Joshua Bloch

Java

EFEKTYWNE PROGRAMOWANIE

Wydanie III

Helion

Tytuł oryginału: Effective Java (3rd Edition)

Tłumaczenie: Rafał Jońca

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-9896-2

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jave3v
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

6

Typy wyliczeniowe i adnotacje

Java obsługuje dwie rodziny specjalnych typów referencyjnych — pewien rodzaj klas nazwany **typem wyliczeniowym** oraz pewien rodzaj interfejsu nazwany **typem adnotacyjnym**. W tym rozdziale przedstawimy najlepsze praktyki związane z wykorzystywaniem obu rodzin typów.

Temat 34. Użycie typów wyliczeniowych zamiast stałych int

Typ wyliczeniowy to typ, którego prawidłowe wartości tworzy stały zbiór, na przykład pory roku, planety w systemie słonecznym lub rodzaje kart w talii. Zanim do języka został dodany typ wyliczeniowy, standardową praktyką reprezentowania takich typów było deklarowanie grupy zmiennych typu `int`, po jednej dla każdej wartości typu.

```
// Wzorzec typu wyliczeniowego z użyciem int - bardzo niedoskonały!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

Technika ta, znana pod nazwą **wzorca wyliczeniowego `int`**, ma wiele wad. Nie zapewnia ona żadnej formy bezpieczeństwa typów i jest mało wygodna. Kompilator nie będzie protestował, jeżeli przekażemy jabłko do metody oczekującej pomarańczy, porównamy jabłko z pomarańczą za pomocą operatora == lub co gorsza:

```
// Smaczny sok jabłkowy z domieszką cytrusów!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Jak można zauważyć, nazwa każdej ze stałych dla jabłka zaczyna się od `APPLE_`, a nazwa każdej stałej pomarańczy zaczyna się od `ORANGE_`. Dzieje się tak, ponieważ Java nie zapewnia przestrzeni nazw dla grup wartości wyliczeniowych typu `int`. Prefiksy zapobiegają powtórzeniu nazw, gdy dwie grupy wyliczeniowe mają tak samo nazwane stałe, co pozwala odróżnić od siebie `ELEMENT_MERCURY` i `PLANET_MERCURY`.

Programy korzystające z wzorca wyliczeniowego `int` są niewygodne. Ponieważ stałe wyliczeniowe `int` są **zmiennymi stałymi** [JLS, 4.12.4], są kompilowane do programów klienckich, które je wykorzystują [JLS, 13.1]. Jeżeli wartość `int` związana ze stałą wyliczeniową zmieni się, klient musi być ponownie skompilowany. W przeciwnym razie będzie nadal działał, ale jego zachowanie będzie niezdefiniowane.

Nie istnieje prosty sposób przetłumaczenia stałej wyliczeniowej na napis do wyświetlenia. Jeżeli drukujemy taką stałą lub wyświetlamy ją w debugerze, widzimy tylko liczbę, co nie jest zbyt pomocne. Nie istnieje niezawodny sposób na iterowanie po wszystkich wartościach `int` w typie wyliczeniowym, a nawet określenie wielkości grupy stałych `int`.

Można również spotkać odmianę tego wzorca, w którym w miejsce stałych `int` są używane stałe `String`. Wariant ten, nazywany **wzorcem wyliczeniowym `String`**, jest nawet mniej polecany. Choć zapewnia on czytelne napisy dla stałych, może prowadzić do problemów z wydajnością, ponieważ korzysta z porównywania napisów. Co gorsza, może doprowadzić do tego, że niedoświadczeni użytkownicy na stałe wpiszą stałe znakowe do kodu klienta, zamiast korzystać z nazw pól. Jeżeli taki wpisany napis zawiera błąd typograficzny, wymknie się spod kontroli w czasie kompilacji i spowoduje błąd w czasie działania aplikacji.

Na szczęście Java zapewnia alternatywę pozwalającą na ominięcie wad wzorców wyliczeniowych `int` i `String`, dającą wiele nowych korzyści. Typ ten jest zdefiniowany w podręczniku jako **typ `enum`** [JLS, 8.9]. W najprostszej postaci wygląda następująco:

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

Na pierwszy rzut oka typ wyliczeniowy może wyglądać podobnie do typów znanych z innych języków, takich jak C, C++ i C#, ale podobieństwo jest mylące. Typ wyliczeniowy w języku Java jest w pełni wartościową klasą, znacznie bardziej zaawansowaną niż odpowiedniki w innych językach, w których typ wyliczeniowy jest właściwie zbiorem wartości `int`.

Podstawowe założenie leżące u podstaw typu wyliczeniowego w języku Java jest proste — są to klasy eksportujące po jednym obiekcie dla każdej stałej wyliczenia z użyciem finalnego statycznego pola publicznego. Typy wyliczeniowe są w efekcie finalne, dzięki temu, że nie jest dostępny konstruktor. Ponieważ klienty nigdy nie tworzą obiektów typu wyliczeniowego ani po nich nie dziedziczą, nie mogą istnieć instancje inne niż zadeklarowane stałe wyliczeniowe. Inaczej mówiąc, typy wyliczeniowe są kontrolowanymi instancjami (temat 1.). Są one generalizacją singletonów (temat 3.), które są w praktyce jednoelementowymi typami wyliczeniowymi.

Typ wyliczeniowy zapewnia bezpieczeństwo typów w czasie kompilacji. Jeżeli zadeklarujemy, że parametr będzie typu `Apple`, to gwarantowane jest, że przekazana do parametru referencja obiektu różna od `null` będzie jedną z trzech prawidłowych wartości typu `Apple`. Próba przekazania wartości niewłaściwego typu spowoduje błąd kompilacji, tak samo jak próba przypisania wyrażenia jednego typu wyliczeniowego do zmiennej innego albo użycia operatora == do porównywania wartości różnych typów wyliczeniowych.

Typy wyliczeniowe o identycznie nazwanych stałych mogą bez problemów współistnieć, ponieważ każdy typ posiada własną przestrzeń nazw. Można dodawać lub zmieniać kolejność stałych w typie wyliczeniowym bez konieczności ponownej kompilacji klientów, ponieważ pola eksportujące stałe zapewniają warstwę izolacyjną pomiędzy typem wyliczeniowym i jego klientami — stałe nie są wkompilowywane w klienty, tak jak miało to miejsce w przypadku wzorca typu wyliczeniowego `int`. Można również tłumaczyć wartości typu wyliczeniowego na czytelne napisy przez wywoływanie ich metody `toString`.

Oprócz naprawiania niedociągnięć typu wyliczeniowego korzystającego z `int` nowe typy wyliczeniowe pozwalają dopisać dowolne metody i implementować dowolne interfejsy. Zapewniają wysokiej jakości implementacje wszystkich metod klasy `Object` (rozdział 3.), implementują `Comparable` (temat 14.) i `Serializable` (rozdział 12.), a ich serializowana postać jest zaprojektowana tak, aby przetrwać większość zmian typu wyliczeniowego.

Do czego jest potrzebna możliwość dodawania do typu wyliczeniowego metod i pól? Przykładem może być możliwość skojarzenia danych ze stałymi. Nasze typy `Apple` i `Orange` mogą na przykład zawierać metodę zwracającą kolor owocu lub nawet jego zdjęcie. Można wzbogacać typ wyliczeniowy o dowolną metodę, która wydaje się nam odpowiednia. Typ wyliczeniowy może rozpocząć życie jako prosta kolekcja stałych wyliczeniowych i przeobrazić się w rozbudowaną abstrakcję.

Dobrym przykładem takiego bogatego typu wyliczeniowego może być typ modelujący osiem planet naszego systemu słonecznego. Każda planeta ma masę i promień, a na podstawie tych dwóch parametrów można obliczyć grawitację na powierzchni.

To z kolei pozwala obliczyć ciężar obiektu na powierzchni planety na podstawie masy obiektu. Poniżej zamieszczony jest przykład takiego typu wyliczeniowego. Liczby w nawiasach po stałych wyliczeniowych są parametrami przekazywanymi do konstruktora. W tym przypadku jest to masa i promień planety:

```
// Typ wyliczeniowy z danymi i operacjami
public enum Planet {
   MERCURY(3.302e+23, 2.439e6),
   VENUS  (4.869e+24, 6.052e6),
   EARTH  (5.975e+24, 6.378e6),
   MARS   (6.419e+23, 3.393e6),
   JUPITER(1.899e+27, 7.149e7),
   SATURN (5.685e+26, 6.027e7),
   URANUS (8.683e+25, 2.556e7),
   NEPTUNE(1.024e+26, 2.477e7);

   private final double mass; // w kilogramach
   private final double radius; // w metrach
   private final double surfaceGravity; // w m / s^2

   // uniwersalna stała grawitacyjna w m^3 / kg s^2
   private static final double G = 6.67300E-11;

   // konstruktor
   Planet(double mass, double radius) {
      this.mass = mass;
      this.radius = radius;
      surfaceGravity = G * mass / (radius * radius);
   }

   public double mass() { return mass; }
   public double radius() { return radius; }
   public double surfaceGravity() { return surfaceGravity; }

   public double surfaceWeight(double mass) {
      return mass * surfaceGravity; // F = ma
   }
}
```

Jak widać, bardzo łatwo jest napisać bogaty typ wyliczeniowy, taki jak `Planet`. **Aby skojarzyć dane ze stałymi wyliczenia, należy zadeklarować pola instancyjne i napisać konstruktor pobierający dane i zapisujący je w polach**. Typy wyliczeniowe są z natury niezmienne, więc wszystkie pola powinny być oznaczone `final` (temat 17.). Mogą być one publiczne, ale znacznie lepiej zadeklarować je jako prywatne i udostępnić publiczne akcesory (temat 16.). W przypadku typu `Planet` konstruktor oblicza i zapisuje grawitację na powierzchni, ale jest to tylko optymalizacja. Grawitacja może być wyliczana na podstawie masy i promienia przy każdym wywołaniu metody `surfaceWeight`, który pobiera masę obiektu i zwraca jego ciężar na planecie reprezentowanej przez stałą.

Choć typ wyliczeniowy `Planet` jest prosty, daje on zaskakująco dużo możliwości. Poniżej znajduje się krótki program pobierający wagę obiektu na Ziemi (w dowolnej jednostce) i wyświetla elegancką tabelę wag obiektu na wszystkich ośmiu planetach (w tej samej jednostce):

```
public class WeightTable {
   public static void main(String[] args) {
      double earthWeight = Double.parseDouble(args[0]);
      double mass = earthWeight / Planet.EARTH.surfaceGravity();
      for (Planet p : Planet.values())
         System.out.printf("Waga na %s wynosi %f%n",
                           p, p.surfaceWeight(mass));
   }
}
```

Należy zwrócić uwagę, że `Planet`, podobnie jak inne typy wyliczeniowe, posiada metodę statyczną `values`, która zwraca tablicę wartości typu w kolejności ich zadeklarowania. Należy również zwrócić uwagę, że metoda `toString` zwraca zadeklarowaną nazwę każdej z wartości typu wyliczeniowego, co pozwala w łatwy sposób drukować ją za pomocą `println` i `printf`. Jeżeli nie jesteśmy usatysfakcjonowani reprezentacją tekstową stałej, można ją zmienić przez nadpisanie metody `toString`. Poniżej zamieszczony jest wynik działania naszego małego programu `WeightTable` (wyliczenie nie przysłania metody `toString`) z przekazaną w wierszu poleceń wartością 185:

```
Waga na MERCURY wynosi 69.912739
Waga na VENUS wynosi 167.434436
Waga na EARTH wynosi 185.000000
Waga na MARS wynosi 70.226739
Waga na JUPITER wynosi 467.990696
Waga na SATURN wynosi 197.120111
Waga na URANUS wynosi 167.398264
Waga na NEPTUNE wynosi 210.208751
```

Aż do roku 2006, czyli dwa lata po dodaniu typu wyliczeniowego do Javy, Pluton był planetą. Warto w tym momencie zadać sobie pytanie, co się stanie, jeśli usuniemy element z typu wyliczeniowego. Dowolny klient, który nie korzystał bezpośrednio z usuniętego elementu, nadal będzie działał prawidłowo. Nasz przykładowy program `WeightTable` nadal działałby prawidłowo — po prostu wyświetlałby jeden wiersz mniej. Co stanie się z programem, który jawnie korzystał z usuniętego elementu (w tym przypadku `Planet.PLUTO`)? Jeśli program będzie kompilowany ponownie, proces ten się nie powiedzie i zostanie wyświetlony komunikat o błędzie wskazujący na odniesienie do zdegradowanej planety. Jeśli nie dojdzie do ponownej kompilacji, program wyświetli podobny komunikat w trakcie działania. To najlepsze zachowanie, na jakie można liczyć, znacznie przyjemniejsze od tego, które dostarczyłby wzorzec wyliczenia `int`.

