helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

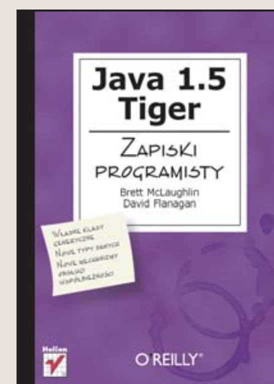
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Java 1.5 Tiger. Zapiski programisty



Autorzy: Brett McLaughlin, David Flanagan
Tłumaczenie: Jaromir Senczyk
ISBN: 83-246-0048-5
Tytuł oryginału: Java 1.5 Tiger A Developers Notebook
Format: B5, stron: 224

Zobacz, co ma do zaoferowania najnowsza wersja Javy

- Zdefiniuj własne klasy generyczne
- Zastosuj w programach nowe typy danych
- Poznaj nowe mechanizmy obsługi współbieżności

Najnowsza wersja języka i środowiska Java, nosząca nazwę Tiger, to nie aktualizacja, ale prawie nowy język programowania. Wprowadzono do niej ponad sto poprawek, zmian i udoskonaleń oraz nowe biblioteki i interfejsy programistyczne. Java 1.5 Tiger oferuje programistom dziesiątki nowych możliwości. Nowa specyfikacja języka jest ogromnym tomiskiem, którego lektura znuży nawet największego fanatyka Javy. Kontakt z najnowszym wcieleniem tego popularnego języka programowania najlepiej zacząć od poznania tego, co faktycznie jest w nim nowe.

„Java 1.5 Tiger. Notatnik programisty” to książka, zawierająca notatki prawdziwych fachowców, którzy analizowali nową wersję Javy jeszcze przed jej publiczną prezentacją. Przedstawia wszystkie nowości Javy 1.5 zilustrowane ponad pięćdziesięcioma przykładami kodu źródłowego. Czytając ją, poznasz zasady stosowania generyczności, zrozumiesz działanie mechanizmów automatycznego opakowywania i rozpakowywania, nauczysz się korzystać z nowych sposobów formatowania tekstów i opanujesz posługiwanie się typami wyliczeniowymi i adnotacjami.

- Generyczność i typy parametryczne
- Tworzenie i stosowanie typów wyliczeniowych
- Automatyczne opakowywanie i rozpakowywanie
- Wykorzystywanie list argumentów o zmiennej długości
- Instrukcja for/in
- Obsługa i synchronizacja wątków

Poznaj i ujarzmij siłę „tygrysa”

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

Spis treści

Seria „Zapiski programisty" **7**

Wstęp **13**

Rozdział 1. Co nowego? **21**

Tablice 22

Kolejki 25

Kolejki uporządkowane i komparatory 28

Przesłanianie typów zwracanych 30

Wykorzystanie zalet Unicode 32

Klasa StringBuilder 34

Rozdział 2. Generyczność **37**

Stosowanie list bezpiecznych typologicznie 38

Stosowanie map bezpiecznych typologicznie 41

Przeglądanie typów parametrycznych 42

Akceptowanie typów parametrycznych jako argumentów 45

Zwracanie typów parametrycznych 46

Stosowanie typów parametrycznych jako parametrów typu 47

Ostrzeżenia lint 48

Generyczność i konwersje typów 49

Stosowanie wzorców typów 53

Tworzenie typów generycznych 55
Ograniczanie parametrów typu 56

Rozdział 3. Typy wyliczeniowe 59

Tworzenie typu wyliczeniowego 60
Typy wyliczeniowe rozwijane w miejscu deklaracji 66
Przeglądanie typów wyliczeniowych 67
Typy wyliczeniowe w instrukcjach wyboru 68
Mapy typów wyliczeniowych 73
Zbiory typów wyliczeniowych 75
Dodawanie metod do typów wyliczeniowych 78
Implementacja interfejsów i typy wyliczeniowe 82
Ciała klas specyficzne dla poszczególnych wartości 83
Samodzielne definiowanie typów wyliczeniowych 86
Rozszerzanie typu wyliczeniowego 87

