Java 9 update

Spis treści

Java 9 update . . . 1
- JShell . . . 2
  - Konsola . . . 2
  - Intellij . . . 6
- Kolekcje . . . 7
  - List.of() . . . 7
  - Set.of() . . . 8
  - Map.of() oraz Map.ofEntries() . . . 9
- Interfejsy . . . 9
- try-with-resources . . . 10
- Optional . . . 10
  - or() . . . 10
  - stream() . . . 11
  - ifPresentOrElse() . . . 11
- Streamy . . . 12
  - ordered i unordered . . . 12
  - takeWhile() . . . 13
  - ordered . . . 13
  - unordered . . . 13
  - dropWhile() . . . 13
  - ordered . . . 14
  - unordered . . . 14
  - iterate() . . . 14
  - ofNullable() . . . 15
- Klasy Anonimowe . . . 15
- Java Modules . . . 16
- Pozostałe . . . 16

Java 9 update

Java 9 została wydana we wrześniu 2017 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.

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.

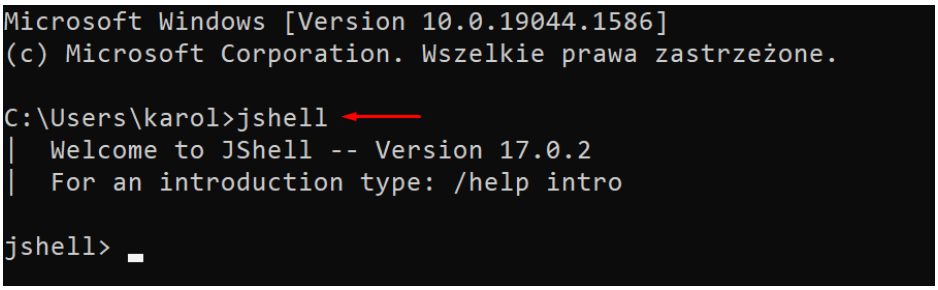
JShell

Java 9 wprowadziła nowe narzędzie zwane `JShell`, którego pełna nazwa brzmi **Java Shell**. Inna nazwa, jaką możemy spotkać to **REPL** (*Read Evaluate Print Loop*). Narzędzie to pozwala na uruchomienie fragmentu kodu w konsoli bez konieczności tworzenia klasy, definiowania metody `main()` oraz kompilacji. Narzędzie to służy do szybkiego przetestowania jak zachowa się dany fragment kodu, a nie do napisania całej aplikacji webowej 😀. Przejdźmy natomiast do tego, jak można z tego narzędzia skorzystać.

Konsola

Zanim przejdziesz dalej, upewnij się, że masz ustawioną zmienną środowiskową `path` i wskazuje ona na katalog `bin` w miejscu instalacji Twojej Javy. Bez tego ustawienia, poniższe komendy nie zadziałają.

Uruchom terminal/konsolę i wpisz polecenie `jshell`.

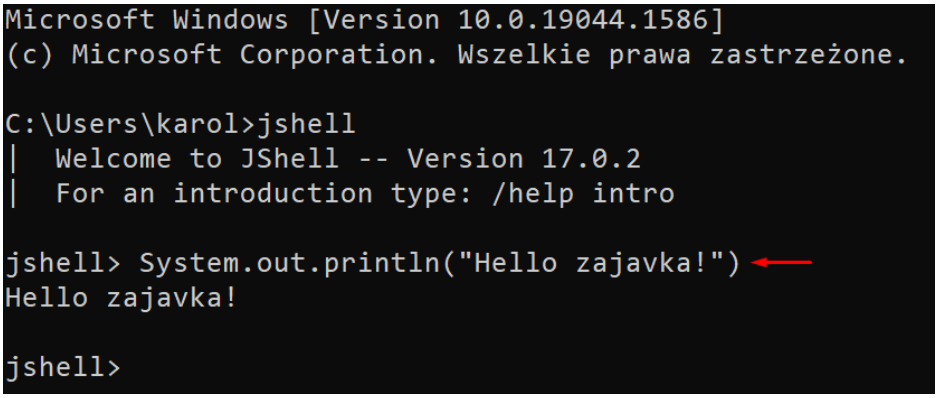


Obraz 1. JShell

W tym momencie uruchomione zostanie narzędzie `JShell`, które pozwala nam wpisywać kod Javovy. Wpisz zatem:

```
System.out.println("Hello Zajavka")
```

i zobaczmy, co zostanie wydrukowane na ekranie:

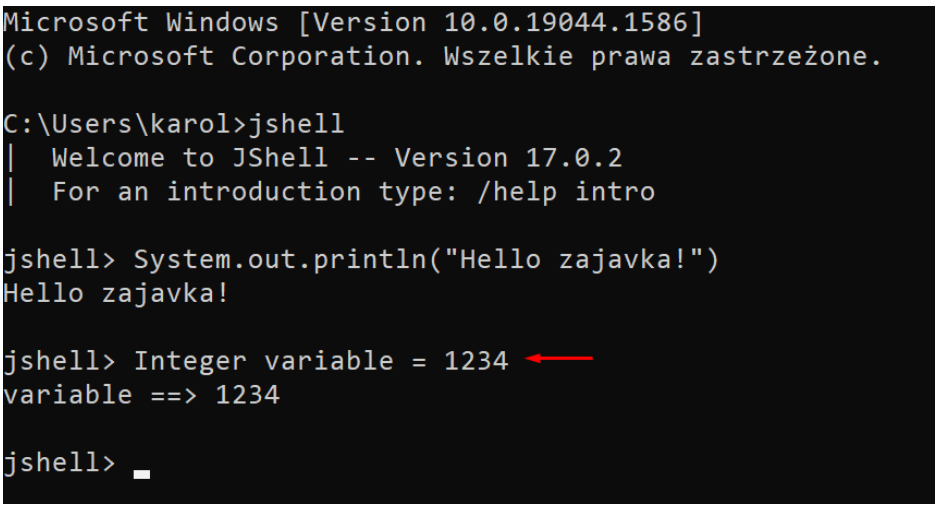


Obraz 2. JShell

Zwróć uwagę, że nie dopisaliśmy na końcu linijki średnika. Teraz wpisz w konsoli:

```
Integer variable = 1234
```

i wciśnij `enter`:

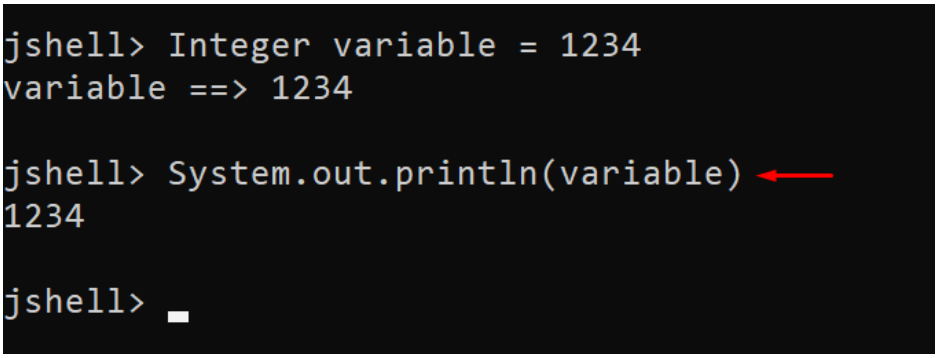


Obraz 3. JShell

Możesz teraz spróbować wydrukować wartość zdefiniowanej zmiennej, wykorzystując polecenie:

```
System.out.println(variable)
```

Zauważ, że nadal nie podajemy średnika.



Obraz 4. JShell

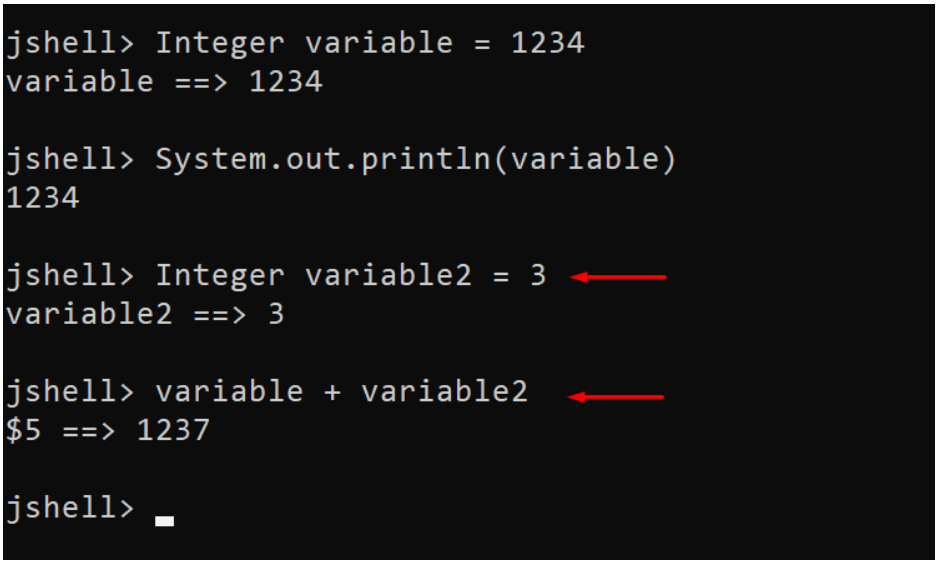
Zdefiniuj teraz zmienną:

```
Integer variable = 3
```

i w następnej kolejności wpisz:

```
variable + variable2
```

zobacz, co zostanie wydrukowane na ekranie:



Obraz 5. JShell

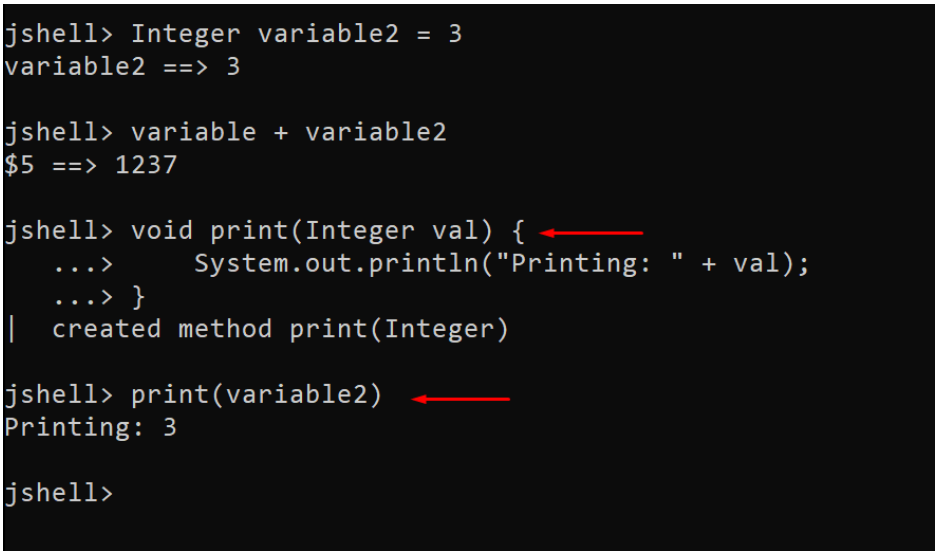
Możemy również w ten sposób definiować klasy i metody. Wpisz zatem treść metody:

```
void print(Integer val) {
    System.out.println("Printing: " + val);
}
```

i spróbuj wywołać tę metodę przy pomocy polecenia:

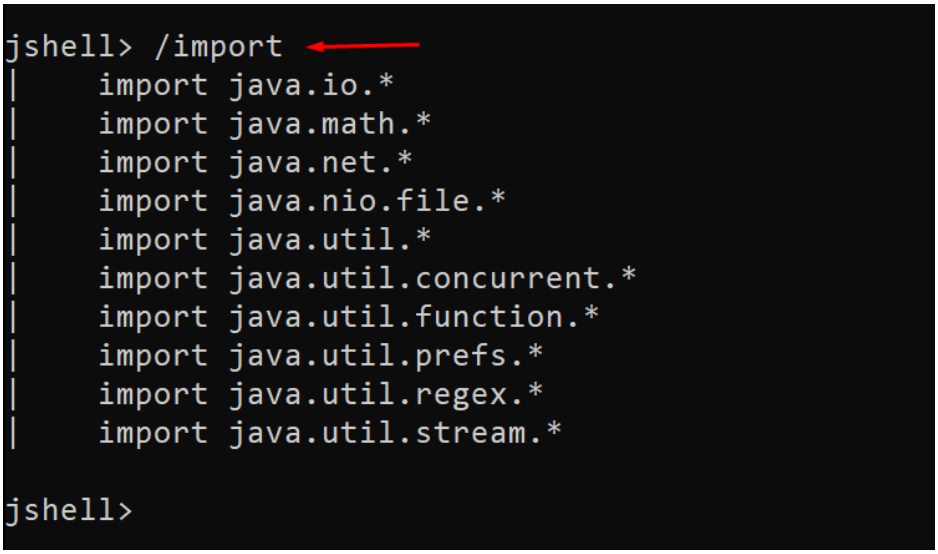
```
print(variable)
```

Na ekranie zostanie wydrukowane:



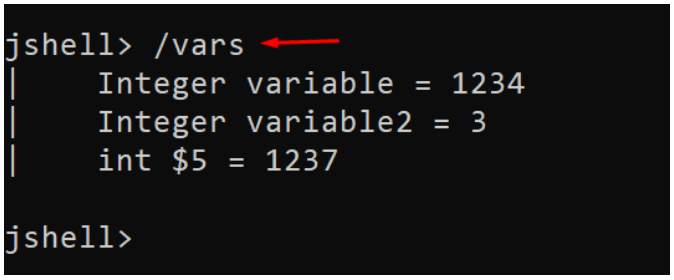
Obraz 6. JShell

Korzystając z `JShell`, możemy również zobaczyć jakie importy zostały dodane domyślne, gdy tylko uruchomiliśmy sesję `JShell`. `JShell` pozwala również na dodanie swoich własnych importów:



Obraz 7. JShell

Możemy również zobaczyć jakie zmienne zostały zadeklarowane od momentu, gdy rozpoczęliśmy sesję `JShell`:



Obraz 8. JShell

Kolejną komendą, jaką możemy wykonać, jest komenda sprawdzająca, jakie mamy zdefiniowane

metody:

```
jshell> /methods
|    void print(Integer)

jshell>
```

Obraz 9. JShell

`JShell` pozwala również na wypisanie całego napisanego przez nas kodu od momentu uruchomienia sesji `JShell`:

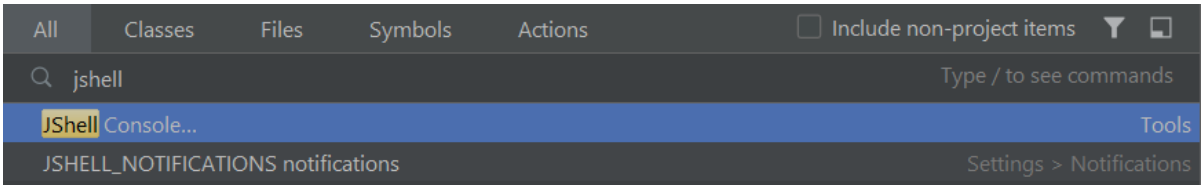
```
jshell> /list

   1 : System.out.println("Hello zajavka!")
   2 : Integer variable = 1234;
   3 : System.out.println(variable)
   4 : Integer variable2 = 3;
   5 : variable + variable2
   6 : void print(Integer val) {
           System.out.println("Printing: " + val);
       }
   7 : print(variable2)

jshell>
```

Obraz 10. JShell

Powyższe przykłady zostały przedstawione przy wykorzystaniu konsoli. `JShell` może być również używany z poziomu `Intellij`.

Intellij

Jeżeli chcemy uruchomić `JShell` z poziomu `Intellij`, wystarczy wcisnąć dwa razy **Shift** i wpisać **jshell**:



Obraz 11. JShell w Intellij

Jeżeli wybierzemy teraz pierwszą pozycję, uruchomimy konsolę `JShell` w `Intellij`. Możemy teraz w tej konsoli pisać analogicznie, tak jak było to pokazane wcześniej. Różnica polega jednak na tym, że tym razem `Intellij` będzie nam pomagał i podpowiadał:

Obraz 12. JShell w Intellij

Za pomocą ikony **play** możemy teraz uruchomić napisany fragment kodu.

Kolekcje

List.of()

Wraz z Javą 9 zostały wprowadzone nowe metody pozwalające na inicjowanie kolekcji immutable w prosty i krótki sposób:

```
public class Example {

    public static void main(String[] args) {
        List<String> listEmpty = List.of(); ①
        List<String> list1 = List.of("String1", "String2", "String3"); ②
        System.out.println(list1); ③
        list1.add("string4"); ④
        System.out.println(list1);
    }
}
```

① Ten zapis pozwala nam stworzyć pustą listę niemutowalną.

② Wykorzystując ten zapis, możemy stworzyć listę niemutowalną, która będzie zawierała zdefiniowane elementy.

③ Lista poprawnie wydrukuje się na ekranie.

④ Zostanie wyrzucony wyjątek `UnsupportedOperationException`, gdyż stworzona lista jest **immutable** (*niemutowalna*). Oznacza to, że nie możemy zmienić jej zawartości.

Zwróć uwagę, że stworzona lista to nie jest ani `ArrayList`, ani `LinkedList`. To jest jakaś implementacja listy, która nie pozwala zmieniać jej zawartości.

Przed Java 9 mieliśmy dostępne metody takie jak `Collections.empty()` oraz `Collections.singletonList()`. Pierwsza z nich tworzyła pustą listę, a druga listę z jednym elementem. Cechą wspólną tych metod było to, że tworzyły one listy **immutable**. Spójrz na kod poniżej:

```
public class Example {

    public static void main(String[] args) {
        List<String> emptyList = Collections.<String>emptyList(); ①
        List<String> singletonList = Collections.singletonList("element"); ②
        try {
            emptyList.add("someElement");
```

```
        } catch (Exception e) {
            System.out.println("Could not add element to emptyList. Exception: " + e.getClass()); ③
        }
        try {
            singletonList.add("someElement");
        } catch (Exception e) {
            System.out.println("Could not add element to singletonList. Exception: " + e.getClass()); ④
        }
    }
}
```

① Ta linijka tworzy pustą listę, do której nie możemy dodawać żadnych elementów.

② Ta linijka tworzy listę z jednym elementem, do której nie możemy dodawać żadnych elementów.

③ Na ekranie zostanie wydrukowane: *Could not add element to emptyList. Exception: class java.lang.UnsupportedOperationException.*

④ Na ekranie zostanie wydrukowane: *Could not add element to singletonList. Exception: class java.lang.UnsupportedOperationException.*

Powyższy przykład pokazuje, że zostały stworzone listy **immutable**, czyli takie, których stanu nie możemy zmienić. W Javie 9 zostały wprowadzone metody, które tworzą niemutowalne listy analogicznie do powyższego przykładu, jednak robią to, stosując krótszy zapis.

Co zrobić, żeby taka lista była mutowalna?

Można to zrobić w ten sposób:

```
List<String> list = new ArrayList<>(List.of("String1", "String2", "String3"));
```

W ten sposób tworzymy `ArrayList`, (której stan możemy już zmieniać) na podstawie danych z listy **immutable**.

Set.of()

Analogicznie do przedstawionych metod na interfejsie `List`, podobne metody zostały dodane do interfejsu `Set`:

```
public class Example {

    public static void main(String[] args) {
        Set<String> setEmpty = Set.of();
        Set<String> set1 = Set.of("String1", "String2", "String3");
        System.out.println(set1);
        set1.add("string4");
        System.out.println(set1);
    }
}
```

Zachowanie będzie analogiczne jak w poprzednim przykładzie. Utworzona w ten sposób zostanie implementacja `Set`, która jest **immutable**, zatem dodanie elementu do takiego `Set` spowoduje wyrzucenie wyjątku `UnsupportedOperationException`.

Map.of() oraz Map.ofEntries()

Java 9 wprowadziła nowe metody do interfejsu `Map`. Analogicznie do poprzednich przykładów, wprowadzone metody tworzą mapę, która jest **immutable**:

```
public class Example {

    public static void main(String[] args) {
        Map<Object, Object> emptyMap = Map.of(); ①
        Map<Integer, String> immutableMap = Map.of(1, "abc", 2, "def", 3, "ghi"); ②
        System.out.println(immutableMap); ③
        immutableMap.put(4, "jkl"); ④
        System.out.println(immutableMap);
    }
}
```

① Ten zapis pozwala nam stworzyć pustą mapę **immutable**.

② Wykorzystując ten zapis, możemy stworzyć mapę niemutowalną, która będzie zawierała zdefiniowane elementy.

③ Mapa poprawnie wydrukuje się na ekranie.

④ Zostanie wyrzucony wyjątek `UnsupportedOperationException`, gdyż stworzona lista mapa **immutable** (*niemutowalna*). Oznacza to, że nie możemy zmienić jej zawartości.

Należy tutaj zwrócić uwagę na pewną kwestię. W przypadku metod `List.of()` oraz `Set.of()` mogliśmy stworzyć `List` albo `Set` z dowolną ilością elementów. W przypadku metody `Map.of()` liczba wpisów (entry - par klucz:wartość) jest ograniczona do 10. Jeżeli chcemy stworzyć `Map`, korzystając z analogicznego podejścia, ale zdefiniować więcej niż 10 elementów, do tego posłuży nam metoda `Map.ofEntries()`:

```
public class Example {

    public static void main(String[] args) {
        Map<Integer, String> manyElements = Map.ofEntries(
            Map.entry(1, "1"), Map.entry(2, "2"), Map.entry(3, "3"),
            Map.entry(4, "4"), Map.entry(5, "5"), Map.entry(6, "6"),
            Map.entry(7, "7"), Map.entry(8, "8"), Map.entry(9, "9"),
            Map.entry(10, "10"), Map.entry(11, "11"), Map.entry(12, "12")
        );
        System.out.println(manyElements);
        manyElements.put(13, "13");
        System.out.println(manyElements);
    }
}
```

W tym przypadku musimy również wykorzystać metodę `Map.entry()`, która pozwoli nam stworzyć parę **klucz:wartość**. Ponownie stworzona mapa jest **immutable**, czyli próba dodania kolejnego elementu skończy się wyrzuceniem wyjątku `UnsupportedOperationException`.

Interfejsy

Java 9 pozwoliła na definiowanie metod prywatnych w interfejsach. Jest to dodatkowa funkcjonalność w stosunku do **default** i **static** metod w interfejsach wprowadzonych w Java 8. Przykład:

```
public interface Swim {

    private String createSwimmer(){
        // implementacja
    }

    private static boolean areYouReady(){
        // implementacja
    }

}
```

try-with-resources

Java 9 wprowadziła nowy zapis konstrukcji `try-with-resources`, przykład:

```
public class Example {

    void beforeJava9() throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader("inpuy.txt"))) {
            System.out.println(reader.readLine());
        }
    }

    void afterJava9() throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader("inpuy.txt"));
        try (reader) {
            System.out.println(reader.readLine());
        }
    }
}
```

Zapis w metodzie `afterJava9()` ma za zadanie uprościć pisany kod. W wersji Javy poniżej 9 przedstawiony zapis spowoduje błąd kompilacji.

