# Java 12 update

## Spis treści



## Java 12 update

Java 12 została wydana w marcu 2019 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.

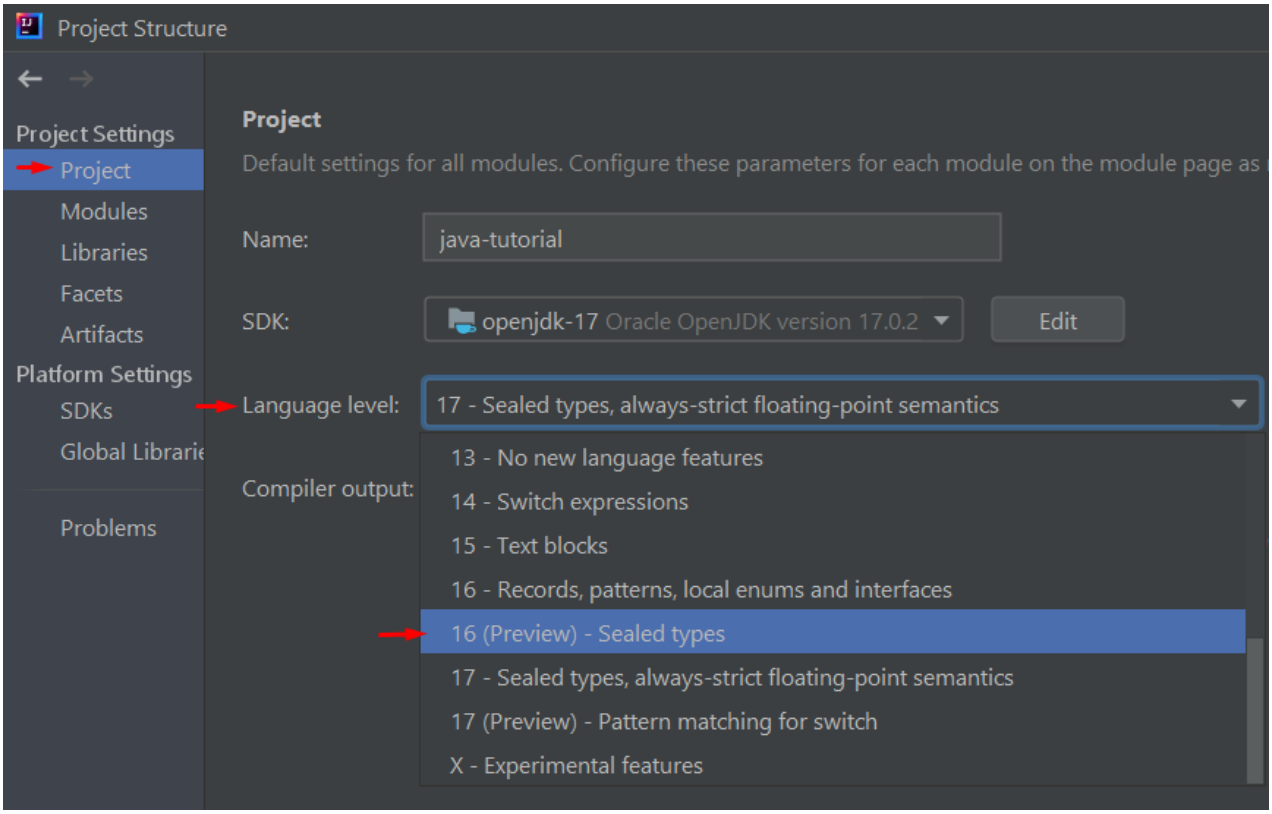
Skoro zmniejszył się odstęp czasu pomiędzy kolejnymi wydaniami Javy, możliwe stało się udostępnianie funkcjonalności, które mogą być przetestowane przez społeczność, żeby zebrać feedback i zdecydować czy daną funkcjonalność zostawić, czy też nie.

Dało to możliwość wprowadzenia **preview features**. Są to funkcjonalności, które są skończone i działają, ale nie ma pewności, że nie zostaną usunięte w kolejnych wydaniach Javy. Oznacza to, że możemy je testować, ale na pewno nie opierać o nie kodu, który ma zostać wdrożony na produkcję i z którego mają korzystać klienci naszej aplikacji. 😀

Jeżeli jakieś zmiany są wprowadzane w trybie standardowym, a nie w trybie **preview** to są one wtedy nazywane **standard feature**.

Zaznaczę też tutaj, że nie będziemy dogłębnie omawiać funkcjonalności na etapie **preview features**. Wrócimy do nich, gdy zostaną udostępnione jako **standard features**. Jako twórcy Zajavki nie chcemy poświęcać czasu na zagadnienia, które mogą zostać finalnie usunięte. Wolimy poświęcić ten sam czas na omówienie kolejnych zagadnień.

Włączyć **preview features** można z poziomu `Intellij`. Należy wejść ustawienia projektu (*CTRL* + *ALT* + *SHIFT* + *S*) i wybrać opcję **Language level** z "preview" w nazwie tak jak na poniższej grafice. Ustawienie **Language level** pozwala na określenie, które funkcje Javy `Intellij` powinien wspierać, a przy których powinien zgłaszać błąd. Wybieramy tutaj zgodność takiego wsparcia z konkretną wersją Javy.




