Java 8 update

Spis treści

Truizm o wersjach języków

Języki programowania (nie tylko Java) są narzędziami/projektami, które są cały czas rozwijane (jak nie są, to warto się zastanowić nad tym, czy nie lepiej pomyśleć o innej technologii). Wspomniane zdanie dotyczy nie tylko języków programowania, ale również technologii, bibliotek, frameworków albo innych rozwiązań. Wpływa to na to, że programista/programistka musi w miarę na bieżąco aktualizować swoją wiedzę dotyczącą używanych technologii, bo może to wpływać na tworzenie rozwiązań szybciej, prościej i ogólnie mówiąc lepiej.

Aktualizacja (wydanie nowej wersji) danej technologii jest o tyle ciekawym zagadnieniem, że twórcy danego języka, technologii albo biblioteki muszą podjąć decyzję, jak długo wspierają i utrzymują funkcjonalności, które zostały wprowadzone na przestrzeni lat, przy rozwoju kolejnych wersji technologii. Oznacza to, że mogą mieć podejście, w którym wypuszczając nową wersję języka, (wymyślona wartość 16.2.1) zdecydują się na usunięcie funkcjonalności, które zostały wprowadzone w wersji 6.0.0, bo są już przestarzałe i nie chcą dalej mieć z nimi nic wspólnego 😉. Z drugiej strony (i takie

podejście ma Java), można dodawać cały czas nowe funkcjonalności i jednocześnie starać się cały czas utrzymywać te istniejące, bo przecież ciągle ktoś może z tego korzystać.

Jak przekłada się to na nas jako użytkowników danej technologii, biblioteki albo języka? (Dla uproszczenia będę dalej cały czas odnosił się do języka, ale pamiętaj, że to ma zastosowanie do bibliotek albo frameworków). Musimy mieć świadomość czy korzystając z jakiejś funkcjonalności, może okazać się, że przy przejściu na wyższą wersję języka albo biblioteki, dana funkcjonalność może zostać usunięta.

Powiedzmy przykładowo, że korzystamy z biblioteki `our-example-library` w wersji `1.5.2` i wykorzystujemy u nas w kodzie klasę `ExampleClass` wywołując metodę `exampleMethod()`. W pewnym momencie decydujemy się na zwiększenie wersji tej biblioteki do wersji `2.0.3` i ta konkretna metoda zostaje oznaczona jako **Deprecated** (w skrócie - nie powinniśmy takiej konstrukcji więcej używać, poniżej wspomnę o tym ponownie za moment). Oznacza to, że powinniśmy się postarać przestać wykorzystywać metodę `exampleMethod()` w naszym kodzie, bo niedługo np. może zostać ona usunięta i od tego momentu będzie to dla nas oznaczało błędy kompilacji. Mówiąc od pewnego momentu mam na myśli, że jeżeli zwiększymy wersję omawianej biblioteki np. do `3.0.1` to może się okazać, że nie mamy już klasy `ExampleClass` i nie ma dostępnej metody `exampleMethod()`. Opisane podejście jest jednym z możliwych, czyli twórcy danej biblioteki albo całego języka dochodzą do wniosku, że przy wydaniu kolejnej wersji języka pozbywają się starych funkcjonalności.

A jak się to ma do Javy? Java ma podejście odwrotne. Założeniem Javy jest utrzymywanie **kompatybilności wstecznej** (*backwards compatibility*), czyli twórcy Javy starają się nie usuwać funkcjonalności, które kiedyś zostały już dodane do języka. Przykładowo, zwiększając wersję Javy np. z 9 do 11 nasz kod nadal powinien działać tak samo, bo teoretycznie nie zostają usunięte funkcjonalności z których potencjalnie możemy korzystać. Dodawane są nowe, które możemy wykorzystywać i które pozwalają pisać nasz kod szybciej, prościej i ogólnie mówiąc lepiej. Czy twórcy nie zmienią tego podejścia w przyszłości? Nie wiadomo. Dziś jest tak jak zostało to opisane ☺.

