Java 15 update

Spis treści

Java 15 update

Java 15 została wydana we wrześniu 2020 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.

Text Blocks

Text Blocks zostały wprowadzone w Java 15 na stałe. Służą one do tego, żeby w przejrzysty sposób deklarować **multiline String**, czyli Stringi wielolinijkowe. Najprostszy przykład **Text Block** wygląda w ten sposób:

```
// ten zapis jest prawidłowy
String example = """
     zajavka!""";

// i ten również
String example = """
    zajavka!
    """;

// tak nie można
String example = """zajavka!""";
```

Text Block rozpoczyna się potrójnym cudzysłowem i nową linijką. Kończy się również potrójnym cudzysłowem. Dzięki takim blokom możemy w bardziej czytelny sposób deklarować bloki tekstu, które wymagają od nas zapisu na kilka linijek. Mogą to być, chociażby zapytania **SQL**.

Dużą zaletą stosowania **Test Block** jest możliwość zachowania wcięć w tekście. Spójrz na przykład

poniżej:

```
public class Example {

    public static void main(String[] args) {
        System.out.println(gimmeHtml());
    }

    static String gimmeHtml() {
        return """
            <html> ①
                <body>
                    <div>zajavka!</div>
                </body>

            </html>""";
    }
}
```

① Zwróć uwagę, że przy wydruku wcięcia są uwzględniane z od najbardziej wysuniętego na lewo fragmentu tekstu. Inaczej mówiąc, pierwsze 12 spacji, które są zapisane po lewej stronie od zaznaczonej linijki, zostaną pominięte.

Na ekranie zostanie to wydrukowane w ten sposób:

```
<html>
    <body>
        <div>zajavka!</div>
    </body>
①
</html>
```

① Pusta linijka jest również uwzględniania poprawnie

Jeżeli natomiast napiszemy kod w ten sposób (w dwóch linijkach zostały dodane 2 spacje):

```
public class Example {

    public static void main(String[] args) {
        System.out.println(gimmeHtml());
    }

    static String gimmeHtml() {
        return """
              <html>
                <body>
                    <div>zajavka!</div>
                  </body>
            </html>"""; ①
    }
}
```

① Zwróć uwagę, że przy wydruku wcięcia są uwzględniane z od najbardziej wysuniętego na lewo fragmentu tekstu. Inaczej mówiąc, pierwsze 12 spacji, które są zapisane po lewej stronie od zaznaczonej linijki, zostaną pominięte.

Rezultat będzie wyglądał w ten sposób:

```
  <html> ①
    <body>
        <div>zajavka!</div>
      </body> ②
</html>
```

① Tutaj są dodane dwie spacje na początku.

② I tutaj również są dodane dwie spacje na początku.

Dla porównania spójrz na przykład tego samego bloku tekstu, ale zapisanego "po staremu":

```
static String gimmeHtmlOld() {
    return "<html>\n"
        + "    <body>\n"
        + "        <div>zajavka!</div>\n"
        + "    </body>\n"
        + "\n"
        + "</html>";
}
```

escaping

Pamiętasz może, że znaki specjalne w Stringach należy escapować (**escaping**)? Przypomnijmy sobie na czym to polegało.

Jeżeli chcemy zapisać prosty `String`, to możemy to zrobić w ten sposób:

```
String simpleString = "Witam na zajavce!";
```

Wiemy na tym etapie, że Stringi zaczynają się i kończą przy wykorzystaniu cudzysłowu. Co w takim razie zrobić, jeżeli chcemy wykorzystać cudzysłów w zdaniu, ale jednocześnie chcemy powiedzieć Javie, że ten cudzysłów nie kończy Stringa. Jeżeli zapisalibyśmy to tak:

```
String simpleString = "Witam na "zajavce!"";
```

To Java czytając linijkę od lewej strony, napotka najpierw cudzysłów rozpoczynający, a potem kończący String, czyli przeczyta zdanie: *"Witam na"*. Następnie pojawi się zapis *zajavce!*, który nie wiadomo, czym jest. Potem odczytany zostanie kolejny String: *""*. Rozumując w ten sposób widać, że musimy w jakiś sposób oznaczyć, który cudzysłów jest "specjalny" i ma nie kończyć Stringa. Robimy to stosując znaczek `\` - backslash. Nazywa się to **character escaping** (*ucieczka postaci* 😸?). Natomiast backslash w zestawieniu z jakimś znakiem nazywany jest **control sequence**. Przykład:

```
String simpleString = "Witam na \"zajavce!\"";
```

Powoduje to powstanie kolejnego problemu. Wyobraź sobie, że będziemy teraz chcieli zapisać taki

`String`:

```
String simpleString = "Mój ulubiony katalog to: C:\zajavka";
```

Zwróć uwagę, że dostaniemy błąd kompilacji. Dlaczego? Bo `\` jest znakiem zarezerowawnym do escapowania. Inaczej mówiąc, backslash jest zarezerwowany do poprzedzania "znaków specjalnych". Nie może wystąpić pojedynczo. Jeżeli chcemy, żeby `String` poprawnie używał backslasha, należy go escapować, czyli dodać backslash przed backslashem. Przykład:

```
String simpleString = "Mój ulubiony katalog to: C:\\zajavka";
```

Jakie **control sequence** możemy wyróżnić?

