Zapis / odczyt plików - serializacja

Czego się dowiesz

- Czym jest serializacja,
- jak zapisywać obiekty w formie binarnej do plików,
- co oznacza słowo kluczowe transient.

Wstęp

Zapis danych do plików w formie tekstowej jest przyjemny i czytelny nawet po wyłączeniu programu, a dane tak zapisane możemy później zaimportować np. do excela. Taki zapis wymaga jednak często budowania rozbudowanych metod, które pozwolą odbudować poprzedni stan aplikacji (utworzenie wszystkich obiektów na podstawie danych tekstowych z pliku).

W Javie 1.5 wprowadzono mechanizm **serializacji**, czyli zapisu do plików bezpośredniego stanu obiektów, który może być później łatwo odtworzony. Serializacja wykorzystywana jest nie tylko w kontekście plików, ale też np. przy przesyłaniu obiektów pomiędzy aplikacjami w sieci, jednak na przykładzie plików mechanizm ten jest najłatwiej zrozumieć.

Interfejs Serializable

Jeżeli chcemy, aby nasz obiekt miał możliwość zapisu serializowanego, jego klasa powinna implementować interfejs **Serializable**. Dzięki temu dajemy możliwość zapisu obiektu przy pomocy obiektu klasy ObjectOutputStream. Brak implementacji interfejsu Serializable spowoduje przy próbie zapisu obiektu wygenerowanie wyjątku NotSerializableException.

Stwórz klasę Person (osoba), której obiekty będziemy zapisywali do plików.

Person.java

```
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 38120171770882265 28L;

    private String firstName;
    private String lastName;
```

```
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Person(String firstName, String lastName) {
        setFirstName(firstName);
        setLastName(lastName);
    }

}
```

Klasa implementuje interfejs Serializable. Dzięki temu wszystkie pola typów prostych i napisy typu String zostaną zapisane do pliku. Jeżeli klasa jest bardziej złożona i zawiera również pola innych typów referencyjnych, to klasy, które je definiują także muszą implementować interfejs Serializable. Stała serialVersionUID jest elementem opcjonalnym, który pozwala na wersjonowanie klas. Jest ona generowana domyślnie przez kompilator na podstawie kilku elementów jak nazwa klasy i pola w niej zawarte. Dlatego jeśli zapiszesz jakiś obiekt, następnie w klasie Person dodasz nowe pole, nawet prywatne, to próba odczytu obiektu z wcześniej utworzonego pliku będzie niemożliwa. Wygenerowanie samodzielnie stałej serialVersionUID pozwoli zachować kompatybilność wsteczną. Korzystając z eclipse możesz to zrobić automatycznie, a w przypadku IntelliJ musisz np. po prostu wymyślić dowolną wartość.

Zapis obiektów do plików

Jeżeli mamy już zdefiniowany typ, który może być serializowany, to czas przejść do faktycznego zapisu. Posłużymy się klasami FileOutputStream oraz opakowującego ten strumień ObjectOutputStream. Jest to sytuacja podobna jak przy FileWriter i BufferedWriter.

ObjectWriter.java

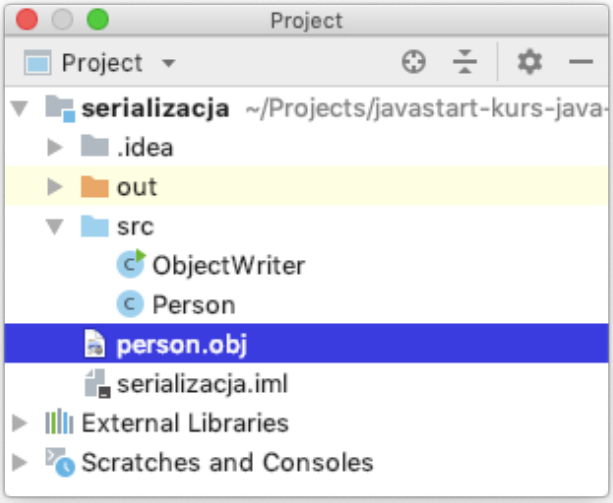
```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

class ObjectWriter {
    public static void main(String[] args) {
        String fileName = "person.obj";
        Person p1 = new Person("Jan", "Kowalski");

        try(
                var fs = new FileOutputStream(fileName);
                var os = new ObjectOutputStream(fs);
        ) {
            os.writeObject(p1);
            System.out.println("Zapisano obiekt do pliku");

        } catch (IOException e) {
            System.err.println("Błąd zapisu pliku " + fileName);
            e.printStackTrace();
        }
    }
}
```

Zdefiniowaliśmy nazwę pliku, do którego chcemy zapisać obiekt (weź pod uwagę, że nazwa i rozszerzenie pliku może być dowolne, może to być równie dobrze .txt jak i .asdf). Utworzyliśmy obiekt `Person`, który chcemy zapisać - jest to możliwe, ponieważ klasa `Person` implementuje interfejs `Serializable`.

