w pierwszej kolejności. (W JDK 8 wprowadzono także możliwość dodawania adnotacji do typów — przedstawię ją w dalszej części rozdziału. Niemniej jednak pewne podstawowe techniki odnoszą się do obu rodzajów adnotacji). Adnotacje można stosować praktycznie dla dowolnych rodzajów deklaracji, w tym deklaracji klas, metod, zmiennych, parametrów i stałych wyliczeniowych. Istnieje nawet możliwość oznaczenia jedną adnotacją deklaracji innej adnotacji. We wszystkich przypadkach treść adnotacji musi znaleźć się przed właściwą deklaracją.

Podczas stosowania adnotacji należy przypisać wartości jej składowym. Oto przykład zastosowania adnotacji dla metody:

```
// Adnotacja dotycząca metody.
@MyAnno(str = "Przykład adnotacji", val = 100)
public static void myMeth() { //...
```

Adnotacja jest powiązana z metodą `myMeth()`. Warto dobrze przyjrzeć się stosowanej składni. Nazwa rodzaju adnotacji jest poprzedzona znakiem `@`, a w nawiasach znajduje się lista inicjalizująca składowe. Aby nadać składowej wartość, wystarczy podać nazwę składowej, znak równości i wartość. W tym przykładzie łańcuch `"Przykład adnotacji"` zostaje przypisany do składowej `str`. Łatwo zauważyć, że w przytoczonym wyrażeniu po identyfikatorze `str` nie użyto nawiasów. Podczas przypisywania wartości składowej adnotacji wystarczy użyć nazwy tej składowej. W tym kontekście metoda wygląda, jakby była zmienną składową.

Określanie strategii zachowywania adnotacji

Zanim przystąpimy do szczegółowego omawiania adnotacji, warto wspomnieć o **strategii zachowywania adnotacji**. Strategia określa, w którym momencie adnotacje są pomijane. Java definiuje trzy takie strategie — znajdują się one w wyliczeniu `java.lang.annotation.RetentionPolicy`. Nazwy tych strategii to `SOURCE`, `CLASS` i `RUNTIME`.

Adnotacja ze strategią `SOURCE` jest zachowywana tylko w pliku kodu źródłowego. Innymi słowy, jest pomijana w trakcie kompilacji kodu.

Adnotacja ze strategią `CLASS` jest zachowywana w pliku *.class* w trakcie kompilacji. Nie jest jednak dostępna dla maszyny wirtualnej w trakcie wykonywania programu.

Adnotacja ze strategią `RUNTIME` jest zachowywana w pliku *.class* w trakcie kompilacji i jest dostępna dla maszyny wirtualnej w trakcie wykonywania programu. Oznacza to, że właśnie strategia `RUNTIME` zapewnia największą trwałość adnotacji.

Adnotacja zastosowana dla deklaracji zmiennej lokalnej nigdy nie jest zachowywana w pliku *.class*.

Strategię zachowywania adnotacji dla notatki można wskazać, stosując jedną z wbudowanych adnotacji `@Retention`. Oto ogólna postać tej adnotacji.

```
@Retention(strategia-zachowywania)
```

Element *`strategia-zachowywania`* musi być jedną z wcześniej omówionych stałych wyliczeniowych. Jeżeli nie określi się jawnie żadnej strategii, domyślnie zostanie użyta strategia `CLASS`.

Poniższa wersja adnotacji `MyAnno` używa adnotacji `@Retention` do określenia strategii zachowywania `RUNTIME`. Oznacza to, że adnotacja `MyAnno` będzie dostępna dla maszyny wirtualnej w trakcie wykonywania programu.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

Odczytywanie adnotacji w trakcie działania programu za pomocą refleksji

Mimo że adnotacje zaprojektowano przede wszystkim z myślą o narzędziach programistycznych i wdrożeniowych, adnotacje stosujące strategię `RUNTIME` można odczytywać w czasie wykonywania programu, korzystając z tzw. **refleksji**. Refleksja umożliwia pobranie informacji na temat klasy w trakcie działania programu. Interfejs API refleksji umieszczono w pakiecie `java.lang.reflect`. Istnieje wiele zastosowań tego mechanizmu, więc nie będę ich tu wszystkich wymieniał. Ograniczymy się tylko do analizy kilku przykładów związanych z adnotacjami.

Pierwszy krok wykorzystania refleksji polega na pobraniu obiektu klasy `Class` reprezentującego klasę, której adnotację chcemy odczytać. Klasa `Class` to jedna z klas wbudowanych w język Java i znajdujących się w pakiecie `java.lang`. Jest ona szczegółowo omówiona w części II książki. Istnieje wiele sposobów uzyskania obiektu `Class`. Jednym z nich jest zastosowanie metody `getClass()` zdefiniowanej w klasie `Object`. Oto ogólna postać metody.

```
final Class<?> getClass()
```

Metoda zwraca obiekt klasy `Class` reprezentujący klasę obiektu wywołującego tę metodę.

Warto zwrócić uwagę na konstrukcję `<?>` użytą po słowie `Class` w powyższej deklaracji metody `getClass()`. Wspomniana konstrukcja ma związek z dostępnym w Javie mechanizmem typów sparametryzowanych. Metoda `getClass()`, jak wiele innych prezentowanych tutaj metod związanych z refleksjami, korzysta z typów sparametryzowanych. Same typy sparametryzowane zostaną omówione w rozdziale 14. Dobre rozumienie typów sparametryzowanych nie jest jednak niezbędne do zrozumienia podstawowych zasad funkcjonowania refleksji.

Po uzyskaniu obiektu klasy `Class` można użyć jego metod do odczytania informacji na temat rozmaitych elementów zadeklarowanych w tej klasie, w tym adnotacji. Aby uzyskać adnotację związaną z konkretnym elementem zadeklarowanym w klasie, należy najpierw pobrać obiekt reprezentujący ten element. Klasa `Class` zawiera między innymi metody `getMethod()`, `getField()` i `getConstructor()`, które pobierają informacje odpowiednio o metodzie, polu i konstruktorze. Metody zwracają obiekty typu `Method`, `Field` i `Constructor`.

