Java 16 update

Spis treści

Java 16 update

Java 16 została wydana w marcu 2021 i jest wersją **non-LTS**. Poniżej omówimy niektóre funkcjonalności udostępnione w tym wydaniu. Przy aktualizacji wersji Javy często poprawianych jest o wiele więcej funkcjonalności i dodawanych o wiele więcej klas lub metod niż te, które wymieniamy tutaj. W obrębie tych materiałów poruszamy tylko te kwestie, które są adekwatne do naszego poziomu zaawansowania jako Java developerów.

Records

W końcu możemy przejść do omówienia zagadnienia Rekordów (*Records*). Powód powstania tego typu rozwiązania wynikał z tego, że Javie często tworzymy klasy, które nie realizują logiki, nie wykonują obliczeń, służą tylko do przetrzymywania informacji. Klasy takie są bardzo powtarzalne, posiadają pola, gettery, settery, konstruktory, nadpisują metody `equals()` i `hashCode()` oraz `toString()`. Rekordy mają uprościć tworzenie takich struktur. Rekord można rozumieć jak taką uproszczoną klasę, która skraca zapis w sytuacji opisanej powyżej. Z innej strony możemy na nie spojrzeć jak na `enum`, który też jest specyficznym rodzajem klasy.

Często w kodzie powielamy tworzenie klas, które są bardzo powtarzalne i służą do przetrzymywania informacji. Inaczej można to sobie wyobrazić jak takie nośniki danych, które są przesyłane pomiędzy innymi obiektami, które faktycznie realizują jakąś logikę biznesową. Przy tworzeniu takich nośników danych często powielamy następujące czynności:

- Tworzymy klasę.
- Dodajemy pola.
- Definiujemy konstruktor.
- Generujemy gettery i settery.
- Nadpisujemy metody `equals()` i `hashCode()`.
- Nadpisujemy metodę `toString()`.

Dowiedzieliśmy się już, że większość z tych czynności możemy wykonać generując kod przy wykorzystaniu `Intellij`. Wadą tego podejścia jest natomiast to, że jeżeli dodamy w takiej klasie np. pole, to musimy regenerować wcześniej wspomniane konstruktory i metody ponownie.

Później poznaliśmy też Project Lombok, który pozwala nam szybciej i łatwiej rozwiązywać wspomniane problemy. Z drugiej strony pojawia się pytanie, czy na prawdę twórcy Javy nie mogliby rozwiązać jakoś tego problemu sami? Nawet stosując **Lombok** trzeba pamiętać, żeby wygenerować `equals()` i `hashCode()` oraz `toString()`. Możemy wtedy stosować adnotacje `@Data` albo `@Value`, ale nadal daje to pole to pomyłki. Lepiej byłoby w takim razie mieć wspomniane aspekty zapewnione w klasie domyślnie.

Jednakże jak przeczytamy jaki jest cel tej zmiany JEP 395 (*JEP* - Java Enhancement Proposal) to znajdziemy taki fragment opisu:

> While it is superficially tempting to treat records as primarily being about boilerplate reduction, we instead choose a more semantic goal: modeling data as data. (If the semantics are right, the boilerplate will take care of itself.) It should be easy, clear, and concise to declare shallowly-immutable, well-behaved nominal data aggregates.

Co oznacza, że główną motywacją nie było pozbycie się **boilerplate** kodu, czyli takiego co trzeba go dużo pisać, ale za wiele to on nie robi. Motywacją było odróżnienie, które klasy służą za nośniki danych. W tym celu wprowadzono **Records**.

W celu dalszego omawiania tej funkcjonalności wprowadźmy na początku klasę `Person`:

Klasa Person

```
public class Person {

    private final String name;
    private final String surname;
    private final Long age;

    public Person(final String name, final String surname, final Long age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
```

```
    }

    public Long getAge() {
        return age;
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        final Person person = (Person) o;

        if (!Objects.equals(name, person.name)) return false;
        if (!Objects.equals(surname, person.surname)) return false;
        return Objects.equals(age, person.age);
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (surname != null ? surname.hashCode() : 0);
        result = 31 * result + (age != null ? age.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
            "name='" + name + '\'' +
            ", surname='" + surname + '\'' +
            ", age=" + age +
            '}';
    }
}
```

Trochę dużo jak na fakt, że potrzebujemy z tej klasy tak naprawdę 3 linijki, te zaznaczone poniżej:

```
public class Person {

    private final String name;
    private final String surname;
    private final Long age;

    // reszta jest nam w sumie niepotrzebna
}
```

Możesz zwrócić uwagę, że `Intellij` nawet sam Ci podpowiada, żeby zamienić klasę `Person` na **record** `Person`.

