Quiz yourself: Secure serialization and deserialization (advanced)

JAVA SE

Quiz yourself: Secure serialization and deserialization (advanced)

See if you know how to make rapidly changing code easier to maintain.

by Simon Roberts and Mikalai Zaikin

June 22, 2020

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. The “intermediate” and “advanced” designations refer to the exams rather than to the questions, although in almost all cases, “advanced” questions will be harder. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

The objective here is to make serialization and deserialization secure while making often-changing code easier to maintain.

For this advanced-level Java SE 11 quiz, imagine you are working on an application that makes extensive use of serialization of business objects. Fields are being added to, and removed from, the business object classes over time; this is causing problems with old serialized representations becoming inconsistent with new code.

Which change best facilitates managing these changes and keeping the business object working? Choose one.

A. Implement `java.io.Externalizable` and choose which variables to serialize.

The answer is A.

B. Make unwanted instance variables `private`.

The answer is B.

C. Make unwanted instance variables `transient`.

The answer is C.

D. Add a `serialPersistentFields` array and refer to it from `writeObject / readObject` when writing or reading required variables.

The answer is D.

Answer. The default behavior of the serialization mechanism saves and restores all the instance state of an entire object. Fields that are added or removed might break the application or, in the worst case, they can introduce security issues. If you rely on default serialization, you should provide a `static final long serialVersionUID` constant and update it every time you modify a class structure in incompatible ways.

Let's look at the options proposed for mitigating these problems.

Option A suggests implementing the `java.io.Externalizable` interface to give control over which part of the object is saved and restored, and to let you choose which variables to serialize. Certainly, this is a very powerful mechanism. However, in this model, the programmer becomes entirely responsible for handling the serialization and deserialization at a low level. Thus, the programmer must strictly maintain the order of serializing and deserializing, and also handle all evolution of the class. All this is error-prone and makes maintenance much harder. So, while this approach is very powerful and might help, it's likely too much work for the problem at hand.

Option A has the power to mitigate the problem, so it's not wrong. However, since this quiz question asks for the *best* option, you need to reserve judgment until you know if another option is better.

Option B suggests using the `private` access modifier to control serialization. However, `private` does not alter a field's serialization. Given that this option simply makes no difference to the situation, you can reject option B as incorrect immediately.

Option C suggests marking some fields as `transient`. This keyword prevents a variable being written to the serialized form, and so it might address the problem at hand. However, `transient` still has some weaknesses. One is that in rapidly changing classes, it's generally easier to maintain code that takes an "allowlist" approach, rather than a "blocklist" approach. An allowlist approach means that the programmer identifies variables that *should* be included in serialization, rather than identifying those that *should not* be included.

A second problem is that `transient` does not address the issue that changing the type of a serialized variable breaks

compatibility. Although this option certainly has some merit, it's not as expressive as option A.

Option D suggests providing a `private static final` array with the reserved name `serialPersistentFields`. This array is populated with `ObjectStreamField` objects that specify the names and types of the serializable fields, like this:

```
class BusinessObject implements Serializable
    List list;
    private static final ObjectStreamField[]
      { new ObjectStreamField("list", List.cl
}
```

This mechanism allows the programmer to explicitly allowlist the fields that should be serialized, and it supports adding or removing fields with some constraints. When code evolves, it's possible to use the `private` methods `writeObject` and `readObject` to perform custom mapping from serialized copies, for example:

```
class BusinessObject implements Serializable
    MyList list; // Changed data type, was pr

    private void readObject(ObjectInputStream
        // Obtain all the data from the origi
        ObjectInputStream.GetField gf = ois.r
        // Extract original list from that da
        // new data type, MyList from the ele
        this.list = MyList.createFromList(gf.
    }
}
```

Now that you have determined three of the four options can provide a degree of mitigation for the problem at hand, let's consider which might be the *best* option. This might be easiest if you visualize the three valid options in a table alongside their features.

	`Externalizable`: Option A	`transient`: Option C	`serialPersistentFields`: Option D
Allows excluding fields from serialized form?	Yes	Yes	Yes
Whitelist approach, which simplifies adding and removing fields?	Yes	No	Yes
Supports changing field types?	Yes	No	Yes
Provides complete control of all aspects of serialization?	Yes	No	No
Field read/write order defined in a single place?	No	Yes	Yes
Simple programming mechanism, which prevents implementation errors?	No	Yes	Yes

From the table, it seems reasonable to conclude that option D, the use of `serialPersistentFields`, provides the best balance, since it's quite easy to implement and change, and yet

it is powerful and expressive enough to deal with the great majority of changes that are likely to occur in a real project.

By contrast, with option A, the use of `Externalizable` is the most expressive option, but it requires great care in implementation. It will be tiresome to change the code correctly with each small change in the data structure. Therefore, option A is incorrect.

Option C, using `transient`, is very simple, but it is incapable of handling changes in data type. Further, since using `transient` requires attention when new fields are added that should not be serialized, this is likely to be more error-prone than an allowlist approach. Therefore, option C is incorrect.

For these reasons, using the `serialPersistentFields` mechanism is considered to be the best general-case approach, and option D is correct.

The correct answer is option D.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Share this Page