Niektóre operacje związane ze stałymi wyliczanymi mogą być używane wyłącznie w klasie lub pakiecie, w którym jest zdefiniowany typ wyliczeniowy. Takie operacje najlepiej zaimplementować jako metody prywatne lub prywatne w ramach pakietu. Każda ze stałych posiada wtedy ukrytą kolekcję operacji, które pozwalają na odpowiednie reagowanie w klasie lub pakiecie w przypadku manipulacji na stałych. Tak samo jak w przypadku innych klas, o ile nie mamy ważnej potrzeby udostępniania metod klientom, należy zadeklarować je jako prywatne lub, jeżeli jest to wymagane, prywatne w ramach pakietu (temat 15.).

Jeżeli typ wyliczeniowy jest ogólnie przydatny, powinien być klasą najwyższego poziomu, natomiast jeżeli jest używany w klasie najwyższego poziomu, powinien być jej składnikiem (temat 24.). Na przykład typ wyliczeniowy `java.math.RoundingMode` reprezentuje tryb zaokrąglania ułamków dziesiętnych. Tryby zaokrąglania są wykorzystywane w klasie `BigDecimal`, ale zapewniają one przydatną abstrakcję, która nie jest fundamentalnie związana z `BigDecimal`. Przez umieszczenie typu wyliczeniowego `RoundingMode` na najwyższym poziomie projektanci biblioteki zachęcają programistów potrzebujących trybu zaokrąglania do ponownego użycia tego typu, dzięki czemu uzyskuje się większą spójność API.

Techniki zademonstrowane w przykładzie `Planet` są wystarczające dla większości typów wyliczeniowych, ale czasami potrzeba więcej. Z każdą stałą `Planet` są związane różne dane, ale czasami potrzebujemy skojarzyć całkowicie inne **działanie** z każdą ze stałych. Załóżmy, że chcemy napisać typ wyliczeniowy reprezentujący operacje na prostym kalkulatorze czterodziałaniowym i chcemy zapewnić metodę do wykonania operacji arytmetycznych reprezentowanych przez każdą ze stałych. Jednym ze sposobów realizacji jest wybór wyrażenia w zależności od wartości wyliczenia:

```
// Typ wyliczeniowy z wyborem spośród własnych wartości - niedoskonały
public enum Operation {
   PLUS, MINUS, TIMES, DIVIDE;

   // wykonanie operacji arytmetycznej reprezentowanej przez stałą
   public double apply(double x, double y) {
      switch(this) {
         case PLUS: return x + y;
         case MINUS: return x - y;
         case TIMES: return x * y;
         case DIVIDE: return x / y;
      }
      throw new AssertionError("Nieznana operacja: " + this);
   }
}
```

Kod ten działa, ale nie jest zbyt ładny. Nie kompiluje się on bez instrukcji `throw`, ponieważ koniec metody jest technicznie osiągalny, nawet jeżeli nie zostanie nigdy osiągnięty [JLS, 14.21]. Co gorsza, kod ten jest delikatny. Jeżeli dodamy nową

stałą wyliczeniową, a zapomnimy dodać odpowiedni przypadek w `switch`, typ nadal będzie się kompilował, ale wystąpi błąd w czasie działania przy próbie wykonania nowej operacji.

Na szczęście istnieje lepszy sposób na kojarzenie różnych operacji z każdą stałą wyliczenia — można zadeklarować abstrakcyjną metodę `apply` w typie wyliczeniowym i nadpisywać ją w konkretnej metodzie dla stałej umieszczonej w **treści klasy specyficznej dla stałej**. Takie metody są znane jako **implementacje metod specyficznych dla stałych**.

```
// Typ wyliczeniowy z implementacją metod specyficznych dla stałej
public enum Operation {
   PLUS { public double apply(double x, double y){return x + y;} },
   MINUS { public double apply(double x, double y){return x - y;} },
   TIMES { public double apply(double x, double y){return x * y;} },
   DIVIDE { public double apply(double x, double y){return x / y;} };

   public abstract double apply(double x, double y);
}
```

Jeżeli dodamy nową stałą do drugiej wersji `Operation`, mało prawdopodobne jest, że zapomnimy napisać metody `apply`, ponieważ znajduje się ona zaraz po deklaracji każdej ze stałych. W mało prawdopodobnym przypadku, gdy jednak jej zapomnimy, kompilator przypomni nam o tym, ponieważ metody abstrakcyjne w typie wyliczeniowym muszą być przesłaniane konkretnymi metodami w każdej ze stałych.

Implementacja metody specyficzna dla stałej może być łączona z danymi specyficznymi dla stałej. Poniżej znajduje się na przykład kolejna wersja `Operation`, w której nadpisujemy metodę `toString` w celu zwrócenia symbolu skojarzonego z operacją.

```
// Typ wyliczeniowy z treścią klasy specyficzną dla stałej oraz z danymi
public enum Operation {
   PLUS("+") {
      public double apply(double x, double y) { return x + y; }
   },
   MINUS("-") {
      public double apply(double x, double y) { return x - y; }
   },
   TIMES("*") {
      public double apply(double x, double y) { return x * y; }
   },
   DIVIDE("/") {
      public double apply(double x, double y) { return x / y; }
   };
   private final String symbol;
   Operation(String symbol) { this.symbol = symbol; }
   @Override public String toString() { return symbol; }

   public abstract double apply(double x, double y);
}
```

Na przykład przedstawiona powyżej implementacja `toString` pozwala łatwo wyświetlać wyrażenia arytmetyczne w sposób pokazany przez ten mały program.

```
public static void main(String[] args) {
   double x = Double.parseDouble(args[0]);
   double y = Double.parseDouble(args[1]);
   for (Operation op : Operation.values())
      System.out.printf("%f %s %f = %f%n",
                        x, op, y, op.apply(x, y));
}
```

Uruchomienie tego programu z wartościami 2 i 4 przekazanymi w wierszu poleceń daje nam następujący wynik:

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

Typy wyliczeniowe mają automatycznie generowaną metodę `valueOf(String)`, która przekształca nazwę stałej na samą stałą. Jeżeli nadpiszemy metodę `toString` w typie wyliczeniowym, warto rozważyć również napisanie metody `fromString` w celu przekształcenia własnej reprezentacji na odpowiednią stałą. Poniższy kod (po odpowiedniej zmianie typu) realizuje to zadanie dla dowolnego typu wyliczeniowego, o ile każda stała będzie miała unikatową reprezentację znakową.

```
// Implementacja metody fromString dla typu wyliczeniowego
private static final Map<String, Operation> stringToEnum =
        Stream.of(values()).collect(
           toMap(Object::toString, e -> e));

// zwraca Operation dla napisu lub null, jeżeli napis jest nieprawidłowy
public static Optional<Operation> fromString(String symbol) {
   return Optional.ofNullable(stringToEnum.get(symbol));
}
```

Należy zwrócić uwagę, że stałe `Operation` są umieszczane w odwzorowaniu `stringToEnum` w bloku statycznym, który jest wykonywany po utworzeniu stałych. Przedstawiony kod używa strumienia (rozdział 7.) dla tablicy zwróconej przez metodę `values()`. Przed Javą 8 musielibyśmy utworzyć pusty obiekt `HashMap`, a następnie przejść po każdej wartości z tablicy i wstawić ją do odwzorowania. Oczywiście jeśli chcesz, możesz nadal korzystać z takiego rozwiązania. Pamiętaj jednak, że próba umieszczenia każdej stałej w odwzorowaniu w jej konstruktorze **nie** zadziała. Spowoduje błąd kompilacji, co jest prawidłowym działaniem, ponieważ nastąpiłoby wygenerowanie `NullPointerException`, jeżeli takie przypisanie byłoby legalne. Konstruktory typu wyliczeniowego nie mogą korzystać z pól statycznych typu z wyjątkiem zmiennych stałych (temat 34.). Takie ograniczenie jest niezbędne, ponieważ te pola statyczne nie są zainicjowane w momencie wykonywania

konstruktorów. Specjalnym przypadkiem tego ograniczenia jest to, że stała wyliczeniowa nie może uzyskać dostępu do innej stałej wyliczenia z poziomu swojego konstruktora.

Zwróć uwagę, że metoda `fromString` zwraca `Optional<String>`. Umożliwia to metodzie wskazanie, że tekst, który został do niej przekazany, nie reprezentuje żadnej poprawnej operacji, co zmusza klienta do uwzględnienia takiej możliwości (temat 55.).

Wadą specyficznych dla stałych implementacji metod jest to, że powodują utrudnienie współdzielenia kodu pomiędzy stałymi wyliczenia. Jako przykład weźmy typ wyliczeniowy reprezentujący dni tygodnia w pakiecie księgowym. Taki typ wyliczeniowy posiada metodę obliczającą wynagrodzenie pracownika za dany dzień na podstawie podstawowego wynagrodzenia (za godzinę) oraz liczby minut przepracowanych danego dnia. W pięciu dniach tygodnia każda minuta przekraczająca zwykłą zmianę powoduje wygenerowanie płacy w nadgodzinach, a w dwóch dniach weekendu cały czas pracy jest traktowany jako nadliczbowy. Przy użyciu instrukcji `switch` można łatwo wykonać takie obliczenia przez zastosowanie wielu etykiet dla dwóch fragmentów kodu.

```
// Typ wyliczeniowy wybierający na podstawie własnych wartości w celu
// współdzielenia kodu - wątpliwe
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    double pay(double minutesWorked, double payRate) {
        double basePay = minutesWorked * payRate;

        double overtimePay;
        switch(this) {
            case SATURDAY: case SUNDAY: // weekend
                overtimePay = basePay / 2;
                break;
            default: // zwykłe dni tygodnia
                overtimePay = minutesWorked <= MINS_PER_SHIFT ?
                    0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        return basePay + overtimePay;
    }
}
```

Kod ten jest bez wątpienia spójny, ale z perspektywy jego utrzymania dosyć niebezpieczny. Załóżmy, że dodajemy element do wyliczenia, być może specjalną wartość oznaczającą urlop, ale zapomnimy dodać odpowiedniego przypadku do instrukcji `switch`. Program nadal będzie się kompilował, ale metoda `pay` po cichu zapłaci pracownikowi taką samą kwotę jak za zwykły dzień tygodnia.

