Efektywne programowanie w języku Java

Autor: Joshua Bloch
Tłumaczenie: Paweł Gonera
ISBN: 83-7197-989-4
Tytuł oryginału: Effective Java Programming Language
Format: B5, stron: 214
Przykłady na ftp: 48 kB

Java to wspaniałe narzędzie w rękach programisty. Ale nawet najlepsze narzędzie może zostać źle użyte. Istnieje wiele książek, które opisują ten język programowania skupiając się na przedstawieniu jego składni. Ta książka jest zupełnie inna. Adresowana jest do osób znających już Javę i przedstawia praktyczne zasady pisania efektywnego, poprawnego kodu.

Każda wskazówka omówiona jest w osobnym podrozdziale opisującym dany problem, przykłady poprawnego (i błędnego!) kodu, a także historie zaczerpnięte z bogatego doświadczenia autora. Ta książka zapozna Cię z idiomami właściwymi językowi Java oraz z istotnymi z praktycznego punktu widzenia wzorcami projektowymi.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Odpowiedniki konstrukcji języka C

Język programowania Java posiada wiele podobieństw do języka C, ale kilka jego konstrukcji zostało pominiętych. W większości przypadków oczywiste jest, dlaczego dana konstrukcja została pominięta i w jaki sposób można sobie bez niej radzić. W rozdziale tym zaproponujemy zamienniki dla kilku pominiętych konstrukcji języka C, których zastąpienie nie jest tak oczywiste.

Najczęstszym wątkiem, który przewija się przez cały ten rozdział, jest twierdzenie, że wszystkie pominięte konstrukcje były zorientowane na dane, a nie zorientowane obiektowo. Język programowania Java zawiera bardzo wydajny system typów i proponowane zamienniki w pełni korzystają z tego systemu w celu zapewnienia wyższego stopnia abstrakcji niż konstrukcje języka C, które są przez nie zastępowane.

Nawet jeżeli zdecydujesz się na pominięcie tego rozdziału, warto zapoznać się z tematem 21., poświęconym *typowi wyliczeniowemu*, który zastępuje konstrukcję `enum`, dostępną w języku C. Wzorzec ten nie był powszechnie znany w czasie pisania tej książki, a posiada znaczną przewagę nad obecnie stosowanymi rozwiązaniami tego problemu.

Temat 19. Zastępowanie struktur klasami

Konstrukcja `struct` języka C została usunięta z języka Java, ponieważ za pomocą klasy można zrealizować wszystko to, co potrafi struktura, a nawet więcej. Struktura jedynie grupuje kilka pól danych w jeden obiekt — klasa zawiera operacje wykonywane na wynikowym obiekcie i pozwala na ukrycie pól danych przed użytkownikiem obiektu. Inaczej mówiąc, klasa *hermetyzuje* dane w obiekcie i umożliwia dostęp do nich jedynie za pomocą metod, co pozwala twórcy klasy na swobodną zmianę reprezentacji danych w późniejszym czasie (temat 12.).

W czasie pierwszych pokazów języka Java niektórzy programiści korzystający z języka C uważali, że klasy są zbyt obszerne, aby w pewnych sytuacjach zastąpić struktury.

Nie będziemy się zajmowali tym problemem. Zdegenerowane klasy składające się jedynie z pól danych są pewnym przybliżeniem struktur z języka C:

```
// Takie zdegenerowane klasy nie powinny być publiczne
class Point {
  public float x;
  public float y;
}
```

Ponieważ takie klasy pozwalają na dostęp do swoich pól, nie pozwalają na skorzystanie z zalet hermetyzacji. Nie można zmienić reprezentacji danych w takiej klasie bez zmiany API, nie można wymuszać żadnych ograniczeń oraz nie można podejmować dodatkowych zadań podczas modyfikacji pola. Ortodoksyjni programiści obiektowi uważają, że takie klasy są zakazane i powinny zawsze być zastępowane klasami z polami prywatnymi oraz publicznymi metodami je udostępniającymi:

```
// Hermetyzowana klasa o funkcjach struktury
class Point {
  private float x;
  private float y;

  public Point(float x, float y) {
    this.x = x;
    this.y = y;
  }

  public float getX() {return x; }
  public float getY() {return y; }

  public float setX() {this.x = x; }
  public float setY() {this.y = y; }
}
```

Oczywiście twierdzenie to jest prawdziwe w odniesieniu do klas publicznych — jeżeli klasa jest dostępna spoza swojego pakietu, rozważny programista powinien zabezpieczyć sobie możliwość zmiany wewnętrznej reprezentacji danych. Jeżeli klasa publiczna udostępnia swoje pola, nie ma możliwości zmiany reprezentacji danych, ponieważ kod klientów, korzystający z klasy publicznej, może być już rozesłany po całym świecie.

Jeżeli jednak klasa jest prywatna w ramach pakietu lub jest to prywatna klasa zagnieżdżona, nie ma nic złego w bezpośrednim udostępnieniu pól danych — zakładając, że naprawdę opisują abstrakcję definiowaną przez klasę. Podejście to generuje mniej kodu niż wykorzystanie metod dostępowych, zarówno w definicji klasy, jak i w kodzie klientów ją wykorzystujących. Ponieważ kod klientów jest ściśle związany z wewnętrzną reprezentacją klasy, jest on ograniczony do pakietu, w którym klasa ta jest zdefiniowana. W przypadku, gdy konieczna jest zmiana reprezentacji danych, możliwe jest wprowadzenie zmian bez konieczności zmiany kodu poza pakietem. W przypadku prywatnej klasy zagnieżdżonej zasięg zmian jest ograniczony do klasy nadrzędnej.

Kilka klas w bibliotekach języka Java nie dotrzymuje zalecenia, aby klasy publiczne nie udostępniały bezpośrednio swoich pól. Przykładami takich klas są klasy `Point` i `Dimension` z pakietu `java.awt`. Przykłady te nie powinny być naśladowane — należałoby raczej

wskazywać je jako przykład negatywny. W temacie 37. opisany został przykład pokazujący, jak udostępnienie pól w klasie `Dimension` spowodowało problemy z wydajnością. Problemy te nie mogą zostać usunięte bez wpływania na kod klientów.

Temat 20. Zamiana unii na hierarchię klas

Konstrukcja języka C — `union` — jest najczęściej wykorzystywana do definiowania struktur, umożliwiających przechowywanie więcej niż jednego typu danych. Struktura taka zwykle posiada co najmniej dwa pola — unię i *znacznik*. Pole znacznika jest zwykłym polem, wykorzystywanym do wskazywania aktualnego typu danych, przechowywanego przez unię. Znacznik jest najczęściej typu wyliczeniowego (`enum`). Struktura zawierająca unię i znacznik jest czasami nazywaną *unią z dyskryminatorem*.

Poniżej przedstawiamy przykład definicji typu `shape_t`, zapisany w języku C. Jest to unia z dyskryminatorem, reprezentująca prostokąt lub koło. Funkcja `area` na podstawie wskaźnika do struktury `shape_t` zwraca pole figury lub `-1.0`, jeżeli struktura zawiera nieprawidłowe dane.

```
/* Unia z dyskryminatorem */
#include "math.h"
typedef enum {RECTANGLE, CIRCLE} shapeType_t;

typedef struct {
    double length;
    double width;
} rectangleDimensions_t;

typedef struct {
    double radius;
} circleDimensions_t;

typedef struct {
    shapeType_t tag;
    union {
        rectangleDimensions_t rectangle;
        circleDimensions_t    circle;
    } dimensions;
} shape_t;

double area(shape_t *shape) {
    switch(shape->tag) {
      case RECTANGLE: {
        double length = shape->dimensions.rectangle.length;
        double width  = shape->dimensions.rectangle.width;
        return length * width;
      }
      case CIRCLE: {
        double r = shape->dimensions.circle.radius;
        return M_PI * (r*r);
      }
      default: return -1.0; /* Nieprawidłowy znacznik */
    }
}
```

Projektanci języka Java postanowili nie wprowadzać konstrukcji `union`, ponieważ istnieje dużo lepszy mechanizm definiowania typów umożliwiających reprezentowanie różnych typów — dziedziczenie. Unie z dyskryminatorem są jedynie mało wydajną imitacją hierarchii klas.

Aby zamienić unię z dyskryminatorem na hierarchię klas, należy zdefiniować klasę abstrakcyjną zawierającą metody abstrakcyjne dla każdej operacji, której działanie jest zależne od wartości znacznika. We wcześniejszym przykładzie przedstawiona była tylko jedna taka operacja — `area`. Ta klasa abstrakcyjna staje się korzeniem hierarchii klas. Jeżeli istnieją inne operacje, niezależne od wartości znacznika, należy zdefiniować odpowiednie metody w klasie bazowej. Podobnie, jeżeli w unii z dyskryminatorem istnieją pola danych poza znacznikiem i unią, reprezentujące typy danych wspólne dla wszystkich typów, powinny być one dodane do klasy bazowej. W naszym przykładzie nie było żadnych składników niezależnych od typów.

Następnie dla każdego typu reprezentowanego w unii z dyskryminatorem definiujemy klasy, dziedziczące z klasy bazowej. W naszym przykładzie typami tymi są: koło i prostokąt. W każdej z klas podrzędnych należy umieścić pola danych odpowiednie dla jej typu. W naszym przykładzie dla koła jest to promień, a dla prostokąta długość i szerokość. Na koniec definiujemy odpowiednie implementacje metod abstrakcyjnych z klasy bazowej. Poniżej przedstawiamy hierarchię klas dla naszego przykładu unii z dyskryminatorem:

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }

    double area() { return length * width; }
}
```

Hierarchia klas posiada wiele zalet w porównaniu z unią z dyskryminatorem. Najważniejszą z nich jest ta, że hierarchia klas zapewnia kontrolę typów. W naszym przykładzie każdy obiekt `Shape` może być jedynie prawidłowym obiektem `Circle` lub `Rectangle`. Bardzo prosto wygenerować strukturę `shape_t`, która będzie całkowicie nieprzydatna, ponieważ połączenie pomiędzy znacznikiem i unią nie jest wymuszane przez język programowania. Jeżeli znacznik wskazuje, że `shape_t` reprezentuje prostokąt, ale unia

została zainicjowana danymi koła, wszystko może się zdarzyć. Nawet, gdy unia z dyskryminatorem zostanie prawidłowo zainicjowana, możliwe jest omyłkowe przekazanie jej do funkcji nieodpowiedniej dla danej wartości znacznika.

Drugą zaletą hierarchii klas jest łatwość rozszerzania, nawet o wiele niezależnie działających części. Aby rozszerzyć hierarchię klas, wystarczy dodać nową kasę pochodną. Jeżeli zapomnisz zdefiniować jednej z metod abstrakcyjnych, natychmiast wskaże Ci to kompilator. Aby rozbudować unię z dyskryminatorem, należy mieć dostęp do kodu źródłowego. Musisz dodać nową wartość do typu `enum` oraz nową gałąź do instrukcji `switch` w każdej funkcji operującej na unii. Na koniec musisz skompilować kod. Jeżeli w którejś funkcji zapomnisz dodać nowego przypadku, kompilator nie będzie w stanie tego wykryć. Pozostaje umieszczenie w kodzie kontroli niespodziewanych wartości znacznika i generowanie w takich sytuacjach komunikatów błędów.

Czwartą zaletą hierarchii klas jest możliwość odwzorowania naturalnych relacji hierarchicznych pomiędzy typami, co pozwala na zwiększenie elastyczności i lepszej kontroli typów w czasie kompilacji. Załóżmy, że do naszego oryginalnego przykładu chcemy dodać obsługę kwadratów. W hierarchii klas możemy odwzorować fakt, że kwadrat jest specjalnym rodzajem prostokąta (zakładając, że oba są niezmienne):

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }

    double side() {
        return length; // może być również width
    }
}
```

