Notatki - Streamy - ObjectInputStream i ObjectOutputStream

Spis treści

ObjectInputStream i ObjectOutputStream . . . 1
- Serializable . . . 2
  - transient . . . 2
  - serialVersionUID . . . 2
  - Jakie klasy serializować? . . . 3
  - Nareszcie przykład . . . 3
- Tworzenie obiektu przy deserializacji . . . 5

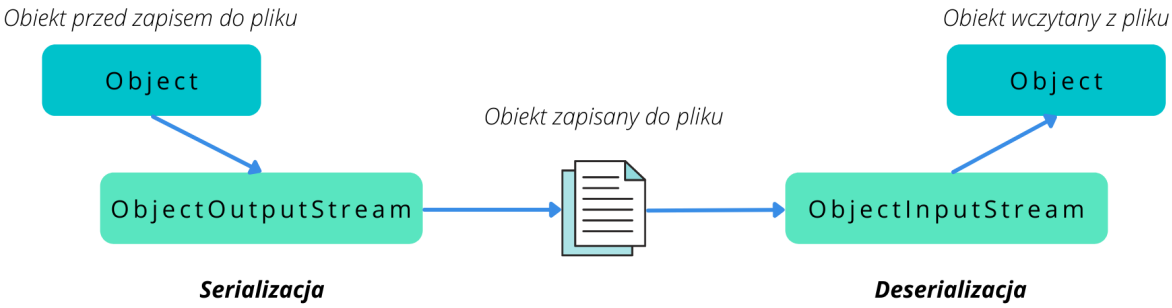
ObjectInputStream i ObjectOutputStream

A może chcemy w trakcie działania programu zapisać nasz cały obiekt do pliku a potem go wczytać?

Proces konwersji obiektu, który mamy w pamięci do jego reprezentacji pozwalającej go zapisać (np. do pliku) nazywamy `serializacją` obiektu. Proces odwrotny, czyli wczytanie obiektu (np. z pliku) do pamięci programu nazywamy `deserializacją` obiektu.

Sama `serializacja` i `deserializacja` w praktyce nie jest może często używana w tej formie, gdzie chcemy zapisać obiekt do pliku aby móc go wczytać gdzieś w przyszłości. Możesz natomiast spotkać się w praktyce z taką sytuacją gdzie pobierzesz dane z jakiegoś systemu zewnętrznego, zapiszesz je do jakiegoś obiektu w pamięci i będziesz potrzebować zapisać gdzieś ten obiekt razem z jego wszystkimi danymi. Przykładowo można wtedy zapisać taki obiekt w bazie danych, z której możemy szybciej pobrać dane niż z systemu zewnętrznego. Dzięki temu takie dane będą o wiele szybciej dostępne w Twoim programie i nie będzie trzeba ich pobierać ponownie z jakiegoś systemu zewnętrznego. Oczywiście przedstawiona sytuacja jest przykładowa 😀.

Swoją drogą to sytuacja opisana powyżej nazywana jest `Cacheowaniem danych`, a samo miejsce gdzie takie dane są przetrzymywane nazywane jest `Cache`. Czyli od teraz jak powiemy, że coś zostało "Skeszowane", to już wiesz, że dane zostały gdzieś zapisane w takiej formie, żebyśmy mieli do nich bardzo szybki dostęp niż gdybyśmy pobierali te dane bezpośrednio ze źródła.



Obraz 1. Schemat procesu serializacji i deserializacji

Serializable

`java.io.Serializable` jest interfejsem potrzebnym do tego, żeby móc przeprowadzić serializację obiektu. Jest to tak zwany **`marker interface`**, czyli interface oznaczający, że z danym obiektem ma się coś stać. Taki interfejs nie ma metod. Oznacza to, że jeżeli chcemy serializować nasz obiekt, klasa na podstawie której został stworzony, musi implementować interfejs `Serializable`. Klasy wbudowane w API Javy często implementują ten interface domyślnie. Jeżeli nasz obiekt ma pola, które są zdefiniowane przez klasy napisane przez nas, również te klasy muszą implementować interfejs `Serializable`.