Aby lepiej prześledzić cały proces, najlepiej przeanalizować przykład uzyskujący adnotacje związane z metodą. W tym celu trzeba pobrać obiekt `Class` reprezentujący klasę, a następnie wywołać dla niego metodę `getMethod()`, przekazując nazwę metody jako parametr. Ogólna postać metody `getMethod()` jest następująca.

```
Method getMethod(String nazwaMetody, Class<?> ... typyParametrów)
```

Nazwę metody przekazuje się w elemencie *nazwaMetody*. Jeśli metoda przyjmuje argumenty, trzeba przekazać obiekty `Class` reprezentujące typy parametrów (element *typyParametrów*). Zauważ, że *typyParametrów* to parametr o zmiennej liczbie argumentów. Oznacza to, iż można przekazać dowolną liczbę typów parametrów, również zero. Metoda `getMethod()` zwraca obiekt `Method` reprezentujący metodę. Jeśli nie uda się odnaleźć metody, zostanie zgłoszony wyjątek `NoSuchMethodException`.

Adnotacje skojarzone z obiektami klas `Class`, `Method`, `Field` i `Constructor` można uzyskać za pośrednictwem metody `getAnnotation()`. Ogólna postać tej metody jest następująca.

```
Annotation getAnnotation(Class<A> typAdnotacji)
```

Element *typAdnotacji* to obiekt `Class` reprezentujący interesującą nas adnotację. Metoda zwraca referencję do adnotacji. Za pośrednictwem tej referencji można uzyskać wartości reprezentowane przez składowe tej adnotacji. Jeśli nie uda się znaleźć żądanej adnotacji (jeśli na przykład zastosowano dla adnotacji inną strategię zachowywania niż `RUNTIME`), metoda zwróci wartość `null`.

Poniżej pokazano program łączący w sobie wszystkie opisane powyżej elementy i używający refleksji do wyświetlenia adnotacji związanej z metodą.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Deklaracja typu adnotacji.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Adnotacja metody.
  @MyAnno(str = "Przykładowa adnotacja", val = 100)
  public static void myMeth() {
    Meta ob = new Meta();

    // Pobranie adnotacji dla tej metody i
    // wyświetlenie wartości jej składowych.
    try {
      // Najpierw pobierz obiekt Class reprezentujący
      // aktualną klasę.
      Class c = ob.getClass();

      // Pobierz obiekt Method reprezentujący
      // aktualną metodę.
      Method m = c.getMethod("myMeth");

      // Następnie pobierz adnotację dla tej metody.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Wyświetl wartości adnotacji.
      System.out.println(anno.str() + " " + anno.val());
```

```
    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

Wynik działania programu jest następujący.

```
Przykładowa adnotacja 100
```

Program używa refleksji do pobrania i wyświetlenia wartości elementów `str` i `val` adnotacji `MyAnno` przypisanej do metody `myMeth()` klasy `Meta`. Warto zwrócić szczególną uwagę na dwa wiersze. Oto pierwszy z nich.

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

Szczególnie interesujące jest wyrażenie `MyAnno.class`, które powoduje zwrócenie obiektu klasy `Class` dla typu `MyAnno`, czyli adnotacji. Taką konstrukcję nazywamy **stałą klasową**. Podobne konstrukcje można stosować wszędzie tam, gdzie jest potrzebny obiekt typu `Class` dla znanej klasy. Na przykład poniższego wyrażenia można by użyć do uzyskania obiektu typu `Class` dla klasy `Meta`:

```
Class<?> c = Meta.class;
```

Oczywiście takie rozwiązanie działa tylko wtedy, gdy z wyprzedzeniem zna się nazwę klasy. Nie zawsze występuje taka sytuacja. Ogólnie stałe klasowe można uzyskiwać dla klas, interfejsów, typów prostych i tablic. (Warto pamiętać, że zapis `<?>` ma związek z typami sparametryzowanymi, które zostaną omówione w rozdziale 14.).

Drugi wiersz, którym warto się zainteresować, dotyczy sposobu pobierania wartości elementów `str` i `val` w momencie ich wyświetlania.

```
System.out.println(anno.str() + " " + anno.val());
```

Zauważ, że jest to zwyczajne wywołanie metody o takiej samej nazwie jak element. Podobne konstrukcje można stosować wszędzie tam, gdzie jest potrzebna wartość składowej adnotacji.

Drugi przykład refleksji

W poprzednim programie metoda `myMeth()` nie miała żadnych parametrów. W wywołaniu `getMethod()` wystarczyło więc przekazać jedynie nazwę metody. Aby pobrać metodę z parametrami, trzeba dodatkowo przekazać do metody `getMethod()` obiekty klas reprezentujące typy parametrów. Oto odrobinę zmieniona wersja wcześniejszego programu.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Metoda myMeth ma teraz dwa parametry.
  @MyAnno(str = "Dwa parametry", val = 19)
  public static void myMeth(String str, int i)
  {
    Meta ob = new Meta();

    try {
      Class c = ob.getClass();
```

```
      // Tutaj pojawia się określenie typów parametrów.
      Method m = c.getMethod("myMeth", String.class, int.class);

      MyAnno anno = m.getAnnotation(MyAnno.class);

      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth("test", 10);
  }
}
```

Wynik działania tej wersji programu jest następujący.

```
Dwa parametry 19
```

Metoda `myMeth()` przyjmuje dwa parametry — jeden typu `String` i drugi typu `int`. W celu pobrania informacji na temat metody konieczne jest wywołanie poniższej wersji metody `getMethod()`.

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Jako dodatkowe argumenty trzeba przekazać obiekty `Class` reprezentujące klasę `String` i typ `int`.

Odczytywanie wszystkich adnotacji

Aby odczytać wszystkie adnotacje skojarzone z danym elementem i stosujące strategie zachowywania `RUNTIME`, należy użyć metody `getAnnotations()` dla tego elementu. Oto ogólna postać metody.

```
Annotation[] getAnnotations()
```

Metoda zwraca tablicę adnotacji. Metodę można wywołać dla obiektów typu `Class`, `Method`, `Constructor` i `Field` oraz kilku innych.

Poniżej pokazano kolejny przykład ilustrujący sposób wykorzystania refleksji do pobrania wszystkich adnotacji związanych z klasą i metodą. Kod deklaruje dwa typy adnotacji. Są one używane do dołączenia dodatkowych informacji do klasy i metody.

```
// Wyświetlanie wszystkich adnotacji klasy i metody.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
  String description();
}

@What(description = "Klasa testowa adnotacji")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

  @What(description = "Metoda testowa adnotacji")
  @MyAnno(str = "Testowanie", val = 100)
  public static void myMeth() {
    Meta2 ob = new Meta2();

    try {
      Annotation annos[] = ob.getClass().getAnnotations();
```

```
      // Wyświetlenie wszystkich adnotacji dla klasy Meta2.
      System.out.println("Wszystkie adnotacje Meta2:");
      for(Annotation a : annos)
        System.out.println(a);

      System.out.println();

      // Wyświetlenie wszystkich adnotacji dla metody myMeth.
      Method m = ob.getClass( ).getMethod("myMeth");
      annos = m.getAnnotations();

      System.out.println("Wszystkie adnotacje myMeth:");
      for(Annotation a : annos)
        System.out.println(a);

    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

Wynik działania programu jest następujący.

```
Wszystkie adnotacje Meta2:
@What(description=Klasa testowa adnotacji)
@MyAnno(str=Meta2, val=99)

Wszystkie adnotacje myMeth:
@What(description=Metoda testowa adnotacji)
@MyAnno(str=Testowanie, val=100)
```

Program używa metody `getAnnotations()` do pobrania tablicy notatek związanych z klasą `Meta2` i metodą `myMeth()`. Metoda ta zwraca tablicę obiektów `Annotation`. Przypomnę, iż klasa `Annotation` stanowi interfejs bazowy dla wszystkich notatek i przesłania metodę `toString()` klasy `Object`. Z tego powodu próba wyświetlenia notatki powoduje wywołanie metody `toString()`, która zwraca pełny opis notatki.

Interfejs AnnotatedElement

Metody `getAnnotation()` i `getAnnotations()` użyte we wcześniejszych przykładach są zdefiniowane w interfejsie `AnnotatedElement` z pakietu `java.lang.reflect`. Interfejs ten wykorzystuje mechanizm refleksji do pobierania adnotacji i jest zaimplementowany przez klasy `Method`, `Field`, `Constructor`, `Class`, `Package` i kilku innych.

Poza wspomnianymi wcześniej metodami `getAnnotation()` oraz `getAnnotations()` interfejs definiuje jeszcze kilka innych metod. Dwie z nich były dostępne już w JDK 5. Pierwszą z nich jest `getDeclaredAnnotations()` o następującej ogólnej postaci.

```
Annotation[] getDeclaredAnnotations()
```

Metoda zwraca wszystkie niedziedziczone adnotacje występujące w wywołującym obiekcie. Druga metoda to `isAnnotationPresent()` o następującej postaci.

```
boolean isAnnotationPresent(Class<? extends Annotation> typAdnotacji)
```

Metoda zwraca wartość `true`, jeśli z wywoływanym obiektem jest skojarzona adnotacja typu *typAdnotacji*. W przeciwnym przypadku zwraca wartość `false`. W JDK 8 zostały do nich dodane trzy kolejne metody: `getDeclaredAnnotation()`, `getAnnotationsByType()` oraz `getDeclaredAnnotationsByType()`. Dwie ostatnie metody automatycznie operują na powtarzających się adnotacjach. (Powtarzające się adnotacje zostały opisane pod koniec tego rozdziału).

Wartości domyślne

Składowym adnotacji można nadawać wartości domyślne, które będą stosowane, jeśli odpowiednie wartości nie zostaną określone podczas tworzenia egzemplarzy adnotacji. Wartości domyślne dodaje się poprzez dodanie klauzuli `default` do deklaracji składowej. Oto ogólna postać deklaracji.

```
typ składowa() default wartość;
```

Wskazywana *wartość* musi być zgodna z typem *typ*.

Oto przykład adnotacji `@MyAnno` z dodanymi wartościami domyślnymi.

```
// Deklaracja typu adnotacji zawierającego wartości domyślne.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testowanie";
  int val() default 9000;
}
```

Składowym `str` i `val` będą przypisywane odpowiednio domyślne wartości `"Testowanie"` i 9000. Oznacza to, że w momencie stosowania tej adnotacji dla klasy lub metody nie jest wymagane określenie jakichkolwiek parametrów. W razie potrzeby można jednak przypisać poszczególnym składowym wartości inne niż domyślne. Oto cztery możliwe sposoby stosowania adnotacji `@MyAnno`:

```
@MyAnno() // obie wartości domyślne
@MyAnno(str = "pewien tekst") // domyślne tylko val
@MyAnno(val = 100) // domyślne tylko str
@MyAnno(str = "Testowanie", val = 100) // brak wartości domyślnych
```

Poniższy program ilustruje wykorzystanie wartości domyślnych adnotacji.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Deklaracja typu adnotacji zawierającego wartości domyślne.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testowanie";
  int val() default 9000;
}

class Meta3 {

  // Adnotacja dla metody używa wartości domyślnych.
  @MyAnno()
  public static void myMeth() {
    Meta3 ob = new Meta3();

    // Pobranie adnotacji dla metody
    // i wyświetlenie wartości jej składowych.
    try {
      Class c = ob.getClass();

      Method m = c.getMethod("myMeth");

      MyAnno anno = m.getAnnotation(MyAnno.class);

      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

Program po uruchomieniu wyświetli następujący wynik.

```
Testowanie 9000
```

Adnotacje znacznikowe

Adnotacja znacznikowa to specjalna forma adnotacji, która nie zawiera żadnych składowych. Jej jedynym zadaniem jest znakowanie elementu kodu — wystarczy sama jej obecność przy jakimś elemencie. Najlepszym sposobem sprawdzenia istnienia adnotacji znacznikowej jest użycie metody `isAnnotationPresent()` zdefiniowanej w interfejsie `AnnotatedElement`.

Poniższy przykład ilustruje sposób stosowania adnotacji znacznikowych. Ponieważ interfejs adnotacji nie zawiera składowych, wystarczy sprawdzenie samego istnienia adnotacji (lub jej braku).

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Adnotacja znacznikowa.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Użycie znacznika dla metody.
  // Zauważ, iż nie są wymagane nawiasy.
  @MyMarker
  public static void myMeth() {
    Marker ob = new Marker();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      // Sprawdzenie istnienia adnotacji znacznikowej.
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("Adnotacja MyMarker istnieje.");

    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

Wynik działania programu potwierdza istnienie adnotacji:

```
Adnotacja MyMarker istnieje.
```

Warto zwrócić uwagę na brak konieczności stosowania nawiasów okrągłych po nazwie adnotacji znacznikowej `@MyMarker`. Ponieważ adnotacja nie zawiera żadnych składowych, wystarczy podać samą jej nazwę.

```
@myMarker
```

Zastosowanie pustych nawiasów nie jest błędem, ale nie są one wymagane.

Adnotacje jednoelementowe

Adnotacje jednoelementowe zawierają tylko jedną składową. Działają dokładnie tak samo jak inne adnotacje, ale dodatkowo umożliwiają stosowanie skróconej formy przypisania wartości jedynej składowej. W momencie stosowania adnotacji jednoelementowej wystarczy użyć samej wartości składowej — nie trzeba podawać jej nazwy. Aby jednak móc skorzystać z takiego skrótu, składowa musi mieć nazwę `value`.

Poniżej znajduje się przykład wykorzystujący adnotację jednoelementową.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Adnotacja jednoelementowa.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // składowa musi mieć nazwę value
}

class Single {

  // Oznaczenie metody adnotacją jednoelementową.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      MySingle anno = m.getAnnotation(MySingle.class);

      System.out.println(anno.value()); // wyświetla 100

    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

Zgodnie z oczekiwaniami program po uruchomieniu wyświetla wartość 100. Adnotacja użyta dla metody `myMeth()` ma następującą postać.

```
@MySingle(100)
```

Zauważ brak `value =`.

Skróconą składnię można zastosować również wtedy, gdy adnotacja zawiera inne składowe, pod warunkiem że zdefiniowano dla tych składowych wartości domyślne. Poniższy kod deklaruje adnotację z dodatkową składową xyz o wartości domyślnej 0.

```
@interface SomeAnno {
  int value();
  int xyz() default 0;
}
```

Wszędzie tam, gdzie ma być stosowana wartość domyślna składowej xyz, można użyć poniższej postaci adnotacji. Wystarczy podać wartość składowej `value` zgodnie ze składnią adnotacji jednoelementowej.

```
@SomeAnno(88)
```

W takiej sytuacji xyz zostanie ustawione na wartość domyślną 0, natomiast `value` otrzyma wartość 88. Gdy chce się ustawić wartości obu składowych, trzeba jawnie podać obie nazwy.

```
@SomeAnno(value = 88, xyz = 99)
```

Warto pamiętać, że składnia adnotacji jednoelementowej wymaga użycia nazwy `value` dla jedynej składowej.

Wbudowane adnotacje

Java definiuje wiele wbudowanych adnotacji. W większości są to adnotacje wyspecjalizowane, ale dziewięć ma uniwersalny charakter. Cztery z nich znajdują się w pakiecie `java.lang.annotation`: `@Retention`, `@Documented`, `@Target` i `@Inherited`. Pięć pozostałych należy do pakietu `java.lang`: `@Override`, `@Deprecated`, `@FunctionalInterface`, `@SafeVarargs` i `@SuppressWarnings`.

W wydaniu JDK 8 do interfejsu `java.lang.annotation` zostały dodane dwie nowe adnotacje: `Repeatable` oraz `Native`. Pierwsza z nich obsługuje adnotacje, które mogą się powtarzać, opisane pod koniec tego rozdziału. Z kolei druga adnotacja, `Native`, służy do oznaczania pól, do których może się odwoływać kod rdzenny.

Adnotacja @Retention

Tę adnotację zaprojektowano z myślą o stosowaniu tylko dla innych adnotacji. Adnotacja `@Retention` określa strategię zachowywania (opisana wcześniej w tym rozdziale).

Adnotacja @Documented

Ta adnotacja pełni funkcję interfejsu znaczników informującego narzędzia o konieczności udokumentowania adnotacji. Powinna być stosowana jedynie przy deklaracji innych adnotacji.

Adnotacja @Target

Ta adnotacja służy do określania typów elementów, dla których można stosować właściwą adnotację. Powinna być stosowana jedynie przy innych adnotacjach. Adnotacja `@Target` przyjmuje jeden argument, który musi być tablicą stałych typu wyliczeniowego `ElementType`. Argument określa, dla jakich typów deklaracji można używać adnotacji. Znaczenie poszczególnych stałych wyliczeniowych opisano w tabeli 12.1.

Tabela 12.1. *Stałe wyliczenia ElementType*

Stałe adnotacji @Target	Adnotację można stosować dla
`ANNOTATION_TYPE`	Innej adnotacji
`CONSTRUCTOR`	Konstruktora
`FIELD`	Pola
`LOCAL_VARIABLE`	Zmiennej lokalnej
`METHOD`	Metody
`PACKAGE`	Pakietu
`PARAMETER`	Parametru
`TYPE`	Klasy, interfejsu lub wyliczenia
`TYPE_PARAMETER`	Parametr typu (dodana w JDK 8)
`TYPE_USE`	Zastosowanie typu (dodana w JDK 8)

W adnotacji `@Target` można wskazać więcej niż jedną ze stałych, ale trzeba je oddzielić przecinkami i umieścić w nawiasach klamrowych. Aby wskazać, że dana adnotacja ma zastosowanie tylko dla pól i zmiennych lokalnych, należy użyć adnotacji `@Target` w następującej formie:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

Jeśli nie zostanie użyta adnotacja `@Target`, to z wyjątkiem parametrów typów adnotacja może być używana w dowolnych deklaracjach. Z tego względu często bardzo dobrym rozwiązaniem jest jawne określanie, dla jakich elementów można stosować daną adnotację, co w jasny i precyzyjny sposób określi jej zamierzone zastosowanie.