Utworzyliśmy obiekt klasy `FileOutputStream` przekazując do konstruktora nazwę pliku. Jest to klasa stosunkowo niskopoziomowa, która pozwala na zapis plików w formie binarnej. Można ją obudować klasą `ObjectOutputStream`, która posiada pokaźny zestaw wygodnych metod do zapisu obiektów.

Musimy przechwycić dwa wyjątki - najwygodniej jest to robić za pomocą bloku try-with-resources, ponieważ nie trzeba się także martwić o zamykanie strumieni.

Po uruchomieniu programu i odświeżeniu projektu, widać, że plik faktycznie został utworzony.



Obiekt jest zapisany w formie mało czytelnej, jednak na upartego jeśli otworzysz taki plik nawet w notatniku, to możesz z niego wyłuskać odpowiednie dane.

Odczyt obiektów z plików

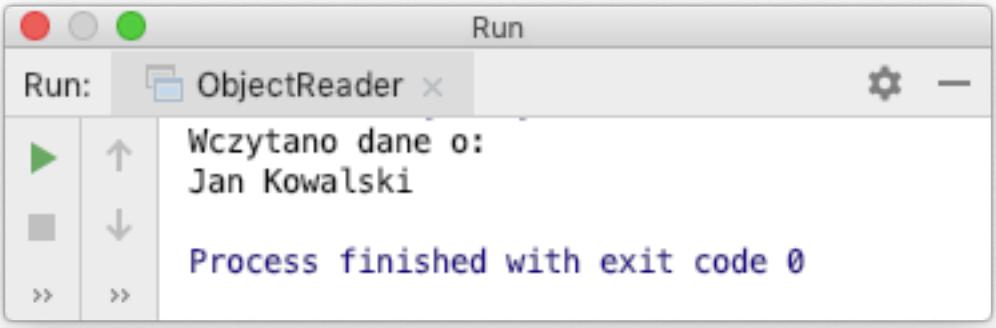
Odczyt obiektów z plików jest bardzo podobny, jednak wykorzystuje się do tego obiekty FileInputStream oraz ObjectInputStream.

ObjectReader.java

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

class ObjectReader {
    public static void main(String[] args) {
        String fileName = "person.obj";
        Person p1 = null;

        try (
                var fis = new FileInputStream(fileName);
                var ois = new ObjectInputStream(fis);
        ) {
            p1 = (Person) ois.readObject();
        } catch (ClassNotFoundException | IOException e) {
            System.err.println("Nie udało się odczytać pliku");
            e.printStackTrace();
        }

        if (p1 != null) {
            System.out.println("Wczytano dane o: ");
            System.out.println(p1.getFirstName() + " " + p1.getLastName());
        }
    }
}
```

Utworzenie obiektów FileInputStream i ObjectInputStream jest analogiczne do obiektów zapisujących dane. Odczytując dane za pomocą metody readObject() otrzymujemy zawsze referencję klasy Object i dlatego wymagane jest jego rzutowanie na odpowiedni typ. Jednocześnie metoda ta generuje kontrolowany wyjątek ClassNotFoundException, który musi być obsłużony lub zadeklarowany za pomocą throws.

Dla potwierdzenia poprawnego wczytania danych wyświetlamy informację odwołując się poprzez gettery do pól obiektu p1.



Weź pod uwagę, że mechanizm serializacji wymaga, aby w klasie nadrzędnej, która nie implementuje interfejsu Serializable był zdefiniowany konstruktor bez argumentów. W naszym przypadku jeżeli klasa Person nie implementowałaby interfejsu Serializable i próbowalibyśmy zdefiniować klasę Employee (pracownik) dziedziczącą po niej, to w poniższej sytuacji otrzymamy błąd:

```
import java.io.Serializable;

class Employee extends Person implements Serializable {

}
```

ponieważ klasa Person nie posiada konstruktora bez parametrów (ani domyślnego, bo zdefiniowany jest konstruktor z 2 parametrami). W celu jego wyeliminowania w klasie Person należałoby zdefiniować konstruktor, który nie musi nic robić:

```
public Person(){}
```

Zablokowanie serializacji - transient

Jeżeli z jakiegoś powodu zależy ci na tym, żeby jakieś składowe klasy nie były serializowane, możesz je zablokować stosując słowo kluczowe **transient**. Dzięki temu zostanie ono pominięte w procesie serializacji i informacje zostaną "zgubione".

```
class Person implements Serializable {
        private static final long serialVersionUID = 38120171770882265291;

        private transient String firstName;
        private String lastName;
//...
}
```

Pamiętaj też, że **serializacji nie podlegają żadne składowe statyczne**, ponieważ nie są one związane z konkretną instancją klasy, ale z samą klasą.

Serializacja a dziedziczenie

Jeżeli korzystamy z dobrodziejstw jakie daje nam dziedziczenie to należy wiedzieć, że jeśli jakaś klasa implementuje interfejs Serializable, to możliwość serializacji będą miały także wszystkie klasy pochodne w hierarchii dziedziczenia.

Warto też pamiętać, że jeśli rozszerzamy klasę, która nie implementuje interfejsu Serializable, a jej klasa pochodna tak, to serializacji podlegają tylko pola z klasy pochodnej, informacje z klasy bazowej zostaną pominięte.