Aby bezpiecznie wykonać obliczenie przy użyciu implementacji specyficznej dla stałej, konieczne będzie powielenie obliczenia nadgodzin dla każdej stałej lub przeniesienie obliczeń do dwóch metod pomocniczych (jednej dla weekendów i jednej dla dni roboczych) i wywoływanie odpowiedniej metody pomocniczej z każdej ze stałych. Oba podejścia powodują powstanie sporej ilości kodu narzędziowego, co znacznie zmniejsza czytelność i zwiększa możliwość popełnienia błędu.

Kod narzędziowy może być zmniejszony przez zastąpienie metody abstrakcyjnej `overtimePay` w `PayrollDay` metodą konkretną, w której wykonamy obliczenie nadgodzin dla dni powszednich. W takim przypadku konieczne jest nadpisanie metody wyłącznie dla weekendów. Ma to jednak takie same wady jak instrukcja `switch` — jeżeli dodamy kolejny dzień bez nadpisywania metody `overtimePay`, odziedziczy ona obliczenia dla dni tygodnia bez żadnej informacji o tym fakcie.

Potrzebujemy więc **wymuszenia** wyboru strategii płacy nadgodzin za każdym razem, gdy dodamy stałą wyliczenia. Na szczęście istnieje łatwy sposób osiągnięcia tego efektu. Polega to na przeniesieniu obliczenia nadgodzin do prywatnego zagnieżdżonego typu wyliczeniowego i przekazanie obiektu tego **typu wyliczeniowego strategii** do konstruktora typu wyliczeniowego `PayrollDay`. Typ wyliczeniowy `PayrollDay` deleguje obliczenie płacy w nadgodzinach do typu wyliczeniowego strategii, co eliminuje potrzebę zastosowania polecenia `switch` lub implementacji metod specyficznych dla stałych w `PayrollDay`. Choć ten wzorzec jest mniej zwięzły niż instrukcja `switch`, jest bezpieczniejszy i bardziej elastyczny:

```
// Wzorzec typu wyliczeniowego strategii
enum PayrollDay {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
   SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

   private final PayType payType;

   PayrollDay(PayType payType) { this.payType = payType; }
   PayrollDay() { this(PayType.WEEKDAY); } // domyślny

   int pay(int minutesWorked, int payRate) {
      return payType.pay(minutesWorked, payRate);
   }

   // typ wyliczeniowy strategii
   private enum PayType {
      WEEKDAY {
         int overtimePay(int minsWorked, int payRate) {
            return minsWorked <= MINS_PER_SHIFT ? 0 :
               (minsWorked - MINS_PER_SHIFT) * payRate / 2;
         }
      },
      WEEKEND {
         int overtimePay(int minsWorked, int payRate) {
```

```
            return minsWorked * payRate / 2;
         }
      };

      abstract int overtimePay(int mins, int payRate);
      private static final int MINS_PER_SHIFT = 8 * 60;

      int pay(int minsWorked, int payRate) {
         int basePay = minsWorked * payRate;
         return basePay + overtimePay(minsWorked, payRate);
      }
   }
}
```

Jeżeli instrukcje `switch` w typach wyliczeniowych nie są dobre do implementowania operacji specyficznych dla stałych w typach wyliczeniowych, to **do** czego się one nadają**? Instrukcje wyboru w typach wyliczeniowych są dobre do wzbogacania zewnętrznych typów wyliczeniowych o operacje specyficzne dla stałych**. Załóżmy, że typ wyliczeniowy `Operation` jest poza naszą kontrolą i chcemy dodać metodę instancyjną zwracającą odwrotność każdej operacji. Można symulować ten efekt za pomocą następującej metody statycznej:

```
// Wybór ze stałej do symulowania brakującej metody
public static Operation inverse(Operation op) {
   switch(op) {
      case PLUS: return Operation.MINUS;
      case MINUS: return Operation.PLUS;
      case TIMES: return Operation.DIVIDE;
      case DIVIDE: return Operation.TIMES;

      default: throw new AssertionError("Nieznana operacja: " + op);
   }
}
```

Skorzystaj z tej techniki również dla typów wyliczeniowych, które **są** pod Twoją kontrolą, jeśli metoda po prostu nie należy do typu wyliczeniowego. Metoda może być potrzebna przy niektórych wyliczeniach, ale nie jest na tyle użyteczna, aby rozważyć dołączenie jej do typu wyliczeniowego.

Typy wyliczeniowe są porównywalne pod względem wydajności ze stałymi `int`. Niewielki spadek wydajności w stosunku do stałych `int` jest związany z ładowaniem i inicjowaniem typu wyliczeniowego. Poza urządzeniami o ograniczonych zasobach, takich jak telefony komórkowe czy tostery, mało prawdopodobne jest, aby było to w praktyce zauważalne.

Kiedy należy więc korzystać z typów wyliczeniowych? **Za każdym razem, gdy potrzebujemy stałego zestawu stałych znanych w momencie kompilacji**. Oczywiście, obejmuje to „naturalne typy wyliczeniowe”, takie jak planety, dni tygodnia

i figury szachowe. Oprócz tego mogą to być inne zbiory, których wszystkie możliwe wartości są znane w czasie kompilacji, takie jak pozycje menu, kody operacji i znaczniki wiersza poleceń. **Nie jest konieczne, aby zbiór stałych w typie wyliczeniowym był stały przez cały czas**. Mechanizm typów wyliczeniowych został zaprojektowany tak, aby pozwolić na binarną zgodność ewolucji typów wyliczeniowych.

Podsumujmy. Zalety typu wyliczeniowego w stosunku do stałych `int` są zachęcające. Typ wyliczeniowy jest znacznie bardziej czytelny, bezpieczny i daje większe możliwości. Wiele typów wyliczeniowych nie wymaga jawnego konstruktora ani składników, ale wiele innych korzysta z zalet skojarzenia danych z każdą stałą i zapewnienia metod, których działanie jest zależne od tych danych. Znacznie mniej typów wyliczeniowych korzysta z kojarzenia wielu operacji z jedną metodą. W tych względnie rzadkich przypadkach lepiej użyć metod specyficznych dla stałych, sterowanych własnymi wartościami. Jeżeli wiele stałych wyliczeniowych korzysta z tych samych operacji, warto rozważyć zastosowanie wzorca typu wyliczeniowego strategii.

Temat 35. Użycie pól instancyjnych zamiast kolejności

Wiele typów wyliczeniowych jest naturalnie skojarzonych z wartościami `int`. Wszystkie takie typy mają metodę `ordinal`, która zwraca pozycję numeryczną każdej stałej wyliczenia w tym typie. Można ulec pokusie skorzystania z wartości `int` metody `ordinal`:

```
// Nadużycie kolejności do uzyskania skojarzonej wartości - NIE RÓB TAK
public enum Ensemble {
   SOLO, DUET, TRIO, QUARTET, QUINTET,
   SEXTET, SEPTET, OCTET, NONET, DECTET;

   public int numberOfMusicians() { return ordinal() + 1; }
}
```

Choć ten typ wyliczeniowy działa, jest on koszmarem utrzymania. Jeżeli kolejność stałych zostanie zmieniona, metoda `numberOfMusicians` zawiedzie. Jeżeli chcemy dodać drugą stałą wyliczenia skojarzoną z wartością `int`, która była już użyta, mamy sporego pecha. Załóżmy, że chcemy dodać stałą do **podwójnego kwartetu**, który podobnie jak oktet składa się z ośmiu muzyków, ale nie ma sposobu na wykonanie tej operacji.

Dodatkowo nie można dodać stałej dla wartości `int` bez dodawania stałej dla kolejnych wartości `int`. Chcemy na przykład dodać stałą reprezentującą **potrójny kwartet**, który składa się z dwunastu muzyków. Nie istnieje standardowy termin

dla zespołu składającego się z jedenastu muzyków, więc jesteśmy zmuszeni do dodania dodatkowej stałej dla nieużywanej wartości `int` (11). Jest to co najmniej niełądne. Jeżeli mamy wiele nieużywanych wartości `int`, jest to niepraktyczne.

Na szczęście istnieje proste rozwiązanie tych problemów. **Nigdy nie należy kojarzyć wartości skojarzonej z wartością wyliczaną na podstawie kolejności; należy zamiast tego przechować ją w polu instancyjnym**:

```
public enum Ensemble {
   SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
   SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
   NONET(9), DECTET(10), TRIPLE_QUARTET(12);

   private final int numberOfMusicians;
   Ensemble(int size) { this.numberOfMusicians = size; }
   public int numberOfMusicians() { return numberOfMusicians; }
}
```

Specyfikacja `Enum` zawiera następujące objaśnienie dla metody `ordinal`: „Większość programistów nie skorzysta z tej metody. Jest zaprojektowana do użycia przez ogólne struktury danych bazujące na typach wyliczeniowych, takie jak `EnumSet` i `EnumMap`". Jeżeli nie piszemy tego typu struktur danych, najlepiej unikać metody `ordinal`.

Temat 36. Użycie EnumSet zamiast pól bitowych

Jeżeli element typu wyliczeniowego był używany przede wszystkim w zbiorach, zwykle wykorzystywany był wzorzec typu wyliczeniowego `int` (temat 34.), w którym każdej stałej przypisywana była inna potęga 2:

```
// Bitowe stałe typu wyliczeniowego - PRZESTARZAŁE!
public class Text {
   public static final int STYLE_BOLD = 1 << 0; // 1
   public static final int STYLE_ITALIC = 1 << 1; // 2
   public static final int STYLE_UNDERLINE = 1 << 2; // 4
   public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

   // parametr jest bitową sumą zero lub więcej stałych STYLE_
   public void applyStyles(int styles) { ... }
}
```

Ta reprezentacja pozwala użyć bitowej operacji `OR` do połączenia kilku stałych ze zbioru, co jest znane pod nazwą **pola bitowego**:

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

Reprezentacja pola bitowego pozwala również na efektywne wykonywanie zbioru operacji, takich jak suma lub część wspólna, przy użyciu arytmetyki bitowej. Pola bitowe mają wszystkie wady stałych `int`, jak również swoje własne. Jest nawet trudniej zinterpretować pole bitowe niż zwykłą stałą wyliczeniową `int` po jej wyświetleniu w postaci liczby. Dodatkowo nie istnieje prosty sposób na iterowanie po wszystkich elementach reprezentowanych przez pole bitowe. Co więcej, trzeba przewidzieć maksymalną liczbę bitów, która kiedykolwiek będzie potrzebna, gdy pisze się API, i na tej podstawie dobrać typ pola (`int` lub `long`). Po wybraniu typu nie można przekroczyć jego szerokości (32 lub 64 bity) bez zmiany API.

