JAVA SE

Quiz yourself: Read and write objects by using serialization

Test your knowledge of the java.io.Serializable interface.

by Simon Roberts and Mikalai Zaikin

November 9, 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. 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 of this Java SE 11 quiz is to read and write objects by using serialization. Imagine you are working on a project that includes these two classes:

```
final class Item {
  String title;

  public Item(String title) {
    this.title = title;
  }

  String getTitle() {
    return title;
  }
}

class Order implements Serializable {
  private int id;
  private List<Item> items = new ArrayList<It

  public Order(int id) {
    this.id = id;
  }

  public void addItem(Item i) {
      items.add(i);
    }

    public void getItem(int i) {
      items.get(i);
    }
  }
```

Also imagine a colleague has already started working on a requirement that needs the ability to serialize order objects. Another requirement prohibits modifications of the `Item` class code.

Which of the following changes allows an order to be properly serialized?

A. Add this new method to the `Order` class:

```
private void writeObject(ObjectOutputStream s
}
```

The answer is A.

B. Add this new method to the `Order` class:

```
private void writeObject(ObjectOutputStream s
  s.defaultWriteObject();
  for (Item itm : items) {
    s.writeObject(itm.getTitle());
  }
}
```

The answer is B.

C. Add this new method to the `Order` class and mark the `items` instance variable `transient`:

```
private void writeObject(ObjectOutputStream s
  s.defaultWriteObject();
  for (Item itm : items) {
    s.writeObject(itm.getTitle());
  }
}
```

The answer is C.

D. None of the above

The answer is D.

Answer. Here are some background rules regarding the behavior of the serialization system. When you serialize an object, the default behavior is to serialize all of the object's instance data transitively—that is, to serialize the instance data of the object, to serialize each object to which the object refers, and so on until the entire object graph is serialized. (Note that `static` members are better viewed as elements of the class, not as elements of the object.)

This behavior can be modified in a number of ways, most simply by marking a field as `transient`. Another rule in the serialization system is that an attempt to serialize an instance of any class is permitted only if that class implements the `Serializable` interface. This interface is a "marker" interface (an idea that predates the introduction of annotations with Java 5). The serialization system will refuse to serialize an object unless the defining class of that object implements the `Serializable` interface—you can imagine that the `Serializable` interface marks the object as a permitted target for serialization.

Now, let's look at the situation presented in the question. Here are three observations:

- First, the `Order` class implements the `Serializable` marker interface. Presumably this is a result of the partial refactoring work already performed.
- Second, the `Order` class has two fields. One is an `int` and the other is an `ArrayList`. Neither is marked `transient`, and both are serializable members of core Java.
- Third, the list contains `Item` objects, but the `Item` class does not implement the `Serializable` interface. You are prohibited from altering the source code of the `Item` class and from subclassing the class, since it is `final`.

Given these three observations, you can conclude that the default serialization must fail when it attempts to serialize any instance of the `Item` class found in the list. Further, since you are prohibited from altering the `Item` class, it will always be impossible for those objects to be serialized.

All is not necessarily lost, however, since it is possible to modify the serialization mechanism and instead of serializing the `Item` directly, serialize some other data—probably the `String` representing the title—from which an `Item` might be reconstructed.

The documentation for the `java.io.Serializable` interface says the following regarding how a customized serialization mechanism can be implemented by adding some new methods to a class:

> Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutput
private void readObject(java.io.ObjectInputSt
```

> The default mechanism for saving the `Object`’s fields can be invoked by calling `out.defaultWriteObject`. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

Now, let’s look at the options for answering the proposed question.

Option A suggests implementing the `writeObject` method and directly calling the `defaultWriteObject` method from within it. However, this approach simply invokes the default serialization mechanism without changing anything. That will fail as soon as there is an attempt to serialize any `Item` instance and throw a `java.io.NotSerializableException`. Because of this, you can see that option A is incorrect.

Option B seems at first sight to address the problem by taking the approach of serializing the title of each `Item`. Since the `title` is a `String`, this aspect would work. However, the overall solution still fails because the first action of the `writeObject` method is still to call the `defaultWriteObject` utility, and that will still fail since it will try to serialize the `Item` objects. This solution will cause a `NotSerializableException` to be thrown before execution ever gets to the manual serialization behavior. Because of this, option B is also incorrect.

Option C improves the situation further because it marks the `items` variable as `transient`. This excludes the list from the default serialization and, therefore, the code does not crash when it calls `defaultWriteObject`. Further, the title of each item is saved to the output stream individually, because the `java.lang.String` object is serializable. From this you can determine that this code would run without throwing any exceptions and a representation of the object would be written to the output file.

However, the question requires “proper serialization.” What does that mean? The `Serializable` documentation notes: “The `writeObject` method is responsible for writing the state of the object for its particular class *so that the corresponding* `readObject` *method can restore it*.”

The problem with the code for option C is that there is no reliable way to deserialize the result. The failure is because the *size* of the list is not written, so there is no way to know how many times to read item titles and re-create `Item` instances. Because the resulting serialized data cannot be restored, option C is also incorrect.

Given that options A, B, and C are all incorrect, you can conclude that the correct answer in this case is option D: “None of the above.”

Of course, it’s rather unsatisfying simply to have determined that all these attempts are unsuccessful. What *should* you do to properly serialize these objects, so you can reliably restore them from the result?

To accomplish that, you can build on the proposal of option C (marking the items list as `transient` and representing the individual `Item` objects using only the `String` representing an item’s title), but you must also save the size of the list in the serialized data stream. You should do this before writing any of the `Items` (which still must be written as `String`s representing the item titles). It seems reasonable to store the size using a primitive `int` value. On the basis of this, a valid method for the `Order` class might look as follows (don’t forget that you must still mark the `items` variable as `transient`):

```
private void writeObject(ObjectOutputStream s
    // does not write items because the list
    s.defaultWriteObject();
    // write size of the list
    s.writeInt(items.size());
    for (Item itm : items) {
        s.writeObject(itm.getTitle());
    }
}
```

And the corresponding deserialization code in the `Order` class would look like this:

```
private void readObject(ObjectInputStream s)
    // restore the non-transient parts of the
    s.defaultReadObject();
    // determine the number of Items to be re
    int n = s.readInt();
    // create a list for the Item objects
    items = new ArrayList<>();
    // read the correct number of titles, cre
    // and insert that item into the list
    for (int i = 0; i < n; i++) {
        String title = (String)s.readObject()
        items.add(new Item(title));
    }
}
```

Conclusion: 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

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices