Theory: Serialization basics

24 minutes 0 / 5 problems solved

The lifetime of all objects within a program is limited by the runtime. When we stop and then reopen the program, the information about previously created objects is lost. Imagine it happens in a computer game: this behavior is not what we actually need! Fortunately, a program can save objects to some permanent storage like a hard drive and read them back the next time the program starts.

§1. Serialization and deserialization

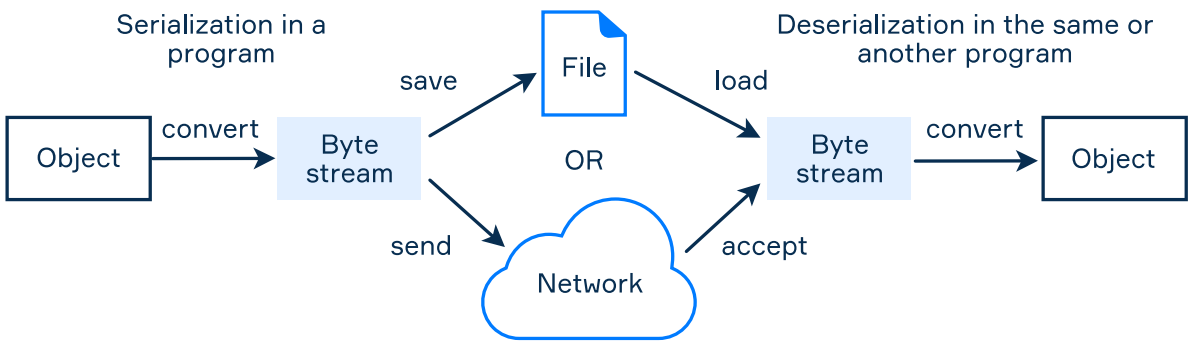
There are two processes to save and restore the state of objects between program launches: **serialization** and **deserialization**.

Serialization is a process that converts the state of an object into a stream of bytes. Objects are saved to some permanent storage for reconstruction at a later time.

Deserialization is the reverse process when the serialized byte form is used to reconstruct the actual object.

There are numerous ways to serialize Java objects into binary and text formats, for example, XML and JSON. The Java platform pays special attention to binary serialization and provides a default binary serialization protocol.

The following picture shows that an object can be serialized into a byte stream to store the data in a file or a database and then reconstructed again to be used in a program.



An additional feature of serialization is the ability to send some serialized objects through the network and then deserialize and use them in another Java program. So, serialization makes it easy for objects to be used over connected systems.

§2. Making a class serializable

To make a class serializable, it must implement the `Serializable` interface. This is a marker interface without methods. This is used to inform the compiler that the class implementing it has some special behavior.

```
class SomeClass implements Serializable {
    // fields and methods
}
```

The class being serialized can contain any primitive type and any other class as its field. All related values and objects will also be serialized. You can easily **prevent** a field from being serialized with the `transient` keyword:

```
private transient String nonSerialziedField;
```

There is a recommendation to add a special field called `serialVersionUID` for every class that implements this interface. The field should be `static`, `final` and of the `long` type:

Current topic:

Serialization basics

Topic depends on:

- The Object class
- Exception handling (Stage 7)
- Output streams
- Input streams

Topic is required for:

- Custom serialization

Table of contents:

```
private static final long serialVersionUID = 7L;
```

The `serialVersionUID` field is used to verify that the sender and the receiver of a serialized object are compatible and have loaded the classes for that object. If the version number of the sender and receiver classes don't match, then the runtime error `InvalidClassException` occurs. The matching of this value happens “under the hood” during serialization and deserialization. An error occurs only in cases of mismatch.

> Although it's not required, it is strongly recommended for a serializable class to explicitly declare its own `serialVersionUID`. Declaring and using this number guarantees a consistent `serialVersionUID` value across different Java compiler implementations. At the same time, there is no need for two different classes to have unique values for this field.

§3. Streams for objects

In Java, the serialization and deserialization mechanisms are based on the standard I/O system and byte streams. They use the `ObjectOutputStream` and `ObjectInputStream` classes accordingly.

The first class provides a method called `void writeObject(Object object)` which writes the state of the specified object to the stream . The second class has a corresponding method `Object readObject()` to restore the object. Both methods throw exceptions when something is wrong.

Here are two of our complete methods for serialization and deserialization put inside the `SerializationUtils` class for convenience.

```
class SerializationUtils {
    /**
     * Serialize the given object to the file
     */

public static void serialize(Object obj, String fileName) throws IOException {
        FileOutputStream fos = new FileOutputStream(fileName);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(obj);

        oos.close();

    }



    /**

     * Deserialize to an object from the file

     */

 public static Object deserialize(String fileName) throws IOException, ClassNotFoundException {

        FileInputStream fis = new FileInputStream(fileName);

        BufferedInputStream bis = new BufferedInputStream(fis);

        ObjectInputStream ois = new ObjectInputStream(bis);

        Object obj = ois.readObject();

        ois.close();

        return obj;

    }

}
```

Both methods use streams of different types: file streams, buffered streams and object streams. Creating new stream objects based on other streams is a common practice in Java. They wrap each other adding new functionality according to the decorator pattern. You can copy this code and try to change it to better understand the example.

Here is a small description of the given code.