Niektórzy programiści, korzystający z typów wyliczeniowych zamiast stałych `int`, nadal upierają się przy korzystaniu z pól bitowych, gdy muszą przekazywać zbiory stałych. Nie ma powodu, aby to robić, ponieważ istnieje lepsza alternatywa. W pakiecie `java.util` znajduje się klasa `EnumSet`, która pozwala efektywnie reprezentować zbiór wartości jednego typu wyliczeniowego. Klasa ta implementuje interfejs `Set`, dzięki czemu daje bogactwo, bezpieczeństwo typów i możliwości współpracy zapewniane przez wszystkie inne implementacje `Set`. Jednak wewnętrznie każdy `EnumSet` jest reprezentowany jako wektor bitowy. Jeżeli bazowy typ wyliczeniowy ma nie więcej niż sześćdziesiąt cztery elementy — czyli w większości przypadków — cały `EnumSet` jest reprezentowany za pomocą jednej liczby `long`, więc wydajność jest porównywalna do pól bitowych. Operacje masowe, takie jak `removeAll` i `retainAll`, są implementowane za pomocą arytmetyki bitowej, tak samo jak wykonujemy to ręcznie w przypadku pól bitowych. Jesteśmy jednak izolowani od brzydoty i podatności na błędy ręcznych manipulacji na bitach — `EnumSet` wykonuje za nas całą ciężką pracę.

Poniżej zamieszczony jest poprzedni przykład korzystający z typu wyliczeniowego zamiast pól bitowych. Jest krótszy, czytelniejszy i bezpieczniejszy:

```
// EnumSet - nowoczesny następca pól bitowych
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // może być przekazany dowolny Set, ale EnumSet jest najlepszy
    public void applyStyles(Set<Style> styles) { ... }
}
```

Poniżej mamy kod klienta, który przekazuje obiekt `EnumSet` do metody `applyStyles`. `EnumSet` zapewnia bogaty zestaw statycznych metod fabrycznych dla ułatwienia tworzenia zbioru i jedna z nich jest użyta w tym fragmencie kodu:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Należy zwrócić uwagę, że metoda `applyStyles` oczekuje `Set<Style>`, a nie `EnumSet<Style>`. Choć wydaje się prawdopodobne, że wszystkie klienty będą przekazywać do metody obiekt `EnumSet`, to jednak dobrą praktyką jest akceptowanie typu interfejsu zamiast typu implementacji (temat 64.). Pozostawia to możliwość przekazania przez klienta w niektórych przypadkach innych implementacji `Set`.

Podsumujmy. **Użycie typu wyliczeniowego w zbiorach nie jest powodem jego reprezentacji przy użyciu pól bitowych**. Klasa `EnumSet` łączy w sobie spójność i wydajność pól bitowych z wieloma zaletami typów wyliczeniowych opisanych w temacie 34. Jedyną rzeczywistą wadą `EnumSet` jest to, że nadal w Java 9 niemożliwe jest utworzenie niezmiennego obiektu `EnumSet`, ale najprawdopodobniej zostanie to poprawione w kolejnej wersji. Na razie można opakować `EnumSet` za pomocą `Collections.unmodifiableSet`, ale wtedy zmniejszona zostanie zwięzłość i wydajność.

Temat 37. Użycie EnumMap zamiast indeksowania kolejnością

Czasami można napotkać kod korzystający z indeksowania tablicy lub listy za pomocą metody `ordinal` (temat 35.). Oto prosty przykład klasy reprezentującej roślinę:

```
class Plant {
   enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }

   final String name;
   final LifeCycle lifeCycle;

   Plant(String name, LifeCycle lifeCycle) {
      this.name = name;
      this.lifeCycle = lifeCycle;
   }

   @Override public String toString() {
      return name;
   }
}
```

Załóżmy teraz, że mamy tablicę reprezentującą rośliny w ogrodzie i chcemy wyświetlić te rośliny zorganizowane względem typu (roczne, wieloletnie lub dwuletnie). W tym celu utworzymy trzy zbiory, po jednym dla każdego cyklu życia i będziemy iterować po elementach ogrodu, umieszczając każdą roślinę w odpowiednim zbiorze. Niektórzy programiści umieściliby te zbiory w tablicy indeksowanej wartością `ordinal` cyklu życia:

```
// Zastosowanie ordinal() do indeksowania tablicy - NIE RÓB TAK!
Set<Plant>[] plantsByLifeCycle =
   (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];
for (int i = 0; i < plantsByLifeCycle.length; i++)
   plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
   plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
```

```
// wyświetlenie wyników
for (int i = 0; i < plantsByLifeCycle.length; i++) {
   System.out.printf("%s: %s%n",
      Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

Technika ta działa, ale powoduje wiele problemów. Ponieważ tablice nie są zgodne z typami ogólnymi (temat 28.), program wymaga niekontrolowanego rzutowania, więc nie skompiluje się bez problemu. Ponieważ tablica nie ma żadnych informacji na temat tego, co reprezentuje indeks, konieczne jest ręczne nadawanie etykiet w czasie wyświetlania. Jednak najpoważniejszym problemem jest to, że przy odwoływaniu się do tablicy poprzez indeks będący kolejnością w typie wyliczeniowym spoczywa na nas odpowiedzialność za użycie właściwej wartości `int`; liczby `int` nie zapewniają bezpieczeństwa typów, tak jak typ wyliczeniowy. Jeżeli użyjemy niewłaściwej wartości, program wykona niewłaściwą operację lub — jeżeli będziemy mieli szczęście — zgłosi wyjątek `ArrayIndexOutOfBoundsException`.

Na szczęście istnieje znacznie lepszy sposób na osiągnięcie tego samego efektu. Tablica efektywnie służy jako odwzorowanie pomiędzy typem wyliczeniowym a wartością, więc można w takim przypadku zastosować `Map`. Dokładniej — dostępna jest bardzo szybka implementacja `Map`, zaprojektowana do wykorzystywania z kluczami typu wyliczeniowego, o nazwie `java.util.EnumMap`. Poprzedni program korzystający z `EnumMap` wygląda następująco:

```
// Zastosowanie EnumMap do skojarzenia typu wyliczeniowego z danymi
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
   new EnumMap<>(Plant.LifeCycle.class);
for (Plant.LifeCycle lc : Plant.LifeCycle.values())
   plantsByLifeCycle.put(lc, new HashSet<>());
for (Plant p : garden)
   plantsByLifeCycle.get(p.lifeCycle).add(p);
System.out.println(plantsByLifeCycle);
```

Program ten jest krótszy, czytelniejszy i bezpieczniejszy, a dodatkowo ma wydajność zbliżoną do pierwszej wersji. Nie ma tu niebezpiecznego rzutowania; nie ma potrzeby ręcznego wyświetlania etykiet, ponieważ klucze odwzorowania są typu wyliczeniowego, który może się przekształcić na czytelną postać, a dodatkowo nie ma możliwości popełnienia błędu przy obliczaniu indeksów tablicy. Obiekty `EnumMap` mają porównywalną wydajność z tablicami indeksowanymi za pomocą kolejności, ponieważ korzystają one wewnętrznie właśnie z tablic. Jednak ukrywają szczegóły implementacji przed programistą, łącząc bogactwo i bezpieczeństwo typów Map z wydajnością tablicy. Należy zwrócić uwagę, że konstruktor `EnumMap` oczekuje jako klucza obiektu `Class` — jest to **token typu związanego**, który zapewnia dostęp do typu ogólnego w czasie działania (temat 33.).

Przedstawiony powyżej program można skrócić jeszcze bardziej, wykorzystując strumienie (temat 45.) do zarządzania odwzorowaniem. Oto najprostszy kod bazujący na strumieniach, który w dużej mierze powiela działanie wcześniejszego przykładu:

```
// Uproszczone podejście wykorzystujące strumienie - jest wątpliwe,
// czy doprowadzi do utworzenia EnumMap!
System.out.println(Arrays.stream(garden)
   .collect(groupingBy(p -> p.lifeCycle)));
```

Problemem w tym kodzie jest to, że wybierze on własną implementację odwzorowania, czyli w praktyce inną klasę niż `EnumMap`, a tym samym nie zapewni wydajności i zwięzłości pamięciowej rozwiązania stosującego `EnumMap` w sposób jawny. Aby rozwiązać ten problem, użyjmy trójparametrowej wersji `Collectors.groupingBy`, która pozwala kodowi wywołującemu wskazać implementację odwzorowania w parametrze `mapFactory`:

```
// Wykorzystanie strumienia i EnumMap do powiązania danych z wyliczeniem
System.out.println(Arrays.stream(garden)
   .collect(groupingBy(p -> p.lifeCycle,
      () -> new EnumMap<>(LifeCycle.class), toSet())));
```

Tego rodzaju optymalizacja nie jest warta dodatkowego nakładu pracy w tak prostym przykładzie jak prezentowany, ale może mieć duże znaczenie w programie wykorzystującym odwzorowania w sposób niezwykle intensywny.

Zachowanie wersji bazującej na strumieniu różni się nieco od wersji z `EnumMap`. Wersja `EnumMap` zawsze tworzy zagnieżdżone odwzorowanie dla każdego cyklu życia rośliny, a wersja strumieniowa tworzy zagnieżdżone odwzorowanie tylko wtedy, gdy ogród zawiera jedną roślinę lub więcej roślin o danym cyklu życia. Jeśli więc ogród zawiera rośliny jednoroczne i wieloletnie, ale nie dwuletnie, rozmiar `plantsByLifeCycle` będzie w wersji `EnumMap` wynosił trzy, a w obu wersjach strumieniowych dwa.

Można się spotkać również z tablicami tablic, indeksowanymi (dwukrotnie!) kolejnością w typie, reprezentującymi odwzorowanie pomiędzy dwoma wartościami wyliczeniowymi. Na przykład poniższy program korzysta z takich tablic do reprezentowania przejść pomiędzy dwoma stanami (z ciekłego na stały to zamarzanie, z ciekłego do gazowego to parowanie i tak dalej).

```
// Zastosowanie ordinal() do indeksowania tablicy tablic - NIE RÓB TAK!
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
      MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

      // wiersze indeksowane kolejnością źródłową, kolumny docelową
      private static final Transition[][] TRANSITIONS = {
         { null, MELT, SUBLIME },
         { FREEZE, null, BOIL },
         { DEPOSIT, CONDENSE, null }
      };
```

```
        // zwraca przejście fazowe z jednego stanu do drugiego
        public static Transition from(Phase from, Phase to) {
           return TRANSITIONS[from.ordinal()][to.ordinal()];
        }
     }
}
```

Program działa i może nawet wydawać się elegancki, ale wrażenie to jest złudne. Podobnie jak w przypadku wcześniejszego przykładu z ogrodem zielnym, kompilator nie ma możliwości poznać relacji pomiędzy kolejnością w typie i indeksami tablicy. Jeżeli popełnimy błąd w tablicy przejść lub zapomnimy zaktualizować ją przy modyfikacji typu wyliczeniowego `Phase` lub `Phase.Transition`, program ulegnie awarii w czasie działania. Awaria może przyjąć postać wyjątku `ArrayIndex` ↪`OutOfBoundsException`, `NullPointerException` lub (co gorsza) nieprawidłowego działania. Dodatkowo wielkość tablicy jest kwadratem tablicy stanów, nawet jeżeli liczba niepustych wpisów jest mała.

I w tym przypadku lepiej zastosować `EnumMap`. Ponieważ każde przejście fazowe jest indeksowane **parą** stałych stanu, najlepiej reprezentować tę relację jako odwzorowanie z jednego typu wyliczeniowego (stan początkowy) na odwzorowanie z drugiego typu wyliczeniowego (stan końcowy) i na wynik (przejście fazowe). Dwa stany skupienia skojarzone z przejściem fazowym najlepiej modelować przez dołączenie danych do typu wyliczeniowego przejść fazowych, które są następnie wykorzystywane do inicjowania zagnieżdżonych obiektów `EnumMap`:

```
// Zastosowanie zagnieżdżonych EnumMap do skojarzenia danych z parą
// wartości wyliczeniowych
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
      MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
      BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
      SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

      private final Phase from;
      private final Phase to;

