Tomasz Nurkiewicz

Java 8 – najbardziej rewolucyjna wersja w historii

Częstotliwość wydawania nowych wersji języka Java pozostawia wiele do życzenia. JDK 6 oraz 7 były też pewnym rozczarowaniem ze względu na małą innowacyjność i opóźnienia. Java 8, spodziewana już w połowie marca, ma szansę radykalnie odmienić dotychczasowe postrzeganie tej dość leciwej platformy. Wyrażenia lambda, współbieżne kolekcje czy zupełnie nowe API do obsługi czasu przybliżają ten jeden z najpopularniejszych języków programowania do konkurencji.

Java 8 wprowadza najwięcej nowości od czasu piątej edycji wydanej blisko 10 lat temu. W tym artykule przyjrzymy się najciekawszym i najbardziej rewolucyjnym zmianom. Java z nudnego i rozwlekłego języka ma szansę ponownie stać się popularnym, wygodnym i ciekawym narzędziem dla nowych aplikacji. Java nadal ustępuje nowoczesnym językom na JVM, niemniej jednak znacząco skraca ten dystans.

DOMYŚLNE METODY W INTERFEJSACH

Do czasu Javy 8 istniał jasny podział na interfejsy - jedynie deklarujące metody bez żadnej implementacji - oraz klasy abstrakcyjne. Aby zachować zgodność wsteczną, Oracle postanowiło wprowadzić tzw. metody domyślne w interfejsach. Teraz możemy dołączyć do dowolnej metody interfejsu także implementację, którą klasa implementująca dany interfejs może, ale nie musi przesłonić (ang. *override*):

Listing 1. Przykład metod domyślnych w interfejsach

```
public interface Encrypter {

   byte[] encode(byte[] bytes);

   default byte[] encodeStr(
      String s, Charset charset) {

     return this.encode(s.getBytes(charset));
   }

   default byte[] encodeChars(
      char[] chars, Charset charset) {

     final String s = String.valueOf(chars);
     return this.encodeStr(s, charset);
   }
 }
```

W powyższym przykładzie interfejs deklaruje trzy metody `encode*()`, ale dwie z nich posiadają implementacje opatrzone modyfikatorem `default` (znanym z poprzednich wersji Javy w innym kontekście). Klasa implementująca taki interfejs musi zaimplementować tylko jedną metodę, a nie wszystkie trzy:

Listing 2. Klasa implementująca interfejs z metodami domyślnymi

```
public class RotEncrypter implements Encrypter {
  @Override
  public byte[] encode(byte[] b) {
    final byte[] result = new byte[b.length];
    for (int i = 0; i < b.length; ++i) {
      result[i] = (byte) (b[i] + 13);
    }
    return result;
  }

}
```

Klasa `RotEncrypter` implementuje tylko jedną metodę, a pozostałe są zaimplementowane na jej bazie. Oczywiście nadal istnieje możliwość przesłonięcia pozostałych metod interfejsu – ale nie ma już takiego obowiązku. W poprzednich wersjach Javy z reguły w takim przypadku tworzyliśmy zwykły interfejs `Encrypter` oraz pomocniczą klasę (np. `abstract class AbstractEncrypter implements Encrypter`) implementującą wszystkie opcjonalne metody. Trudno nie odnieść wrażenia, że interfejsom w Javie 8 dużo bliżej do cech (ang. *traits*) w Scali, chociaż oczywiście nie są one aż tak rozbudowane i elastyczne.

Powyżej przedstawiono jedno z możliwych zastosowań metod domyślnych. Ich pierwotnym przeznaczeniem było jednak zachowanie wstecznej zgodności przy dodawaniu nowych metod d o istniejących interfejsów. W przeszłości operacja taka natychmiast skutkowała błędem kompilacji (w końcu stary kod nie mógł implementować nieistniejącej natenczas metody), teraz wystarczy dostarczyć domyślną implementację nowych metod. Z tej cechy skorzystali twórcy, pisząc nową metodę `java.util.Collection.stream()` – o której w szczegółach za chwilę.

Uważni czytelnicy zastanawiają się zapewne, jak Java radzi sobie z konfliktami – gdy ma do dyspozycji więcej niż jedną implementację pochodzącą z wielu interfejsów bazowych, które implementuje?

Listing 3. Konflikt domyślnych metod - kod nie kompiluje się!

```
interface Car {
  default void fuel() {}
}

interface Boat {
  default void fuel() {}
}

//Nie kompiluje się!
class Hovercraft implements Car, Boat {

}
```

Klasa `Hovercraft` implementuje dwa interfejsy, z których oba dostarczają nie tylko deklarację, ale również implementację metody `fuel()`. Prowadzi to do konfliktu, którego kompilator nie jest w stanie rozwiązać. Na szczęście istnieje składnia pozwalająca explicite wywołać dowolną spośród skonfliktowanych metod lub dostarczyć trzecią, niezależną wersję:

Listing 4. Rozwiązywanie konfliktów metod domyślnych

```
class Hovercraft implements Car, Boat {
  @Override public void fuel() {
    Car.super.fuel();
  }
}
```