Rozdział 4. Automatyczne opakowywanie i rozpakowywanie 89

Przekształcanie wartości typów podstawowych w obiekty typów opakowujących 90
Przekształcanie obiektów typów opakowujących w wartości typów podstawowych 92
Operacje zwiększania i zmniejszania dla typów opakowujących 93
Boolean i boolean 94
Wyrażenia warunkowe i rozpakowywanie 95
Instrukcje sterujące i rozpakowywanie 97
Wybór metody przeciążonej 98

Rozdział 5. Listy argumentów o zmiennej długości 101

Tworzenie list argumentów o zmiennej długości 103
Przeglądanie list argumentów o zmiennej długości 106
Listy argumentów o zerowej długości 108
Obiekty jako argumenty zamiast typów podstawowych 110
Zapobieganie automatycznej konwersji tablic 112

Rozdział 6. Adnotacje ... 115
Stosowanie standardowych typów adnotacji ... 116
Adnotowanie przesłaniania metody ... 119
Adnotowanie przestarzałej metody ... 121
Wyłączanie ostrzeżeń ... 123
Tworzenie własnych typów adnotacji ... 125
Adnotowanie adnotacji ... 128
Definiowanie elementu docelowego dla typu adnotacji ... 129
Zachowywanie typu adnotacji ... 130
Dokumentowanie typów adnotacji ... 131
Konfigurowanie dziedziczenia adnotacji ... 134
Refleksje i adnotacje ... 137

Rozdział 7. Instrukcja for/in ... 143
Pozbywanie się iteratorów ... 144
Przeglądanie tablic ... 147
Przeglądanie kolekcji ... 148
Zapobieganie niepotrzebnemu rzutowaniu ... 150
Adaptacja klas do pracy z pętlą for/in ... 152
Określanie pozycji na liście i wartości zmiennej ... 157
Usuwanie elementów listy w pętli for/in ... 158

Rozdział 8. Import składowych statycznych ... 161
Importowanie składowych statycznych ... 162
Stosowanie wzorców podczas importowania ... 164
Importowanie wartości typów wyliczeniowych ... 165
Importowanie wielu składowych o tej samej nazwie ... 167
Przesłanianie importu składowych statycznych ... 169

Rozdział 9. Formatowanie ... 171
Tworzenie instancji klasy Formatter ... 171
Formatowanie danych wyjściowych ... 172

Stosowanie metody format() 178
Stosowanie metody printf() 180

Rozdział 10. Wątki 181

Obsługa wyjątków 182
Kolekcje i wątki 186
Stosowanie kolejek blokujących 189
Określanie limitów czasu dla blokowania 193
Separacja logiki wątku od logiki wykonania 195
Stosowanie interfejsu ExecutorService 197
Stosowanie obiektów Callable 199
Wykonywanie zadań bez użycia interfejsu ExecutorService 201
Planowanie zadań 202
Zaawansowana synchronizacja 205
Typy atomowe 207
Blokowanie a synchronizacja 209

Skorowidz 215

ROZDZIAŁ 4

Automatyczne opakowywanie i rozpakowywanie

W tym rozdziale:

- ✓ Przekształcanie wartości typów podstawowych w obiekty typów opakowujących
- ✓ Przekształcanie obiektów typów opakowujących w wartości typów podstawowych
- ✓ Operacje zwiększania i zmniejszania dla typów opakowujących
- ✓ `Boolean` i `boolean`
- ✓ Wyrażenia warunkowe i rozpakowywanie
- ✓ Instrukcje sterujące i rozpakowywanie
- ✓ Wybór metody przeciążonej

Gdy rozpoczynamy naukę programowania w języku Java, jedna z pierwszych lekcji zawsze dotyczy obiektów. Można powiedzieć, że klasa `java.lang.Object` stanowi kamień węgielny języka Java. Praktycznie 99% kodu posługuje się tą klasą lub jej klasami pochodnymi. Jednak pozostały 1% może nam sprawiać sporo kłopotów, wymagając ciągłego przekształcania obiektów na wartości typów podstawowych i z powrotem.