      Transition(Phase from, Phase to) {
         this.from = from;
         this.to = to;
      }

      // inicjalizacja odwzorowania przejść między stanami
      private static final Map<Phase, Map<Phase, Transition>>
        m = Stream.of(values()).collect(groupingBy(t -> t.from,
        () -> new EnumMap<>(Phase.class),
        toMap(t -> t.to, t -> t,
           (x, y) -> y, () -> new EnumMap<>(Phase.class))));
```

```
        public static Transition from(Phase from, Phase to) {
            return m.get(from).get(to);
        }
    }
}
```

Kod inicjujący odwzorowanie przejść fazowych może wydawać się nieco skomplikowany, ale nie jest taki zły. Typem odwzorowania jest `Map<Phase, Map<Phase, Transition>>`, co oznacza „odwzorowanie ze stanu (źródłowego) na odwzorowanie ze stanu (docelowego) na przejście". To odwzorowanie odwzorowań jest inicjalizowane kaskadową wersją dwóch kolektorów. Pierwszy kolektor grupuje przejścia na podstawie stanu początkowego, a drugi tworzy `EnumMap` z odwzorowaniami ze stanu docelowego do przejścia. Funkcja łącząca w drugim kolektorze (`(x, y) -> y`) nie jest używana; potrzebujemy jej tylko dlatego, że musimy wskazać fabrykę obiektów map w celu uzyskania `EnumMap`, a `Collectors` oferuje fabryki teleskopowe. Poprzednie wydanie książki stosowało jawną iterację w celu inicjalizacji odwzorowań. Tamten kod był dłuższy, ale łatwiejszy do zrozumienia.

Teraz załóżmy, że chcemy dodać do systemu nowy stan — **plazmę**, czyli zjonizowany gaz. Z tym stanem są związane tylko dwa przejścia — **jonizacja**, która powoduje przejście gazu w plazmę, oraz **dejonizacja**, czyli przejście plazmy w gaz. Aby zaktualizować program bazujący na tablicach, należy dodać nową stałą do `Phase` oraz dwie do `Phase.Transition` i zamienić oryginalną dziewięcioelementową tablicę tablic na nową, szesnastoelementową. Jeżeli dodamy zbyt dużo lub za mało elementów do tablicy albo umieścimy element w niewłaściwym miejscu, wszystko pójdzie źle — program się skompiluje, ale ulegnie awarii w czasie działania. Aby zaktualizować wersję bazującą na `EnumMap`, wystarczy dodać `PLASMA` do listy stanów oraz `IONIZE(GAS, PLASMA)` i `DEIONIZE(PLASMA, GAS)` do listy przejść fazowych.

```
// Dodanie nowe fazy za pomocą zagnieżdżonej implementacji EnumMap
public enum Phase {
    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
        ... // pozostały kod pozostaje bez zmian
    }
}
```

Program zajmie się resztą, a my niemal nie będziemy mieli możliwości popełnienia błędu. Wewnętrznie odwzorowanie odwzorowań jest implementowane jako tablica tablic, więc nie poświęcimy zbyt wiele przestrzeni ani narzutu czasowego w zamian za czytelność, bezpieczeństwo i łatwość utrzymania.

Warto w tym miejscu zastanowić się nad rozwiązaniem z wcześniejszych przykładów, w których używano `null` do wskazania braku zmiany stanu (czyli wartości `from` i `to` były identyczne). Nie jest to dobra praktyka, bo może doprowadzić do wystąpienia wyjątku `NullPointerException` w trakcie działania programu. Zaprojektowanie czystego, eleganckiego rozwiązania dla analizowanego problemu jest zadziwiająco trudne. Wynikowe programy byłyby dosyć długie, co odciągałoby uwagę od głównego celu tego tematu.

Podsumujmy. **Korzystanie z wartości `ordinal` jako indeksów tablicy jest rzadko właściwe — zamiast nich należy używać `EnumMap`**. Jeżeli reprezentowana relacja jest wielowymiarowa, należy skorzystać z `EnumMap<..., EnumMap<...>>`. Jest to specjalny przypadek ogólnej zasady, że programiści aplikacji powinni bardzo rzadko, o ile nie wcale, korzystać z `Enum.ordinal` (temat 35.).

Temat 38. Emulowanie rozszerzalnych typów wyliczeniowych za pomocą interfejsów

Typy wyliczeniowe są lepsze niemal we wszystkich aspektach od wzorca bezpiecznego dla typów typu wyliczeniowego opisywanego w pierwszym wydaniu tej książki [Bloch01]. Jednak jednym problemem może być rozszerzalność, która jest możliwa przy zastosowaniu wzorca, ale nie jest wspierana przez konstrukcje języka. Inaczej mówiąc, przy użyciu wzorca można mieć jeden typ wyliczeniowy będący rozszerzeniem innego, natomiast przy użyciu konstrukcji języka — nie. Nie jest to przypadek. W większości przypadków rozszerzalność typów wyliczeniowych okazuje się bardzo złym pomysłem. Mylące jest, że elementy typu rozszerzonego są obiektami typu bazowego i że nie zachodzi relacja odwrotna. Nie istnieje dobry sposób wyliczenia wszystkich elementów typu bazowego i jego rozszerzeń. Na koniec — rozszerzalność może skomplikować wiele aspektów projektu i implementacji.

Trzeba jednak pamiętać, że istnieje co najmniej jeden kuszący przypadek użycia rozszerzalnych typów wyliczeniowych — **kody operacji**. Kody operacji są typem wyliczeniowym, którego elementy reprezentują operacje w pewnej maszynie, którego przykładem jest typ `Operation` z tematu 34., reprezentujący funkcje prostego kalkulatora. Czasami pożądane jest umożliwienie użytkownikom API dodawanie własnych operacji, co jest rozszerzaniem zbioru operacji udostępnianych przez API.

Na szczęście istnieje łatwy sposób osiągnięcia tego efektu z użyciem typów wyliczeniowych. Podstawą tego pomysłu jest wykorzystanie możliwości implementowania przez typy wyliczeniowe dowolnych interfejsów przez zdefiniowanie

interfejsu dla typu kodu operacji oraz typu wyliczeniowego będącego standardową implementacją interfejsu. Poniżej zamieszczona jest rozszerzalna wersja typu `Operation` z tematu 34.

```
// Emulowanie rozszerzalnego typu wyliczeniowego z użyciem interfejsu
public interface Operation {
   double apply(double x, double y);
}

public enum BasicOperation implements Operation {
   PLUS("+") {
      public double apply(double x, double y) { return x + y; }
   },
   MINUS("-") {
      public double apply(double x, double y) { return x - y; }
   },
   TIMES("*") {
      public double apply(double x, double y) { return x * y; }
   },
   DIVIDE("/") {
      public double apply(double x, double y) { return x / y; }
   };

   private final String symbol;

   BasicOperation(String symbol) {
      this.symbol = symbol;
   }

   @Override public String toString() {
      return symbol;
   }
}
```

Choć typ wyliczeniowy (`BasicOperation`) nie jest rozszerzalny, typ interfejsu (`Operation`) jest, a ten typ interfejsu jest używany do reprezentowania operacji w API. Możemy zdefiniować kolejny typ wyliczeniowy, który implementuje ten interfejs, i użyć obiektów tego nowego typu w miejsce typu bazowego. Dla przykładu zdefiniujemy rozszerzenie typu operacji przedstawionego powyżej, składające się z operacji potęgowania i reszty z dzielenia. Wystarczy napisać typ wyliczeniowy implementujący interfejs `Operation`:

```
// Emulowanie rozszerzenia typu wyliczeniowego
public enum ExtendedOperation implements Operation {
   EXP("^") {
      public double apply(double x, double y) {
         return Math.pow(x, y);
      }
   },
   REMAINDER("%") {
      public double apply(double x, double y) {
```

```
            return x % y;
        }
    };

    private final String symbol;

    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

Można użyć naszych nowych operacji w każdym miejscu, gdzie korzystaliśmy z operacji podstawowych, pod warunkiem że API korzysta z typu interfejsu (`Operation`), a nie implementacji (`BasicOperation`). Warto zauważyć, że nie musimy definiować abstrakcyjnej metody `apply` w typie wyliczeniowym, tak jak musieliśmy to zrobić w nierozszerzalnym typie wyliczeniowym z implementacjami metod specyficznymi dla instancji (temat 34.). Możemy tak zrobić, ponieważ metoda abstrakcyjna (`apply`) jest składnikiem interfejsu (`Operation`).

Nie tylko możliwe jest przekazanie jednej instancji „rozszerzonego wyliczenia" w każdym miejscu, gdzie spodziewane jest „bazowe wyliczenie", ale również możliwe jest przekazanie całego rozszerzonego typu wyliczeniowego i użycie jego elementów oprócz lub zamiast tych z typu bazowego. Poniżej znajduje się na przykład program testowy z tematu 34., który korzysta z wszystkich zdefiniowanych wcześniej rozszerzonych operacji:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(
        Class<T> opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

Należy zwrócić uwagę, że literał klasy dla rozszerzonego typu operacji (`Extended`
`↪Operation.class`) jest przekazywany z `main` do `test`, co pozwala na opisanie zbioru operacji rozszerzonych. Ten literał klasy służy jako **token typu związanego** (temat 33.). Skomplikowana deklaracja parametru `opEnumType` (`<T extends Enum<T> & Operation> Class<T>`) zapewnia, że obiekt `Class` reprezentuje zarówno typ wyliczeniowy, jak i podtypy `Operation`, co jest wymagane do iterowania po elementach i wykonywania operacji związanych z każdą stałą.

Drugą możliwością jest przekazanie `Collection<? extends Operation>`, będącego **związanym szablonem typu** (temat 31.), zamiast obiektu klasy:

```
public static void main(String[] args) {
   double x = Double.parseDouble(args[0]);
   double y = Double.parseDouble(args[1]);
   test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
   for (Operation op : opSet)
      System.out.printf("%f %s %f = %f%n",
                        x, op, y, op.apply(x, y));
}
```

Wynikowy kod jest nieco mniej skomplikowany, a metoda `test` jest bardziej elastyczna — pozwala to wywołującemu na połączenie operacji z wielu typów implementacji. Z drugiej strony uniemożliwiamy sobie użycie `EnumSet` (temat 36.) oraz `EnumMap` (temat 37.) dla wymienionych operacji.

Oba przedstawione powyżej programy zwrócą ten sam, zamieszczony poniżej wynik, gdy zostaną uruchomione z argumentami `2` i `4`:

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

Niewielką wadą użycia interfejsu do emulowania rozszerzalnych typów wyliczeniowych jest to, że implementacje nie mogą dziedziczyć po jednym typie wyliczeniowym w innym. Jeśli kod implementacji nie zależy od żadnego stanu, można go umieścić w interfejsie, wykorzystując przy tym metody domyślne (temat 20.). W przypadku naszego przykładu typu `Operation` logika przechowywania i odczytywania symboli związanych z operacjami jest powielona w `BasicOperation` oraz `ExtendedOperation`. W tym przypadku nie ma to znaczenia, ponieważ powielone jest niewiele kodu. Jeżeli znajdowałoby się tam więcej wspólnych funkcji, w celu wyeliminowania powtarzania kodu konieczne byłoby umieszczenie ich w klasie pomocniczej lub statycznych metodach pomocniczych.

