BEST PRACTICES FOR SERIALIZATION IN JAVA

This article lays down some of the best practices which you can use during your design or code reviews, and assumes that you are familiar with the basics of serialization in java. I have learned many of these best practices from Effective Java by Joshua Bloch, which tells how to use java to best effect.

Best Practices for Serialization in Java

1. **Consider using custom serialized form and don't use the default serialized form** provided by Java even if you have decided to design only the APIs perfectly for time being releasing a throwaway implementation and then replace it with a better one in future release. This is because, if you use default serialized form, some implementation specific details in addition to the actual data that needs to be serialized, like a linked list implementation detail, will leak into the exported API and if you change the implementation in a future release you won't be able to successfully serialize and de-serialize between different versions of the class. Serializing and de-serializing unnecessary data also has performance problems as it consumes excessive space and time.
2. **If** your custom serialized form is identical to the default serialize form, **you** may **decide to use the default serialized form**. Even in this case **you should implement writeObject() and readObject() methods and call defaultWriteObject() and defaultReadObject() methods respectively from within them**. You can write any other data including transient data which are not taken care by these methods before and or after calling defaultWriteObject() and can then read it back in the same order along with defaultReadObject(). These methods gives you the flexibility to add non-transient instance fields in a later release while preserving backward and forward compatibility.
3. **Use serialization specific annotations like @Serial** for tags **and @SerialData** for methods **within javadoc comments.**
4. Every **instance field that can be made transient must be made so**.
5. Transient variables will be initialized to their default values during default serialization or when you call defaultReadObject() and if the default values of transient variables are unacceptable, **restore transient fields to acceptable values after you call defaultReadObject().**
6. You **should use proper synchronization, if needed**, during serialization process, like in any other case.
7. **Declare an explicit serial version uid** in every serialization class you write. A new version of a class will be compatible to the older one only if they have the same serial version uid number. By running the servialver utility on a class version, we can get the automatically generated value of serial version uid for that class.
8. The readObject will return a newly created object every time you de-serialize and hence singleton objects should not in general implement serializable. However **if you have a singleton that implement**

serializable, to ensure singleton for a serializable object, we should implement a readResolve() method and return the original instance instead of the newly created one.

9. **A better solution than using readResolve is to use enums for singleton** as you don't have to worry about serialization related issues or even synchronization related issues. However if you have a serializable instance control class whose instances are not known at compile time, you will not be able to represent this class as an enum type.

10. **If you depend on readResolve for instance control by discarding the newly created object and returning original instance, all instance fields of object reference types should be transient** or an attacker might make use of the fact that deserialization happens before readResolve is run and he can get a reference to the deserialized object before it is discarded by the readResolve method.

11. **Do not use the writeUnshared() and readUnshared() methods** introduced in java 1.4 for preventing object reference attacks, as these methods apply well only to base level objects according to java specification. Instead you can use defensive copying inside readObject method.

12. The **readObject() method must check the validity of input stream data and make defensive copies**of the instance reference fields, if required, similar to constructors. However the fields should not be final as you need to change their values. For final fields, you can consider the serialization proxy pattern as explained in the book 'Effective Java 2' by Joshua Bloch.

13. 'Effective Java 2' Joshua Bloch explains another approach called serialization proxy pattern where you actually serialize the object of another class called serialization proxy class with your class data and when this serialization proxy class object is de-serialized, it will construct an object of the original type and return back. The serialization proxy pattern uses writeReplace() on the original class to return a serialization proxy class instance during serialization and uses readReplace() on the serialization proxy class to return the original class or a compatible class during deserialization. EnumSet in java uses this pattern to return a RegularEnumSet or a JumboEnumSet based on the number of elements. Since this pattern is not compatible with extendable classes (as it is tied to the original class), **it is recommended to use serialization proxy pattern instead of writing readObject() and writeObject() on a class that is not extendable by its clients.**

14. **If we have used a default serialized form, we need to make sure we test thoroughly for binary compatibility as well as semantic compatibility.** In binary compatibility, we check wehther it is possible to serialize and de-serialize between different versions of the class. In semantic compatibility we need to make sure the de-serialized object is an exact replica of the original object.

15. **You should provide parameter less constructors on non-serialized classes designed for inheritance**, and if it needs to implement serializable. When an instance of a class is de-serialized, their constructor is not run, and is the same for all serializable super classes. However, for a non serializable super class, the super class's non-parameterize (or default) constructor is called, and if not available, an InvalidClassException will be thrown.

16. **Inner classes should not implement serializable**, mainly due to the use of compiler generated synthetic fields. A static member class can, however implement serializable.

17. **You should not call an overridable method from within a readObject() or readObjectNoData() method,** similar to a constructor, as the object is not fully constructed yet, which can result in the invocation of a subclass method before the super class is fully constructed.

18. **Always check again if you really need serialization**, like, if the class needs to participate in a framework that required serialization, then serialization is required.

Source : http://javajee.com/best-practices-for-serialization-in-java