- `FileOutputStream` and `FileInputStream` are used for working with a file;
- `BufferedInputStream` and `BufferedOutputStream` are optional but useful for speeding up the I/O operations;
- `ObjectOutputStream` and `ObjectInputStream` perform serialization and deserialization of objects and also invoke wrapped streams to work with the file;
- both methods also close the streams to avoid resource leaks.

§4. An example: citizens

Suppose, you need to develop an information system that persistently stores all citizens of a country. Here are two related classes for this system: `Citizen` and `Address`. Both classes implement the `Serializable` interface and contain the `serialVersionUID` fields.

The `Citizen` class represents a citizen of the country. It has a name, an address and a non-serializable field called `passport`.

```
public class Citizen implements Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private Address address;
    private transient String passport;

    // getters and setters


    @Override

    public String toString() {

        return "Citizen{" +

                "name='" + name + '\'' +

                ", passport='" + passport + '\'' +

                ", address=" + address +

                '}';

    }

}
```

The `Address` class represents an address in the country where the citizen lives. It has three string fields `state`, `city` and `street`.

```
class Address implements Serializable {
    private static final long serialVersionUID = 1L;

    private String state;
    private String city;
    private String street;

    // getters and setters


    @Override

    public String toString() {

        return "Address{" +

                "state='" + state + '\'' +

                ", city='" + city + '\'' +

                ", street='" + street + '\'' +

                '}';

    }

}
```

We removed all getters and setters from the code snippets to keep them shorter.

The value of `serialVersionUID` of a class should be increased whenever you make a change that adds/updates/removes a field. Otherwise, you may encounter exceptions during the deserialization process for objects which were saved before this change.

§5. Serializing and deserializing objects

Finally, it's time to see how serialization works. Here is a method that generates an array of citizens.

```
public static Citizen[] initCitizens() {
    Citizen mark = new Citizen();
    mark.setName("Mark Olson");
    mark.setPassport("503143798"); // the passport was set

    Address markAddress = new Address();
    markAddress.setState("Arkansas");
    markAddress.setCity("Conway");
    markAddress.setStreet("1661  Dawson Drive");



    mark.setAddress(markAddress);



    Citizen anna = new Citizen();

    anna.setName("Anna Flores");

    anna.setPassport("605143321"); // the passport was set



    Address annaAddress = new Address();

    annaAddress.setState("Georgia");

    annaAddress.setCity("Atlanta");

    annaAddress.setStreet("4353  Flint Street");



    anna.setAddress(annaAddress);



    return new Citizen[]{ mark, anna };

}
```

Actually, there are only two citizens, which doesn't sound realistic. But it is enough for an example. Note, we set passports to both citizens.

Here is the `main` method which runs all the work and uses the `SerializationUtils` class.

```
public static void main(String[] args) {
    String filename = "citizens.data";
    try {
        SerializationUtils.serialize(initCitizens(), filename);

Citizen[] citizens = (Citizen[]) SerializationUtils.deserialize(filename);
        System.out.println(Arrays.toString(citizens));
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }

}
```

It serializes two citizens in the file called `citizens.data` and then load them from the file and prints to the standard output. Here we also organized a simple exception handling. But in real-world applications, you need to think it through better.

As expected, the program outputs an array of two citizens with their fields.

```

[Citizen{name='Mark Olson', passport='null', address=Address{state='Arkansas', cit
y='Conway', street='1661  Dawson Drive'}},

Citizen{name='Anna Flores', passport='null', address=Address{state='Georgia', city
='Atlanta', street='4353  Flint Street'}}]
```

Both citizens were successfully deserialized from the file. The field `passport` is `null` since it was not serialized (`transient`).

You may be wondering what a serialized data in the file looks like. It is something like this:

```
1  ⍰⍰ ⍰ur "
[Lorg.hyperskill.problems.Citizen;i⍰ ⍰⍰⍰⍰⍰  xp   ⍰sr  org.hyperskill.problem
s.Citizen       ⍰⍰ ⍰L ⍰addresst !Lorg/hyperskill/problems/Address;L ⍰namet ⍰Ljava/
lang/String;xpsr  org.hyperskill.problems.Address       ⍰⍰ ⍰L ⍰cityq ~ ⍰L ⍰stateq
~ ⍰L ⍰streetq ~ ⍰xpt ⍰Conwayt ⍰Arkansast ⍰1661  Dawson Drivet
2
Mark Olsonsq ~ ⍰sq ~ ⍰t ⍰Atlantat ⍰Georgiat ⍰4353  Flint Streett  Anna Flores
```

It is possible to find some familiar parts here, but actually it is not a human-readable data format.

The complete code example is available on GitHub. It has a slightly different package structure which is closer to a real project. You can navigate it in the GitHub web interface.

§6. Conclusion

Now you are familiar with the concept of serialization and considered a specific example.

Here are a few points to remember:

- a class to be serialized must implement the `Serializable` interface;
- it is a good practice to add the `serialVersionUID` field to be consistent with the versions during deserialization;
- you must specify in which place to save the state of objects using I/O streams;
- use `writeObject` and `readObject` methods to serialize and deserialize any objects;
- do not forget to handle exceptions in real-world applications.

Report a typo

108 users liked this theory. **3** didn't like it. **What about you?**

😍 🙂 😐 ☹ 😡

Start practicing

Comments (10) Hints (0) Useful links (1) Show discussion