Poznajemy głębiej API Javy

Spis treści

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

StringBuilder

`StringBuilder` jest klasą, która jest polecana w użyciu, jeżeli często modyfikujemy Stringi, zanim dojdą one do finalnego kształtu. `StringBuilder` reprezentuje zmienialną/mutowalną (mutable) sekwencję znaków. Dla porównania **String jest immutable**, czyli raz stworzonego Stringa nie da się już modyfikować, można tylko stworzyć kolejny String. `StringBuilder` można modyfikować, stąd piszę o mutability/immutability. Można dlatego powiedzieć, że **StringBuilder jest alternatywą dla używania Stringa**.

Oprócz wymienionych mamy jeszcze taką klasę jak StringBuffer, która również wprowadza funkcjonalność mutowalnej sekwencji znaków. Różnice polegają na tym, że `StringBuffer` jest przystosowana do używania w środowisku wielowątkowym, przez co jest wolniejsza. `StringBuilder` nie jest do tego dostosowana. Jak przyjdzie nam używać `StringBuilder` w środowisku wielowątkowym, zaleca się wtedy stosowanie `StringBuffer`. Przykład zastosowania:

```
String str = "Hello ";
// błąd kompilacji
str + "World!";
// String jest immutable, zatem należy przypisać wynik do zmiennej,
// inaczej rezultat wywołania jest ignorowany
str.concat("World!");
System.out.println(str);

StringBuilder sb = new StringBuilder("Hello ");
// StringBuilder jest mutable,
// więc nie musimy wywołania przypisywać do zmiennej
sb.append("World!");
System.out.println(sb);
```

Intellij sugeruje nam automatyczną zamianę `Stringa` na `StringBuilder`, gdy zaczynamy modyfikować Stringa w pętli. Modyfikować, tzn:

```
String someString = "";
for (int i = 0; i < 100; i++) {
    someString += "zajavka;";
}
```

BigInteger

Klasa `BigInteger` powinna być używana do obliczeń, która zakładają używanie liczb większych niż zakres typów prymitywnych. Częste przykłady zastosowań to liczenie dużych wartości silni, np. 86!

```
public class BigIntExamples {

    public static void main(String[] args) {
        System.out.println(factorial(86));
        System.out.println(factorialBigInteger(86));
    }

    private static int factorial(final int n) {
        int f = 1;

        for (int i = 2; i <= n; i++)
            f = f * i;

        return f;
    }

    private static BigInteger factorialBigInteger(final Integer n) {
        BigInteger f = BigInteger.ONE;

        for (int i = 2; i <= n; i++)
            f = f.multiply(BigInteger.valueOf(i));

        return f;
    }
}
```

Obie metody `factorial()` liczą dobrze silnię do n = 12. Powyżej zaczyna się już robić rozjazd w wynikach, bo są one zbyt duże, aby się zmieścić w zakresie `int`.

Przykłady często używanego kodu.

Przykład 1

Jak można tworzyć BigInteger:

```
System.out.println(BigInteger.ZERO);
System.out.println(BigInteger.ONE);
System.out.println(BigInteger.TWO);
System.out.println(BigInteger.TEN);
System.out.println(BigInteger.valueOf(12344));
System.out.println(new BigInteger("123559329423949230432032040 32"));
```

Przykład 2

W jaki sposób zwrócić wartość prymitywną albo Stringa:

```
BigInteger bigA = BigInteger.valueOf(3243);
System.out.println("bigA.intValue: " + bigA.intValue());
System.out.println("bigA.longValue: " + bigA.longValue());
System.out.println("bigA.toString: " + bigA.toString());
```

Przykład 3

Należy pamiętać o dwóch rzeczach, gdy zaczniemy używać klas `BitInteger` oraz `BigDecimal`. Z tymi klasami nie funkcjonują operatory dodawania, odejmowania, porównania itp. Dlatego w przypadku `BigInteger` operacje matematyczne są wykonywane przy wykorzystaniu metod:

```
BigInteger bigA = BigInteger.valueOf(3243);
BigInteger bigC = bigA.add(new BigInteger("23879823874"));
System.out.println("bigC: " + bigC);
bigC = bigA.multiply(BigInteger.valueOf(123456789));
System.out.println("bigC: " + bigC);
```