Rekord Person

```
public record Person(String name, String surname, Long age) { }
```

Zwróć uwagę, że "pola w klasie" zostały tutaj zadeklarowane tak jak parametry w metodzie. Jest to sposób zapisu specyficzny dla rekordu. Czyli pola nie są definiowane w ciele rekordu, tylko w nawiasach

zaraz za jego nazwą.

Rekordy mają kilka cech wspólnych z klasami:

- mają określoną paczkę,
- możemy do nich dodawać importy,
- mogą być `public`, albo `package-private`,
- mogą mieć zdefiniowane pola, tyle, że zapisuje się to w inny sposób,
- mogą mieć kilka konstruktorów,
- możemy w nich definiować własne metody,
- mogą implementować interfejsy,
- mogą mieć pola i metody statyczne,
- mogą używać typów generycznych.

Zdefiniowany wcześniej rekord `Person`:

- posiada 3 pola `private final`: *String name*, *String surname* oraz *Long age*,
- posiada dostępny konstruktor ze wszystkimi 3 parametrami,
- posiada metody, które pozwalają pobrać wartość pól (celowo nie nazywam ich getterami, zaraz zobaczysz dlaczego),
- posiada nadpisane metody `equals()` i `hashCode()`
- posiada nadpisaną metodę `toString()`.

Rekordy możemy traktować jak obiekty **immutable**, co oznacza, że nie będziemy mogli zmieniać ich stanu. Dlatego właśnie nie posiadają zdefiniowanych setterów.

```
public class Runner {

    public static void main(String[] args) {
        Person person = new Person("Karol", "Zajavka", 915L);
        String surname = person.surname();
        System.out.println(surname);
    }
}
```

Zwróć uwagę, że metoda, która pozwala na pobranie wartości pola `surname`, nie ma w nazwie słówka `get`. W angielskiej nomenklaturze spotkamy się ze stwierdzeniem **accessor method**, tłumacząc na nasz, możemy to nazywać metodą dostępową. Zanim przejdziemy do omówienia kolejnych możliwości rekordu, należy jeszcze zaznaczyć jedną rzecz.

javap

Zanim przejdziemy do sedna, powiedzmy sobie o narzędziu `javap`. Narzędzie `javap` jest używane żeby pozyskać informacje o klasie lub interfejsie. Inna nazwa tego narzędzia to **Java Disassembler**. Jeżeli użylibyśmy narzędzia `javap` na interfejsie `java.util.List`:

```
javap java.util.List
```

To zobaczymy taki wynik na ekranie:

```
Compiled from "List.java"
public interface java.util.List<E> extends java.util.Collection<E> {
  public abstract int size();
    // pozostałe metody z klasy ...
  public static <E> java.util.List<E> of(E...);
  public static <E> java.util.List<E> copyOf(java.util.Collection<? extends E>);
}
```

Co oznacza, że stosując to narzędzie możemy zobaczyć w jaki sposób skonstruowana jest dana struktura, klasa, enum albo interfejs.

Wywołajmy teraz poniższą komendę i zobacz, jaki otrzymasz wynik na ekranie. Widać jakie klasy implementuje `ArrayList` i jakie interfejsy implementuje.

```
javap java.util.ArrayList
```

Stwórzmy teraz enum `Color` o poniższej treści:

```
public enum Color {
    REB,
    BLUE,
    GREEN
}
```

I wywołajmy ponownie polecenie `javap` na skompilowanym pliku `Color.class`. Na ekranie zobaczysz poniższy zapis. Z drugiej strony znajdując się w klasie `Color` możesz wcisnąć skrót *CTRL + U* i zostaniesz przekierowany/-a do klasy `java.lang.Enum`.

```
Compiled from "Color.java"
public final class Color extends java.lang.Enum<Color> {
  public static final Color REB;
  public static final Color BLUE;
  public static final Color GREEN;
  public static Color[] values();
  public static Color valueOf(java.lang.String);
  static {};
}
```