Jakie ma to przełożenie na naukę takiego języka? Jednocześnie będzie to odpowiedź na pytanie czemu Zajavka została skonstruowana w ten sposób, że na samym początku **nie są** poruszane najnowsze nowości w Javie. Zaczynamy uczyć się Javy od jej podstaw, bo wszystkie działające konstrukcje cały czas mogą być wykorzystywane w praktyce. Dopiero na pewnym etapie następuje aktualizacja naszej wiedzy do najnowszej wersji Javy i ten moment przyszedł właśnie teraz.

Zaczęliśmy od "podstaw", gdzie taką najwcześniejszą Javą poniżej której nie zeszliśmy była Java 8 (decyzja twórców zajavki oparta o praktyczne doświadczenie). Po drodze (w kolejnych warsztatach) pojawiały się wzmianki o funkcjonalnościach, które były dodawane w kolejnych wydaniach Javy, ale nie zostało to zgromadzone w jednym miejscu. Ten warsztat jest takim miejscem gdzie zbierzemy informacje o funkcjonalnościach, które były dodawane w kolejnych wersjach Javy. Oczywiście zrobimy to patrząc na te zagadnienia perspektywy osoby, która dopiero uczy się Javy, więc nie będziemy tutaj rozmawiać o wszystkim, powiemy tylko o najistotniejszych funkcjonalnościach.

Należy pamiętać, że ucząc się jakiejkolwiek technologii możemy mieć taki sam dylemat - czyli jaki zakres materiału przyswoić na początek, a jakiego uczyć się potem, bo może się okazać, że niektóre funkcjonalności, których się uczymy są już przestarzałe.

Deprecated

W Javie możliwe jest stosowanie adnotacji **@Deprecated**, aby poinformować wszystkich użytkowników

naszego fragmentu kodu, że z danego fragmentu kodu nie powinniśmy korzystać. Mówiąc fragmentu kodu mam na myśli np. klasę albo metodę. Wystarczy, że np. jakąś klasę oznaczymy tą adnotacją i jest to informacja dla innych developerów, żeby nie korzystali z tej klasy z różnych powodów (tak to określa dokumentacja). Jednym z powodów może być to, że dana klasa lub metoda ma zostać usunięta w kolejnych wersjach biblioteki lub aplikacji. Przykład użycia w kodzie:

```
@Deprecated
public class DeprecatedClass {

    @Deprecated
    void someMethod() {

    }

    void nonDeprecatedMethod() {

    }
}
```

Jeżeli spróbujemy teraz skorzystać z takiej klasy lub metody, to Intellij zareaguje w poniższy sposób:

```
public class Runner {

    public static void main(String[] args) {
        DeprecatedClass deprecatedClass = new DeprecatedClass(); 1
        deprecatedClass.someMethod(); 2
        deprecatedClass.nonDeprecatedMethod(); 3
    }
}
```

Obraz 1. Intellij i wykorzystanie kodu oznaczonego przez @Deprecated

Zwróć uwagę na poszczególne oznaczenia:

1. Tak Intellij wskazuje użycie klasy oznaczonej przez `@Deprecated`,
2. Tak Intellij wskazuje użycie metody oznaczonej przez `@Deprecated`,
3. A tutaj dla porównania mamy normalną metodę, bez oznaczenia `@Deprecated`. Co ciekawe, ta metoda nie jest przekreślona, pomimo że cała klasa jest oznaczona jako `@Deprecated`.

Adnotacji tej możesz używać, żeby pokazać innym deweloperom, że odradzasz korzystanie z oznaczonego fragmentu kodu, np. klasy albo metody. Ten sposób może służyć do tego, żeby poinformować innych użytkowników Twojego kodu, że dany fragment kodu może być usunięty w przyszłości.