Adnotacja @Inherited

`@Inherited` to adnotacja znacznikowa, którą można stosować tylko przed deklaracjami innych adnotacji. Co więcej, ta adnotacja wpływa tylko na adnotacje stosowane przed definicjami klas. Adnotacja powoduje dziedziczenie adnotacji klasy bazowej przez jej podklasę. Oznacza to, że w odpowiedzi na żądanie konkretnej, brakującej adnotacji podklasy Java sprawdza, czy odpowiednia adnotacja została zdefiniowana dla klasy bazowej. Jeśli klasa bazowa została oznaczona żądaną adnotacją i jeśli oznaczono tę adnotację adnotacją `@Inherited`, to właśnie ona zostanie zwrócona.

Adnotacja @Override

Ta adnotacja znacznikowa może być stosowana tylko przed deklaracjami metod. Metoda oznaczona taką adnotacją musi przykrywać odpowiednią metodę klasy bazowej. Jeśli ten warunek nie zostanie spełniony, kompilator zgłosi stosowny błąd. W ten sposób można zagwarantować, że metoda nadklasy rzeczywiście zostanie przykryta, nie przeciążona.

Adnotacja @Deprecated

`@Deprecated` to adnotacja znacznikowa. Za pomocą tej adnotacji można wskazać przestarzałą deklarację i zasugerować istnienie nowszej, zalecanej formy.

Adnotacja @FunctionalInterface

`@FunctionalInterface` jest adnotacją znacznikową dodaną w JDK 8 i przeznaczoną do oznaczania interfejsów. Informuje ona, że interfejs, do którego została dodana, jest interfejsem funkcyjnym. Mianem **funkcyjnych** określane są interfejsy, które deklarują jedną i tylko jedną metodę abstrakcyjną. Interfejsy funkcyjne są używane przez wyrażenia lambda. (Więcej informacji o interfejsach funkcyjnych oraz wyrażeniach lambda można znaleźć w rozdziale 15.). Jeśli interfejs, do którego dodano tę adnotację, nie jest interfejsem funkcyjnym, to kompilator zgłosi błąd. Koniecznie należy zrozumieć, że adnotacja ta wcale nie jest niezbędna do tworzenia interfejsów funkcyjnych. Każdy interfejs deklarujący dokładnie jedną metodę abstrakcyjną z definicji jest interfejsem funkcyjnym. Z tego względu adnotacja `@FunctionalInterface` ma czysto informacyjny charakter.

Adnotacja @SafeVarargs

`@SafeVarargs` to adnotacja znacznikowa, którą można stosować tylko dla metod i konstruktorów. Adnotacja oznacza, że metoda lub konstruktor nie zawiera żadnych niebezpiecznych działań związanych z parametrem typu `vararg`. Adnotacja służy do wstrzymywania generowania ostrzeżeń związanych z tworzeniem egzemplarzy nieuprzedmiotowionych (ang. *non-reifiable*) typów `vararg` i tablic sparametryzowanych. (Typ nieuprzedmiotowiony to w istocie typ sparametryzowany; typy sparametryzowane zostaną omówione w rozdziale 14.). Adnotację `@SafeVarargs` można stosować tylko dla statycznych i finalnych metod i konstruktorów otrzymujących parametry typu `vararg`. Adnotację wprowadzono w wydaniu JDK 7.

Adnotacja @SuppressWarnings

Ta adnotacja wymusza wstrzymanie generowania określonych ostrzeżeń kompilatora. W adnotacji należy podać nazwę wstrzymywanego ostrzeżenia w formie łańcucha. Adnotacje tego typu można stosować dla dowolnych deklaracji.

Adnotacje typów

Zaczynając od wydania JDK 8, zwiększyła się także liczba miejsc w kodzie, w których można umieszczać adnotacje. Jak już wcześniej wspominałem, początkowo można je było dodawać wyłącznie do deklaracji. Jednak w JDK 8 adnotacje można w większości przypadków umieszczać także w miejscach, gdzie są podawane typy. Ten rozszerzony aspekt adnotacji jest nazywany **adnotacjami typów**. Na przykład adnotację można dodać do typu wyniku zwracanego przez metodę, do typu referencji `this`

wewnątrz metody, do rzutowania, do poszczególnych poziomów tablic, do dziedziczonych klas oraz do klauzuli `throws`. Adnotacje można także dodawać do typów uogólnionych, w tym także do ograniczeń parametrów typów oraz argumentów typów ogólnych. (Szczegółowe omówienie zagadnień typów uogólnionych można znaleźć w rozdziale 14.).

Adnotacje typów są ważne, gdyż pozwalają wykonywać dodatkowe testy, chroniące przed występowaniem błędów. Trzeba wiedzieć, że kompilator `javac` z założenia sam nie przeprowadza tych testów. Do tego celu służy odrębne narzędzie, choć może ono działać jako wtyczka kompilatora.

Adnotacje typów można dodawać do elementów określanych jako zastosowanie typu (`Element ➥Type.TYPE_USE`). (Warto sobie przypomnieć, że ważne elementy, dla których można stosować adnotacje, są określane przy użyciu opisanej wcześniej adnotacji `@Target`). Adnotacja typu odnosi się do typu, przed którym została ona podana. Na przykład przy założeniu, że dostępna jest adnotacja typu o nazwie `@TypeAnno`, poniższa definicja będzie prawidłowa:

```
void myMeth() throws @TypeAnno NullPointerException { //...
```

W tym przypadku adnotacja `@TypeAnno` została dodana do typu `NullPointerException` w klauzuli `throws`.