Po co Ci ta informacja? Dzięki zastosowaniu narzędzia `javap` możemy zobaczyć więcej szczegółów dotyczących konkretnej struktury. Spróbujmy zatem wywołać to samo polecenie na skompilowanej klasie `Person.class`:

```
javap Person.class
```

Na ekranie zostanie wydrukowane:

```
Compiled from "Person.java"
final class Person extends java.lang.Record {
  Person(java.lang.String, java.lang.String, java.lang.Long);
  public final java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public java.lang.String name();
  public java.lang.String surname();
  public java.lang.Long age();
}
```

Dlatego właśnie napisałem wcześniej, że **record** może być rozumiane analogicznie jak **enum**. I jedna i druga struktura jest klasą `final`, oznacza to, że nie możemy dziedziczyć ani z enuma, ani z recordu. Widzimy też, że domyślnie mamy dostępne metody `equals()` i `hashCode()` oraz `toString()`. Do tego widzimy accessor methods, czyli `name()`, `surname()` oraz `age()`. Jasno teraz widać, że jest to struktura **immutable**.

Co możemy zrobić w rekordach

Fields

Rekordy pozwalają nam na definiowanie pól w nawiasach okrągłych zaraz po nazwie rekordu. Oznacza to, że nie możemy wykorzystać takiego zapisu:

```
record Person(String name, String surname, Long age) {
    private final String pesel; ①
}
```

① Błąd kompilacji

Możemy natomiast definiować w rekordach stałe:

```
record Person(String name, String surname, Long age) {

    public static final String COMMON_NAME = "COMMON_NAME"; ①

    private static final Integer COUNTER; ②
    static {
        COUNTER = 100;
    }
}
```

① Możemy definiować stałe w ten sposób.

② Możemy również stosować statyczne bloki inicjalizacyjne.

Methods

Rekordy pozwalają na definiowanie własnych metod:

```
record Person(String name, String surname, Long age) {

    boolean isMature() {
        return age > 18;
    }
}
```

Rekordy mogą również implementować interfejsy:

Interfejs Name

```
public interface Name {

    String name();
}
```

Rekord Person

```
record Person(String name, String surname, Long age) implements Name { ①

    @Override
    public String surname() {
        return "Zwracam surname po swojemu, bo nie podoba mi się ten z rekordu: " + surname; ②
    }
}
```

① Implementujemy interfejs `Name`, co oznacza, że musimy implementować metodę `name()`. Rekord zapewnia nam dostępną metodę `name()`, co oznacza, że jest ona automatycznie implementowana.

② Możemy nadpisać metody dostarczane przez record i dodać swoją własną implementację.

Generics

Możemy stosować typy generyczne w rekordach. Przykład:

Rekord Person

```
record Person<T>(String name, String surname, T generic) {}
```

I uruchomimy teraz poniższy fragment kodu:

```
Person<Function<String, Integer>> person = new Person<>("Karol", "Zajavka", String::length); ①
Integer len = person.generic().apply(person.surname()); ②
System.out.println(len);
```

① Referencja określa typ generyczny, gdzie w tym przypadku jest to *Function<String, Integer>*. Jako trzeci argument wywołania konstruktora przekazujemy **method reference** będące implementacją. Równie dobrze możemy w tym miejscu napisać lambdę.

② Najpierw zwracamy wartość z metody `generic()` i z racji, że jest to typ `Function` to możemy wywołać metodę `apply()`.

Na ekranie wydrukowana zostanie długość słowa `Zajavka`.

Constructors

Rekord automatycznie generuje konstruktor ze wszystkimi określonymi polami i taki konstruktor nazywa się konstruktorem **kompaktowym** (*compact constructor*). Możliwe jest natomiast dodanie swojego własnego konstruktora, ale pierwsza linijka, która zostanie napisana musi stanowić wywołanie konstruktora kompaktowego.