Adnotacja `@Deprecated` może również wystąpić w takim wariancie:

```
@Deprecated(forRemoval = true)
```

Oznacza to jawną intencję autora mówiącą, że klasa taka zostanie w przyszłości usunięta. Możemy przy tym dodać kolejny parametr:

```
@Deprecated(since = "1.6.12", forRemoval = true)
```

Dzięki takiemu zapisowi możemy określić wersję aplikacji lub biblioteki, od której dana funkcjonalność zostanie usunięta. Z racji, że ten parametr jest typu `String`, w praktyce można tutaj wpisać datę z przyszłości, kiedy developer planuje usunąć daną klasę. Jest to wtedy informacja dla kolegów/koleżanek z zespołu.

Wersje i zarys historyczny

Zrozumienie metodologii stojącej za nazewnictwem kolejnych wersji Javy i za odstępami czasu pomiędzy kolejnymi wydaniami nie jest proste. Wynika to z tego, że podejście do nazewnictwa i częstotliwości aktualizacji Javy zmieniało się na przestrzeni lat. Gdybyśmy chcieli sprawdzić, jak wyglądało to w poprzednich latach, to możemy przykładowo spojrzeć na ten artykuł na Wikipedii. Zwróć uwagę, że początkowo (w latach 1995 - 2017) częstotliwość kolejnych wydań Javy wynosiła około 2-3 lata. Możesz się domyślić, że zmiany wprowadzane z taką częstotliwością były raczej duże. Można powiedzieć, że stabilizacja nazewnictwa nastąpiła z Javą **Java SE 5**. Jeżeli natomiast spojrzymy na częstotliwość wydań, to w okolicach roku 2018 częstotliwość się zwiększyła.

W czerwcu 2018 została ogłoszona zmiana modelu wydań kolejnych wersji Javy. Wcześniej, co 2-3 lata wydawane było "duże" wydanie Javy (po angielsku zwane **major release**). W praktyce mogło się to przeciągnąć do 3-4 lat. Nowy model zakładał, że co 2-3 lata wydawana jest wersja **LTS** (o znaczeniu tego stwierdzenia przeczytasz niżej). Natomiast co 6 miesięcy miało być wydawane "małe" wydanie Javy. Pierwszą wersja Javy, od której zaczęto stosować to podejście, była Java 11. We wrześniu 2021 zaproponowane zostało skrócenie okresu wydań **major** do 2 lat.

LTS

Każda wersja Javy związana jest z datą wspierania danej wersji przez twórców. Data wspierania (*support date*) określa datę, do której twórcy będą zapewniali poprawki znalezionych błędów, poprawki bezpieczeństwa itp. dla danej wersji wydania Javy. Trzeba również pamiętać, że Java ma wielu Vendorów i każdy Vendor może mieć inny czas wspierania jego wydania JDK. Przykładowo na stronie Oracle znajdziemy tabelkę, w której Oracle umieszcza informacje dotyczące różnych rodzajów wsparcia i różnych dat określających koniec takiego wsparcia. Jeżeli natomiast chcielibyśmy zrozumieć, co oznacza każdy rodzaj wsparcia, to możemy posłużyć się poniższą tabelką ze strony Oracle:

Lifetime Support stages	Features
Premier Support	Provides comprehensive maintenance and software upgrades for your Oracle Database, Oracle Fusion Middleware, and Oracle Applications for five years from the general availability (GA) date.
Extended Support	Puts you in control of your database, middleware, and applications upgrade strategy by providing additional maintenance and upgrades for Oracle Database, Oracle Fusion Middleware, and Oracle Applications for an additional fee.

Lifetime Support stages	Features
Sustaining Support	Maximizes your investment protection by providing maintenance for as long as you use your Oracle software. Features include access to Oracle online support tools, upgrade rights, pre-existing fixes, and assistance from technical support experts.