Przedstawiona hierarchia klas nie jest jedynym rozwiązaniem naszego problemu. Hierarchia ta powstała po podjęciu kilku decyzji, o których warto wspomnieć. Klasy w hierarchii, poza `Square`, udostępniają swoje pola, nie oferując metod dostępowych. W przypadku klas publicznych jest to nie do zaakceptowania, ale nam zależało na zwięzłości kodu (temat 19.). Klasy te są niezmienne. Czasami nie jest to najlepsze, jednak najczęściej właśnie takie rozwiązanie jest właściwe (temat 13.).

Ponieważ język Java nie zawiera konstrukcji `union`, można uważać, że nie ma niebezpieczeństwa utworzenia unii z dyskryminatorem. Możliwe jest jednak napisanie kodu, który będzie posiadał te same wady. Jeżeli kiedykolwiek będziesz chciał napisać klasę z polem znacznikowym, należy pomyśleć o eliminacji pola znacznikowego przez modyfikację hierarchii klas.

Innym zastosowaniem konstrukcji `union` w języku C, całkowicie niezwiązanym z uniami z dyskryminatorem, jest możliwość oglądania wewnętrznej reprezentacji danych poprzez umyślne omijanie systemu typów. Metoda ta demonstrowana jest przez poniższy fragment kodu w języku C, który drukuje wewnętrzną postać liczby `float` w postaci szesnastkowej:

```
union {
  float f;
  int   bits;
```

```
} sleaze;

sleaze.f = 6.699e-41;        /* Umieszczenie danych w jednym z pól unii */
printf("%x\n", sleaze.bits); /* ... i odczytanie z drugiego */
```

Choć może być to użyteczne, szczególnie dla programistów systemowych, takie nieprzenośne zastosowanie nie ma odpowiednika w języku Java. Działanie takie nie może być dopuszczalne w języku, który gwarantuje bezpieczeństwo typów i nieomal izoluje programistów od wewnętrznej reprezentacji danych.

Pakiet `java.lang` zawiera metody pozwalające przekształcić liczby zmiennoprzecinkowe na ich bitową reprezentację, ale działanie tych metod jest bardzo dokładnie zdefiniowane w celu zapewnienia ich przenośności. Poniższy fragment kodu jest luźnym odpowiednikiem przedstawionego kodu w języku C, ale gwarantuje uzyskanie identycznych wyników bez względu na platformę, na której jest uruchomiony:

```
System.out.println( Integer.toHexString(Float.floatToIntBits(6.699e-41f)));
```

Temat 21. Zastępowanie konstrukcji enum za pomocą klas

Konstrukcja `enum` również nie została przeniesiona do języka Java. Konstrukcja ta służy do definiowania typu wyliczeniowego — typu, składającego się ze stałego zbioru wartości. Niestety, konstrukcja ta nie jest zbyt zaawansowana. Definiuje ona tylko zbiór nazwanych stałych typu `integer`, nie zapewniając żadnego mechanizmu kontroli typów. W języku C można wykonać następujące wyrażenia:

```
typedef enum {FUJI, PIPPIN, GRANNY_SMITH} apple_t;  /* rodzaje jabłek */
typedef enum {NAVEL, TEMPLE, BLOOD} orange_t;   /* rodzaje pomarańczy */
orange_t myFavorite = PIPPIN;      /* Mieszanie jabłek z pomarańczami */
```

ale takie jest nieprawidłowe:

```
orange_t x = (FUJI - PIPPIN)/TEMPLE;
```

Konstrukcja `enum` nie zawiera przestrzeni nazw dla tworzących ją stałych. Dlatego poniższa deklaracja, zawierająca użytą już nazwę, pozostaje w konflikcie z deklaracją typu `orange_t`:

```
typedef enum {BLOOD, SWEAT, TEARS} fluid_t;
```

Typy definiowane za pomocą konstrukcji `enum` są niepewne. Dodanie stałych do takiego typu bez ponownej kompilacji klientów powoduje nieprzewidywalne działanie, niezależnie od tego, jak dokładnie są sprawdzane istniejące wartości stałych. Poszczególne zespoły nie mogą niezależnie dodawać stałych do tych typów, ponieważ nowe typy wyliczeniowe bardzo często są ze sobą w konflikcie. Konstrukcja `enum` nie zapewnia żadnego mechanizmu, ułatwiającego zamianę stałych wyliczanych na ciągi znaków lub przeglądanie stałych w typie.

Niestety, najczęściej spotykany sposób emulowania typu wyliczeniowego w języku Java posiada wszystkie wady konstrukcji `enum` z języka C:

```
// sposób emulacji emun za pomocą int - problematyczny!!!
public class PlayingCard {
    public static final int SUIT_CLUBS    = 0;
    public static final int SUIT_DIAMONDS = 1;
    public static final int SUIT_HEARTS   = 2;
    public static final int SUIT_SPADES   = 3;
   ...
}
```

Możesz się również spotkać z odmianą tego wzorca, wykorzystującą stałe typu `String`. Wariant taki nigdy nie powinien być używany. Choć pozwala na bezpośrednie drukowanie nazw stałych, może powodować obniżenie wydajności, ponieważ korzysta z porównywania ciągów. Dodatkowo niedoświadczeni użytkownicy mogą wbudować stałe w kod zamiast korzystania z odpowiednich nazw pól. Jeżeli taka stała posiada błąd (literówkę), to błąd ten nie będzie wykryty w czasie kompilacji i będzie powodował powstanie błędów wykonania.