Opisany w tym miejscu wzorzec jest wykorzystywany w bibliotekach Javy. Przykładowo typ wyliczeniowy `java.nio.file.LinkOption` implementuje interfejsy `CopyOption` i `OpenOption`.

Podsumujmy. **Choć nie można pisać rozszerzalnych typów wyliczeniowych, można emulować je przez napisanie interfejsu dla bazowego typu wyliczeniowego i implementowanie tego interfejsu**. Pozwala to klientom na pisanie własnych typów wyliczeniowych implementujących ten interfejs. Te typy wyliczeniowe mogą być używane wszędzie tam, gdzie może być wykorzystywany bazowy typ wyliczeniowy, przy założeniu że API wykorzystuje typ interfejsu.

Temat 39. Korzystanie z adnotacji zamiast wzorców nazw

Dawniej standardowym działaniem było korzystanie z **wzorców nazw** w celu wskazania, że pewne elementy programu wymagają specjalnego traktowania przez narzędzia lub framework. Na przykład przed wersją 4. framework testowania JUnit wymagał od użytkowników wyznaczenia metod testowych przez rozpoczęcie ich nazwy od `test` [Beck04]. Technika ta działa, ale ma kilka dużych wad. Po pierwsze, błędy typograficzne mogą powodować niewykryte awarie. Na przykład możemy przypadkowo nazwać metodę testującą `tsetSafetyOverride` zamiast `testSafety` `↪Override`. JUnit 3 nie wykryje błędu, ale nie wykona tych testów, dając fałszywe poczucie bezpieczeństwa.

Drugą wadą wzorca nazw jest brak możliwości upewnienia się, że są użyte we właściwych elementach programu. Załóżmy, że nazwaliśmy klasę `testSafety` `↪Mechanisms` w nadziei, że JUnit 3 przetestuje automatycznie jej wszystkie metody, niezależnie od ich nazw. I w takim przypadku JUnit 3 nie będzie nic sygnalizował, ale nie wykona żadnych testów.

Trzecią wadą wzorców nazw jest brak dobrej metody łączenia wartości parametrów z elementami programu. Na przykład możemy chcieć obsługiwać kategorię testów, które są wykonane prawidłowo, jeżeli zgłoszą określony wyjątek. Typ wyjątku jest faktycznie parametrem testu. Moglibyśmy zakodować nazwę wyjątku w nazwie metody testowej z użyciem rozbudowanego wzorca nazwy, ale będzie to brzydkie i niepewne rozwiązanie (temat 62.). Kompilator nie będzie miał żadnej możliwości sprawdzenia, czy napis, który definiuje nazwę wyjątku jest prawidłowy. Jeżeli nazwana klasa nie istnieje lub nie jest wyjątkiem, nie dowiemy się o tym do momentu uruchomienia testu.

Adnotacje [JLS, 9.7] rozwiązują te problemy w elegancki sposób. Framework JUnit umożliwił korzystanie z nich od wersji 4. Załóżmy, że chcemy zdefiniować typ adnotacji do oznaczania prostych testów uruchamianych automatycznie, które nie udają się, jeżeli zgłoszą wyjątek. Poniżej pokazane jest, jak może wyglądać taki typ adnotacji o nazwie `Test`.

```
// Deklaracja znacznikowego typu adnotacji
import java.lang.annotation.*;

/**
 * Wskazuje, że adnotowana metoda jest metodą testującą.
 * Używać tylko dla bezparametrowych metod statycznych.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Deklaracja dla typu adnotacyjnego `Test` posiada również adnotacje `Retention` oraz `Target`. Takie adnotacje dla typu adnotacyjnego są również nazywane **metaadnotacjami**. Metaadnotacja `@Retention(RetentionPolicy.RUNTIME)` wskazuje, że adnotacja `Test` powinna być utrzymywana w czasie działania. Bez tego adnotacja `Test` nie byłaby widoczna dla narzędzia testującego. Metaadnotacja `@Target ↪(ElementType.METHOD)` wskazuje, że adnotacja `Test` jest legalna tylko dla deklaracji metod — nie może być stosowana dla deklaracji klas, pól lub innych elementów programu.

Trzeba zauważyć, że przed deklaracją adnotacji `Test` znajduje się komentarz: „Używać tylko dla bezparametrowych metod statycznych". Lepiej byłoby, gdyby kompilator pozwalał wymusić takie ograniczenie, ale tego nie robi, chyba że napiszemy **procesor adnotacji**. Aby dowiedzieć się więcej na ten temat, odwiedź stronę dokumentacji dotyczącą `javax.annotation.processing`. W przypadku braku procesora adnotacji, jeżeli umieścimy adnotację `Test` w metodzie instancyjnej lub metodzie mającej jeden lub więcej parametrów, program testujący nadal będzie się kompilował, ale narzędzie testujące będzie musiało poradzić sobie z problemem w czasie działania.

Przedstawimy teraz, jak adnotacja `Test` działa w praktyce. Jest ona nazywana `adnotacją znacznikową`, ponieważ nie posiada parametrów i tylko „oznacza" adnotowany element. Jeżeli programista źle wpisze `Test` lub zastosuje adnotację `Test` do innego elementu programu niż deklaracja metody, program się nie skompiluje:

```
// Program zawierający adnotacje znacznikowe
public class Sample {
   @Test public static void m1() { } // test powinien się udać
   public static void m2() { }
   @Test public static void m3() { // test powinien zawieść
      throw new RuntimeException("Bum!");
   }
   public static void m4() { }
   @Test public void m5() { } // NIEWŁAŚCIWE UŻYCIE:
                              // metoda niestatyczna
   public static void m6() { }
   @Test public static void m7() { // test powinien zawieść
      throw new RuntimeException("Awaria");
   }
   public static void m8() { }
}
```

Klasa `Sample` posiada osiem metod, z których cztery są adnotowane jako testowe. Dwie z nich, `m3` i `m7`, zgłaszają wyjątki, a dwie, `m1` i `m5` — nie. Jednak jedna z adnotowanych metod, które nie zgłaszają wyjątku, `m5`, jest metodą instancyjną, więc nie jest to prawidłowe użycie adnotacji. W sumie `Sample` zawiera cztery testy — jeden wykona się prawidłowo, dwa nieprawidłowo i jeden jest niewłaściwy. Cztery metody nieposiadające adnotacji `Test` zostaną zignorowane przez narzędzie testujące.

Adnotacja `Test` nie ma bezpośredniego wpływu na klasę `Sample`. Służy ona wyłącznie w celu dostarczania informacji dla „zainteresowanych" programów. Mówiąc bardziej ogólnie, adnotacje nigdy nie zmieniają semantyki adnotowanego kodu, ale pozwalają na jego specjalne traktowanie przez narzędzia, takie jak prosty program do uruchamiania testów:

```
// Program przetwarzający adnotacje znacznikowe
import java.lang.reflect.*;

public class RunTests {
   public static void main(String[] args) throws Exception {
      int tests = 0;
      int passed = 0;
      Class<?> testClass = Class.forName(args[0]);
      for (Method m : testClass.getDeclaredMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            tests++;
            try {
               m.invoke(null);
               passed++;
            } catch (InvocationTargetException wrappedExc) {
               Throwable exc = wrappedExc.getCause();
               System.out.println(m + " nieudany: " + exc);
            } catch (Exception exc) {
               System.out.println("Niewłaściwy @Test: " + m);
            }
         }
      }
      System.out.printf("Udanych: %d, Nieudanych: %d%n",
                        passed, tests - passed);
   }
}
```

Narzędzie do uruchamiania testów pobiera w pełni kwalifikowaną nazwę klasy z wiersza polecenia i uruchamia wszystkie metody adnotowane przez `Test` z użyciem refleksji przez wywołanie `Method.invoke`. Metoda `isAnnotationPresent` informuje program, którą metodę należy uruchomić. Jeżeli metoda testowa zgłosi wyjątek, mechanizmy refleksji opakowują go w `InvocationTargetException`. Narzędzie przechwytuje wyjątek i wyświetla raport o niewłaściwym wykonaniu, zawierający oryginalny wyjątek zgłoszony przez metodę testową, wyodrębniony z `InvocationTargetException` za pomocą metody `getCause`.

Jeżeli próba wywołania metody testowej za pomocą refleksji zgłosi wyjątek inny niż `InvocationTargetException`, oznacza to niewłaściwe zastosowanie adnotacji `Test`, które nie zostało wychwycone w czasie kompilacji. Takie użycia obejmują adnotację metody instancyjnej, metody z co najmniej jednym parametrem lub niedostępnej metody. Drugi blok `catch` w programie uruchamiającym testy przechwytuje błędy użycia i wyświetla odpowiedni komunikat o błędzie. Poniżej przedstawiony jest wynik wykonania programu `RunTests` z parametrem `Sample`:

```
public static void Sample.m3() nieudany: RuntimeException: Bum!
Niewłaściwy @Test: public void Sample.m5()
public static void Sample.m7() nieudany: RuntimeException: Awaria
Udanych: 1, Nieudanych: 3
```

Dodajmy teraz obsługę testów, które udają się, jeżeli zgłoszą określony wyjątek. Będziemy potrzebować dla nich nowej adnotacji:

```
// Typ adnotacji z parametrem
import java.lang.annotation.*;
/**
 * Wskazuje, że adnotowana metoda jest metodą testową,
 * która w razie powodzenia musi zgłosić wyznaczony wyjątek.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

Typem tego parametru dla adnotacji jest `Class<? extends Exception>`. Ten typ szablonowy jest oczywiście bardzo ogólny. Mówiąc po ludzku, oznacza on „obiekt `Class` pewnej klasy dziedziczącej po `Exception`", co pozwala użytkownikowi adnotacji określić dowolny typ wyjątku. Jest to przykład zastosowania **tokena typu związanego** (temat 33.). Jak to działa w praktyce? Warto zwrócić uwagę, że literał klasy jest użyty jako wartość parametru adnotacji:

```
// Program zawierający adnotacje z parametrem
public class Sample2 {
   @ExceptionTest(ArithmeticException.class)
   public static void m1() { // test powinien się powieść
      int i = 0;
      i = i / i;
   }
   @ExceptionTest(ArithmeticException.class)
   public static void m2() { // test owinien zawieść (zły wyjątek)
      int[] a = new int[0];
      int i = a[1];
   }
   @ExceptionTest(ArithmeticException.class)
   public static void m3() { } // test powinien zawieść (brak wyjątku)
}
```

Zmodyfikujemy teraz nasz program do uruchamiania testów, aby przetwarzał nowe adnotacje. Wymaga to dodania do metody `main` następującego kodu:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
   tests++;
   try {
      m.invoke(null);
      System.out.printf("Test %s nieudany: brak wyjątku%n", m);
```