Istnieje także możliwość dodania adnotacji do typu referencji `this` (nazywanej **odbiorcą**). Jak wiadomo, `this` jest niejawnym argumentem wszystkich metod instancyjnych, które odwołują się do wywoływanego obiektu. Aby dodać adnotację do typu `this`, konieczne jest zastosowanie kolejnej nowej możliwości, dodanej w JDK 8. Otóż zaczynając do JDK 8, istnieje możliwość jawnego deklarowania `this` jako pierwszego parametru metody. W takiej deklaracji typ `this` musi odpowiadać klasie, do której należy metoda; oto przykład:

```
class SomeClass {
  int myMeth(SomeClass this, int i, int j) { //...
```

Ponieważ metoda `myMeth()` jest definiowana w klasie `SomeClass`, zatem typem `this` musi być `SomeClass`. W przypadku zastosowania takiej deklaracji zyskujemy możliwość dodania adnotacji do typu `this`. Jeśli ponownie założymy, że dostępna jest adnotacja typu o nazwie `@TypeAnno`, to poniższy wiersz kodu będzie prawidłowy:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { //...
```

Warto zwrócić uwagę, że nie trzeba jawnie deklarować referencji `this`, chyba że chcemy dodać do niej adnotację. (Nawet jeśli `this` nie zostanie jawnie zadeklarowana jako parametr, to i tak będzie ona przekazywana do metody. Pod tym względem w JDK 8 *nic nie uległo* zmianie). Poza tym jawne deklarowanie `this` w żaden sposób nie zmienia sygnatury metody, gdyż i tak parametr ten jest domyślnie deklarowany, choć niejawnie. Warto zatem powtórzyć, że parametr `this` będziemy deklarować wyłącznie w sytuacjach, kiedy będziemy chcieli dodać do niego adnotację typu. Jeśli jednak `this` zostanie zadeklarowany, to *musi* on być pierwszym parametrem.

Zamieszczony poniżej program przykładowy przedstawia kilka miejsc, w których można stosować adnotacje typów. Definiuje on kilka adnotacji, z których parę jest adnotacjami typów. Nazwy adnotacji, jak również elementy, dla których można je stosować, zostały przedstawione w tabeli 12.2.

Tabela 12.2. *Zastosowane w przykładzie adnotacje oraz elementy, dla których można je stosować*

Nazwa adnotacji	Określenie elementu, dla którego można ją stosować:
`@TypeAnno`	`ElementType.TYPE_USE`
`@MaxLen`	`ElementType.TYPE_USE`
`@NotZeroLen`	`ElementType.TYPE_USE`
`@Unique`	`ElementType.TYPE_USE`
`@What`	`ElementType.TYPE_PARAMETER`
`@EmptyOK`	`ElementType.FIELD`
`@Recommended`	`ElementType.METHOD`

Warto zwrócić uwagę, że `@EmptyOK`, `@Recommended` oraz `@What` nie są adnotacjami typów. Zostały one dołączone do programu w celach porównawczych. Spośród nich szczególnie interesująca jest adnotacja `@What`, która jest przeznaczona do dodawania do deklaracji parametru typu. To kolejna nowa możliwość stosowania adnotacji wprowadzona w JDK 8. Komentarze zamieszczone w kodzie programu wyjaśniają zastosowanie każdej z adnotacji.

```
// Prezentacja kilku adnotacji typów.
import java.lang.annotation.*;
import java.lang.reflect.*;

// Adnotacja znacznikowa, którą można dodawać do typów.
@Target(ElementType.TYPE_USE)
@interface TypeAnno { }

// Inna adnotacja znacznikowa, którą można dodawać do typów.
@Target(ElementType.TYPE_USE)
@interface NotZeroLen {
}

// Jeszcze inna adnotacja znacznikowa, którą można dodawać
// do typów.
@Target( ElementType.TYPE_USE )
@interface Unique { }

// Adnotacja sparametryzowana, którą można dodawać do typów.
@Target(ElementType.TYPE_USE)
@interface MaxLen {
  int value();
}

// Adnotacja, którą można dodawać do parametru typu.
@Target(ElementType.TYPE_PARAMETER)
@interface What {
  String description();
}

// Adnotacja, którą można dodawać do deklaracji pola.
@Target(ElementType.FIELD)
@interface EmptyOK { }

// Adnotacja, którą można dodawać do deklaracji metody.
@Target(ElementType.METHOD)
@interface Recommended { }


// Zastosowanie adnotacji dla parametru typu
class TypeAnnoDemo<@What(description = "Ogólny typ danych") T> {

  // Zastosowanie adnotacji typu dla konstruktora.
  public @Unique TypeAnnoDemo() {}

  // Adnotacja typu (w tym przypadku typu String), a nie pola.
  @TypeAnno String str;

  // A to jest adnotacja pola test.
  @EmptyOK String test;

  // Zastosowanie adnotacji typu dla this (odbiorcy).
  public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
    return 10;
  }

  // Adnotacja dodana do typu wyniku zwracanego przez metodę.
  public @TypeAnno Integer f2(int j, int k) {
```

```
    return j+k;
  }

  // Adnotacja dodana do deklaracji metody.
  public @Recommended Integer f3(String str) {
    return str.length() / 2;
  }

  // Zastosowanie adnotacji typu w klauzuli throws.
  public void f4() throws @TypeAnno NullPointerException {
    // ...
  }

  // Adnotacje zastosowane do różnych poziomów tablicy.
  String @MaxLen(10) [] @NotZeroLen [] w;

  // Adnotacja zastosowana dla typu elementów tablicy.
  @TypeAnno Integer[] vec;

  public static void myMeth(int i) {

    // Zastosowanie adnotacji typu dla argumentu typu.
    TypeAnnoDemo<@TypeAnno Integer> ob =
                            new TypeAnnoDemo<@TypeAnno Integer>();

    // Zastosowanie adnotacji typu dla new.
    @Unique TypeAnnoDemo<Integer> ob2 = new @Unique TypeAnnoDemo<Integer>();