Czyli jeżeli mamy obiekt samochód, który ma koła oraz drzwi i chcemy ten samochód zserializować, to zarówno samochód, koło oraz drzwi muszą implementować interfejs `Serializable`. Inaczej dostaniemy wyjątek `NotSerializableException` w trakcie działania programu.

transient

Jeżeli natomiast chcielibyśmy aby w naszym samochodzie serializowalne były koła, ale drzwi już nie, musielibyśmy oznaczyć pole 'drzwi' jako `transient`. Czyli tak samo jak w definicji pola możemy wpisać `final`, tak samo możemy wpisać `transient`. W takim przypadku Java będzie wiedziała, że drzwi są do pominięcia w procesie serializacji.

Do tego należy dodać, że pola statyczne również nie będą serializowane, co jest całkiem logiczne, bo przecież statyczny oznacza, że nie przynależy do obiektu, tylko do klasy. Serializujemy obiekty, a nie klasy.

serialVersionUID

Do całego procesu serializacji/deserializacji należy jeszcze dołożyć taką zmienną jak:

```
private static final long serialVersionUID = 1L;
```

Jeżeli chcemy oznaczyć, że nasz obiekt może być serializowalny, konieczne jest tylko implementowanie przez ten obiekt interfejsu `Serializable`. Pole `serialVersionUID` nie jest konieczne, jest to natomiast dobra praktyka, żeby zdefiniować to pole i aktualizować jego wartość za każdym razem gdy zmieniamy schemat naszej klasy (dodajemy, usuwamy lub modyfikujemy pola klasy).

Pole `serialVersionUID` oznacza wersję naszej klasy, w której obiekt był serializowany. Tak jak mamy wydania książki 1, 3, 5 i 9. Podczas serializowania obiektu, zapisywana jest również informacja o

`serialVersionUID`, dzięki czemu podczas procesu deserializacji jesteśmy w stanie stwierdzić, czy podczas serializowania mieliśmy tę samą wersję klasy jak podczas deserializacji. Jeżeli wersje się nie zgadzają, możemy szybko wykryć, że będziemy mieli problem z deserializacją.

Jeżeli sami nie zadeklarujemy tego pola podając jawnie, zostanie ono wygenerowane automatycznie, a wartość zostanie określona na podstawie "różnych aspektów klasy" (tak fajnie opisuje to dokumentacja Oracle Serializable Docs).

Jednakże mocno zaleca się generowanie tej wartości samodzielnie, gdyż jeżeli będziemy generować ją automatycznie może to prowadzić do dziwnych, trudnych do ogarnięcia błędów. Takie rozjazdy mogą się pojawić np. przy innych kompilatorach. Zaleca się też aby to pole było definiowane jako `private`.

Co do wartości pola `serialVersionUID`, często zdarza się że jego wartość jest ustawiona na `1`. Jest to robione po to, żeby przy dodawaniu kolejnych pól do klasy można było próbować deserializować obiekty, które zostały zserializowane przed dodaniem tych pól. Gdybyśmy używali wartości pola `serialVersionUID`, które byłby unikalne i zmieniane przy każdej edycji klasy, nie bylibyśmy w stanie deserializować 'starego' obiektu po dodaniu do klasy nowych pól.

Jakie klasy serializować?

W nagłówku zostało zadane dobre pytanie. Dodajmy do tego jeszcze czemu w zasadzie wszystkie klasy nie są automatycznie oznaczane jako `Serializable`? Powodem jest to, że mogą wystąpić klasy (a nawet występują niektóre klasy wbudowane), których nie chcemy serializować. Są to takie klasy, które byłoby bardzo ciężko serializować, a czasem mogłoby być to wręcz niemożliwe. Przykładem mogą być klasy Streamów. Stosując to podejście, deweloperzy muszą zatem jawnie określić które klasy chcą serializować. Czyli muszą zrobić to świadomie z pełnym rozumieniem konsekwencji tego co robią 😀.