Optional

Java 9 wprowadziła nowe metody w klasie `Optional`.

or()

Metoda `or()` może zostać wykorzystana w sytuacji, gdy w przypadku pustego `Optional` będziemy chceli zapewnić jakąś wartość domyślną. Przykład:

```
public class OptionalExample {

    public static void main(String[] args) {
        String result = Optional.<String>ofNullable(null) ①
            .or(OptionalExample::provideFallback)
            .orElse("orElse");
        System.out.println(result);
    }
```

```
    private static Optional<String> provideFallback() {
        System.out.println("Providing fallback");
        return Optional.of("fallback");
    }
}
```

① Zwróć uwagę, że jeżeli przekażemy tutaj `null`, to na ekranie zostanie wydrukowane *Providing fallback*. Oznacza to, że została wykonana metoda `provideFallback()`, która ma dać nam inny `Optional`, gdy ten w punkcie 1 okaże się pusty. Jeżeli natomiast w tej samej linijce zamiast `null` wstawimy np. `"1"`, na ekranie nie zostanie wydrukowane *Providing fallback*. Oznacza to, że metoda `or()` jest uruchamiana tylko wtedy, gdy oryginalny `Optional` jest pusty i chcemy zapewnić jakąś wartość domyślną.

stream()

W Java 9 do klasy `Optional` została dodana metoda `stream()`. Pozwala ona na to, żebyśmy z klasy `Optional` przeszli na operowanie na zdefiniowanej wartości jak na Streamie. Przykład:

```
public class OptionalExample {

    public static void main(String[] args) {
        long count0 = Optional.<String>ofNullable(null)
            .stream()
            .count();
        System.out.println(count0); ①
        long count1 = Optional.<String>ofNullable("1")
            .stream()
            .count();
        System.out.println(count1); ②
    }
}
```

① Na ekranie zostanie wydrukowane *0*.

② Na ekranie zostanie wydrukowane *1*.

Na stworzonym `Stream` możemy operować tak samo, jak operowaliśmy na `Streamach` w przypadku kolekcji.

ifPresentOrElse()

Wraz z wprowadzeniem klasy `Optional` została wprowadzona metoda `ifPresent()`, która pozwalała na zdefiniowanie akcji, która miała zostać wykonana, jeżeli `Optional` zawierał jakąś wartość. Brakowało natomiast metody, która pozwoliłaby wykonać jakąś akcję, jeżeli `Optional` byłby pusty. Do tego służy metoda `ifPresentOrElse()`. Przyjmuje ona dwa interfejsy: `Consumer` i `Runnable`. Nie znamy jeszcze interfejsu `Runnable`. Na ten moment wystarczy nam jedynie, że jest on funkcyjny, posiada jedną metodę, która nie przyjmuje parametrów i zwraca `void`, oraz że możemy go zaimplementować przy wykorzystaniu lambdy. Przykład:

```
public class OptionalExample {

    public static void main(String[] args) {
```

```
        Optional.<String>ofNullable(null)
            .ifPresentOrElse(OptionalExample::onPresent, OptionalExample::onEmpty); ①
        Optional.<String>ofNullable("1")
            .ifPresentOrElse(OptionalExample::onPresent, OptionalExample::onEmpty); ②
    }

    private static void onPresent(String input) {
        System.out.println("Calling on present: " + input);
    }

    private static void onEmpty() {
        System.out.println("Calling on empty");
    }
}
```

① W tym przypadku zostanie wywołana metoda `onEmpty()`.

② W tym przypadku zostanie wywołana metoda `onPresent()`.

Streamy

ordered i unordered

Wraz z Java 9 zostały dodane nowe metody do interfejsu `Stream`. Zanim jednak do tego przejdziemy, omówmy co oznaczają stwierdzenia **ordered** i **unordered** Stream. Wcześniej nie zostało zaznaczone jawnie, kiedy `Stream` jest **ordered** i kiedy jest **unordered**, chociaż w pewnym sensie było to naturalne, że `Streamy` przetwarzały elementy w jakiejś kolejności. Jeżeli zależało nam na wymuszeniu takiej kolejności, to mogliśmy wywołać operację `sorted()`. Naturalne też było, że jeżeli `Stream` był tworzony na podstawie `List`, to zachowywał kolejność jej elementów. Jednocześnie, jeżeli `Stream` był tworzony na podstawie `HashSet`, to kolejność elementów była nieprzewidywalna, ale mogliśmy ją narzucić, wykorzystując operację `ordered()`. Na tym właśnie polega różnica między **ordered** `Stream` i **unordered** `Stream`. Cytując dokumentację:

> Streams may or may not have a defined encounter order. Whether or not a stream has an encounter order depends on the source and the intermediate operations. Certain stream sources (such as List or arrays) are intrinsically ordered, whereas others (such as HashSet) are not. Some intermediate operations, such as sorted(), may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered, such as BaseStream.unordered(). Further, some terminal operations may ignore encounter order, such as forEach().
>
> If a stream is ordered, most operations are constrained to operate on the elements in their encounter order; if the source of a stream is a List containing [1, 2, 3], then the result of executing map(x → x*2) must be [2, 4, 6]. However, if the source has no defined encounter order, then any permutation of the values [2, 4, 6] would be a valid result.
>
> For sequential streams, the presence or absence of an encounter order does not affect performance, only determinism. If a stream is ordered, repeated execution of identical stream pipelines on an identical source will produce an identical result; if it is not ordered, repeated execution might produce different results.