Co możemy wywnioskować ze wspomnianych tabelek? Mamy wersje **non-LTS** oraz **LTS**. Wersje **LTS** są wspierane o wiele dłużej niż te pozostałe. Wersje **non-LTS** są wspierane do wydania kolejnej wersji **non-LTS**. Jeżeli z drugiej strony pomyślimy o tym, że z perspektywy Vendora prościej i wygodniej jest aktualizować tylko wybrane wersje, a nie wszystkie, które zostały wydane, to układa się to w całość. ☺

Wiedząc o tym, można podjąć świadomą decyzję dotyczącą tego, z której wersji Javy chcemy korzystać w naszym projekcie. Należy zwyczajnie pamiętać o konsekwencjach naszego wyboru.

Skrót **LTS** oznacza *Long-Term Support*. Jest to oznaczenie, które mówi, że dana wersja będzie miała zapewnione o wiele dłuższe wsparcie techniczne niż pozostałe wersje. Wsparcie w tym kontekście będzie oznaczało, że np. przez kilka kolejnych lat dany Vendor będzie zapewniał poprawki do konkretnej wersji, ale **nie** będzie do niej dodawał nowych funkcji. Dzięki oparciu się o wersje **LTS** firmy/projekty korzystające z tych wersji mogą liczyć na większą stabilność i przewidywalność. Patrząc jeszcze raz na tę samą tabelkę, ciekawe jest to, że wersja 8 ma być wspierana dłużej niż wersja 11.

Wersje **LTS** to Java 7, Java 8, Java 11, Java 17 lub Java 21 i w przyszłości kolejne zgodnie z konwencją. Oznacza to, że te wersje będą otrzymywały aktualizacje stabilności, wydajności oraz bezpieczeństwa. Natomiast to dana organizacja może zdecydować, jaką ma politykę, czy korzysta tylko z wersji **LTS**, czy może też z wersji **non-LTS**.

Java 7 i Java 8 pochodzą ze starszego modelu wydań głównych. Wsparcie jest jednak nadal takie samo, ponieważ każda wersja **LTS** Java otrzymuje tylko aktualizacje wydajności, stabilności i bezpieczeństwa. Pamiętajmy jednocześnie, że Java jest **kompatybilna wstecznie**. Czyli jeżeli korzystamy tylko z wersji **LTS** to w końcu i tak będziemy mieli dostęp do nowszych funkcjonalności wprowadzanych w wersjach **non-LTS**, zwyczajnie nastąpi to później, ale dzięki temu zapewniamy większą stabilność.

Okres 6-miesięczny

Okres 6-miesięczny oznacza, że jeżeli twórcy Javy chcą bardzo mocno wprowadzić jakąś nową funkcjonalność, nie muszą się stresować, że nie zdążą w danym oknie czasowym, bo następne wydanie będzie za 6 miesięcy, a nie za 3 lata. Developerzy natomiast (czyli my) otrzymujemy teraz nowe funkcjonalności w mniejszych paczkach i częściej. Jednocześnie model ten zakłada przewidywalność, tzn. wiemy, że za 6 miesięcy dostaniemy coś nowego. Oczywiście nie wszystkie projekty pisane w Javie są oparte za każdym razem o najnowszą wersję Javy. To jest kwestia tego, jaka jest polityka firmy albo można za każdym razem utrzymywać najnowszą wersję Javy, albo skupić się na tym, żeby zawsze mieć wersję **LTS**, czyli Java 8, Java 11, Java 17 lub Java 21.

Dlaczego tak? Każde nowe wydanie wiąże się z potencjalnymi błędami. Niektóre firmy podchodzą do projektów bardziej zachowawczo i wolą korzystać z wersji **LTS** albo poczekać aż najnowsze wydanie się "uleży" i zostaną poprawione "choroby wieku dziecięcego".

Java 8? To już było!