    Object x = new Integer(10);
    Integer y;

    // Zastosowanie adnotacji typu dla rzutowania.
    y = (@TypeAnno Integer) x;
  }

  public static void main(String args[]) {
    myMeth(10);
  }

  // Zastosowanie adnotacji typu w deklaracji klasy.
  class SomeClass extends @TypeAnno TypeAnnoDemo<Boolean> {}
}
```

Choć w większości przypadków element, do którego odnoszą się adnotacje, w tym przykładzie jest oczywisty, to jednak cztery zastosowania wymagają dokładniejszego omówienia. Pierwszym z nich jest adnotacja typu wyniku zwracanego przez metodę oraz adnotacja deklaracji metody. W powyższym programie należy zatem zwrócić szczególną uwagę na dwie przedstawione poniżej deklaracje metod:

```
// Adnotacja dodana do typu wyniku zwracanego przez metodę.
public @TypeAnno Integer f2(int j, int k) {
  return j+k;
}

// Adnotacja dodana do deklaracji metody.
public @Recommended Integer f3(String str) {
  return str.length() / 2;
}
```

W tych deklaracjach należy zwrócić uwagę na to, że adnotacja jest umieszczona przed typem zwracanego wyniku (którym w obu metodach jest `Integer`). Jednak w każdej z tych metod adnotacja została dodana do innego elementu. W pierwszym przypadku adnotacja `@TypeAnno` dotyczy typu wyniku zwracanego przez metodę `f2()`. Wynika to z faktu, że typ elementu, dla którego można stosować tę adnotację, został określony jako `ElementType.TYPE_USE`, co oznacza, że można jej używać

w miejscach zastosowania typów. W drugim przypadku adnotacja `@Recommended` dotyczy samej deklaracji metody. Wynika to z faktu, że typ elementu, dla którego można stosować tę adnotację, został określony jako `ElementType.METHOD`. W efekcie adnotacja `@Recommended` odnosi się do deklaracji metody, a nie typu jej wyniku. Jak widać, określenie elementu, dla którego można stosować adnotację, pozwala wyeliminować pozorną niejednoznaczność pomiędzy adnotacją deklaracji metody oraz adnotacją typu wyniku zwracanego przez metodę.

I jeszcze jedna uwaga dotycząca adnotacji typu wyniku zwracanego przez metodę: jeśli tym typem jest `void`, to nie będzie można dodać do niego adnotacji.

Kolejnym interesującym aspektem przedstawionym w powyższym programie przykładowym są adnotacje pól. Ich przykłady pokazano ponownie w poniższym fragmencie kodu:

```
// Adnotacja typu (w tym przypadku typu String), a nie pola.
@TypeAnno String str;

// A to jest adnotacja pola test.
@EmptyOK String test;
```

W tym przypadku adnotacja `@TypeAnno` dotyczy typu `String`, natomiast adnotacja `@EmptyOK` — pola `test`. Choć każda z ich jest umieszczona na samym początku deklaracji, to elementy, których dotyczą te adnotacje, są inne i zależą od typu elementów, dla których dane adnotacje można stosować. Jeśli typ elementów, dla których można stosować adnotację, został określony jako `ElementType.TYPE_USE`, to adnotacja dotyczy typu. Jeśli natomiast typ elementów, dla których można stosować adnotację, został określony jako `ElementType.FIELD`, to adnotacja dotyczy pola. A zatem sytuacja jest dosyć podobna do tej, która występowała wcześniej w przypadku metod, więc nie ma mowy o żadnych niejednoznacznościach. Dokładnie ten sam mechanizm pozwala uniknąć niejednoznaczności w razie dodawania adnotacji do zmiennych lokalnych.

Trzeba także zwrócić uwagę na sposób, w jaki dodano adnotację do referencji `this` (odbiorcy):

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Jak widać, `this` jest pierwszym parametrem metody `f()`, a jego typ został określony jako `TypeAnnoDemo` (czyli odpowiada klasie, której składową jest metoda `f()`). Jak już zaznaczyłem, zaczynając od JDK 8, istnieje możliwość jawnego podawania w deklaracjach parametru `this`, właśnie po to, by można było do niego dodawać adnotacje typu.

I w końcu adnotacje poziomów tablicy zastosowane w poniższej instrukcji:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

W tej deklaracji adnotacja `@MaxLen` dotyczy typu pierwszego poziomu tablicy, a adnotacja `@NotZeroLen` — typu jej drugiego poziomu. Z kolei w poniższej deklaracji:

```
@TypeAnno Integer[] vec;
```

adnotacja została dodana do typu elementów tablicy, którym jest `Integer`.

Adnotacje powtarzalne

Kolejną nowością związaną z adnotacjami wprowadzoną w JDK 8 jest możliwość powtarzania adnotacji odnoszących się do tego samego elementu. Są one nazywane **adnotacjami powtarzalnymi**. Aby zapewnić możliwość powtarzania konkretnej adnotacji, należy do niej dodać adnotację `@Repeatable`, zdefiniowaną w pakiecie `java.lang.annotation`. Jej składowa `value` określa typ **kontenera** dla adnotacji powtarzalnej. Kontener jest określany jako adnotacja, której składowa `value` jest tablicą, przy czym jej typ odpowiada typowi adnotacji powtarzalnej. Innymi słowy, aby utworzyć aplikację powtarzalną, należy utworzyć adnotację kontenera, a następnie zastosować typ tej adnotacji jako argument adnotacji `@Repeatable`.

Aby uzyskać dostęp do adnotacji powtarzalnych na przykład za pomocą metody `getAnnotation()`, należy skorzystać z adnotacji kontenera, a nie z samej adnotacji powtarzalnej. Poniższy program przedstawia takie rozwiązanie — zmienia on przedstawioną wcześniej adnotację `MyAnno` w adnotację powtarzalną i pokazuje, jak można jej używać.