Na szczęście język Java pozwala na utworzenie innej metody emulacji typu `enum`, która nie posiada wszystkich wad użycia poprzedniej metody z wartościami `int` lub `String`, a ponadto ma kilka dodatkowych zalet. Jest ona nazywana *bezpiecznym typem wyliczeniowym*. Typ ten nie jest niestety zbyt dobrze znany. Pomysł jest prosty — należy zdefiniować klasę, reprezentującą pojedynczy element typu wyliczeniowego, nie definiując publicznego konstruktora. Zamiast tego należy udostępnić publiczne pola statyczne typu `final`, po jednym dla każdej ze stałych typu wyliczeniowego. Wzorzec ten w najprostszej postaci wygląda następująco:

```
// bezpieczny typ wyliczeniowy
public class Suit {
    private final String name;

    private Suit(String name) { this.name = name; }

    public String toString()  { return name; }

    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

Ponieważ klienci nie mogą utworzyć obiektów tej klasy ani jej rozszerzać, nie mogą istnieć inne obiekty tego typu poza udostępnianymi przez pola statyczne. Choć klasa nie jest zadeklarowana jako `final`, nie można po niej dziedziczyć — konstruktor klasy pochodnej musi wywołać konstruktor klasy bazowej, a on jest niedostępny.

Jak można się domyślić na podstawie nazwy, ten wzorzec pozwala na sprawdzanie typów w czasie kompilacji. Jeżeli zadeklarujesz metodę z parametrem typu `Suit`, masz pewność, że każda referencja różna od `null` będzie prawidłowym obiektem, reprezentującym jedną ze stałych. Wszystkie próby przekazania obiektu o niewłaściwym typie zostaną wykryte w czasie kompilacji, podobnie jak próby przypisania wyrażenia

jednego typu wyliczeniowego do zmiennej innego typu. Można tworzyć wiele typów wyliczeniowych z identycznie nazywającymi się stałymi, ponieważ każda klasa posiada swoją przestrzeń nazw.

Do takiej reprezentacji typu wyliczeniowego można dodawać kolejne stałe i nie powoduje to konieczności rekompilacji klientów, ponieważ publiczne statyczne pola, zawierające referencje do stałych, izolują klientów od klasy, realizującej typ wyliczeniowy. Same stałe nie są wkompilowywane w kod klienta, tak jak często zdarza się to w przypadku realizacji konstrukcji `enum` za pomocą rozwiązania, korzystającego z pól `int` lub `String`.

Ponieważ przedstawiona realizacja typu wyliczeniowego jest zwykłą klasą, może ona przedefiniować metodę `toString`, co pozwoli na zmianę wartości zmiennych na postać nadającą się do wydrukowania. Jeżeli jest to potrzebne, można również zwracać komunikaty w odpowiednim języku. Zwróć uwagę, że ciągi znaków są wykorzystywane jedynie przez metodę `toString`. Nie są używane do porównywania obiektów, ponieważ odziedziczona po klasie `Object` metoda `equals` porównuje referencje do obiektów.

Można również dodać do klasy implementującej typ wyliczeniowy dowolne metody, jakie mogą być potrzebne. W naszej klasie `Suit` może przydać się metoda zwracająca kolor lub rysunek, skojarzony z odpowiednią stałą. Klasa była początkowo prostą realizacją typu wyliczeniowego i z czasem zaczęła się zamieniać w bogatą w funkcję abstrakcję opisywanego obiektu.

Ponieważ do klasy implementującej typ wyliczeniowy można dodawać dowolne metody, może ona implementować interfejsy. Na przykład załóżmy, że chcemy, aby nasza klasa `Suit` implementowała interfejs `Comparable`, co pozwoli klasom sortować karty według koloru. Przedstawimy teraz modyfikację oryginalnego wzorca, która implementuje ten interfejs. Zmienna statyczna `nextOrdinal` używana jest do przypisywania numeru kolejnego dla każdego z tworzonych obiektów. Numery te są używane przez metodę `compareTo` do porządkowania obiektów.

```
// Typ wyliczeniowy numerowany
public class Suit implements Comparable {
    private final String name;

    // Numer kolejny tworzonego koloru
    private static int nextOrdinal = 0;

    // Przypisanie numeru kolejnego do koloru
    private final int ordinal = nextOrdinal++;

    private Suit(String name) { this.name = name; }

    public String toString()  { return name; }

    public int compareTo(Object o) {
        return ordinal - ((Suit)o).ordinal;
    }

    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

Ponieważ stałe typu wyliczeniowego są obiektami, można umieszczać je w kolekcjach. Załóżmy na przykład, że chcemy w klasie `Suit` udostępnić niezmienną listę kolorów w standardowym porządku. Wystarczy dodać do klasy deklarację dwóch klas pól.

```
private static final Suit[] PRIVATE_VALUES =
    { CLUBS, DIAMONDS, HEARTS, SPADES };
public static final List VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

W przeciwieństwie do najprostszej postaci typu wyliczeniowego klasy w wersji korzystającej z numerów kolejnych mogą być serializowane (rozdział 9.) z zachowaniem szczególnej ostrożności. Nie wystarczy dodać do deklaracji klasy klauzuli `implements Serializable`. Wymagane jest również utworzenie metody `readResolve` (temat 57.).