Jeżeli natomiast chodzi o operatory porównania, stosowana jest metoda `compareTo()`, której wyniki w przypadku takiego wywołania `bigA.compareTo(bigB)` należy interpretować następująco:

Tabela 1. Znaczenie wyniku metody `compareTo()` dla BigInteger

Wynik	Znaczenie
-1	bigA < bigB
0	bigA == bigB
1	bigA > bigB

```
BigInteger bigA = BigInteger.valueOf(3243);
BigInteger bigB = BigInteger.valueOf(3432);
// Błąd kompilacji
// System.out.println("bigA > bigB: " + (bigA > bigB));
System.out.println("compareTo: " + (bigA.compareTo(bigB)));
```

Jeżeli chodzi o równość, to zaleca się stosowanie metody equals, powód był wyjaśniany już parę razy wcześniej ☺.

```
BigInteger bigC = BigInteger.valueOf(100);
BigInteger bigD = BigInteger.valueOf(100);
// Intellij pokazuje przy porównaniu referencji warning
System.out.println("equal: " + (bigC == bigD));
System.out.println("equal: " + (bigC.equals(bigD)));
```

BigDecimal

`BigDecimal` jest zalecany przy każdego rodzaju operacjach, w których zależny nam na dokładności otrzymywanych wyników. Jeżeli na jakiejkolwiek rozmowie rekrutacyjnej padnie pytanie: *Jakiej klasy powinniśmy używać do przeprowadzania kalkulacji finansowych?*, zawsze odpowiadamy `BigDecimal`. Wykonywanie operacji na `Double` czy `Float` prowadzi do nieprzewidzianych błędów ze względu na dokładność obliczeniową tych typów. `BigDecimal` jest jednocześnie w stanie prowadzić obliczenia na naprawdę dużych wartościach.

Przykład niedokładności Double:

```
double a = 0.01;
double b = 0.03;
double c = b - a;
// 0.019999999999999997
System.out.println(c);
```

Jeżeli to samo wykonamy przy wykorzystaniu BigDecimal:

```
BigDecimal bd1 = new BigDecimal("0.01");
BigDecimal bd2 = new BigDecimal("0.03");
BigDecimal bd3 = bd1.subtract(bd2);
// -0.02
System.out.println(bd3);
```

Rozbieżność między Double a BigDecimal

Dwa podstawowe typy zmiennoprzecinkowe w Javie (`Double` i `Float`), przetrzymywane są w pamięci w postaci binarnej reprezentacji ułamka i wykładnika potęgi. Inne typy prymitywne (oprócz `boolean`) są zmiennymi stałoprzecinkowymi. W uproszczeniu, liczby zmiennoprzecinkowe zwracają wartość z małym błędem, rzędu `10^-19`. Dlatego w przykładzie nie otrzymaliśmy dokładnego wyniku `0.02` i dlatego powinniśmy stosować `BigDecimal` do operacji obliczeniowych.

Liczba zmiennoprzecinkowa oraz stałoprzecinkowa odnoszą się do binarnej reprezentacji liczb. Pamiętajmy, że komputer myśli zerami i jedynkami, musi więc sobie jakoś te liczby zapisać i przetworzyć. Zachęcam do Googlowania ☺.

Jak można tworzyć BigDecimal:

```
System.out.println(BigDecimal.ZERO);
System.out.println(BigDecimal.ONE);
System.out.println(BigDecimal.TEN);
System.out.println(BigDecimal.valueOf(235345.23523535));
System.out.println(new BigDecimal("9084387293874.092384082934"));
```

W jaki sposób zwrócić wartość prymitywną albo Stringa:

```
BigDecimal bigA = BigDecimal.valueOf(3243);
System.out.println("bigA.intValue: " + bigA.intValue());
System.out.println("bigA.longValue: " + bigA.longValue());
System.out.println("bigA.toString: " + bigA.toString());
```