```
// Prezentacja adnotacji powtarzalnych.

import java.lang.annotation.*;
import java.lang.reflect.*;

// Zapewnia możliwość powtarzania adnotacji MyAnno.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
  String str() default "Testujemy";
  int val() default 9000;
}

// To jest adnotacja kontenera.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
  MyAnno[] value();
}

class RepeatAnno {

  // Do myMeth() dodawane są dwie  adnotacje MyAnno.
  @MyAnno(str = "Pierwsza adnotacja", val = -1)
  @MyAnno(str = "Druga adnotacja", val = 100)
  public static void myMeth(String str, int i)
  {
    RepeatAnno ob = new RepeatAnno();

    try {
      Class<?> c = ob.getClass();

      // Pobiera adnotacje dotyczące metody myMeth().
      Method m = c.getMethod("myMeth", String.class, int.class);

      // Wyświetla powtórzone adnotacje MyAnno.
      Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
      System.out.println(anno);

    } catch (NoSuchMethodException exc) {
       System.out.println("Nie znaleziono metody.");
    }
  }

  public static void main(String args[]) {
    myMeth("test", 10);
  }
}
```

Ten program generuje następujące wyniki:

```
@MyRepeatedAnnos(value=[@MyAnno(val=-1, str=Pierwsza adnotacja), @MyAnno(val=100, str=Druga adnotacja)])
```

Zgodnie z wcześniejszymi wyjaśnieniami, aby adnotacja `MyAnno` mogła być powtarzana, należy do niej dodać adnotację `@Repeatable` określającą adnotację kontenera. W tym przykładzie adnotacją kontenera jest `MyRepeatedAnnos`. Program pobiera adnotacje powtarzalne, wywołując metodę `getAnnotation()` i przekazując do niej klasę adnotacji kontenera, a nie klasę adnotacji powtarzalnych. Jak pokazują wyniki działania programu, adnotacje powtarzalne są od siebie oddzielane przecinkami, a nie zwracane pojedynczo.

Innym sposobem pobierania adnotacji powtarzalnych jest korzystanie z jednej z nowych metod dodanych w JDK 8 do interfejsu `AnnotatedElement` i pozwalających na bezpośrednie operowanie na adnotacjach powtarzalnych. Chodzi konkretnie o dwie metody: `getAnnotationsByType()` oraz `getDeclaredAnnotationsByType()`. Przyjrzymy się pierwszej z nich:

```
<T extends Annotation> T[] getAnnotationsByType(Class<T> annoType)
```

Metoda ta zwraca tablicę adnotacji typu `annoType`, dodanych do obiektu, na rzecz którego metoda została wywołana. Jeśli takich adnotacji nie ma, metoda zwróci pustą tablicę. Poniżej przedstawiony został krótki przykład użycia metody `getAnnotationsByType()` w celu pobrania adnotacji typu `MyAnno`:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
  System.out.println(a);
```

Jak widać, w wywołaniu metody `getAnnotationsByType()` przekazywany jest typ adnotacji powtarzalnej. Tablica zwrócona przez tę metodę zawiera wszystkie egzemplarze adnotacji `MyAnno` dodane do metody `myMeth()`; w tym przykładzie są dwie takie adnotacje. Do każdej ze zwróconych adnotacji można odwołać się przy użyciu indeksu tablicy. Powyższy przykład wyświetla te adnotacje, korzystając z wersji `for-each` pętli `for`.

Ograniczenia

Ze stosowaniem deklaracji adnotacji wiąże się kilka ograniczeń. Po pierwsze, żadna adnotacja nie może dziedziczyć po innej adnotacji. Po drugie, wszystkie metody deklarowane w adnotacji muszą być bezparametrowe. Co więcej, metody adnotacji mogą zwracać tylko wartości następujących typów:

- typ prosty, na przykład `int` lub `double`;
- obiekt typu `String` lub `Class`;
- typ wyliczeniowy;
- typ innej adnotacji;
- tablicę jednego z wymienionych wcześniej typów.

Adnotacje nie mogą korzystać z typów sparametryzowanych (ich dokładne omówienie znajduje się w rozdziale 14.). Nie mogą też zawierać klauzuli `throws`.

Skorowidz

A

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Y

Z

Ż

Kompletne źródło informacji na temat języka Java!

Od czasu swojej premiery w 1995 roku Java cieszy się niesłabnącą popularnością wśród programistów oraz pracodawców. Język ten znajduje zastosowanie wszędzie tam, gdzie wymagane są najwyższa wydajność, niezawodność i bezpieczeństwo. Ponadto każda kolejna wersja Javy wprowadza nowości, które sprawiają, że życie programisty staje się łatwiejsze, a tworzony kod bardziej przejrzysty.

Ukazanie się na rynku wersji ósmej tego języka jest świetną okazją do zapoznania się z najnowszym wydaniem cenionego podręcznika na jego temat, poprawionego oraz uzupełnionego o wiadomości o wprowadzonych nowościach. W trakcie lektury zdobędziesz informacje na temat składni Javy, typowych konstrukcji oraz najlepszych technik programowania. Ponadto poznasz platformę JavaFX i wyrażenia lambda oraz odkryjesz sposoby radzenia sobie z zagadnieniem wielowątkowości. Książka ta jest doskonałą i pasjonującą lekturą zarówno dla programistów języka Java, jak i osób chcących poznać ten język.

Dzięki tej książce:

- **poznasz składnię języka Java**
- **zaznajomisz się z nowościami w Javie 8**
- **wykorzystasz potencjał platformy JavaFX**
- **opanujesz wyrażenia lambda**
- **nauczysz się tworzyć wydajne rozwiązania wielowątkowe**

Herbert Schildt — były członek komitetu ANSI/ISO, który dokonał standaryzacji języka C++. Autor popularnych publikacji poświęconych programowaniu, głównie w językach C, C++, C# i Java. Jego książki, wykorzystywane przez profesjonalistów, szkoleniowców i studentów, rozeszły się w ponad 3 milionach egzemplarzy.

Helion

34787 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

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

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

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-0812-1

9 788328 308121

cena: 179,00 zł

Oracle Press