Cytat potwierdza to, co napisałem wcześniej. Jeżeli tworzymy `Stream` na podstawie listy - zachowamy kolejność wystąpień elementów - **ordered**. Jeżeli natomiast utworzymy `Stream` na podstawie `HashSet` - kolejność elementów może być zmienna - **unordered**.

Należy też zaznaczyć, że cały czas rozmawiamy tutaj o **sequential** Streamach. Oznacza to, że dane/wartości/obiekty są przetwarzane przez `Stream` w sekwencji, inaczej mówiąc, oznacza to, że dane/wartości/obiekty są przetwarzane pojedynczo, element po elemencie. Możliwe jest przetwarzanie danych równolegle, zwyczajnie o tym jeszcze nie rozmawialiśmy i nie umiemy tego zrobić ☺.

takeWhile()

ordered

Operacja pośrednia `takeWhile()` jest bardzo podobna do operacji `filter()`, ale jednak lekko się od niej różni. Obie operacje przyjmują `Predicate` jako swój argument wywołania. Operacja `filter()` zwraca `Stream` tylko z tymi elementami, które spełniły przekazany predykat. Operacja `takeWhile()` zwraca elementy z oryginalnego Streama, dopóki (*while*) spełniają one predykat. Oznacza to, że jeżeli jeden element w trakcie przetwarzania przestanie spełniać ten predykat, to reszta Streama jest odrzucana. Przykład:

```
List<Integer> list = List.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
List<Integer> result = list.stream()
    .takeWhile(element -> element % 2 != 0)
    .collect(Collectors.toList());
System.out.println(result);
```

Na ekranie zostanie wydrukowane: *[1, 3, 5, 7]*. Czyli operacja `takeWhile()` zwraca elementy ze Streama, dopóki spełniony jest podany predykat. Pierwszy element, który tego predykatu nie spełnia, powoduje odrzucenie pozostałych wartości - dlatego na ekranie nie drukuje się 13 i 15, które przecież są nieparzyste. Pokazane zachowanie jest powtarzalne w przypadku Streamów, które są **ordered**.

unordered

Gdy wykorzystamy `Stream` **unordered**, będzie to wyglądało w ten sposób:

```
Set<Integer> set = Set.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
List<Integer> result = set.stream()
    .takeWhile(element -> element % 2 != 0)
    .collect(Collectors.toList());
System.out.println(result); ①
```

① Jeżeli zaczniemy teraz wielokrotnie uruchamiać ten program, to za każdym razem na ekranie może zostać drukowana inna wartość, np.: *[]*, albo *[3, 5, 7]*, albo *[7]*. Wynika to z tego, że rozpoczęty `Stream` jest **unordered**. Inaczej mówiąc, operacja taka jest niedeterministyczna, wynika to z tego, że operacja `takeWhile()` jest wykonywana na tym samym zbiorze elementów, ale za każdym razem ten zbiór może mieć inną kolejność.

dropWhile()

ordered

Można powiedzieć, że operacja `dropWhile()` jest przeciwieństwem `takeWhile()`. Zamiast zostawiać elementy, które spełniają predykat, takie elementy będą odrzucane - do pierwszego elementu, który takiego predykatu nie spełnia. Przykład:

```
List<Integer> list = List.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
List<Integer> result = list.stream()
    .dropWhile(element -> element % 2 != 0)
    .collect(Collectors.toList());
System.out.println(result);
```

Na ekranie zostanie wydrukowane: *[8, 10, 12, 13, 15]*. Czyli odrzucone zostały wartości, które spełniają podany predykat, do pierwszej, która tego predykatu nie spełnia. Od tego momentu nie odrzucamy żadnych wartości, nawet jeżeli spełniłyby one predykat.

unordered

Gdy napiszemy ten sam przykład, ale przy wykorzystaniu **unordered** `Stream` - wynik działania będzie nieprzewidywalny:

```
Set<Integer> set = Set.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
List<Integer> result = set.stream()
    .dropWhile(element -> element % 2 != 0)
    .collect(Collectors.toList());
System.out.println(result);
```