W poprzednich materiałach skupialiśmy się cały czas na pokazywaniu mechanizmów, które są dostępne w Javie, od czasu do czasu zaznaczając wersję Javy, w której dany mechanizm został wprowadzony. Tak jak wspomniałem wcześniej, nigdzie nie zostało jawnie to zebrane, żeby dać Ci takie jedno źródło wiedzy co i kiedy było wprowadzane. Chciałbym, żebyśmy zaczęli od tego, co zostało wprowadzone jako nowość w Java 8. Z jednej strony chcę Ci pokazać, ile już umiesz, a z drugiej strony zaznaczyć, dlaczego Java 8 jest określana jako swojego rodzaju game-changer w świecie programowania.

Java 8 została udostępniona światu w marcu **2014**. 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.

Interfejsy

Przed Java 8, w interfejsach mogliśmy mieć tylko i wyłącznie metody `public abstract`. Oznaczało to, że dodanie jakiejkolwiek metody do interfejsu wymuszało implementację tej metody we wszystkich klasach, które dany interfejs implementują. Java 8 wprowadziła możliwość definiowania w interfejsach metod **default** oraz **static**.

Default

Metoda **default** w interfejsie daje możliwość napisania domyślnej implementacji metody. Jednocześnie nie wymusza na klasach implementujących dany interfejs, żeby metoda **default** była w nich zaimplementowana. Przykład:

```
interface Swim {
    default String canSwim() { ①
        return "No, I can't";
    }
}

class Person implements Swim {

    void goSwimming() {
        System.out.println("Should I go swimming? " + canSwim()); ②
    }

    public static void main(String[] args) {
        new Person().goSwimming(); ③
    }
}
```

① Metoda **default** jest w tym momencie **public** i nie jest **abstract**.

② Możemy odwołać się do metody **default**, gdyż jej domyślna implementacja znajduje się w interfejsie `Swim`.

③ Na ekranie zostanie wydrukowane: *Should I go swimming? No, I can't*

Możemy również nadpisać metodę **default**. Przykład:

```
interface Swim {
    default String canSwim() { ①
        return "No, I can't";
    }
}

class Person implements Swim {

    void goSwimming() {
        System.out.println("Should I go swimming? " + canSwim());
    }

    @Override
    public String canSwim() {
        return "Yes I can!"; ①
    }

    public static void main(String[] args) {
        new Person().goSwimming(); ②
    }
}
```

① Ta metoda nadpisze domyślną implementację metody `canSwim()` z interfejsu `Swim` i to zachowanie zostanie wykorzystane.

② Na ekranie zostanie wydrukowane: *Should I go swimming? Yes I can!*

Static

Od Javy 8 możemy również zdefiniować metodę **static** w interfejsie.

```
interface Swim {

    static void someMethod() { ①
        System.out.println("Calling static method");
    }
}

class Person {

    public static void main(String[] args) {
        Swim.someMethod(); ②
    }
}
```

① Metoda **static** jest w tym momencie **public** i nie jest **abstract**.

② Wywołując taką metodę statyczną musimy wywołać ją bezpośrednio na interfejsie.

Date & Time

Klasy, które były omawiane wcześniej, które pozwalały nam na obsługę dat i czasów, czyli `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetDateTime`, `ZonedDateTime`, `Instant` i inne z nimi związane również zostały

wprowadzone w Java 8. Możemy zgeneralizować to, mówiąc, że Java 8 wprowadziła paczkę `java.time`. Poprzednio w Javie stosowane były inne klasy, które były dosyć ciężkie w użyciu, dlatego twórcy Javy zdecydowali się na nowe Date & Time API. W ramach materiałów nie omawialiśmy poprzednich rozwiązań i nie będziemy tego robić 😈. W ramach tego warsztatu nie będziemy również omawiać ponownie API Date & Time oraz wspomnianych klas, bo zostało to wszystko dosyć szczegółowo omówione w ramach Bootcampu. ☺

Programowanie funkcyjne

Java 8 wprowadziła całą koncepcję programowania funkcyjnego, którą omawialiśmy w warsztacie o programowaniu funkcyjnym. Z perspektywy poprzednich wersji Javy jest to duży game-changer bo umożliwił stosowanie wielu konstrukcji, które wcześniej nie były możliwe. Nawet na przykładzie wzorców projektowych widzieliśmy, że Java 8 wprowadziła konstrukcje, które pozwoliły spojrzeć na rozwiązanie tych samych problemów z kompletnie innej perspektywy. Nowe funkcje zostały dodane w paczce `java.util.function`.