```
private Object readResolve() throws ObjectStreamException {
    return PRIVATE_VALUES[ordinal]; // Zmiana na postać kanoniczną
}
```

Metoda ta, wywołana automatycznie przez system serializacji, zabezpiecza przed powielaniem istniejących stałych, powstałych w procesie deserializacji. Zapewnia ona, że będzie istniał tylko jeden obiekt dla każdej stałej typu wyliczeniowego, co pozwala uniknąć konieczności przedefiniowywania metody `Object.equals`. Bez tej gwarancji metoda `Object.equals` będzie zwracała nieprawidłowe wyniki, jeżeli będzie porównywała dwie równe stałe, ale o różnych kolejnych numerach. Zwróć uwagę, że metoda `readResolve` korzysta z tablicy `PRIVATE_VALUES`, więc musisz zadeklarować tę tablicę, nawet jeżeli nie chcesz udostępniać kolekcji `VALUES`. Należy również zwrócić uwagę, że pole `name` nie jest wykorzystywane przez metodę `readResolve`, więc jest ono nietrwałe i takie powinno pozostać.

Wynikowa klasa jest jednak krucha — konstruktor dla dowolnej nowej wartości musi występować po wszystkich istniejących wartościach, dzięki czemu zapewniamy, że wszystkie serializowane wcześniej obiekty nie zmienią swoich wartości w trakcie deserializacji. Dzieje się tak, ponieważ postać serializowana stałych (temat 55.) opiera się jedynie na ich kolejnych numerach. Jeżeli stała, będąca składnikiem typu wyliczeniowego, zmieni swój numer kolejny, stała o tym numerze poddana serializacji zmieni swoją wartość podczas procesu deserializacji.

Może istnieć jedna lub więcej operacji skojarzonych z każdą ze stałych, które są wykorzystywane jedynie wewnątrz pakietu, zawierającego klasę realizującą typ wyliczeniowy. Operacje takie najlepiej implementować jako metody prywatne w ramach pakietu. Każda stała typu wyliczeniowego zawiera ukrytą kolekcję operacji, pozwalających reagować odpowiednio dla odpowiednich stałych.

Jeżeli klasa implementująca typ wyliczeniowy posiada metody, których działanie znacznie się różni w zależności od wartości stałej, powinieneś skorzystać z osobnych klas prywatnych lub anonimowych klas, zagnieżdżonych dla każdej ze stałych. Pozwala to każdej ze stałych posiadać własną implementację danej metody i automatycznie wywoływać odpowiednią implementację. Alternatywą jest tworzenie wielościeżkowych rozgałęzień, które wybierają odpowiednią metodę w zależności od stałej, na rzecz której wywołujemy tę metodę. Jest to rozwiązanie niechlujne, podatne na błędy i często wydajność tego rozwiązania jest niższa od rozwiązania, korzystającego z automatycznego wybierania metod przez maszynę wirtualną.

Obie techniki opisane w poprzednim akapicie są zilustrowane jeszcze jedną klasą, realizującą typ wyliczeniowy. Klasa ta, `Operation`, reprezentuje działania wykonywane przez prosty kalkulator czterodziałaniowy. Poza pakietem, w którym zdefiniowana jest ta klasa, można skorzystać ze stałych klasy `Operation` do wywołania metod klasy `Object` (`toString`, `hashCode`, `equals` itd.). Wewnątrz pakietu można dodatkowo wykonywać operacje matematyczne związane ze stałymi. Przypuszczalnie pakiet może eksportować obiekt kalkulatora udostępniający jedną lub więcej metod, które jako parametrów oczekują stałych klasy `Operation`. Zwróć uwagę, że sama klasa `Operation` jest klasą abstrakcyjną, zawierającą jedną prywatną w ramach pakietu metodę abstrakcyjną — `eval`, która wykonuje odpowiednią operację matematyczną. Dla każdej stałej zdefiniowana jest wewnętrzna klasa anonimowa, więc każda stała może zdefiniować własną wersję metody `eval`.

```
// Typ wyliczeniowy z operacjami skojarzonymi ze stałymi
public abstract class Operation {
    private final String name;

    Operation(String name)   { this.name = name; }

    public String toString() { return this.name; }

    // Wykonaj operację matematyczną dla bieżącego kontekstu
    abstract double eval(double x, double y);

    public static final Operation PLUS = new Operation("+") {
        double eval(double x, double y) { return x + y; }
    };

    public static final Operation MINUS = new Operation("-") {
        double eval(double x, double y) { return x - y; }
    };

    public static final Operation TIMES = new Operation("*") {
        double eval(double x, double y) { return x * y; }
    };