```
        } catch (InvocationTargetException wrappedEx) {
            Throwable exc = wrappedEx.getCause();
            Class<? extends Exception> excType =
                m.getAnnotation(ExceptionTest.class).value();
            if (excType.isInstance(exc)) {
                passed++;
            } else {
                System.out.printf(
                    "Test %s nieudany: oczekiwano %s, otrzymano %s%n",
                    m, excType.getName(), exc);
            }
        } catch (Exception exc) {
            System.out.println("Niewłaściwy @Test: " + m);
        }
    }
```

Kod ten jest podobny do kodu używanego przy przetwarzaniu adnotacji `Test`, z jednym wyjątkiem — pobiera on wartość parametru adnotacji i korzysta z niej do sprawdzenia, czy zgłoszony wyjątek jest odpowiedniego typu. Nie ma tu jawnego rzutowania, dzięki czemu nie ma niebezpieczeństwa zgłoszenia `ClassCastException`. To, że program testowy kompiluje się, gwarantuje, że jego parametry adnotacji reprezentują prawidłowe typy wyjątków, poza jednym przypadkiem — możliwe jest, że parametry adnotacji są prawidłowe w czasie kompilacji, ale plik klasy reprezentującej typ wyjątku nie jest dostępny w czasie działania. W tym dosyć rzadkim przypadku program zgłosi wyjątek `TypeNotPresentException`.

Rozszerzając nasz przykład testowania wyjątków, możliwe jest, że test będzie sprawdzał, czy metoda zgłasza jeden z kilku wymienionych wyjątków. Mechanizm adnotacji posiada mechanizmy ułatwiające obsługę takich przypadków. Załóżmy, że zmienimy typ parametru adnotacji `ExceptionTest` na tablicę obiektów `Class`:

```
// Typ adnotacji z parametrem tablicowym
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

Składnia parametrów tablicowych w adnotacjach jest elastyczna. Jest zoptymalizowana dla tablic jednoelementowych. Wszystkie poprzednie adnotacje `Exception` ↪`Test` są nadal prawidłowe dla nowej wersji z parametrem tablicowym i będą dawać w wyniku tablice jednoelementowe. Aby przekazać tablicę wieloelementową, należy ująć elementy w nawiasy klamrowe i rozdzielić je przecinkami:

```
// Kod zawierający adnotację z parametrem tablicowym
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<>();
```

```
        // specyfikacja pozwala tej metodzie zgłosić
        // IndexOutOfBoundsException lub NullPointerException
        list.addAll(5, null);
    }
```

Dosyć łatwo jest zmienić nasz program testujący, aby przetwarzać nową wersję `ExceptionTest`. Poniższy kod zastępuje oryginalną wersję:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
   tests++;
   try {
      m.invoke(null);
      System.out.printf("Test %s nieudany: brak wyjątku%n", m);
   } catch (Throwable wrappedExc) {
      Throwable exc = wrappedExc.getCause();
      int oldPassed = passed;
      Class<? extends Exception>[] excTypes =
         m.getAnnotation(ExceptionTest.class).value();
      for (Class<? extends Exception> excType : excTypes) {
         if (excType.isInstance(exc)) {
            passed++;
            break;
         }
      }
      if (passed == oldPassed)
         System.out.printf("Test %s nieudany: %s %n", m, exc);
   }
}
```

Wraz z Javą 8 pojawił się nowy sposób tworzenia adnotacji wielowartościowych. Zamiast deklarować typ adnotacyjny z parametrem w postaci tablicy, można oznaczyć deklarację adnotacji metaadnotacją `@Repeatable`, która oznacza, że do jednego elementu można tę samą adnotację stosować wielokrotnie. Ta metaadnotacja przyjmuje jeden parametr, który jest obiektem klasy **typu adnotacyjnego zawierającego**, którego jedynym parametrem jest tablica typu adnotacyjnego [JLS, 9.6.3]. Poniżej przedstawiamy, jak wyglądałaby adnotacja, gdybyśmy kontynuowali przykład z adnotacją `ExceptionTest`. Zwróć uwagę, że typ adnotacji zawierającej musi zostać oznaczony odpowiednią regułą retencji i celu, bo w przeciwnym razie deklaracji nie uda się skompilować:

```
// Typ adnotacji pozwalający na powtarzalność adnotacji
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
   Class<? extends Exception> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
```

```
public @interface ExceptionTestContainer {
   ExceptionTest[] value();
}
```

Oto jak wyglądałby test `doublyBad` wykorzystujący adnotację powtarzalną zamiast adnotacji przyjmującej tablicę:

```
// Kod wykorzystujący adnotację powtarzalną
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

Przetwarzanie powtarzalnych adnotacji wymaga uwagi. Adnotacja powtarzalna generuje sztuczny zawierający typ adnotacyjny. Metoda `getAnnotationsByType` jest tego świadoma i może być stosowana do dostępu zarówno do powtórzonych, jak i niepowtórzonych adnotacji typu powtarzalnego. Z drugiej strony `isAnnotation` ↪`Present` jawnie wskazuje, że adnotacja powtarzalna nie jest typu adnotacyjnego, ale zawierającego typu adnotacyjnego. Jeśli element posiada adnotację powtarzalną pewnego typu i użyje się metody `isAnnotationPresent` do sprawdzenia, czy element posiada adnotację tego typu, okaże się, że tak nie jest. Użycie tej metody do sprawdzania istnienia typu adnotacyjnego spowoduje, że program będzie po cichu ignorował adnotacje powtarzalne. Z drugiej strony, jeśli użyjemy tej metody do sprawdzania zawierającego typu adnotacyjnego, program będzie po cichu ignorował adnotacje nieobsługujące powtarzania. Aby za pomocą `isAnnotationPresent` wykrywać wersje z powtarzaniem i bez niego, trzeba sprawdzać zarówno typ adnotacji jak i typ adnotacji zawierającej. Oto jak wyglądałby fragment programu `RunTests` po wprowadzeniu poprawek pozwalających na obsługę powtarzanej wersji adnotacji `ExceptionTest`:

```
// Przetwarzanie adnotacji powtarzalnych
if (m.isAnnotationPresent(ExceptionTest.class)
   || m.isAnnotationPresent(ExceptionTestContainer.class)) {
   tests++;
   try {
      m.invoke(null);
      System.out.printf("Test %s nieudany: brak wyjątku", m);
   } catch (Throwable wrappedExc) {
      Throwable exc = wrappedExc.getCause();
      int oldPassed = passed;
      ExceptionTest[] excTests =
           m.getAnnotationsByType(ExceptionTest.class);
      for (ExceptionTest excTest : excTests) {
         if (excTest.value().isInstance(exc)) {
            passed++;
            break;
         }
      }
      if (passed == oldPassed)
         System.out.printf("Test %s nieudany: %s %n", m, exc);
   }
}
```

Adnotacje powtarzalne wprowadzono, aby poprawić czytelność kodu źródłowego, który w sposób logiczny stosuje wiele instancji tego samego typu adnotacyjnego dla danego elementu programu. Jeśli uważasz, że faktycznie będzie to poprawiało czytelność kodu źródłowego, korzystaj z nich, ale pamiętaj, że deklaracja i przetwarzanie adnotacji powtarzalnych wymaga więcej kodu, a przetwarzanie jest też narażone na błędy.

Framework testujący opracowany w tym temacie jest tylko zabawką, ale jasno demonstruje przewagę adnotacji nad wzorcami nazw. Pokazuje on tylko czubek góry lodowej możliwości, jakie dają adnotacje. Jeżeli napiszemy narzędzie wymagające od programisty dodania informacji do plików źródłowych, wystarczy zdefiniować zbiór odpowiednich adnotacji. **Nie istnieje dobry powód używania wzorców nazw, gdy mamy do dyspozycji adnotacje**.

Jednak oprócz twórców narzędzi większość programistów nie będzie musiała definiować typów adnotacyjnych. **Wszyscy programiści powinni jednak używać predefiniowanych typów adnotacji oferowanych przez platformę Java** (tematy 40. oraz 27.). Dodatkowo należy rozważyć użycie adnotacji definiowanych przez używane IDE lub narzędzia analizy statycznej. Takie adnotacje pozwalają poprawić jakość informacji diagnostycznych udostępnianych przez te narzędzia. Należy jednak pamiętać, że te adnotacje nie zostały ustandaryzowane, więc być może będzie potrzeba włożyć nieco pracy przy zmianie narzędzi lub po powstaniu standardu.

Temat 40. Spójne użycie adnotacji Override

Biblioteki Javy zawierają kilka typów adnotacji. Dla typowego programisty najważniejszy jest `@Override`. Adnotacja ta może być używana tylko w deklaracjach metod i wskazuje, że adnotowana deklaracja metody przesłania deklarację w typie bazowym. Jeżeli będziemy spójnie korzystać z tej adnotacji, zabezpieczymy się przed dużą grupą nieprzyjemnych błędów. Przeanalizujmy poniższy program, w którym klasa `Bigram` reprezentuje **bigram**, czyli uporządkowaną parę liter:

```
// Czy możesz wskazać błąd?
public class Bigram {
   private final char first;
   private final char second;
   public Bigram(char first, char second) {
      this.first = first;
      this.second = second;
   }
   public boolean equals(Bigram b) {
      return b.first == first && b.second == second;
```

```
    }
    public int hashCode() {
       return 31 * first + second;
    }

    public static void main(String[] args) {
       Set<Bigram> s = new HashSet<Bigram>();
       for (int i = 0; i < 10; i++)
          for (char ch = 'a'; ch <= 'z'; ch++)
             s.add(new Bigram(ch, ch));
       System.out.println(s.size());
    }
}
```

Główny program w pętli dodaje do zbioru dwadzieścia sześć bigramów, z których każdy składa się z dwóch identycznych małych liter. Następnie wyświetlana jest wielkość zbioru. Można oczekiwać, że program wyświetli 26, ponieważ zbiory nie mogą zawierać duplikatów. Jeżeli spróbujemy uruchomić ten program, okaże się, że wyświetla nie 26, ale 260. Gdzie tkwi błąd?

Jasne jest, że autor klasy `Bigram` miał zamiar nadpisać metodę `equals` (temat 10.) i pamiętał nawet o nadpisaniu metody `hashCode` (temat 11.). Niestety nasz nieszczęsny programista nie nadpisał `equals`, ale ją przeciążył (temat 52.). Aby nadpisać `Object.equals`, należy zdefiniować metodę `equals`, której parametrem jest `Object`, ale parametrem metody `equals` w klasie `Bigram` nie jest `Object`, więc `Bigram` dziedziczy metodę `equals` z `Object`. Ta metoda `equals` testuje **identyczność** obiektów, podobnie jak operator `==`. Każda z dziesięciu kopii każdego `bigramu` jest inna od pozostałych dziewięciu, więc dla metody `Object.equals` są różne, co wyjaśnia, dlaczego program wyświetlił 260.

Na szczęście kompilator może pomóc nam znaleźć ten błąd, ale tylko wtedy, gdy nieco mu w tym pomożemy przez wskazanie, że nadpisujemy `Object.equals`. W tym celu adnotujemy `Bigram.equals` za pomocą `@Override`, tak jak poniżej:

```
@Override public boolean equals(Bigram b) {
   return b.first == first && b.second == second;
}
```

Jeżeli wstawimy tę adnotację i spróbujemy skompilować program, kompilator wygeneruje komunikat o błędzie, taki jak ten:

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

Można natychmiast zorientować się, co zrobiliśmy źle, klepnąć się w czoło i zamienić błędną implementację `equals` na prawidłową (temat 10.):

```
@Override public boolean equals(Object o) {
   if (!(o instanceof Bigram))
      return false;
   Bigram b = (Bigram) o;
   return b.first == first && b.second == second;
}
```

Z tego powodu **powinno się używać adnotacji `Override` przy każdej deklaracji metody, która ma nadpisywać deklarację w klasie bazowej**. Istnieje jeden niewielki wyjątek od tej reguły. Jeżeli piszemy klasę, która nie jest oznaczona jako abstrakcyjna, i uważamy, że będzie nadpisywać metodę abstrakcyjną, nie trzeba dodawać adnotacji `Override` dla tej metody. W klasie, która nie jest zadeklarowana jako abstrakcyjna, kompilator będzie generował komunikat o błędzie, jeżeli nie nadpiszemy abstrakcyjnej metody z klasy bazowej. Jednak można również skupić uwagę na wszystkich metodach swojej klasy, które nadpisują metody bazowe, i w takim przypadku można bez żadnych problemów dodać do nich adnotacje. Większość środowisk IDE można skonfigurować tak, aby wstawiały adnotacje `@Override` automatycznie, gdy decydujemy się przesłonić metodę.