- `\t` - Tab.
- `\'` - Apostrof. W Stringu możemy go używać normalnie, ten zapis przyda się, gdy będziemy chcieli zapisać `char a = '\'';`.
- `\"` - Cudzysłów.
- `\\` - Backslash.
- `\b` - Backspace, w ten sposób możemy skasować znak w Stringu.
- `\n` - Przejście do nowej linijki.
- `\r` - Powrót karetki (*carriage return*). Spróbuj wywołać taki fragment kodu: `System.out.println("Hej\rzajavka!");` i zwróć uwagę, że następuje powrót do początku linijki i wypisanie reszty tekstu. Możemy to rozumieć jak powrót karetki w maszynach do pisania. (wikipedia: *karetka* – przesuwająca się ruchem posuwisto-zwrotnym część maszyny do pisania i drukarki).
- `\f` - Form feed - można to traktować jako zaszłość, więc nie będziemy się na tym skupiać.

Skoro już ten temat został poruszony, należy jeszcze wspomnieć o tym, że każdy system operacyjny w inny sposób dokonuje zakończenia linijki i przejścia do nowej.

- Unix i nowe MacOS - `\n`,
- Windows - `\r\n`,
- Starsze systemy Macintosh - `\r`.

Ma to o tyle znaczenie, że jeżeli będziemy pisali ręcznie w Stringu znaki `\n` lub `\r` to teoretycznie może to doprowadzić do powstania nieprzewidywalnych błędów na różnych systemach operacyjnych, dlatego dla bezpieczeństwa zaleca się stosowanie takich zapisów:

```
String str1 = "Hello " + System.lineSeparator() + "zajavka!"; ①
String str2 = String.format("Hello %nzajavka!"); ②
```

Pierwszy zapis jest trochę długi, dlatego można posłużyć się zapisem nr 2. Stosowanie znaku `%n` jest niezależne od systemu operacyjnego, Java sama sobie dalej z tym poradzi. Wiadomo, że często dla uproszczenia stosuje się zwyczajnie `\n` w Stringach, ale teoretycznie może to prowadzić do błędów.

Jeszcze jedna ciekawostka. Możemy zapisywać Stringi, posługując się bezpośrednio znakami **Unicode** w poniższy sposób. Znaki są zapisane w systemie szesnastkowym. Spróbuj uruchomić poniższy kod i zobacz, co zostanie wydrukowane na ekranie:

```
System.out.println("\u0043\u0068\u0069\u006e\u0065\u0073\u0065\u003a \u6bdb\u6cfd\u4e1c");
```

Po co tyle tego, jak my tu o blokach tekstów rozmawiamy?

Żebyśmy mogli teraz wrócić do wyjaśnienia jak escapować **Text Block**. ☺ Stosując **Text Block**, możemy zapisać tekst w poniższy sposób:

```
public class Example {

    public static void main(String[] args) {
        System.out.println(gimme());
    }

    static String gimme() {
        return """
           Oto linijka 1
           "zajavka" ①
           \\ ②
           \""" ③
           """;
    }
}
```

① Nie musimy escapować pojedynczych cudzysłowów.

② Backslash już musimy, bo jak tego nie zrobimy, to nie zostanie on wydrukowany na ekranie.

③ Potrójny cudzysłów musi być escapowany, ale wystarczy tylko jeden backslash.

Na ekranie zostanie wydrukowane:

```
Oto linijka 1
"zajavka"
\
"""
```

spacje

Text Block automatycznie usunie tzw. *trailing spaces*, czyli spacje, które znajdują się na końcu danej linijki. Możemy jednak wymusić, żeby nie były one usuwane, wykorzystując poniższy zapis:

```
static String getEscapedSpaces() {
    return """
        line 1 ①
        line 2       \s ②
        """;
}
```

① Tutaj spacje zostaną usunięte.

② A tutaj już nie.

String.format

Jak zestawić ze sobą zachowanie `String.format()`, które pozwala na określanie parametrów w Stringu z **Text Block**. Można to zrobić w poniższy sposób:

```
public class Example {

    public static void main(String[] args) {
        System.out.println(withParam("abc"));
    }

    static String withParam(String param) {
        return """
            zajavka: %s
            """.formatted(param); ①
    }
}
```

① Metoda `formatted()` wywołana na **Text Block** pozwala na podstawienie parametru zdefiniowanego w `String`.

Pattern Matching instanceof (preview)

Java 15 utrzymała nową koncepcję `instanceof` jako **preview feature**, dlatego nie poruszamy tej tematyki w obrębie Java 15.

Records (preview)

Java 15 utrzymała nową koncepcję rekordów jako **preview feature**, dlatego nie poruszamy tej tematyki w obrębie Java 15.

Sealed Classes (preview)

Java 15 wprowadziła nową koncepcję **sealed classes** (*klasy zapieczętowane*), przy czym jest to **preview feature**. Dzięki **sealed classes** zarówno klasy i interfejsy mogą ograniczyć, które klasy lub interfejsy mogą z nich dziedziczyć albo je implementować. 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 15 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę JDK 15. Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. ☺