Podobnie jak w przypadku `BigInteger`, `BigDecimal` nie obsługuje operatorów, zatem operacje matematyczne przeprowadzamy przy wykorzystaniu metod:

```
BigDecimal bd1 = new BigDecimal("900230949023985.09233095345");
BigDecimal bd2 = new BigDecimal("923900593405.982340290432");

System.out.println("bd1.add: " + bd1.add(bd2));
System.out.println("bd1.multiply: " + bd1.multiply(bd2));
System.out.println("bd1.subtract: " + bd1.subtract(bd2));
System.out.println("bd1.divide: " + bd1.divide(bd2, RoundingMode.HALF_UP));
System.out.println("bd1.pow: " + bd1.pow(2));
System.out.println("bd1.negate: " + bd1.negate());
```

I ponownie, `BigDecimal` nie obsługuje operatorów porównania, zatem należy używać metody `compareTo()`, która działa tak samo, jak w przypadku `BigInteger`. Wynik wywołania kodu `bigA.compareTo(bigB)` i interpretacja w tabelce poniżej:

Tabela 2. Znaczenie wyniku metody `compareTo()` dla BigDecimal

Wynik	Znaczenie
-1	bigA < bigB
0	bigA == bigB
1	bigA > bigB

I ponownie, jeżeli chodzi o równość, to zaleca się stosowanie metody equals, powód był wyjaśniany już parę razy wcześniej ☺.

```
BigDecimal bigC = BigDecimal.valueOf(90234.1245);
BigDecimal bigD = BigDecimal.valueOf(90234.1245);
// Intellij pokazuje przy porównaniu referencji warning
System.out.println("equal: " + (bigC == bigD));
System.out.println("equal: " + (bigC.equals(bigD)));
```

Optional

Do tego momentu poznaliśmy już wyjątek `NullPointerException`. Nawet jeżeli jeszcze nie udało Ci się o tym przekonać na własnej skórze, to uwierz mi, że w praktyce jest on częsty i denerwujący. Jeżeli chcemy zacząć się przed nim zabezpieczać w kodzie, to w pewnym momencie kod ten zaczyna się mocno komplikować i "brudzić". W wielu miejscach trzeba zacząć dodawać ify sprawdzające, czy przypadkiem dana referencja nie jest `null`em. A potem wywołujemy na niej getter, jego wynik też może być `null`em i możliwe, że też trzeba się przed tym zabezpieczyć.

W tym celu twórcy Javy wprowadzili klasę `java.util.Optional`. Pomaga nam ona pisać czystszy kod unikając częstego sprawdzania, czy referencja nie jest `null`em, a do tego `Optional` pozwala nam zapewnić wartości domyślne, alternatywne w przypadku gdy dana referencja jest `null`em.

Po co to komu?

Przykładowy fragment kodu bez Optionala:

```
public class OptionalExample {
    public static void main(String[] args) {
        String result = someString().toUpperCase();
        System.out.print(result);
    }

    private static String someString() {
        return null;
    }
}
```

Jak już możesz się domyślić, na ekranie pojawi się `java.lang.NullPointerException`. Możemy temu zapobiec stosując klasę `Optional`.

```
import java.util.Optional;

public class OptionalDemo {
    public static void main(String[] args) {
        String result = someString();
        Optional<String> isNull = Optional.ofNullable(result);
        if (isNull.isPresent()) {
            System.out.println(result.toUpperCase());
        } else
            System.out.println("NULL");
    }
}
```

`Optional` jest swojego rodzaju opakowaniem na obiekty, które mogą albo istnieć, albo być `null`em. Możesz też zwrócić uwagę, że tak jak w przypadku `List`, `Optional` pozwala nam określić, jaki typ danych jest w nim przechowywany. `Optional` nie jest inicjowany przez konstruktor, gdyż ten on prywatny.

Tworzenie Optional

3 sposoby na stworzenie `Optional`.

```
Optional<String> empty = Optional.empty();
Optional<String> ofFull = Optional.of("Hello");
Optional<String> ofNull = Optional.ofNullable(null);
```

Należy pamiętać, że:

- **`Optional.empty()`** - stosujemy, jeżeli chcemy stworzyć Optional, który jest pusty,
- **`Optional.of()`** - stosujemy, jeżeli jesteśmy pewni, że obiekt, który opakowujemy `Optional` nigdy nie będzie `null`, bo jeżeli przekażemy do tego wywołania null, to w trakcie działania programu zostanie wyrzucony wyjątek. Jeżeli chcemy stworzyć Optional i wiemy, że obiekt, który do niego przekazujemy może być `null`, używamy następnej metody `Optional.ofNullable()`,
- **`Optional.ofNullable()`** - stosujemy, jeżeli obiekt, który opakowujemy może być `null`em.

