Notatki - Obsługa plików a wyjątki

Spis treści

Try-With-Resources

Zacznijmy od fragmentu kodu, żeby mieć o czym rozmawiać. Załóżmy, że mamy 2 pliki tekstowe na dysku i chcemy napisać program, który podczas działania odczyta coś z jednego pliku i zapisze do drugiego. W przykładach pokażemy jak można to zrobić, natomiast na ten moment, nie będziemy się skupiać na wyjaśnianiu kodu, który faktycznie operuje na plikach, to będzie potem. Skupimy tylko na części dotyczącej wyjątków.

```
public void example(Path path1, Path path2) { ①
    BufferedReader in = null; ②
    BufferedWriter out = null; ③
    try {
        in = Files.newBufferedReader(path1);
        out = Files.newBufferedWriter(path2);
        String line = in.readLine();
        out.write(line);
    } catch (IOException e) { ④
        e.printStackTrace();
    } finally { ⑤
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) { ⑥
                e.printStackTrace(); ⑦
            }
        }
        if (out != null) { ⑧
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

① Path - klasa, która jest w stanie operować na ścieżce do pliku na dysku.

② BufferedReader - klasa, która pozwoli nam czytać z pliku.

③ BufferedWriter - klasa, która pozwoli nam pisać do pliku.

④ IOException - wyjątek, który może zostać wyrzucony jak coś pójdzie nie tak z odczytem lub zapisem do pliku.

⑤ Zabawa polega na tym, że musimy zadbać o to, żeby na koniec "zamknąć" otwarty plik. Z dokumentacji: *close() - closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.*

⑥ Podczas samego zamykania `.close()`, też może zostać wyrzucony `IOException`.

⑦ Dodam, że z `.close()` trzeba uważać, bo w tym przykładzie napisaliśmy je w oddzielnych `try-catch` (oddzielne dla `in` i oddzielne dla `out`), ale jeżeli byśmy zamiast robić `try-catch` dodali `throws IOException` w definicji metody, mogłaby wystąpić taka sytuacja, że pierwszy `close()` się wykonał, wyrzucił wyjątek i wtedy drugi `close()` się nie wykona, bo obsługa wyjątku ma nastąpić w metodzie wywołującej metodę `example()` i ktoś o tym zwyczajnie zapomni.

⑧ Po co w ogóle to zamykać? Żeby nie doprowadzać do wycieków pamięci, które nastąpią, bo zarezerwowaliśmy jakiś fragment pamięci, używamy go i następnie trzeba go zamknąć. Pamiętasz, że Java robi to za nas bo używa `Garbage Collectora`? Nie w tej sytuacji 😁.

Wygląda na skomplikowane, prawda? W Javie 7 zostało wprowadzone ułatwienie, które nazywa się **`try-with-resources`**. Poniższy fragment kodu robi dokładnie to samo co poprzednik. Zwróć uwagę na ciekawy zapis `try`.

```
public void tryWithResourcesExample(Path path1, Path path2) throws IOException {
    try (
        BufferedReader in = Files.newBufferedReader(path1);
        BufferedWriter out = Files.newBufferedWriter(path2)
    ) {
        out.write(in.readLine());
    }
}
```

Cały kod się uprościł, zatem przejdźmy do tego co się tutaj dzieje. W Javie 7 został wprowadzony zapis `try-with-resources`, który umożliwia nam zapisanie samego `try` (pamiętasz, że `try` musiał mieć albo `catch` albo `finally`? Nie można było napisać samego `try`? W tym przypadku można).

```
try
( ①
    BufferedReader read = Files.newBufferedReader(path1); ②
    BufferedWriter write = Files.newBufferedWriter(path2) ③
) { ④
⑤
} ⑥
```

① Nawias zwykły.

② Jeżeli definiujemy 2 zmienne, musimy oddzielić je średnikiem.

③ Tutaj średnik nie jest konieczny.

④ Koniec zwykłego nawiasu.

⑤ Kod który się wykona w obrębie `try-with-resources`.

⑥ W tym miejscu następuje automatyczne zamknięcie zasobów.

Z takim zapisem normalnie funkcjonuje zapis `catch` oraz `finally`. Nadal możemy mieć kilka `catchy` i

jedno `finally` (**nie możemy mieć kilku `finally`**)

Ważne do zapamiętania jest tutaj również to, że jeżeli zdefiniujemy jakąś zmienną w `try` w nawiasach, to możemy jej używać tylko w obrębie `try` - `catch` i `finally` już jej nie widzą. Czyli w poprzednim przykładzie zmienne `read` i `write` są widoczne tylko w zakresie bloku `try`.

AutoCloseable

Jeżeli zaczniesz pisać przykłady na własną rękę to szybko zwrócisz uwagę, że nie można napisać czegoś takiego:

```
public void nonWorkingExample() {
    try (String variable1 = "zajavka") {

    }
}
```

Dlaczego się tak dzieje? Bo `try-with-resources` ma taki wymóg, że w obrębie nawiasu (tego zwykłego, a nie klamrowego) w `try`, możemy tworzyć tylko obiekty klas, które implementują interfejs `AutoCloseable`. `String` tego nie robi. Napisana przez Ciebie klasa też tego nie robi, dopóki nie zaimplementujesz tego jawnie (tak jak w przypadku np. `Comparable`).

Co się natomiast stanie gdy zaimplementujemy taki interfejs?