Do typów podstawowych w języku Java należą int, short, char i tym podobne. Ich wartości nie są obiektami. W rezultacie w języku Java udostępniono dodatkowo klasy opakowujące, takie jak Integer, Short czy Character, które stanowią obiektowe wersje typów podstawowych. Jednak ciągłe przekształcanie obiektów w wartości typów podstawowych i z powrotem bywa irytujące. Nagle okazuje się, że kod programu zostaje zdominowany przez wywołania metod w rodzaju intValue().

W wersji Tiger dostarczono rozwiązania tego problemu za pomocą dwóch nowych mechanizmów konwersji: opakowywania i rozpakowywania. W obu przypadkach nazwy wspomnianych mechanizmów można dodatkowo opatrzyć przymiotnikiem „automatyczne".

Przekształcanie wartości typów podstawowych w obiekty typów opakowujących

Literały w języku Java zawsze są jednego z typów podstawowych. Na przykład wartość 0 jest typu int i może zostać zamieniona w obiekt za pomocą następującego kodu:

```
Integer i = new Integer(0);
```

W wersji Tiger wyeliminowano konieczność tworzenia takiego kodu.

Jak to osiągnąć?

Nareszcie możemy zapomnieć o ręcznych konwersjach i pozwolić wykonywać je maszynie wirtualnej Java:

```
Integer i = 0;
```

Zamieni ona automatycznie wartość typu podstawowego w obiekt opakowujący. Ta sama konwersja odbywa się też w przypadku zmiennych typów podstawowych:

```
int foo = 0;
Integer integer = foo;
```

Aby przekonać się o przydatności tego rozwiązania, należy spróbować skompilować powyższy fragment za pomocą jednej z wersji języka

wcześniejszych od wersji Tiger. Uzyskamy wtedy dość dziwne komunikaty o błędach:

```
compile-1.4:
     [echo] Compiling all Java files...
    [javac] Compiling 1 source file to classes
    [javac] src\com\oreilly\tiger\ch04\Conversion.java:6: incompatible
types
    [javac] found    : int
    [javac] required: java.lang.Integer
    [javac]     Integer i = 0;
    [javac]                 ^
    [javac] src\com\oreilly\tiger\ch04\ConversionTester.java:9:
incompatible types
    [javac] found   : int
    [javac] required: java.lang.Integer
    [javac]     Integer integer = foo;
    [javac]                       ^
    [javac] 2 errors
```

Przykłady zamieszczone w tym rozdziale możemy skompilować z opcją -source 1.4, używając celu compile-1.4.

Błędy te znikną w „magiczny" sposób w wersji Tiger, gdy zastosujemy opcję kompilacji `-source 1.5`.

Jak to działa?

Wartości typów podstawowych zostają automatycznie opakowane. Przez opakowanie rozumiemy konwersję typu podstawowego do odpowiadającego mu obiektowego typu opakowującego: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` lub `Double`. Ponieważ konwersja ta zachodzi automatycznie, nazywamy ją *automatycznym opakowywaniem*.

W języku Java, oprócz opakowania wartości, może również dojść do konwersji poszerzającej typ:

```
Number n = 0.0f;
```

W tym przypadku literał zostanie najpierw opakowany obiektem typu `Float`, którego typ zostanie następnie poszerzony do `Number`.

Istnieje możliwość jawnego zażądania konwersji opakowującej w sposób przypominający rzutowanie. Łatwiej jednak pozostawić wykonanie tego zadania maszynie wirtualnej.

Specyfikacja języka Java informuje, że pewne wartości typów prostych są zawsze opakowywane za pomocą tych samych obiektów opakowujących, które nie mogą być modyfikowane. Obiekty te są przechowywane w buforze i używane wielokrotnie. Do wartości traktowanych w ten sposób należą `true` i `false`, wszystkie wartości typu byte, wartości typów short i int należące do przedziału od `-128` do `127` oraz wszystkie znaki

typu char o kodach od \u0000 do \u007F. Jednak ponieważ wszystko to odbywa się automatycznie, jest to właściwie tylko szczegół implementacji i nie musimy się tym przejmować.

