Theory: Custom serialization

19 minutes 0 / 5 problems solved

We have discussed the default serialization of objects. To disable serialization of fields we used the `transient` keyword. But sometimes this is not enough. You may need some kind of validation fields when deserializing a project. To achieve it, you should prefer to use **custom serialization**. There can be some design constraints as well: the class is expected to be changed in future releases which could break the deserialization of previously serialized objects.

§1. How to customize serialization?

Java gives us two methods that we can use to customize the serialization process. These methods are:

- `writeObject()`
- `readObject()`

Now, this could be a bit strange to you. This is just a built-in feature of Java serialization. None of these methods are inherited, overridden or overloaded. You simply have to implement these two methods in your serializing class with your custom logic for serialization. This is how it should look:

```
public class ClassName implements Serializable {

    // transient and non-transient fields

    private void writeObject(ObjectOutputStream oos) throws Exception {
       // write the custom serialization code here
    }

    private void readObject(ObjectInputStream ois) throws Exception {

       // write the custom deserialization code here

    }

}
```

When you call the `oos.writeObject()` method, **JVM** first checks whether you have implemented the `writeObject()` method in your serializing class. If so, JVM executes the code inside that method instead of doing default serialization. Similarly, JVM will call the `readObject()` method in the serializing class when you call the `ois.readObject()` method.

§2. Initialize transient variables

Let's come back to our previous example. We know that `oos.writeObject()` doesn't serialize the password. We can solve this problem by initializing the password when deserializing the object. We can do it like this

```
public class User implements Serializable {
    String userName = "admin";
    transient String password = "password";

    private void readObject(ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        password = new String(" ");
    }
}
```

Here, we don't have to implement the `writeObject()` method as we don't want to add anything to the serialization process. We only have to implement the `readObject()` method. Our first line is `ois.defaultReadObject()`

Current topic:

Custom serialization ...

Topic depends on:

× Serialization basics ...

Table of contents:

↑ Custom serialization

§1. How to customize serialization?

§2. Initialize transient variables

§3. More examples of custom serialization

§4. Conclusion

Feedback & Comments

which will perform the default deserialization. It means that after the `ois.defaultReadObject()` method you have the normal values for non-transient fields and null values for transient fields.

Next, we instantiate the password with `password = new String(“ ”)`. The code in the `TransientExample` class will remain the same.

When you run the updated code, you will get the following output:

```
Before serialization : Username : admin, Password : password
After serialization and deserialization : Username : admin, Password :
```

You can see that the password is not null, it is just empty.

§3. More examples of custom serialization

There are many other reasons to use custom serialization. For example:

- When you want to **encrypt** important fields of a class
- When you want to use a more **compressed** serialization.

Let’s see how to encrypt the fields of a class. We have two functions, `encrypt` and `decrypt`, that we can use for encryption. Their implementations are not of importance here, let's just assume that they are available to us.

This code uses both `writeObject()` and `readObject()` methods:

```
public class User implements Serializable {
    String userName = "admin";
    transient String password = "password";

    private void writeObject(ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String encryptPassword = encrypt(password);
        oos.writeObject(encryptPassword);
    }



    private void readObject(ObjectInputStream ois) throws Exception {

        ois.defaultReadObject();

        String password = decrypt((String)ois.readObject());

    }

}
```

First, `oos.defaultWriteObject()` in `writeObject()` method will perform the default serialization on non-transient fields. Then we will encrypt the password using `encrypt()` method. Next, we will serialize the encrypted password. Likewise, `ois.defaultReadObject()` method will deserialize non-transient fields. Then using the `readObject()` method, you can retrieve the `encryptPassword` field. Finally, use the `decrypt()` method to decrypt the variable.

§4. Conclusion

In this topic, we've covered a couple of things. First, we've explained why we need to prevent some fields from being serialized and how the `transient` keyword can help you do that. Next, we've discussed custom serialization and how to implement it with just two methods. It's obvious that Java provides us with an easy way to customize the serialization, but what matters is how we can write an effective code for custom serialization.

Report a typo

92 users liked this theory. **20** didn't like it. **What about you?**

Start practicing

Comments (7) Hints (0) Useful links (0) Show discussion