Na ekranie może zostać wydrukowane: *[12, 10, 8, 7, 5, 3, 1]*, albo *[12, 13, 15, 1, 3, 5, 7, 8, 10]*, albo *[12, 10, 8, 7, 5, 3]*. Jeżeli chcemy, żeby powyższy przykład był przewidywalny, należy dodać operację `sorted()`.

```
Set<Integer> set = Set.of(1, 3, 5, 7, 8, 10, 12, 13, 15);
List<Integer> result = set.stream()
    .sorted() ①
    .dropWhile(element -> element % 2 != 0)
    .collect(Collectors.toList());
System.out.println(result);
```

① Dzięki temu zapisowi, na ekranie będzie drukowane: *[8, 10, 12, 13, 15]* i będzie to drukowane w sposób powtarzalny.

iterate()

W Java 9 została wprowadzona nowa odsłona operacji `iterate()`. Poniżej przykład:

```
System.out.println("#1");
IntStream.iterate(2, previous -> previous * previous) ①
    .limit(4)
    .forEach(System.out::println);

System.out.println("#2");
```

```
IntStream.iterate(2, previous -> previous <= 256, previous -> previous * previous) ②
    .forEach(System.out::println);
```

① Tę operację poznaliśmy już wcześniej. Należało tutaj skorzystać z `limit()`. Inaczej otrzymalibyśmy **infinite** `Stream`.

② Nowa wersja operacji `iterate()` pozwala na dodanie `Predicate`, określający warunek, który pozwala nam na zakończenie generowania wartości w `Stream`. Jeżeli dobrze napiszemy ten warunek, to nie otrzymamy **infinite** `Stream`.

Powyższe dwa przykłady dadzą ten sam efekt na ekranie.

ofNullable()

W Java 9 została dodana nowa operacja rozpoczynająca `Stream.ofNullable()`. Działa ona analogicznie do `Optional.ofNullable()`. Możemy w ten sposób stworzyć `Stream` z jednym elementem albo pusty `Stream`:

```
System.out.println(Stream.ofNullable(null).count()); ①
System.out.println(Stream.ofNullable("element").count()); ②
```

① Na ekranie zostanie wydrukowane: *0*.

② Na ekranie zostanie wydrukowane: *1*.

Klasy Anonimowe

Java 9 wprowadziła również uproszczenie zapisu klas anonimowych. Przykład dla Java 8:

```
public class Example {

    public static void main(String[] args) {
        SomeAbstractClass<String> someAbstractClass = new SomeAbstractClass<String>() {
            @Override
            void call(String t1, String t2) {
                System.out.println("Running");
            }
        };
    }
}
```

I teraz ten sam przykład przy zastosowaniu uproszczenia w Java 9:

```
public class Example {

    public static void main(String[] args) {
        SomeAbstractClass<String> someAbstractClass = new SomeAbstractClass<>() {
            @Override
            void call(String t1, String t2) {
                System.out.println("Running");
            }
        };
    }
```

```
}
```

Różnicą jest to, że zniknął `String` w tzw. **Diamond operator**, czyli w nawiasach trójkątnych. Taki zapis jest możliwy dopiero od Java 9.

Jednocześnie zwróć uwagę, że jest to sytuacja analogiczna do tego, co było udoskonalone przy przejściu z Javy 6 na Javę 7. Java 6 wymagałaby od nas takiego zapisu:

```
List<String> list1 = new ArrayList<String>();
```

Java 7 wprowadziła tzw. **Diamond operator** i wraz z Javą 7 ten sam kod można zapisać w ten sposób:

```
List<String> list1 = new ArrayList<>();
```

Znaczek `<>` nazywany jest **Diamond operator**.

Sytuacja analogiczna ma miejsce w Java 9 przy korzystaniu z klas anonimowych.

Java Modules

Dużą zmianą, która została wprowadzona w Java 9, są **Java Modules**. Java 9 wprowadziła nowy poziom abstrakcji, który można rozumieć jak poziom ponad paczkami, czyli kolejna warstwa abstrakcji grupująca paczki. Moduł możemy rozumieć jak grupę blisko związanych ze sobą paczek albo inaczej jako paczkę paczek. Modularność ma poprawić reużywalność kodu. Stosując moduły, możemy zdefiniować tylko jeden moduł dla jednego pliku `.jar`. Oczywiście funkcjonalność ta została wprowadzona w taki sposób, żeby przejście z Javy 8 na Javę 9 odbyło się bez większych problemów. To tylko teoria i w praktyce mogą pojawić się problemy. Nie chcę wchodzić dalej w szczegóły implementacyjne systemu modułów, bo wydaje mi się to zbyt skomplikowane i zbyt złożone na ten moment. Zakładam, że jak będzie Ci to potrzebne w praktyce to i tak będziesz się edukować ponownie na ten temat, dlatego nie schodzimy głębiej w tematykę modułów ☺.

Pozostałe

Wymienione funkcjonalności nie są wszystkimi, jakie zostały wprowadzone w Javie 9. 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, 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 9 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany.

Możesz również zerknąć na tę stronę JDK 9. Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. ☺