

Java 11 update

Spis treści

Java 11 update	1
javac	2
String	2
isBlank()	2
lines()	2
repeat()	3
strip()	3
var i lambda	4
Files	4
Podsumowanie	5

Java 11 update

Java 11 została wydana we wrześniu 2018 i jest wersją **LTS**.



Niektóre z poruszanych zagadnień będą dla Ciebie tylko przypomnieniem, bo poruszaliśmy je już wcześniej. Z jednej strony chcę Ci przez to pokazać, ile już umiesz, a z drugiej strony zaznaczyć, które funkcjonalności były dodawane do języka na przestrzeni kolejnych wydań Javy.

Wcześniej wspomnieliśmy o zmianie polegającej na skróceniu okresów wydań kolejnych wersji. Czyli co 6 miesięcy ma się pojawiać szybka aktualizacja, natomiast **LTS** co 2 lata. Zaznaczam to w tym miejscu, gdyż Java 11 jest **LTS**.

Java 11 jest o tyle istotnym wydaniem Javy, że z tą wersją **Oracle** zmodyfikował (czy zmodyfikowała? W końcu to wyrocznia ☺) swój model wsparcia, pomocy technicznej i licencjonowania. Zmiana licencjonowania zakłada, że komercyjne wykorzystanie Javy od Oracle oznacza konieczność opłaty takiej licencji. Czy to oznacza, że od tego momentu musisz zacząć płacić za uczenie się Javy? **Nie**. Założenie jest takie, że Oracle JDK przestaje być darmowe przy wykorzystaniu komercyjnym produkcyjnie. Oznacza to, że jeżeli będziemy uruchamiali naszą aplikację napisaną w Java w wersji 11 produkcyjnie w oparciu o Oracle JDK, to musimy opłacić licencję.



Pamiętasz, że można pobrać Javę od wielu vendorów? [Link](#)

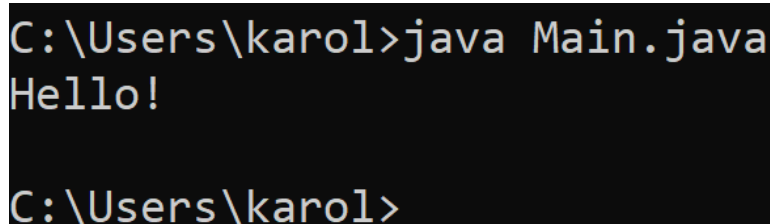
Należy jednak pamiętać, że konieczność opłacania licencji dotyczy **Oracle JDK**. Możemy natomiast spokojnie i z powodzeniem używać **Open JDK** albo JDK od innych vendorów, np. AdoptOpenJDK, IBM, Red Hat itp. Generalnie, jeżeli nie zależy Ci na wsparciu od Oracle (a nie jest Ci to potrzebne o ile nie uruchamiasz aplikacji dla jakiegoś klienta produkcyjnie) to możesz spokojnie korzystać z **OpenJDK**. Zatem tematyka opłacania licencji zacznie Cię interesować dopiero jak będziesz odpowiedzialny/-a za uruchomienie aplikacji napisanej w Java u Twojego klienta na produkcji. Wtedy też zaczniesz się

dowiadując jakie są różnice między produkcyjnym uruchomieniem aplikacji w oparciu o Oracle JDK w porównaniu do OpenJDK. Nie zgłębiamy dalej tego tematu i dla uproszczenia zakładamy, że korzystamy z **OpenJDK**.

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.

javac

Od Javy 11 (jeżeli kompilujesz i uruchamiasz kod w terminalu) nie musisz najpierw wpisywać polecenia **javac**, a potem dopiero uruchamiać skompilowanego kodu przy wykorzystaniu polecenia **java**. Możesz to zrobić w jednym kroku:



```
C:\Users\karol>java Main.java
Hello!

C:\Users\karol>
```

Obraz 1. Komenda java

String

Java 11 wprowadza nowe metody do klasy **String**.

isBlank()

Jak sama nazwa wskazuje, metoda ta sprawdza, czy **String** jest pusty. Przykład:

```
System.out.println("").isBlank()); //true
System.out.println(" ").isBlank()); //true
System.out.println("  ").isBlank()); //true
System.out.println("-").isBlank()); //false
System.out.println("Zajavka").isBlank()); //false
```

lines()

Metoda ta zwraca **Stream<String>**, który operuje na Stringach powstałych z podziału pierwotnego Stringa na linijki. Przykład:

```
String line = "A\nA\nA\nA\nA";
System.out.println(line.lines().count());
```

W przykładzie mamy 5 literek **A**, które są oddzielone przejściem do nowych linii. Oznacza to, że otrzymamy 5 linijek, dlatego na ekranie drukowane jest 5.

repeat()

Metoda powtarza `String`, na którym jest wywołana. Przykład:

```
String line = "zajavka!".repeat(2);
System.out.println(line);
```

Na ekranie jest drukowane: *zajavka!zajavka!*

strip()

Java 11 wprowadza metody takie jak `strip()`, `stripLeading()` oraz `stripTrailing()`. Służą one do usuwania białych znaków. Można zadać sobie pytanie: *Po co, skoro jest `trim()`?* Metody `strip()` to kolejna ewolucja `trim()`, która jest zgodna z `Unicode`. Metoda `strip()` uznaje więcej znaków z tablicy `Unicode` za **białe znaki** (*whitespace*), oznacza to, że występują znaki, których `trim()` nie usunie, a `strip()` już tak.