Przekształcanie obiektów typów opakowujących w wartości typów podstawowych

Oprócz konwersji wartości typów podstawowych do typów opakowujących, w wersji Tiger dokonywana jest również konwersja w kierunku przeciwnym. Podobnie jak opakowywanie, również rozpakowywanie nie wymaga interwencji programisty.

Jak to osiągnąć?

Poniżej przedstawiony został prosty przykład kodu, w którym zachodzi opakowywanie i rozpakowywanie bez żadnych dodatkowych instrukcji:

```
// Opakowywanie
int foo = 0;
Integer integer = foo;

// Rozpakowywanie
int bar = integer;

Integer counter = 1;      // opakowywanie
int counter2 = counter;   // rozpakowywanie
```

Zupełnie proste, prawda?

A co…

…z wartością null? Ponieważ wartość null dozwolona jest dla dowolnego obiektu i tym samym dowolnego typu opakowującego, poniższy kod jest poprawny:

```
Integer i = null;
int j = i;
```

Podstawienie wartości null do i jest dozwolone. Następnie i zostaje rozpakowane do j. Ponieważ null nie jest poprawną wartością typu podstawowego, zostaje wygenerowany wyjątek NullPointerException.

Operacje zwiększania i zmniejszania dla typów opakowujących

Jeśli zastanowimy się nad konsekwencjami opakowywania i rozpakowywania, dojdziemy do wniosku, że są one daleko idące. Na przykład każda operacja dostępna dla typu podstawowego powinna być również dostępna dla typu opakowującego i na odwrót. Przykładem mogą być operacje zwiększania i zmniejszania (`++` i `--`), które obecnie działają także w przypadku typów opakowujących.

Jak to osiągnąć?

Nie jest to trudne:

```
Integer counter = 1;
while (true) {
  System.out.printf("Iteracja %d%n", counter++);
  if (counter > 1000) break;

}
```

Zmienna `counter` traktowana jest w tym przykładzie jak zmienna typu `int`.

Jak to działa?

W rzeczywistości w naszym przykładzie zdarzyło się więcej, niż można by przypuszczać. Weźmy następujący jego fragment:

```
counter++
```

Przypomnijmy, że `counter` jest typu `Integer`. Obiekt `counter` został więc najpierw automatycznie rozpakowany do wartości typu `int`, który wymagany jest przez operator `++`.

WSKAZÓWKA

Zauważmy, że operator `++` nie został przystosowany do działania z typami opakowującymi. Kod ten działa jedynie dzięki automatycznemu rozpakowywaniu.

Dopiero po rozpakowaniu wykonana zostaje operacja zwiększania. Następnie nowa wartość musi zostać z powrotem umieszczona w obiekcie

`counter`, co wymaga wykonania operacji opakowywania. A wszystko to odbywa się niezauważalnie, w ułamku sekundy!

W naszym przykładzie również w przypadku porównania obiektu `counter` z literałem `1000` zachodzi automatyczne opakowywanie.

Boolean i boolean

Typ `boolean` wyróżnia się wśród typów podstawowych dostępnością operatorów logicznych, takich jak `!` (negacja), `||` (alternatywa) i `&&` (koniunkcja). Dzięki automatycznemu rozpakowywaniu możemy stosować je także dla obiektów typu `Boolean`.

Jak to osiągnąć?

Za każdym razem, gdy w wyrażeniu zawierającym operator `!`, `||` lub `&&` pojawia się obiekt `Boolean`, zostaje on automatycznie rozpakowany do wartości typu `boolean`, która zostaje użyta do wyznaczenia wartości całego wyrażenia.

```
Boolean case1 = true;
Boolean case2 = true;
boolean case3 = false;

Boolean result = (case1 || case2) && case3;
```

Wartość typu `boolean` będąca wynikiem wyrażenia zostanie w tym przypadku z powrotem opakowana za pomocą obiektu typu `Boolean`.