    public static final Operation DIVIDED_BY =
        new Operation("/") {
            double eval(double x, double y) { return x / y; }
    };
}
```

Przedstawiony typ wyliczeniowy ma wydajność porównywalną do wydajności klasy korzystającej ze stałych wyliczeniowych typu `int`. Dwa różne obiekty klasy reprezentującej typ wyliczeniowy nigdy nie reprezentują tej samej wartości, więc porównanie referencji, które jest bardzo szybkie, wystarczy do sprawdzenia równości logicznej. Klienci klasy mogą nawet użyć operatora == zamiast metody `equals` — wynik będzie identyczny, a operator == może nawet działać szybciej.

Jeżeli klasa typu wyliczeniowego jest przydatna, może być klasą najwyższego poziomu — jeżeli jest związana z inną klasą najwyższego poziomu, powinna być statyczną klasą zagnieżdżoną tej klasy (temat 18.). Na przykład klasa `java.math.BigDecimal` zawiera zbiór stałych wyliczeniowych typu `int`, określających *tryby zaokrąglania*

części dziesiętnych. Te tryby zaokrąglania stanowią użyteczny model abstrakcji, który nie jest zasadniczo przywiązany do klasy `BigDecimal` — byłoby lepiej utworzyć osobną klasę `java.math.RoundingModes`. Udostępnienie takiej klasy wszystkim programistom korzystającym z trybów zaokrągleń pozwoliłoby na zwiększenie spójności między różnymi API.

Podstawowa implementacja wzorca bezpiecznego typu wyliczeniowego, zilustrowana za pomocą dwóch implementacji klasy `Suit`, jest *zamknięta*. Użytkownicy nie mogą dodawać nowych elementów typu wyliczeniowego, ponieważ klasa nie udostępnia im konstruktora. Powoduje to, że klasa zachowuje się tak, jakby została zdefiniowana jako `final`. Najczęściej jest to najlepsze rozwiązanie, ale istnieją przypadki, w których chcemy utworzyć *rozszerzalną* klasę typu wyliczeniowego. Może być to potrzebne na przykład do reprezentowania formatów kodowania rysunków, gdy chcesz, aby inni programiści mogli umożliwiać obsługę nowych formatów.

Aby umożliwić rozszerzanie typu wyliczeniowego, wystarczy udostępnić zabezpieczony konstruktor. Użytkownicy będą mogli dzięki temu dziedziczyć po istniejącej klasie, dodając w podklasach własne stałe. Nie musisz się obawiać, że stałe typu wyliczeniowego spowodują konflikt, tak jak było to w przypadku typu wyliczeniowego, korzystającego ze stałych `int`. Rozszerzalny wariant wzorca bezpiecznego typu wyliczeniowego korzysta z przestrzeni nazw pakietu do tworzenia „magicznie administrowanych" przestrzeni nazw dla rozszerzalnych wyliczeń. Różne zespoły mogą niezależnie rozszerzać wyliczenia i nie pozostaną one nigdy w konflikcie.

Dodanie elementu do rozszerzalnego typu wyliczeniowego nie gwarantuje jeszcze, że nowe elementy będą w pełni obsługiwane. Metody operujące na elementach typu wyliczeniowego muszą przewidywać ewentualność przekazania elementu nieznanego programiście. Wielokrotne rozgałęzienia są dyskusyjne w przypadku zamkniętego typu wyliczeniowego, a w przypadku typu rozszerzalnego są nie do przyjęcia, ponieważ nie mogą samoczynnie dodawać nowej gałęzi dla typu, dodanego przez programistę rozszerzającego klasę.

Jedynym sposobem radzenia sobie z tym problemem jest wyposażenie klasy bezpiecznego typu wyliczeniowego we wszystkie metody niezbędne do opisania zachowania się stałych tej klasy. Metody niezbyt użyteczne dla klientów powinny być zadeklarowane jako `protected` w celu ich ukrycia, natomiast klasy pochodne mogą je przedefiniować. Jeżeli metoda taka nie ma rozsądnego działania domyślnego, oprócz `protected` powinna być również zadeklarowana jako `abstract`.

Dla rozszerzalnych klas bezpiecznego typu wyliczeniowego dobrze jest przedefiniować metody `equals` i `hashCode` jako metody finalne, wywołujące odpowiednie metody z klasy `Object`. Zapewni to, że żadna podklasa przypadkowo nie przedefiniuje tych metod, co zagwarantuje, że wszystkie równe obiekty są również identyczne (`a.equals(b)` tylko wtedy, gdy `a==b`):

```
// Metody zabezpieczające przed przesłanianiem
public final boolean equals(Object that) {
    return super.equals(that);
}
```

```
public final int hashCode() {
    return super.hashCode();
}
```

Trzeba pamiętać, że wariant rozszerzalny nie jest zgodny z wariantem implementującym interfejs `Comparable` — jeżeli będziesz próbował je ze sobą połączyć, porządkowanie elementów w podklasach będzie działało w porządku inicjalizacji podklas, co może się zmieniać w różnych programach i w różnych wywołaniach.

Rozszerzalny wariant wzorca bezpiecznego typu wyliczeniowego jest zgodny z wariantem implementującym interfejs `Serializable`, ale łączenie tych wariantów wymaga nieco uwagi. Każda podklasa musi przypisywać własne numery kolejne i napisać własną metodę `readResolve`. Zasadniczo każda klasa jest odpowiedzialna za serializację i deserializację swoich obiektów. Przedstawiamy teraz kolejną wersję klasy `Operation`, która jest rozszerzalna i może być serializowana.

```
// Serializowalny rozszerzalny bezpieczny typ wyliczeniowy
public abstract class Operation implements Serializable {
    private final transient String name;
    protected Operation(String name) { this.name = name; }

    public static Operation PLUS = new Operation("+") {
        protected double eval(double x, double y) { return x+y; }
    };
    public static Operation MINUS = new Operation("-") {
        protected double eval(double x, double y) { return x-y; }
    };
    public static Operation TIMES = new Operation("*") {
        protected double eval(double x, double y) { return x*y; }
    };
    public static Operation DIVIDE = new Operation("/") {
        protected double eval(double x, double y) { return x/y; }
    };

    // Wykonaj operację matematyczną dla bieżącego kontekstu
    protected abstract double eval(double x, double y);

    public String toString() { return this.name; }
    // Zablokowanie możliwości przedefiniowywania Object.equals przez klasy
    ↳pochodne
    public final boolean equals(Object that) {
        return super.equals(that);
    }
    public final int hashCode() {
        return super.hashCode();
    }

    // 4 poniższe deklaracje są niezbędne dla serializacji
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES = { PLUS, MINUS, TIMES, DIVIDE };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal];  // Zmiana na postać kanoniczną
    }
}
```

Przedstawimy również podklasę klasy `Operation`, która dodaje operacje logarytmowania i podnoszenia do potęgi. Ta klasa pochodna może być zdefiniowana poza pakietem zawierającym klasę bazową. Powinna być ona publiczna i umożliwiać dziedziczenie. Możliwe jest utworzenie wielu niezależnych podklas, współistniejących bez żadnych konfliktów.