Interfejsy funkcyjne

Functional interface to taki interfejs, który ma tylko jedną metodę abstrakcyjną. Może jednocześnie mieć wiele innych metod **default** oraz **static**, ale metodę abstrakcyjną może mieć tylko jedną. Jeżeli chcemy zaznaczyć, że jakiś interface jest funkcyjny i w ten sposób dać o tym znać albo innym developerom albo sobie z przyszłości to zastosujemy adnotację `@FunctionalInterface`. Taki interfejs możemy następnie zaimplementować przy wykorzystaniu **lambdy**.

Lambda

Wyrażenia **lambda** były sposobem żeby wyrazić programowanie funkcyjne w świecie programowania zorientowanego obiektowo. Przypomnijmy, że każda klasa w Javie dziedziczy z klasy `Object`, więc twórcy musieli jakoś podejść do rozwiązania zagadnienia podejścia funkcyjnego w świecie obiektowym. Rozwiązanie to polega na tym, że wyrażenie lambda może zostać wykorzystane tylko w zestawieniu z interfejsem funkcyjnym. Taki interfejs ma tylko jedną metodę abstrakcyjną, więc ta jedna metoda może zostać zaimplementowana przy wykorzystaniu wyrażenia lambda. Przypomnijmy tylko sobie, że wyrażenie **lambda** wygląda w poniższy sposób:

```
(argument) -> (ciało)
```

Nie będziemy tutaj wracać do przykładów gdyż zostały one omówione w warsztacie o programowaniu funkcyjnym.

Method Reference

Razem z **lambdą** został również wprowadzony mechanizm zwany **method reference**. **Method reference** (który można rozumieć jako referencję do metody) jest mechanizmem, który został wprowadzony aby skrócić zapis lambdy. Bazuje na tym, że niektóre lambdy mogą zostać zastąpione nazwą metody, której sygnatura pasuje w danym wywołaniu i może być ona użyta zamiast konkretnej lambdy. Czyli zamiast pisać lambdę możemy wskazać "referencję" do metody, która może zostać użyta zamiast lambdy. Możemy wykorzystywać ten mechanizm razem z metodami statycznymi, metodami instancyjnymi i konstruktorami.

Stream

W Javie 8 została również dodana nowa paczka `java.util.stream`, w której znajdziemy klasy dające nam możliwość pracy ze Stream API. Streamy służą do tego, żeby móc w sposób funkcyjny operować na sekwencjach danych. Aby móc otrzymać Stream z kolekcji, zostały dodane 2 metody:

- `.stream()` - dzięki niej otrzymamy Stream, na którym możemy wykonywać operacje,
- `.parallelStream()` (o tym nie rozmawialiśmy) - dzięki niej otrzymamy Stream, który pozwoli nam wykonywać operacje wielowątkowo - na razie się na nim nie skupiamy,

Stream możemy również utworzyć wykorzystując np. `Stream.of()`.

Nie będziemy tutaj wracać do przykładów gdyż zostały one omówione w warsztacie o programowaniu funkcyjnym.

Optional

W Javie uciążliwa jest sytuacja gdzie zostaje wyrzucony wyjątek `NullPointerException`. Jednocześnie próba pisania kodu, gdzie co chwila pojawia się **if**, który dokonuje sprawdzenia `!= null` prowadzi często do mało przejrzystego kodu, który często zajmuje dużo miejsca. Aby ułatwić obsługę takich sytuacji, wprowadzona została klasa `Optional`, która ma służyć jako opakowanie. Opakowanie to ma informować, że potencjalnie może być puste, albo może zawierać jakąś wartość. Wykorzystując metody takie jak `.map()`, `.flatMap()` albo `.filter()` możemy operować na wartości w opakowaniu tylko gdy faktycznie ta wartość jest dostępna. Kod taki będzie wyglądał czyściej niż gdybyśmy za każdym razem obsługiwali wszystkie przypadki ręcznie i jednocześnie zmniejszamy ryzyko wystąpienia wyjątku `NullPointerException`.