Wartości typów podstawowych zostają opakowane w porównaniach za pomocą operatora ==. W przypadku operatorów takich jak <, >= i tym podobnych, typy opakowujące zostają rozpakowane do typów podstawowych.

A co...

...z bezpośrednimi porównaniami obiektów? Działają one w taki sam sposób jak dotąd:

```
Integer i1 = 256;
Integer i2 = 256;

if (i1 == i2) System.out.println("Równe!");
else System.out.println("Różne!");
```

Wynikiem wykonania tego kodu, przynajmniej przez moją maszynę wirtualną, jest komunikat `"Różne!"`. W tym przykładzie nie zachodzi operacja rozpakowywanie. Literał `256` zostaje opakowany za pomocą

dwóch różnych obiektów typu Integer (przynajmniej przez moją maszynę), a następnie obiekty te zostają porównane za pomocą operatora ==. Wynikiem porównania jest false, ponieważ porównywane są dwie różne instancje mające różne adresy w pamięci. Ponieważ po obu stronach operatora == znajdują się obiekty, nie zachodzi operacja rozpakowywania.

OSTRZEŻENIE

Nie należy polegać na wyniku działania tego przykładu, zamieściłem go jedynie dla ilustracji. Inne implementacje maszyny wirtualnej Java mogą optymalizować kod i stworzyć jedną, wspólną instancję dla obu literałów. W takim przypadku wynikiem wyrażenia zawierającego operator porównania == będzie wartość logiczna true.

Ale uwaga! Przypomnijmy (z podrozdziału „Przekształcanie wartości typów podstawowych w obiekty typów opakowujących"), że niektóre wartości typów podstawowych są opakowywane za pomocą obiektów, które nie mogą być modyfikowane. Dlatego też wynik działania poniższego kodu może stanowić pewną niespodziankę:

```
Integer i1 = 100;
Integer i2 = 100;

if (i1 == i2) System.out.println("Równe!");
else System.out.println("Różne!");
```

W tym przypadku wynikiem tym będzie komunikat "Równe!". Przypomnijmy, że wartości typu int z zakresu od -128 do 127 są opakowywane właśnie za pomocą stałych, niemodyfikowalnych obiektów. Maszyna wirtualna używa więc tego samego obiektu dla i1 i i2. W rezultacie wynikiem porównania jest wartość true. Należy o tym pamiętać, ponieważ przeoczenia tego faktu mogą powodować trudne do znalezienia błędy.

Wyrażenia warunkowe i rozpakowywanie

Jedną z dziwniejszych możliwości języka Java jest operator warunkowy zwany również *operatorem ternarnym*. Operator ten stanowi wersję instrukcji warunkowej if/else reprezentowaną za pomocą znaku ?.

Ponieważ operator ten wymaga wyznaczenia wartości wyrażeń, również jest związany z mechanizmem automatycznego opakowywania wprowadzonym w wersji Tiger. Operatora tego możemy używać dla różnych typów.

Jak to osiągnąć?

Oto składnia operatora warunkowego:

```
[wyrażenie warunkowe] ? [wyrażenie1] : [wyrażenie2]
```

Jeśli wynikiem wyrażenia *[wyrażenie warunkowe]* jest wartość logiczna `true`, opracowane zostanie *[wyrażenie1]*; w przeciwnym razie *[wyrażenie2]*. W wersjach poprzedzających wersję Tiger wynikiem wyrażenia *[wyrażenie warunkowe]* musiała być wartość typu `boolean`. Było to mało wygodne w sytuacji, gdy jakaś metoda zwracała obiekty typu opakowującego `Boolean` lub obiekt tego typu był wynikiem wyrażenia. W wersji Tiger nie stanowi to już problemu, ponieważ operator działa dla wartości będących wynikiem rozpakowania obiektów `Boolean`:

```
Boolean arriving = false;
Boolean late = true;

System.out.println(arriving ? (late ? "Najwyższy czas!" : "Cześć!") :
                              (late ? "Pośpiesz się!" : "Do
zobaczenia!"));
```

Jak to działa?