Sprawdzenie zawartości Optional

Gdy już mamy stworzony Optional, możemy wywołać na nim metody sprawdzające:

```
System.out.println("ofFull.isPresent: " + ofFull.isPresent()); ①
System.out.println("ofFull.isEmpty: " + ofFull.isEmpty()); ②
```

① `isPresent()` - sprawdzi, czy w Optionalu znajduje się obiekt,

② `isEmpty()` - sprawdzi, czy Optional jest pusty.

Pobranie wartości Optional

Jeżeli jesteśmy pewni, że `Optional` nie jest pusty, możemy wywołać na nim:

```
String unpacked1 = ofNullableFull.get();
```

ifPresent

Teoretycznie, gdy `Optional` jest pusty, też możemy wywołać na nim `.get()`, ale wtedy w trakcie działania programu zostanie wyrzucony wyjątek `NoSuchElementException`.

`Optional` dostarcza nam metodę `ifPresent()`, która pozwalają na wykonanie jakiejś akcji, tylko wtedy, gdy nasz `Optional` nie jest pusty:

```
ofFull.ifPresent(a -> System.out.println("ifPresent Doing something with: " + a));
```

`ifPresent()` przyjmuje lambdę jako argument, gdzie lambda może mieć jeden parametr wchodzący i nie może nic zwracać.

Wartość domyślna

Wspomniałem też wcześniej, że `Optional` dostarcza nam mechanizm podania wartości alternatywnej, gdy `Optional` jest pusty:

```
Optional<String> ofFull = Optional.of("Hello");
String unpacked = ofFull.orElse("Other");
System.out.println("Unpacked: " + unpacked);
```

`orElse()` zwraca nam "Other" i przypisuje wynik do zmiennej, gdy nasz `Optional` jest pusty. Istnieje jeszcze wariant zwracania wartości alternatywnej, który może wyrzucić wyjątek, gdy `Optional` jest pusty:

```
String unpacked5 = ofNullableNull.orElseThrow(() -> new RuntimeException("No value present"));
```

Często też używa się `.orElseGet()`, który przyjmuje lambdę. `orElseGet()` różni się od metody `orElse()`, że `orElseGet()` wykonuje się w sposób odroczony, czyli tylko wtedy gdy zajdzie taka potrzeba, bo `Optional` jest faktycznie pusty. `orElse()` wykonuje się zawsze. Przeanalizuj poniższy fragment kodu i jego wynik na ekranie:

```
public class OptionalExamples {

    public static void main(String[] args) {
        System.out.println("Optional.of()");
        Optional<String> ofNullableFull = Optional.of("zajavka");
        String unpacked1 = ofNullableFull.orElse(createDefault1());
        String unpacked2 = ofNullableFull.orElseGet(() -> createDefault2());
        System.out.println();

        System.out.println("Optional.ofNullable()");
        Optional<String> ofNullableNull = Optional.ofNullable(null);
        String unpacked3 = ofNullableNull.orElse(createDefault1());
        String unpacked4 = ofNullableNull.orElseGet(() -> createDefault2());
    }

    private static String createDefault1() {
        System.out.println("Getting default object 1");
        return "Default";
    }

    private static String createDefault2() {
        System.out.println("Getting default object 2");
        return "Default";
    }
}
```

Na ekranie zostanie wydrukowane:

```
Optional.of()
Getting default object 1

Optional.ofNullable()
Getting default object 1
Getting default object 2
```

Na tej podstawie widać, że `orElse()` wykona się zawsze, niezależnie, czy `Optional` jest pusty, czy wypełniony. `orElseGet()` wykonuje się faktycznie tylko wtedy, gdy `Optional` jest pusty. To jest przykład **deferred execution**, o którym pisałem w notatkach o lambdach. Warto o tym pamiętać, jeżeli operacja, którą umieszczamy w `orElse()` lub `orElseGet()` zajmuje dużo czasu.

map()