Nowoczesne środowiska IDE dają kolejny powód na spójne korzystanie z adnotacji `Override`. Środowiska te wykonują automatyczne testy nazywane inspekcją kodu. Jeżeli włączymy odpowiednią inspekcję, IDE będzie generowało ostrzeżenia w przypadku napotkania metody, która nie ma adnotacji `Override`, ale nadpisuje metodę w klasie bazowej. Jeżeli będziemy konsekwentnie korzystać z adnotacji `Override`, ostrzeżenie to będzie szybko informować o nieoczekiwanym nadpisywaniu. Ostrzeżenia te uzupełniają komunikaty o błędach kompilatora, które ostrzegają nas o nieoczekiwanym błędzie nadpisania. Dzięki IDE i kompilatorowi możemy być pewni, że nadpisujemy metody wszędzie tam, gdzie chcemy je nadpisać, i nigdzie indziej.

Można używać adnotacji `Override` w deklaracjach metod, które nadpisują deklaracje z interfejsów, jak również klas. Wraz z pojawieniem się metod domyślnych dobrą praktyką stało się korzystanie z `Override` na konkretnych implementacjach interfejsów metod, aby upewnić się co do poprawności sygnatury. Jeśli wiesz, że interfejs nie zawiera metod domyślnych, możesz pominąć adnotacje `Override` w konkretnych implementacjach metod interfejsu, aby niepotrzebnie nie zaśmiecać kodu.

W klasie abstrakcyjnej lub interfejsie **warto** jednak adnotować **wszystkie** metody, które będą nadpisywały metody klasy lub interfejsu nadrzędnego, niezależnie, czy konkretne, czy abstrakcyjne. Na przykład interfejs `Set` dodaje nowe metody do interfejsu `Collection`, więc powinien zawierać adnotacje `Override` we wszystkich deklaracjach metod w celu upewnienia się, czy przypadkowo do interfejsu `Collection` nie zostały dodane nowe metody.

Podsumujmy. Kompilator może ochronić nas przed wieloma błędami, o ile będziemy używać adnotacji `Override` w każdej deklaracji metody, która będzie nadpisywała deklarację w typie bazowym — z jednym wyjątkiem. W klasie konkretnej nie ma potrzeby adnotować metod, które będą nadpisywały deklaracje metod abstrakcyjnych (choć nic nie przeszkadza to zrobić).

Temat 41. Użycie interfejsów znacznikowych do definiowania typów

Interfejs znacznikowy jest interfejsem niezawierającym deklaracji metod. Oznacza on klasę implementującą ten interfejs jako mającą pewne właściwości. Przykładem może być interfejs `Serializable` (rozdział 12.). Przez implementowanie tego interfejsu klasa wskazuje, że te obiekty mogą być zapisywane do `ObjectOutput ↪Stream` (lub serializowane).

Można spotkać się z opinią, że adnotacje znacznikowe (temat 39.) spowodowały, że interfejsy znacznikowe są przestarzałe. Założenie to jest błędne. Interfejsy znacznikowe mają dwie zalety w stosunku do adnotacji znacznikowych. **Pierwszą i najważniejszą jest to, że interfejsy znacznikowe definiują typ implementowany przez obiekty oznaczanej klasy, a adnotacje znacznikowe tego nie zapewniają**. Istnienie tego typu pozwala przechwytywać błędy w czasie kompilacji, które w przypadku użycia adnotacji znacznikowej mogą być przechwycone dopiero w czasie działania programu.

Mechanizm serializacji Javy (rozdział 11.) używa interfejsu znacznikowego `Serializable` do oznaczenia typu, który można serializować. Metoda `ObjectOutput ↪Stream.write`, która odpowiada za serializację przekazanego do niej obiektu, wymaga zastosowania tego interfejsu w przekazywanym obiekcie. Gdyby argumentem metody był typ `Serializable`, próbę serializacji nieodpowiedniego obiektu można by wykryć już na etapie kompilacji. Wykrywanie błędów na etapie kompilacji to główne zadanie interfejsów znacznikowych. Z niewyjaśnionych przyczyn metoda `ObjectOutputStream.write` nie skorzysta z interfejsu `Serializable`. Typem argumentu metody jest `Object`. Jak się okazuje, próba wywołania `ObjectOutputStream.write` na obiekcie, który nie implementuje `Serializable`, spowoduje błąd tylko w czasie wykonania.

Inną przewagą interfejsu znacznikowego nad adnotacjami znacznikowymi jest to, że mogą być bardziej precyzyjnie kierowane. Jeżeli typ adnotacji jest zadeklarowany z docelowym `ElementType.TYPE`, może być stosowany do **dowolnej** klasy lub interfejsu. Załóżmy, że mamy znacznik stosowany tylko do implementacji określonego interfejsu. Jeżeli zdefiniujemy go jako interfejs znacznikowy, będziemy mogli rozszerzać jedyny interfejs, do którego się on odnosi, gwarantując, że wszystkie oznaczone typy są również podtypami jedynego interfejsu, do którego się odnosi.

Można się spierać, że interfejs `Set` jest takim **ograniczonym interfejsem znacznikowym**. Jest stosowany tylko do podtypów `Collection`, ale nie dodaje żadnych innych metod poza zdefiniowanymi przez `Collection`. Nie jest on uważany powszechnie za interfejs znacznikowy, ponieważ precyzuje założenia kilku metod `Collection`, w tym `add`, `equals` i `hashCode`. Jednak łatwo sobie wyobrazić interfejs znacznikowy, który jest stosowany tylko do podtypów pewnego interfejsu i który **nie** precyzuje założeń żadnej z metod interfejsu. Takie interfejsy znacznikowe mogą opisywać pełny niezmiennik całego obiektu lub wskazywać, że obiekty mogą być przetwarzane przez metodę pewnej innej klasy (w sposób, w jaki interfejs `Serializable` oznacza, że obiekt może być przetwarzany przez `ObjectOutputStream`).

Główną zaletą adnotacji znacznikowych względem interfejsów znacznikowych jest to, że są częścią większego mechanizmu adnotacji. Dzięki temu adnotacje znacznikowe pozwalają na zachowanie spójności we frameworkach pozwalających na adnotowanie różnych elementów programu.

Kiedy powinniśmy korzystać z adnotacji znacznikowych, a kiedy z interfejsów znacznikowych? Jasne jest, że konieczne jest używanie adnotacji, jeżeli znacznik odnosi się do dowolnego elementu programu innego niż klasa lub interfejs, ponieważ tylko klasy i interfejsy mogą implementować lub rozszerzać interfejs. Jeżeli znacznik odnosi się wyłącznie do klasy lub interfejsu, należy odpowiedzieć sobie na pytanie, czy potrzebuję napisać jedną lub więcej metod akceptujących wyłącznie obiekty z tym znacznikiem? Jeżeli tak, powinniśmy użyć interfejsu znacznikowego zamiast adnotacji. Umożliwia to użycie interfejsu jako typu parametru metody, co pozwoli skorzystać z kontroli typów w czasie kompilacji. Jeżeli odpowiedź na pierwsze pytanie brzmi „nie”, zadajemy sobie drugie — czy chcemy ograniczyć użycie tego znacznika do elementów określonego interfejsu na zawsze. Jeżeli tak, sensowne jest zdefiniowanie znacznika jako podinterfejsu tego interfejsu. Jeżeli odpowiedź na oba pytania brzmi „nie”, powinniśmy prawdopodobnie użyć adnotacji.

Podsumujmy. Interfejsy znacznikowe i adnotacje znacznikowe mają swoje zastosowania. Jeżeli chcemy zdefiniować typ, który nie ma skojarzonych żadnych metod, pozwoli na to typ znacznikowy. Jeżeli chcemy oznaczyć elementy programu inne niż klasy i interfejsy, pozostawiając możliwość dodania w przyszłości do znacznika większej ilości informacji, albo dodać znacznik do frameworku intensywnie wykorzystującego typ adnotacyjny, to właściwym wyborem jest adnotacja znacznikowa. **Jeżeli okazuje się, że piszemy typ znacznikowy, którego elementem docelowym jest `ElementType.TYPE`, warto poświęcić nieco czasu na zorientowanie się, czy faktycznie powinien to być typ znacznikowy, czy też lepszy byłby interfejs znacznikowy**.

W pewnym sensie jest to odwrotność porady z tematu 22.: „Jeżeli nie potrzebujesz typu, nie używaj interfejsu”. Można powiedzieć, że jest to odpowiednik wskazówki: „Jeżeli chcesz zdefiniować typ, użyj interfejsu”.

Skorowidz

E

F

G

H

I

J

K

L

M

R

S

Ś

T

U

W

Z

Poznaj najlepsze praktyki programowania z użyciem platformy Java

Java jest konsekwentnie udoskonalana i unowocześniana dzięki zaangażowaniu wielu ludzi. Wieloparadygmatowość nowoczesnych wersji tego języka sprawia, że stosowanie najlepszych praktyk w coraz większym stopniu determinuje jakość kodu. Obecnie napisanie kodu, który prawidłowo działa i może być łatwo zrozumiany przez innych programistów, nie wystarczy — należy zbudować program w taki sposób, aby można było go łatwo modyfikować. Jako że Java to dziś obszerna i złożona platforma konieczne stało się uaktualnienie najlepszych praktyk.

Ta książka jest kolejnym, trzecim wydaniem klasycznego podręcznika programowania w Javie. Poszczególne rozdziały zostały gruntownie przejrzane, zaktualizowane i wzbogacone o sporo ważnych treści. Znalazło się tu wiele wartościowych porad dotyczących organizowania kodu w taki sposób, aby stał się przejrzysty, co ułatwi przyszłe modyfikacje i usprawnienia. Poza takimi zagadnieniami jak programowanie zorientowane obiektowo czy korzystanie z różnych typów obszernie omówiono stosowanie lambd i strumieni oraz zasady obsługi wyjątków a także korzystania ze współbieżności i z serializacji. Książka składa się z dziewięćdziesięciu tematów pogrupowanych w dwanaście rozdziałów. Taki układ pozwala na szybkie odnalezienie potrzebnego rozwiązania.

W książce między innymi:

- **interfejsy funkcyjne, wyrażenia lambda, referencje do metod oraz strumienie**
- **metody domyślne i statyczne w interfejsach**
- **wnioskowanie typów**
- **korzystanie z @SafeVarargs**
- **instrukcja try z zasobami**
- **nowe elementy bibliotek Javy**

Dr Joshua Bloch wykłada na Uniwersytecie Carnegie Mellon. Wcześniej był głównym architektem Javy w firmie Google, wyróżniającym się inżynierem w firmie Sun Microsystems i starszym projektantem systemów w Transarc. Kierował projektowaniem i implementacją wielu funkcjonalności platformy Java, w tym rozszerzenia języka w JDK 5.0 oraz Collection Framework. Jego książki są uważane za lekturę obowiązkową każdego, kto chce pisać dobry i wydajny kod w Javie.

Java: jakość kodu, efektywność działania i przyjemność programowania!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9896-2
9 788328 398962
Cena: 99,00 zł