Omawiając działanie operatora ternarnego, zarówno w wersji Java 1.4, jak i Tiger, warto wspomnieć o pewnych dodatkowych szczegółach. W wersjach poprzedzających wersję Tiger, *[wyrażenie1]* i *[wyrażenie2]* musiały być tego samego typu lub musiała istnieć możliwość przypisania jednego drugiemu. Czyli na przykład oba wyrażenia musiały być typu `String` lub jedno typu `int`, a drugie typu `float` (ponieważ wartość typu `int` może zostać przekształcona na typ `float`). W wersji Tiger ograniczenia te są nieco luźniejsze ze względu na możliwość zastosowania rozpakowywania. Jedno lub oba wyrażenia mogą zostać rozpakowane, i dlatego jedno może być na przykład typu `Integer`, a drugie typu `Float`. W tym przypadku rozpakowane zostaną *oba* wyrażenia, a powstała wartość typu `int` zostanie rozszerzona do typu `float`. Wyni-

kiem wyrażenia będzie więc wartość typu float, która *nie* zostanie z powrotem opakowana w typ Float.

Kolejną dodatkową właściwością wprowadzoną w wersji Tiger jest automatyczne rzutowanie referencji na ich wspólny typ. Wyjaśnia to poniższy przykład:

```
String s = "pewien";
StringBuffer sb = new StringBuffer("tekst");
boolean mutable = true;

CharSequence cs = mutable ? sb : s;
```

Przykład ten pochodzi z piątego wydania książki Java in a Nutshell (O'Reilly).

W wersjach poprzedzających wersję Tiger podczas kompilacji tego kodu wystąpił by błąd, ponieważ sb (typu StringBuffer) i s (typu String) nie mogą być do siebie przypisywane. Ponieważ oba typy implementują interfejs CharSequence, kod ten uda się jednak skompilować, jeśli zastosujemy rzutowanie:

```
CharSequence cs = mutable ? (CharSequence)sb : (CharSequence)s;
```

W wersji Tiger można użyć dowolnego *wspólnego* typu dla obu wyrażeń. W tym przypadku CharSequence spełnia to wymaganie i może być poprawnym typem wyniku wyrażenia.

Efektem takiego rozwiązania jest to, że każde dwa obiekty zawsze mają wspólny typ java.lang.Object, wobec czego wynik działania operatora ternarnego na wyrażeniach, które nie są typów podstawowych, może zostać przypisany do obiektu typu java.lang.Object.

Z punktu widzenia technicznego możliwość ta związana jest z obsługą generyczności w wersji Tiger, ale wydaje mi się, że lepiej było wspomnieć o niej w tym miejscu. Generyczność została omówiona szczegółowo w rozdziale 2.

Instrukcje sterujące i rozpakowywanie

W języku Java istnieje kilka instrukcji sterujących, których argumentem jest wartość typu boolean lub wyrażenie, którego wynik jest typu boolean. Fakt, że w instrukcjach tych mogą teraz występować również obiekty typu Boolean, nie powinien już stanowić zaskoczenia. Dodatkowo instrukcja wyboru switch akceptuje szereg nowych typów.

Jak to osiągnąć?

Wprowadzona w wersji Tiger możliwość automatycznego rozpakowywania obiektów typu Boolean do wartości typu boolean używana jest

także w przypadku instrukcji `if/else`, `while` i `do`. Poniższy przykład nie wymaga szczegółowych objaśnień:

```
Boolean arriving = false;
Boolean late = true;

Integer peopleInRoom = 0;
int maxCapacity = 100;
boolean timeToLeave = false;
while (peopleInRoom < maxCapacity) {
  if (arriving) {
    System.out.println("Miło Cię widzieć.");
    peopleInRoom++;
  } else {
    peopleInRoom--;
  }
  if (timeToLeave) {
    do {
      System.out.printf("Osoba numer %d musi opuścić pokój!%n",
peopleInRoom);
      peopleInRoom--;
    } while (peopleInRoom > 0);
  }
}
```

Uruchamiając ten przykład, należy pamiętać, że wykonuje on nieskończoną pętlę.