Zagłębiając się w tematykę programowania zobaczysz, że generalnie w kodowaniu jest dużo takich zagadnień filozoficznych, czy robić coś domyślnie, czy wymusić na programistach określenie czegoś w sposób jawny.

Nareszcie przykład

Klasa Car

```
public class Car implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String name;

    private final Long year;

    // jeżeli pole byśmy oznaczyli jako transient, nie zostałoby ono zserializowane
    private final List<Door> doors;

    // ... konstruktor, gettery i toString
}
```

Klasa Door

```
public class Door implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String which;

    // ... konstruktor, gettery i toString
}
```

Klasa ObjectSerializer

```
public class ObjectSerializer {

    public static void serializeCars(List<Car> cars, File dataFile) throws IOException {
        try (ObjectOutputStream outputStream = new ObjectOutputStream(
            new BufferedOutputStream(
                new FileOutputStream(dataFile)))
        ) {
            for (Car car : cars) {
                outputStream.writeObject(car);
            }
        }
    }

    public static List<Car> deserializeCars(File dataFile) throws IOException, ClassNotFoundException {
        List<Car> cars = new ArrayList<Car>();
        try (ObjectInputStream inputStream = new ObjectInputStream(
            new BufferedInputStream(
                new FileInputStream(dataFile)))
        ) {
            while (true) {
                Object objectRead = inputStream.readObject();
                if (!(objectRead instanceof Car)) {
                    System.out.println("Object is not Car");
                }
                cars.add((Car) objectRead);
            }
            // jak dojdziemy do końca pliku, zostanie wyrzucony EOFException
        } catch (EOFException e) {
            System.out.println("end of file");
        }
        return cars;
    }

}
```

Klasa ObjectStreamExample

```
public class ObjectStreamExample {

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        List<Car> cars = new ArrayList<Car>();
        cars.add(new Car("Ford Mustang", 1969L, List.of(new Door("left"), new Door("right"))));
        cars.add(new Car("BMW 5", 2015L, List.of(new Door("left"), new Door("right"))));
        cars.add(new Car("Mercedes G-class", 2004L, List.of(new Door("left"), new Door("right"))));

        // rozszerzenie pliku nie ma znaczenia
        File destination = new File("car.whatever");

        ObjectSerializer.serializeCars(cars, destination);
        List<Car> carsDeserialized = ObjectSerializer.deserializeCars(destination);
        System.out.println(carsDeserialized);
    }
}
```

Należy w tym miejscu też dodać, że referencje `null` również mogą podlegać serializacji i deserializacji.

Tworzenie obiektu przy deserializacji

Jak dokonujemy deserializacji to te obiekty przecież muszą się jakoś stworzyć. I tutaj dzieje się coś niespodziewanego, gdyż podczas tworzenia obiektu w procesie deserializacji nie jest wywoływany żaden konstruktor z naszej klasy serializowanej. Pola, które nie były serializowane są inicjowane wartościami domyślnymi, `null`, `0` itp. Natomiast nie jest wywoływany żaden konstruktor z naszej serializowanej klasy. Żeby być precyzyjnym, dokładnie to jest wołany bezargumentowy konstruktor pierwszej superklasy w hierarchii, która nie jest serializowalna, czyli w tym przypadku - `Object`.

W przypadku pól deserializowanych, początkowo są one inicjowane wartościami domyślnymi, a potem te wartości są nadpisywane wartościami deserializowanymi.

W filmach był pokazany przykład z `Kotem` i `Persem`. `Pers` jako klasa serializowalna nie miał wywoływanych konstruktorów na etapie deserializacji, natomiast klasa `Cat`, jako pierwsza w hierarchii, która nie jest `Serializable`, miała normalnie wywołany blok inicjalizacyjny i konstruktory.

Java natomiast odtwarza wartości pól dla obiektu serializowanego na podstawie danych, które zostały zapisane podczas serializacji.

Zmienne statyczne nie są serializowalne, zatem ich wartości są przypisywane podczas inicjowania klasy.