*Obraz 1. Wybranie preview features*

Można to zrobić również w terminalu, np. w ten sposób:

```
javac --release 12 --enable-preview Main.java
```

# Wyrażenia switch (preview)

Java 12 przedstawiła propozycję nowego zapisu wyrażenia `switch`, natomiast było ono dostępne tylko w trybie **preview**.

Propozycja wyrażenia `switch` została zmodyfikowana w Javie 13 (też jako **preview**) i została faktycznie wdrożona dopiero w Javie 14. Wspominam o tym na tym poziomie, żeby móc zaznaczyć czym jest **preview features**, natomiast do faktycznego omówienia zmian w wyrażeniu `switch` przejdziemy przy omówieniu nowości dodanych w Javie 14.

# Streams - Collectors

Java 12 wprowadziła nowy `Collector` - `Collectors.teeing()`. Jeżeli zastanawiasz się, skąd wzięła się ta nazwa, to wpisz w Google grafika plumbing tee.

Wspomniany kolektor pozwala zakończyć `Stream` na "dwa" rezultaty. Potrzebujemy do tego jednak klasy, która będzie w stanie zwrócić rezultat jako parę. Przykład:

*Klasa Pair*

```
@Value
@AllArgsConstructor
```

```
class Pair {
    Long count;
    Integer sum;
}
```

```
public class Example {

    public static void main(String[] args) {
        Pair result = Stream.of(5, 1, 32, 12)
                .collect(Collectors.teeing(
                    Collectors.counting(), ①
                    Collectors.summingInt(element -> element), ②
                    Pair::new ③
                ));

        System.out.println(result); ④
    }
}
```

① Pierwszy kolektor zliczający ilość elementów.

② Drugi kolektor zliczający sumę wartości elementów.

③ Argument wywołania, który definiuje obiekt, który ma przetrzymywać rezultat.

④ Na ekranie zostanie wydrukowane: *Pair(count=4, sum=50)*.

Definicja metody `Collectors.teeing` wygląda w następujący sposób:

```
public static <T, R1, R2, R> Collector<T, ?, R> teeing(
    Collector<? super T, ?, R1> downstream1,
    Collector<? super T, ?, R2> downstream2,
    BiFunction<? super R1, ? super R2, R> merger ①
)
```

① Jeżeli spojrzymy na tę linijkę, to można wywnioskować, że `merger` może być wykorzystane nie tylko do zwrócenia pary wartości, ale do tego, żeby wykorzystać wartości obliczone przez poprzednie dwa kolektory do obliczenia trzeciej wartości.

Możemy zatem równie dobrze napisać taki fragment kodu:

```
public class Example {

    public static void main(String[] args) {
        String result = Stream.of(5, 1, 32, 12)
                .collect(Collectors.teeing(
                    Collectors.counting(),
                    Collectors.summingInt(element -> element),
                    (count, sum) -> "count: " + count + ", sum: " + sum
                ));

        System.out.println(result); ①
    }
}
```

① Na ekranie zostanie wydrukowane: *count: 4, sum: 50*.

# String

Klasa String ponownie otrzymała nowe metody. Omówimy dwie z nich.

## indent

Metoda ta służy do ustawiania wcięć w przekazanym Stringu. Możemy do niej przekazać jeden parametr, który wpłynie na wynik wywołania metody w następujący sposób:

- Jeżeli jego wartość `value == 0` - wcięcie nie ulegnie zmianie.
- Jeżeli jego wartość `value < 0` - wcięcie zostanie zmniejszone o `value` znaków. Tabulator jest traktowany jak jeden znak. Gdy `value` będzie określał większą ilość znaków, niż jest dostępne, usunięte zostaną wszystkie możliwe wcięcia.
- Jeżeli jego wartość `value > 0` - na początku linijki zostanie dodane tyle spacji, ile wynosi `value`.

Spróbuj wykonać przykład poniżej:

```
String input = "###\n\tzajavka\n\t\tzajavka\n\t\tzajavka\n###";

System.out.println(input.indent(0)); ①
System.out.println(input.indent(-1)); ②
System.out.println(input.indent(1)); ③
```

Na ekranie zostanie wydrukowane:

```
###
    zajavka
        zajavka
        zajavka
###①

###
zajavka
    zajavka
    zajavka
###②

 ###
    zajavka
        zajavka
        zajavka
 ###③
```

① Wynik działania kodu oznaczonego cyfrą 1. Wydruk jest taki sam jakbyśmy wydrukowali `input` bez wykorzystania metody `indent()`.

② Wynik działania kodu oznaczonego cyfrą 2. Wszystkie wcięcia zostały zmniejszone o jeden znak. Zwróć uwagę, że tabulacja została potraktowana jak jeden znak.

③ Wynik działania kodu oznaczonego cyfrą 3. Każda linijka zwiększyła swoje wcięcie o jeden znak.

### transform

Metoda `transform()` pozwala na wywołanie funkcji `Function` na podanym Stringu, a wynik tego działania będzie wynikiem działania funkcji. Przykład:

```
String input = "Hej zajavkowicze! Co tam słychać?";
String[] transform = input.transform(s -> s.split(" "));
System.out.println(Arrays.toString(transform));
```

Na ekranie zostanie wydrukowane: *[Hej, zajavkowicze!, Co, tam, słychać?]*.

## Podsumowanie

Wymienione funkcjonalności nie są wszystkimi, jakie zostały wprowadzone w Javie 12. 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 12 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę JDK 12. Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. ☺