Jeżeli nie wiesz, czym jest i jak działa system szesnastkowy, zapoznaj się najpierw ze stosowanymi systemami liczbowymi: **dwójkowym**, **ósemkowym**, **dziesiętnym** oraz **szesnastkowym**. Systemu dziesiętnego używamy na co dzień i do niego jesteśmy najbardziej przyzwyczajeni.

Przejdź do tablicy `Unicode`. Jeżeli najedziesz teraz na każdy znaczek, to zobaczysz tutaj oznaczenie, np. `U+0042` | *Dec: 66*. Pierwszy zapis jest zapisany w systemie szesnastkowym. Drugi natomiast w systemie dziesiętnym. Wartości te możemy przypisać do zmiennej `char` i zobaczyć, jaki zostanie wtedy wydrukowany wynik:

```
char char1 = '\u0042';
char char2 = 62;
System.out.println(char1); ①
System.out.println(char2); ②
```

① Na ekranie zostanie wydrukowane: *B*.

② Na ekranie zostanie wydrukowane: *>*.

W ten sposób możemy zapisywać znaki w Javie.

Wracając natomiast do przykładu:

```
char whitespace1 = '\u2000'; ①
char whitespace2 = '\u0020'; ②
String data1 = whitespace1 + "zajavka" + whitespace1;
String data2 = whitespace2 + "zajavka" + whitespace2;

System.out.println("isWhitespace1: " + Character.isWhitespace(whitespace1)); ③
System.out.println("isWhitespace2: " + Character.isWhitespace(whitespace2));
System.out.println("data1.trim#" + data1.trim() + "#"); ④
System.out.println("data1.strip#" + data1.strip() + "#"); ⑤
System.out.println("data2.trim#" + data2.trim() + "#"); ⑥
```

```
System.out.println("data2.strip#" + data2.strip() + "#"); ⑦
```

- ① Pierwszy biały znak.
- ② Drugi biały znak.
- ③ W ten sposób możemy sprawdzić, czy przekazany znak jest rozumiany jako **whitespace**.
- ④ Na ekranie zostanie wydrukowane: `data1.trim#zajavka#`. Oznacza to, że metoda `trim()` **nie** zadziałała.
- ⑤ Na ekranie zostanie wydrukowane: `data1.strip#zajavka#`. Oznacza to, że metoda `strip()` zadziałała.
- ⑥ Na ekranie zostanie wydrukowane: `data2.trim#zajavka#`. Oznacza to, że metoda `trim()` zadziałała.
- ⑦ Na ekranie zostanie wydrukowane: `data2.strip#zajavka#`. Oznacza to, że metoda `strip()` zadziałała.

Jeżeli natomiast chodzi o metody `stripLeading()` oraz `stripTrailing()`:

```
char whitespace1 = 8192;  
String data1 = whitespace1 + "zajavka" + whitespace1;  
System.out.println("leading#" + data1.stripLeading() + "#"); ①  
System.out.println("trailing#" + data1.stripTrailing() + "#"); ②
```

- ① Na ekranie zostanie wydrukowane: `leading#zajavka#`.
- ② Na ekranie zostanie wydrukowane: `trailing#zajavka#`.

var i lambda

W Javie 11 pozwolono używać **var** razem z wyrażeniem **lambda**. Przykład:

```
// tak można  
BiFunction<Integer, String, Boolean> impl = (var e1, var e2) -> (e1.toString() + e2).length() > 2;  
  
// tak nie można  
BiFunction<Integer, String, Boolean> impl = (var e1, e2) -> (e1.toString() + e2).length() > 2;  
  
// tak też nie można  
BiFunction<Integer, String, Boolean> impl = (var e1, String e2) -> (e1.toString() + e2).length() > 2;  
  
// i tak też nie można  
Consumer<String> impl = var p -> System.out.println(p);
```

Czyli możemy wykorzystać **var** w lambdzie pod warunkiem, że korzystamy z nawiasów przy definiowaniu parametrów i wszystkie parametry są poprzedzone słówkiem **var**.

Files

Do klasy `Files` zostały wprowadzone metody `writeString()` oraz `readString()`. Metoda `writeString()` zapisuje `String` do pliku, z domyślnym kodowaniem UTF-8. Metoda `readString()` odczytuje natomiast całą zawartość pliku i przypisuje ją do zmiennej `String`. Przykład:

```
static void call() throws IOException {  
    Path path = Files.createFile(Paths.get("test.txt"));  
    Path created = Files.writeString(path, "Zajavka file content!");  
    System.out.println(created); ①  
    String read = Files.readString(created);  
    System.out.println(read); ②  
}
```

- ① Na ekranie zostanie wydrukowane: *test.txt*.
- ② Na ekranie zostanie wydrukowane: *Zajavka file content!*.

Podsumowanie

Wymienione funkcjonalności nie są wszystkimi, jakie zostały wprowadzone w Javie 11. 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 11 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę [JDK 11](#). Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. 😊