W przykładzie tym zachodzi wiele operacji opakowywania i rozpakowywania w różnych instrukcjach sterujących. Warto dokładnie przeanalizować ich działanie.

Inną instrukcją, która zyskała na wprowadzeniu automatycznego rozpakowywania, jest instrukcja wyboru `switch`. We wcześniejszych wersjach języka Java akceptowała ona jedynie wartości typów `int`, `short`, `char` i `byte`. Oprócz typów wyliczeniowych wprowadzonych w wersji Tiger obsługuje ona dzięki rozpakowywaniu teraz również obiekty typów `Integer`, `Short`, `Char` i `Byte`.

Typy wyliczeniowe omówiłem w rozdziale 3.

Wybór metody przeciążonej

Opakowywanie i rozpakowywanie stanowią rozwiązanie wielu typowych problemów (lub przynajmniej czynią życie programisty wygodniejszym). Równocześnie jednak same mogą wprowadzać pewne problemy, zwłaszcza w obszarze *wyboru metody*. Wybór metody jest procesem, w którym kompilator języka Java ustala metodę, jaką należy wywołać. Należy pamiętać, że opakowywanie i rozpakowywanie mają wpływ na przebieg tego procesu.

Jak to osiągnąć?

W zwykłej sytuacji wybór metody w języku Java odbywa się na podstawie nazwy metody. Jednak w przypadku, gdy nazwa metody jest przeciążona, konieczne staje się wykonanie dodatkowego kroku. Wersja przeciążonej metody wybierana jest na podstawie dopasowania listy argumentów. Jeśli dopasowanie to się nie uda, kompilator zgłosi błąd. Brzmi prosto, prawda? Weźmy pod uwagę poniższe dwie metody:

```
public void doSomething(double num);

public void doSomething(Integer num);
```

Załóżmy teraz, że metodę `doSomething()` wywołaliśmy w następujący sposób:

```
int foo = 1;
doSomething(foo);
```

Która metoda zostanie wywołana? We wcześniejszych wersjach języka Java jej ustalenie nie sprawi nam kłopotu. Wartość typu `int` zostanie rozszerzona do typu `double` i wywołana zostanie metoda `doSomething` `➥(double num)`. Jednak w wersji Tiger wydaje się, że nastąpi opakowanie wartości typu `int` i wywołana zostanie metoda `doSomething(int num)`. Chociaż taki sposób działania wydaje się sensowny, jednak tak się nie stanie.

Wyobraźmy sobie sytuację, w której napisalibyśmy powyższy program, skompilowali go i przetestowali w języku Java 1.4, a następnie w wersji Tiger. Okazałoby się, że działa on różnie w zależności od wersji języka. Dlatego też obowiązuje zasada, że w wersji Tiger *zawsze* zostanie wybrana ta sama metoda, która zostałaby wybrana w wersji 1.4. W praktyce powinniśmy unikać takiego korzystania z przeciążania jak w powyższym przykładzie. Jeśli będziemy dokładnie specyfikować listy argumentów metod, problem ten przestanie istnieć.

Jak to działa?

Ze względu na wspomnianą zasadę w wersji Tiger wybór metody odbywa się w trzech etapach:

Listy argumentów o zmiennej długości omówione zostaną w rozdziale 5.

1. Kompilator próbuje określić właściwą metodę bez stosowania operacji opakowywania i rozpakowywania, a także zmiennych list argumentów. Na skutek wykonania tego etapu wybranie zostania taka sama metoda jak w przypadku Java 1.4.
2. Jeśli pierwszy etap zawiedzie, kompilator próbuje ponownie znaleźć odpowiednią metodę, ale tym razem stosując opakowywanie i rozpakowywanie. Metody mające listy argumentów o zmiennej długości na tym etapie nie są brane pod uwagę.
3. Jeśli zawiedzie również drugi etap, kompilator podejmuje ostatnią próbę, uwzględniając opakowywanie i rozpakowywanie, a także listy argumentów o zmiennej długości.

Zastosowanie powyższych zasad zapewnia spójność działania ze wcześniejszymi wersjami języka Java.