Kolejną ważną metodą w `Optionalu` jet `map()`, czyli przemapowanie obiektu na inny. Nie mylić tego z Mapą z kolekcji, ta operacja polega na tym, że chcemy np, przemapować `Integera` na `Stringa`, ale tylko wtedy gdy taki `Optional` istnieje. Operacja `map()` przyjmuje lambdę, w której parametrem wejściowym jest zawartość Optionala. Lambda ta się nie wykona, jeżeli `Optional` jest pusty. Wprowadźmy klasę `Car` i `SteeringWheel`:

Klasa Car

```
public class Car {

    private final SteeringWheel steeringWheel;

    public Car() {
        this.steeringWheel = new SteeringWheel(0.6);
    }

    public Car(final SteeringWheel steeringWheel) {
        this.steeringWheel = steeringWheel;
    }

    public SteeringWheel getSteeringWheel() {
        System.out.println("Getting steering wheel");
        return steeringWheel;
    }

    public Optional<SteeringWheel> getSteeringWheelOpt() {
        System.out.println("Getting steering wheel optional");
        return Optional.ofNullable(steeringWheel);
    }

    @Override
    public String toString() {
        return "Car{" +
            "steeringWheel=" + steeringWheel +
            '}';
    }
}
```

Klasa SteeringWheel

```
public class SteeringWheel {

    private final double diameter;

    public SteeringWheel(final double diameter) {
        this.diameter = diameter;
    }

    public double getDiameter() {
        System.out.println("Getting diameter");
        return diameter;
    }

    public Optional<Double> getDiameterOpt() {
        System.out.println("Getting diameter optional");
        return Optional.of(diameter);
    }

    @Override
    public String toString() {
        return "SteeringWheel{" +
            "diameter=" + diameter +
            '}';
    }
}
```

Mając samochód zdefiniowany jak wyżej możemy się nim pobawić stosując `Optional`:

```
public class OptionalExamples {

    public static void main(String[] args) {
        // Tworzymy Optional z naszego nowego samochodu
        Optional<Car> car = Optional.of(new Car());

        // Mapujemy ten Optional, żeby teraz w środku była kierownica zamiast Samochodu.
        // W tym celu do metody map przekazujemy lambdę,
        // która jako parametr wejściowy przyjmuje samochód,
        // Rezultatem lambdy jest kierownica, która została zwrócona z gettera z samochodu
        Optional<SteeringWheel> optional1 = car.map(c -> c.getSteeringWheel());
        // Kierownica ma getter zwracający jej średnicę,
        // Mogę zatem przemapować Optional, żeby teraz w środku była średnica tej kierownicy
        Optional<Double> optional2 = optional1.map(sw -> sw.getDiameter());
        // Na końcu chcę odpakować Optional, mogę to zrobić albo wykonując metodę get(),
        // Albo określając, daj mi to co masz w Optionalu, albo wartość domyślną 0.0
        Double diameterOrDefault = optional2.orElse(0.0);

        // Tutaj mogę to wydrukować
        System.out.println("Diameter: " + diameterOrDefault);
    }
}
```

Zwróć uwagę, że po każdym kroku zwracałem dalej Optional. Dzięki temu kolejne wywołania mogę chainować (łączyć). Kod poniżej jest równoważny z tym powyżej:

```
public class OptionalExamples {

    public static void main(String[] args) {
        Optional<Car> car = Optional.of(new Car());

        Double diameterOrDefault = car
            .map(c -> c.getSteeringWheel())
            .map(sw -> sw.getDiameter())
            .orElse(0.0);

        System.out.println("Diameter: " + diameterOrDefault);
    }
}
```

Zobacz jak zrobiło się zwięźle ☺. Możesz spróbować napisać taką samą logikę bez stosowania Optionala żeby przekonać się jak ułatwia to życie ☺.

flatMap()