IO

Do klas, które nazwiemy zbiorczo **IO**, czyli do klas odpowiedzialnych za operacje na plikach zostały dodane następujące metody:

- `Files.list(Path dir)` - metoda, która służy do listowania zawartości podanej ścieżki. Zwracany jest `Stream<Path>`, dzięki czemu możemy wykonywać operacje takie jak `map()`, `filter()` itp.
- `Files.lines(Path path)` - metoda zwraca `Stream<String>` i jednocześnie nie ładuje całej zawartości pliku do pamięci programu. Operujemy na `Stream`, a przypomnę, że `Streamy` są **lazy**.
- `Files.walk()` - Metoda służy do przeszukiwania katalogów. Metoda "schodzi" po katalogach coraz głębiej (czyli inaczej rekursywnie) i zwraca `Stream<Path>` ze ścieżkami, które zostały znalezione. Dzięki temu, że typ zwracany to `Stream<Path>` możemy wykonywać dalej operacje na ścieżkach w sposób funkcyjny.

Kolekcje

forEach()

Z racji wprowadzenia funkcji programowania obiektowego, do kolekcji została dołożona metoda `forEach()`, która przyjmuje interfejs funkcyjny `Consumer`. Metoda ta została dodana w interfejsie

`java.lang.Iterable`, dzięki czemu kolekcje, które implementują ten interfejs, mają również dostępną tę metodę. Przykład:

```
public class Example {

    public static void main(String[] args) {
        List<Integer> source = IntStream.rangeClosed(1, 10).boxed().collect(Collectors.toList());
        source.forEach(element -> {
            System.out.println("Calling consumer with element: " + element);
        });
    }
}
```

removeIf()

Do interfejsu `java.util.Collection` została dodana metoda `removeIf()`, która usuwa element z kolekcji jeżeli podany `Predicate` jest spełniony. Przykład:

```
public class Example {

    public static void main(String[] args) {
        List<Integer> source = IntStream.rangeClosed(1, 10).boxed().collect(Collectors.toList());
        System.out.println(source); ①
        source.removeIf(element -> element > 3);
        System.out.println(source); ②
    }
}
```

① Na ekranie zostanie wydrukowane: *[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*

② Na ekranie zostanie wydrukowane: *[1, 2, 3]*

replaceAll()

Metoda służy do wymienienia każdej wartości w podanej mapie zgodnie z przekazaną implementacją interfejsu `BiFunction`. Przykład:

```
public class Example {

    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "1");
        map.put(2, "2");
        map.put(3, "3");
        map.put(4, "4");
        System.out.println(map); ①
        map.replaceAll((key, value) -> value + "replaced:k" + key);
        System.out.println(map); ②
    }
}
```

① Na ekranie zostanie wydrukowane: *{1=1, 2=2, 3=3, 4=4}*

② Na ekranie zostanie wydrukowane: *{1=1replaced,k1, 2=2replaced,k2, 3=3replaced,k3, 4=4replaced,k4}*

Pozostałe

Wymienione funkcjonalności nie są wszystkimi, jakie zostały wprowadzone w Javie 8. 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. W obrębie tych materiałów poruszamy tylko te kwestie, które są adekwatne do naszego poziomu zaawansowania jako Java developerów. Nie poruszamy też zagadnień, które na tym etapie nie są aż tak istotne i lepiej poświęcić ten sam czas nauki 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 8 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę JDK 8. Zaznaczam jednak, że wiele funkcjonalności będzie na tym etapie niezrozumiałych. ☺