```
public class AutoCloseableExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            System.out.println("Doing something with Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        }
    }

    static class Door implements AutoCloseable {

        @Override
        // Nie musimy tu pisać throws Exception.
        // Natomiast jak jest napisane to trzeba je obsłużyć w bloku catch pod try.
        public void close() throws Exception {
            System.out.println("I'm closing my door!");
        }
    }
}
```

Musimy też wtedy zaimplementować metodę `close()` z tego interfejsu, która określa co ma się stać na końcu bloku `try-with-resources`. Na tej podstawie Java wie, co ma się stać na etapie zamykania zasobów przydzielonych na początku bloku `try-with-resources`. Inaczej mówiąc, to w metodzie `close()` piszemy w jaki sposób mają zostać zwolnione zasoby zarezerwowane w `try() {}`.

Dlaczego to działało w przypadku klas `BufferedReader` oraz `BufferedWriter` i nie dostawaliśmy błędu kompilacji? Obie te klasy implementują interfejs `AutoCloseable`.

Istnieją też 2 zalecenia co robić, a czego nie robić w metodzie `close()`:

- Nie wyrzucać wyjątku `Exception`, tylko jakiś bardziej konkretny, mówiący co faktycznie się stało.
- Tak napisać metodę `close()` aby była **`idempotentna`** (bardzo fajne słowo).

Idempotentność oznacza, że możemy tę samą metodę wywoływać ile nam się razy podoba i za każdym razem będzie to miało ten sam efekt. Czy wywołamy ją po raz pierwszy czy 14, zawsze efekt powinien być ten sam. Inaczej mówiąc, taka metoda nie ma efektów ubocznych. Taka rekomendacja pojawia się żebyśmy nie zrobili sobie kuku jeżeli `close()` zostanie wywołane 2 razy.

W ramach ciekawostki dodam, że istnieje taki interface jak `Closeable`, który istniał przed `AutoCloseable`. Jak zaczniesz oglądać jak wzajemnie dziedziczą z siebie te interfejsy, to zobaczysz, że `Closeable` dziedziczy z `AutoCloseable`. To był taki trick, żeby zachować kompatybilność wsteczną, która jest przecież jedną z myśli przewodnich Javy. Możesz zwrócić uwagę, że w definicji `Closeable` w metodzie `close()` jest napisane `throws IOException`, podczas gdy `AutoCloseable` ma już bardziej poluźnione reguły i pozwala na wyrzucenie samego `Exception`. Przypomnę, że `IOException`, które jest rzucane przy sytuacjach wyjątkowych, podczas operacji na plikach, dziedziczy z `Exception`.

Suppressed exceptions

Może nastąpić taka sytuacja, że w trakcie wywołania metody `close()` zostanie wyrzucony wyjątek. Dla jasności jeszcze raz ten sam fragment kodu.

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            System.out.println("Doing something with Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
            throw new RuntimeException("Can't close my Door!");
        }
    }
}
```

A co jeżeli w środku bloku `try`, też zostanie wyrzucony wyjątek?

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            throw new RuntimeException("Exception while opening Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
          throw new RuntimeException("Can't close my Door!");
        }
    }
}
```

Jeżeli wystąpi sytuacja jak powyżej, wywołanie kodu:

```
throw new RuntimeException("Exception while opening Door");
```

powoduje zatrzymanie wywołania bloku `try` i przejście do wywołania metody `.close()`. Ale przecież metoda `close()` również wyrzuca wyjątek. Nazywany jest on `SuppressedException`. Na ekranie wydrukowany wtedy zostaje wyjątek główny, czyli wywołanie kodu:

```
System.out.println("Handling exception thrown by close(): " +  e.getMessage());
```

Drukuje na ekranie:

```
Handling exception thrown by close(): Exception while opening Door
```

Ale, w ten sposób, nie widzimy jaki wyjątek został wyrzucony w metodzie `close()`. Jeżeli natomiast napiszemy w catch `e.printStackTrace()`, dostaniemy taki (albo zbliżony) StackTrace na ekranie:

```
java.lang.RuntimeException: Exception while opening Door
    at SuppressedExample.main(SuppressedExample.java:13)
Suppressed: java.lang.RuntimeException: Can't close my Door!
    at SuppressedExample$Door.close(SuppressedExample.java:26)
    at SuppressedExample.main(SuppressedExample.java:12)
```

Zauważ, że `StackTrace`, również nazywa wyjątek wyrzucony w metodzie `close()` jako `Suppressed`. Aby się do niego dostać, można napisać kod w ten sposób:

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            throw new RuntimeException("Exception while opening Door");
        } catch (Exception e) {
            for (Throwable throwable: e.getSuppressed()) {
                System.out.println(throwable.getMessage());
            }
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
          throw new RuntimeException("Can't close my Door!");
        }
    }
}
```