Może się zdarzyć, że jak zaczniemy mapować jakieś wartości, to metoda, którą wywołamy w środku lambdy też zwróci nam `Optional`. Przykład:

```
public class OptionalExamples {

    public static void main(String[] args) {
        Optional<Car> car = Optional.of(new Car());

        // metoda getSteeringWheelOpt() zwraca Optional,
        // więc po wykonaniu mapowania mamy Optional w Optionalu.
        // Można napisać takiego potworka, nikt nam nie zabrania.
        Optional<Optional<SteeringWheel>> steeringWheelOpt1 = car.map(c -> c.getSteeringWheelOpt());
        // Potem możemy "odwinąć" takiego Optionala w Optionalu stosując w mapowaniu metodę get()
        Optional<SteeringWheel> steeringWheelOpt2 = steeringWheelOpt1.map(c -> c.get());
        // Tutaj znowu mamy taką samą sytuację, metoda getDiameterOpt() zwraca Optional,
        // zatem znowu otrzymamy Optional w Optionalu
        Optional<Optional<Double>> diameterOpt1 = steeringWheelOpt2.map(sw -> sw.getDiameterOpt());
        // Tutaj ponownie takiego Optionala odwijamy stosując metodę get()
        Optional<Double> diameterOpt2 = diameterOpt1.map(d -> d.get());
        // Tutaj get() finalnie zwraca nam średnicę kierownicy
        Double diameter2 = diameterOpt2.get();
    }
}
```

Fragment kodu:

```
Optional<Optional<SteeringWheel>> steeringWheelOpt1 = car.map(c -> c.getSteeringWheelOpt());
Optional<SteeringWheel> steeringWheelOpt2 = steeringWheelOpt1.map(c -> c.get());
```

Można napisać szybciej stosując metodę `flatMap()`:

```
Optional<SteeringWheel> steeringWheel3 = car.flatMap(c -> c.getSteeringWheelOpt());
Optional<Double> diameterOpt3 = steeringWheel3.flatMap(c -> c.getDiameterOpt());
Double diameter3 = diameterOpt3.get();
```

Efekt będzie dokładnie ten sam. To, co robi `flatMap()` to "odwija" `Optionala` w `Optionalu` w locie.

Należy jednak pamiętać, że `flatMap()` kompiluje się poprawnie tylko wtedy, gdy faktycznie mamy sytuacje, że `map()` zwróci nam `Optionala` w `Optionalu`. Taki kod już się nie kompiluje:

```
public class OptionalExamples {

    public static void main(String[] args) {
        Optional<Car> car = Optional.of(new Car());

        car.flatMap(c -> c.getSteeringWheel());
    }
}
```

filter()

Ostatnia już poruszana w tej notatce ciekawa metoda. Metoda `filter()` przyjmuje lambdę jako argument. Lambda ta musi implementować `Predicate`, gdyż metoda `filter()` ma określoną klasę `Predicate` jako argument.

Metoda `filter()` zachowuje się w ten sposób, że jeżeli `Optional` jest pusty, to `filter()` się nie wykona. Jeżeli `Optional` jest pełny, a `filter()` zwróci `false`, to po wykonaniu `filter()`, `Optional` będzie pusty. Jeżeli `filter()` zwróci `true`, to `Optional()` dalej będzie wypełniony. Przykład:

```
public class OptionalExamples {

    public static void main(String[] args) {
        Optional<Car> car = Optional.of(new Car());

        Optional<Double> diameterOptional = car
            .map(c -> c.getSteeringWheel())
            .map(sw -> sw.getDiameter());

        Double diameterOrThrow = diameterOptional
            .filter(d -> d >= 0.5)
            .filter(d -> d < 0.7)
            .orElseThrow();
        System.out.println(diameterOrThrow);
    }
}
```

Przypomnę tylko, że w konstruktorze klasy `Car` jest taki kod:

```
public Car() {
    this.steeringWheel = new SteeringWheel(0.6);
}
```

Czyli wartość domyślna pola `diameter` w klasie `SteeringWheel` wynosi `0.6`. Czyli zmienna `diameterOptional` ma w środku `Double` z wartością `0.6`. Następnie na takim Optionalu wykonujemy pierwszy `filter()`, czyli `.filter(d -> d >= 0.5)`, który zwraca `true`, bo `0.6 >= 0.5`, zatem `Optional` nadal pozostaje wypełniony. Później wykonujemy drugi `filter()`, czyli `.filter(d -> d < 0.7)`, który również zwraca `true`, gdyż `0.6 < 0.7`, zatem `Optional` nadal zwraca `true`. Finalnie do zmiennej `diameterOrThrow` przypisujemy wartość `0.6` i jest ona drukowana na ekranie.

Jeżeli natomiast którykolwiek `filter()` zwróciłby `false`, to fragment kodu `.orElseThrow()` wyrzuciłby wyjątek. Przetestuj to we własnym zakresie ☺.