Klasa Runner

```
public class Runner {

    public static void main(String[] args) {
        Person person = new Person("Karol", "Zajavka", 3L);
        Long age = person.age();
        System.out.println(age);
    }
}
```

```
record Person(String name, String surname, Long age) {

    Person { ①
        System.out.println("Calling compact constructor: " + age); ②
    }

    public Person(final String surname) {
        this(null, surname, null); ③
        System.out.println("Calling other constructor"); ④
    }
}
```

① Konstruktor kompaktowy możemy nadpisać, chociażby po to, żeby dodać jakiś swój kod. Zwróć uwagę, że nie podajemy tutaj parametrów w nawiasach. Możemy podać parametry w nawiasach (wtedy taki konstruktor nazywa się **canonical** (*kanoniczny*)) i napisać konstruktor ze wszystkimi parametrami zamiast konstruktora kompaktowego. Ale po co, jak można korzystać z krótszej składni. 😉

② Jeżeli dodamy jakiś kod do konstruktora kompaktowego, to możemy w nim korzystać z zainicjowanych wartości. Zwróć uwagę, że jak uruchomisz teraz kod z klasy `Runner` to na ekranie wydrukuje się zdanie z zainicjowaną wartością.

③ Pierwsza linijka konstruktora kanonicznego musi wołać konstruktor kompaktowy. Nie możemy kolejnością zamienić linijek oznaczonych 3 i 4.

④ Tutaj możemy dodać nasz kod.

Podsumowując

Podsumowując wszystko, co zostało pokazane, **do czego przydadzą nam się rekordy?**.

Możemy dzięki nim stworzyć obiekty **immutable**, w których możemy skupić się tylko na aspekcie przechowywania danych. Nie musimy wtedy ręcznie pisać albo generować całego **boilerplate** kodu, dzięki czemu nasze struktury stają się mniejsze i czytelniejsze. Możliwe, że w przyszłości spowoduje to

ograniczenie potrzeby stosowania projektu **Lombok**, ale z drugiej strony **Lombok** dostarcza nam o wiele więcej funkcji niż rekordy wprowadzone w Java 16. Przykładowo mogą to być adnotacje `@With` albo `@Builder`. Kolejnym przykładem jest adnotacja `@Data`, która pozwala nam stworzyć klasę mutowalną i daje nam wspomniane gettery, settery, `equals()` i `hashCode()` oraz `toString()`. Czas pokaże jak dalej wszystko będzie się zmieniało, ale na pewno warto jest przyswoić sobie tę konstrukcję. ☺

Static nested class

Przed Java 16, taki zapis kończył się błędem kompilacji:

```
public class Outer {

    class Inner {

        public static int a = 2; ①

        public void call() {
            System.out.println(a);
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.call();
    }
}
```

① W tej linijce dostawaliśmy błąd kompilacji w Java < 16.

W Javie 16 taki zapis stał się poprawny i możemy deklarować zmienne statyczne w *Inner Class*. Przyczyną tej zmiany było wprowadzenie rekordów. Wyjaśnione zostało to w ten sposób (źródło):

> It is currently specified to be a compile-time error if an inner class declares a member that is explicitly or implicitly static, unless the member is a constant variable. This means that, for example, an inner class cannot declare a record class member, since nested record classes are implicitly static.
>
> We relax this restriction in order to allow an inner class to declare members that are either explicitly or implicitly static. In particular, this allows an inner class to declare a static member that is a record class.

Pattern Matching instanceof

Nowością, która została wprowadzona do Java 16 jest `Pattern Matching instanceof`. Tak jak zostało wspomniane wcześniej, w Java 16 nie jest to już **preview feature**, tylko **standard feature**. Nowy zapis został zaproponowany, żeby uprościć taką sytuację:

```
void method(Animal animal) {
    if (animal instanceof Cat) {
```

```
        Cat cat = (Cat) animal;
        System.out.println("Cat is happy: " + cat.mrr());
    }
}
```

Przypomnę tylko, że takie sprawdzenie stosuje się, żebyśmy byli zabezpieczeni przed wyrzuceniem wyjątku `ClassCastException`, jeżeli okaże się, że staramy się rzutować nieprawidłowy typ obiektu. Czyli przykładowo, gdy pod referencją `animal` znajdowałby się obiekt typu `Dog`. `Pattern Matching instanceof` ma za zadanie uprościć ten zapis:

```
void method(Animal animal) {
    if (animal instanceof Cat cat) { ①
        System.out.println("Cat is happy: " + cat.mrr());
    }
}
```

① Zmienna `cat` zostanie zainicjowana tylko, jeżeli spełniony zostanie warunek `instanceof`.