```
// Podklasa serializowalnego rozszerzalnego bezpiecznego typu wyliczeniowego
abstract class ExtendedOperation extends Operation {
    ExtendedOperation(String name) { super(name); }

    public static Operation LOG = new ExtendedOperation("log") {
        protected double eval(double x, double y) {
            return Math.log(y) / Math.log(x);
        }
    };
    public static Operation EXP = new ExtendedOperation("exp") {
        protected double eval(double x, double y) {
            return Math.pow(x, y);
        }
    };

    // 4 poniższe deklaracje są niezbędne dla serializacji
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES = { LOG, EXP };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal];  // Zmiana na postać kanoniczną
    }
}
```

Zwróć uwagę, że metoda `readResolve` w przedstawionej klasie nie jest prywatna, a tylko prywatna w ramach pakietu. Jest to niezbędne, ponieważ obiekty `Operation` i `ExtendedOperation` są właściwie obiektami podklas anonimowych, więc prywatna metoda `readResolve` nie może zostać zastosowana (temat 57.).

Wzorzec bezpiecznego typu wyliczeniowego posiada jednak nieco więcej wad w porównaniu do typu wyliczeniowego, korzystającego z wartości `int`. Prawdopodobnie jedyną poważną niedogodnością jest niewygodne łączenie bezpiecznych stałych wyliczeniowych w zbiory. W przypadku typu wyliczeniowego, korzystającego z wartości `int`, jest to zwykle realizowane przez nadanie stałym wyliczeniowym wartości będących potęgami dwójki i reprezentowanie zbioru jako bitowej sumy logicznej odpowiednich stałych:

```
// Odmiana typu wyliczeniowego korzystającego z bitowych wartości liczb int
public static final int SUIT_CLUBS    = 1;
public static final int SUIT_DIAMONDS = 2;
public static final int SUIT_HEARTS   = 4;
public static final int SUIT_SPADES   = 8;

public static final int SUIT_BLACK = SUIT_CLUBS | SUIT_SPADES;
```

Reprezentowanie w ten sposób zbiorów stałych typu wyliczeniowego jest zwięzłe i niezmiernie szybkie. W przypadku zbiorów stałych bezpiecznego typu wyliczeniowego możesz skorzystać z implementacji zbioru z biblioteki `Collections`, ale nie jest to tak zwięzłe i szybkie.

```
Set blackSuits = new HashSet();
blackSuits.add(Suit.CLUBS);
blackSUITS.add(SUIT.SPADES);
```

Choć zbiory stałych prawdopodobnie nie mogą być zrealizowane ani tak zwięźle, ani tak szybko, jak zbiory stałych wyliczeniowych typu `int`, możliwe jest zmniejszenie tej różnicy przez utworzenie specjalnej implementacji klasy `Set`, która akceptuje jedynie elementy jednego typu i wewnętrznie reprezentuje zbiór jako wektor bitów. Taki zbiór najlepiej zdefiniować w tym samym pakiecie, co klasa reprezentująca przechowywane elementy, co umożliwia dostęp do pól lub metod prywatnych w ramach pakietu. Rozsądne jest utworzenie publicznych konstruktorów, które jako parametry akceptują krótkie sekwencje elementów, dzięki czemu możliwe jest tworzenie następujących wyrażeń:

```
hand.iscard(new SuitSet(Suit.CLUBS, Suit.SPADES));
```

Mniejszą niedogodnością bezpiecznego typu wyliczeniowego, w porównaniu do typu korzystającego z typu `int`, jest to, że bezpieczny typ wyliczeniowy nie może być wykorzystany w instrukcji `switch`, ponieważ nie są to wartości całkowite. Zamiast tego należy skorzystać z instrukcji `if`:

```
if (suit == Suit.CLUBS) {
    ...
} else if (suit == Suit.DIAMONDS) {
    ...
} else if (suit == Suit.HEARTS) {
    ...
} else if (suit == Suit.SPADES) {
    ...
} else {
    throw new NullPointerException("Pusty kolor"); // suit == null
}
```

Instrukcja `if` może nie wykonywać się tak szybko jako `switch`, ale różnica nie powinna być zauważalna. Tworzenie wielokrotnych rozgałęzień w przypadku bezpiecznego typu wyliczeniowego powinno być rzadkie, ponieważ rozgałęzianie kodu należy zastępować automatycznym wyborem metod przez maszynę wirtualną, tak jak w przytoczonej klasie `Operation`.

Kolejny niewielki problem wiąże się ze spadkiem wydajności, ponieważ potrzeba czasu i miejsca w pamięci na załadowanie klasy typu wyliczeniowego i utworzenie obiektów stałych. Problem ten nie powinien być zauważalny w praktyce, poza niektórymi urządzeniami o ograniczonych zasobach, jak na przykład telefony komórkowe czy tostery.

Podsumujmy. Przewagi bezpiecznego typu wyliczeniowego nad typem korzystającym z wartości `int` są bezapelacyjne i żadna z wad nie może być przyczyną zaprzestania korzystania z niego, chyba że jest on głównie używany jako element zbioru lub w środowisku o ograniczonych zasobach. Dlatego **bezpieczny typ wyliczeniowy powinien być jako pierwszy brany pod uwagę, jeśli konieczne jest zastosowania typu wyliczeniowego**. API korzystające z bezpiecznego typu wyliczeniowego jest dużo łatwiejsze w użyciu dla programistów niż to, które korzysta z typu `int`. Jedynym powodem, dla którego bezpieczny typ wyliczeniowy nie jest intensywniej wykorzystywany w bibliotekach platformy Java, jest fakt, że wzorzec ten nie był jeszcze znany w czasie tworzenia

większości klas tego API. Na koniec należy przypomnieć, że stosowanie typu wyliczeniowego powinna być dosyć rzadkie, ponieważ w większości zastosowań tego typu z powodzeniem można skorzystać z dziedziczenia (temat 20.).

Temat 22. Zastępowanie wskaźników do funkcji za pomocą klas i interfejsów

Język C pozwala na stosowanie *wskaźników do funkcji*, które umożliwiają zapamiętywanie i przesyłanie odwołań do określonych funkcji. Wskaźniki do funkcji są najczęściej wykorzystywane w celu umożliwienia funkcji wywołującej modyfikacji swojego działania poprzez przekazanie wskaźnika do drugiej funkcji, czasami nazywanej *funkcją wywołania zwrotnego*. Na przykład, funkcja `qsort` ze standardowej biblioteki języka C wymaga jako parametru wskaźnika do funkcji, której zadaniem jest *porównywanie* elementów. Funkcja porównująca wymaga podania dwóch parametrów — wskaźników do elementów. Zwraca ona ujemną liczbę całkowitą, jeżeli element wskazywany przez pierwszy element jest mniejszy od drugiego, zero, gdy elementy są równe i dodatnią liczbę całkowitą, jeżeli pierwszy element jest większy od drugiego. Można uzyskać różny porządek sortowania, przekazując inną funkcję porównującą. Jest to przykład wzorca *Strategy* (strategia) [Gamma95, str. 315] — funkcja porównująca określa strategię sortowania elementów.

Wskaźniki do funkcji nie znalazły się w języku Java, ponieważ korzystając z referencji do obiektów, można uzyskać takie samo działanie. Wywołując metodę obiektu, zwykle wykonuje się operacje na *tym obiekcie*. Można również zdefiniować obiekt, którego metody wykonują operacje na *innym obiekcie*, przekazanym jawnie do metody. Obiekt klasy udostępniającej tylko jedną taką metodę jest, efektywnie, wskaźnikiem do tej metody. Obiekty takie są zwane *obiektami funkcjonalnymi*. Przyjrzyjmy się takiej właśnie klasie:

```
class StringLengthComparator {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
}
```

Klasa ta udostępnia jedną metodę, która oczekuje dwóch ciągów znaków jako parametrów. Zwraca ona liczbę ujemną w przypadku, gdy pierwszy ciąg jest krótszy od drugiego, zero, gdy ciągi są równe i liczbę dodatnią, gdy pierwszy ciąg jest dłuższy od drugiego. Metoda ta jest *komparatorem*, porównującym długości ciągów. Referencja do obiektu `StringLengthComparator` służy jako „wskaźnik do funkcji”, umożliwiając wywołanie metody dla dwóch podanych ciągów.

Bezstanowość klasy `StringLengthComparator` jest typowa dla klas strategii — klasa ta nie posiada pól, przez co wszystkie jej obiekty są funkcjonalnie identyczne. Aby uniknąć tworzenia niepotrzebnych obiektów, możemy utworzyć tę klasę jako klasę typu *singleton* (temat 4., temat 2.).

```
class StringLengthComparator {
  private StringLengthComparator() {}

  public static final StringLengthComparator
    INSTANCE = new StringLengthComparator();

  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
}
```

Aby przekazać obiekt `StringLengthComparator` do metody, potrzebujemy odpowiedniego typu dla parametrów. Skorzystanie z typu `StringLengthComparator` nie będzie odpowiednie, ponieważ klienci nie będą mogli przekazywać innych strategii porównywania. Można natomiast zdefiniować interfejs `Comparator` i zmienić klasę `StringLengthComparator`, aby implementowała ten interfejs. Inaczej mówiąc, definiujemy *interfejs strategii*, implementowany przez konkretne klasy strategii:

```
// Interfejs strategii
public interfeace Comparator {
  public int compare(Object o1, Object o2);
}
```

Ta definicja interfejsu `Comparator` znajduje się w pakiecie `java.util`, ale równie dobrze możesz ją utworzyć samemu. W przypadku obiektów innych niż ciągi możliwe jest, że ich metody `compare` będą oczekiwały parametrów typu `Object`, a nie `String`. Dlatego w celu prawidłowej implementacji interfejsu `Comparator` przytoczona wcześniej klasa `StringLengthComparator` musi być nieco zmodyfikowana. W celu wywołania metody `length` parametry typu `Object` muszą być rzutowane na `String`.

Klasy strategii są często definiowane jako klasy anonimowe (temat 18.). Poniższe wyrażenie powoduje posortowanie tablicy ciągów według ich długości.

```
Arrays.sort( stringArray, new Comparator() {
  public int compare( Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.length() - s2.length();
  }
});
```

Ponieważ interfejsy strategii służą jako typy dla wszystkich obiektów strategii, w celu udostępnienia strategii jej klasa nie musi być publiczna. Zamiast tego „klasa główna" może udostępniać publiczne statyczne pole (lub statyczną metodę *factory*) typu określanego przez interfejs strategii, a klasa strategii może być prywatną klasą składową klasy głównej. W poniższym przykładzie pokazujemy zastosowanie statycznej klasy zagnieżdżonej, tworzącej klasę strategii do implementacji drugiego interfejsu, `Serializable`.

```
// Udostępnianie strategii
class Host {
  //... większość klasy pominięta

  private static class StrLenCmp
    implements Comparator, Serializable {
```

```
        public int compare( Object o1, Object o2) {
          String s1 = (String)o1;
          String s2 = (String)o2;
          return s1.length() - s2.length();
        }
      }

      // Zwracany komparator można serializować
      public static final Comparator
          STRING_LENGTH_COMPARATOR = new StrLenCmp();
  }
```

Klasa `Host` korzysta z tego wzorca do udostępnienia za pomocą pola `STRING_LENGTH_COMPARATOR` komparatora porównującego długości ciągów.

Podsumujmy. Podstawowym zastosowaniem wskaźników do funkcji jest utworzenie wzorca `Strategia`. W języku Java wzorzec ten można utworzyć, deklarując interfejs reprezentujący strategię i klasę implementującą interfejs dla konkretnej strategii. Gdy strategia jest zastosowana tylko raz, jej klasa deklarowana jest najczęściej jako klasa anonimowa. Gdy strategia jest udostępniana, jej klasa jest zwykle prywatną statyczną klasą zagnieżdżoną i jest udostępniana poprzez publiczne statyczne pole `final`, którego typ jest zgodny z interfejsem strategii.