Stream

Java 16 wprowadziła również pewne usprawnienia do Streamów.

```
public class StreamExample {

    public static void main(String[] args) {
        List<Integer> list = List.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
        List<Integer> collect1 = list.stream()
            .filter(e -> e > 6)
            .collect(Collectors.toList()); ①

        List<Integer> collect2 = list.stream()
            .filter(e -> e > 6)
            .toList(); ②
    }
}
```

① Od Javy 16 ten zapis został uproszczony.

② Możemy to teraz zapisać w ten sposób. Należy pamiętać, że nie mamy w tym przypadku gwarancji jaka implementacja listy zostanie wykorzystana. Najlepiej jest jednak pamiętać, że lista taka jest **immutable**, czyli nie możemy dodawać do niej wartości.

Zwracam tutaj uwagę, że nie została dodana analogiczna metoda `toSet()`.

Day Period

Do możliwych sposobów formatowania daty została wprowadzona kolejna możliwość. Od Javy 16 możemy używać symbolu `B`. Przykład:

```
LocalTime time1 = LocalTime.parse("07:00:00");
LocalTime time2 = LocalTime.parse("15:00:00");
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("hh:mm:ss B");
System.out.println(formatter.format(time1)); ①
System.out.println(formatter.format(time2)); ②
```

① Na ekranie zostanie wydrukowane: *07:00:00 rano*

② Na ekranie zostanie wydrukowane: *03:00:00 po południu*

Można zauważyć, że literka `B` jest alternatywą do formatu `am/pm`, a wiadomość drukowana na ekranie jest w powyższym przykładzie zależna od domyślnego `Locale`.

jpackage

Narzędzie `jpackage` jest narzędziem konsolowym, które możemy uruchomić analogicznie do `jshell`. Służy ono do tego, żeby przygotować plik instalacyjny z wytworzonym przez nas oprogramowaniem. Dzięki temu narzędziu możemy przygotować plik dedykowany pod konkretne systemy operacyjne:

- Windows: `msi` lub `exe`,
- Linux: `deb` lub `rpm`,
- macOS: `pkg` lub `dmg`.

Plik taki zawierałby wszystkie potrzebne zależności do uruchomienia takiej aplikacji. Dzięki tak przygotowanej paczce, będziemy mogli zainstalować przygotowaną aplikację na naszym systemie operacyjnym. Tematyka ta jest poruszana bardziej jako ciekawostka, nie będziemy się w ten temat dalej zagłębiać.

Sealed Classes (preview)

W Java 16 funkcjonalność ta nadal została utrzymana jako **preview feature** i została opublikowana w Java 17. Dlatego do jej omówienia przejdziemy, gdy będziemy omawiać Java 17.

Podsumowanie

Przypomnę, że przy aktualizacji wersji Javy często poprawianych jest o wiele więcej funkcjonalności i dodawanych o wiele więcej klas lub metod niż te, które wymieniamy tutaj. Z kolejnymi wersjami wprowadzane są również rozmaite poprawki lub usprawnienia w samym działaniu JVM albo przykładowo Garbage Collectora (w tym przypadku mogą to być, chociażby różne algorytmy, o których działanie oparty jest GC). Zmianom mogą ulegać również kwestie dotyczące zarządzania pamięcią. Oprócz tego kolejne wersje Javy mogą również wprowadzać dodatkowe narzędzia, które programista może wykorzystywać w swojej pracy. Do tego poprawkom mogą podlegać istniejące implementacje metod. W obrębie tych materiałów poruszamy tylko te kwestie, które są adekwatne do naszego poziomu zaawansowania jako Java developerów. Nie poruszamy też zagadnień, co do których twórcy Zajavki uznali, że z naszego punktu widzenia zmiany te nie są aż tak istotne i lepiej poświęcić ten sam czas na skupienie się na dalszych zagadnieniach.

Jeżeli natomiast interesuje Cię, jakie jeszcze zmiany są wprowadzane z każdą wersją — wystarczy, że wpiszesz w Google np. "Java 16 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę JDK 16. Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. ☺