March 16, 2017 | Vojtech Ruzicka

Should I explicitly declare serialVersionUID?

May 3, 2016 by

Should you explicitly declare serialVersionUID or leave it to be automatically generated?

What is serialVersionUID?

This is the description of serialversionUID according the javadoc of Serializable:

> The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an `InvalidClassException`. A serializable class can declare its own serialVersionUID explicitly by declaring a field named `"serialVersionUID"` that must be static, final, and of type `long`:
>
> ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
>
> If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification.

Why to omit

If a class implements serializable and does not explicitly declare serialVersionUID, it is automatically generated depending on the class details such as fields and methods. Great. By not declaring the version number, it is automatically generated and changes every time the class changes. Therefore, two different versions of the same class are automatically rendered incompatible without any developer action.

That is good. This way it could never happen, that two versions of the class still have the same version number, because someone forgot to update it. And let's face it, it is bound to happen. Sooner or later. It is like with the javadocs – after changes or some bigger refactorings, developers tend to forget (or are too lazy) to update comments to reflect the latest changes. The same with version numbers. It's likely that at least one of the changed classes will keep its original version number.

Now two things can happen depending on the severity of the change – it is going to be compatible and incompatible. Incompatible will result in runtime exception during deseralization. That is at least fail fast and it is immediately obvious that something is wrong. However compatible changes are far more dangerous as it is not obvious right away that something is not right. It may just result in subtle bugs and odd behavior. This is often hard to detect and debug. Automatic generation of version number prevents this as new number is generated whenever the class changes.

Why to declare

Backwards compatibility

If you are using serialization for a short-term storage, locally by your own application only, you probably should not worry about explicitly declaring serialVersionUID. However, if the storage period is longer (let's say more than one release cycle), you need to consider backwards compatibility. In this case auto-generated version number is no longer sufficient as every change makes the new version incompatible. However, the goal is to maintain the backwards compatibility as long as possible.

The good news is that you can keep the class backwards compatible if you're making only compatible changes such as adding new fields. In that case you want your version number to stay the same. Otherwise update it. This is especially important if your class is used by third parties or remote apps, where you may not be able to make them update to the new version once backwards compatibility is lost.

JVM compatibility issues

The good news is that the process of calculating serialVersionUID is described in detail in the official Java Object Serialization Specification. Ideally, using that algorithm, all the compilers would auto-generate the same version id for the same class. Unfortunately, that is not the case. In reality generated numbers vary per compiler. Which means in distributed environments, it is likely that the two different JVMs (eg. Oracle vs IBM java) will produce different version numbers for the same class rendering the serialization incompatible. Unless you are 100% sure (and is there such thing?) that the same JVM will be used across all the participants, it is very risky to let the version number be auto-generated. In such cases it is thus recommended to explicitly declare serialVersionUID to prevent version number mismatch.

When to update

Okay, so maybe it is a good idea to have version number explicitly declared after all. Now the question is – when should you update it? Many developers update whenever they make change to the class. This way the compatibility across multiple JVMs is no longer an issue. That's good. However, there is no backwards compatibility at all since every change to the class results in new version number and is incompatible with all the other versions.

Actually, there are two types of changes (as defined by the Serialization Specification) – compatible and incompatible. When you make incompatible change, you need to update the version number as well. Otherwise, the deserialization will fail. There's no way around. However, if you make some changes, which are compatible, you can keep the current version number. Backwards compatibility is still maintained. You need to make sure that the new version will behave correctly with the old versions – some fields may be null and so on. You may even need to write custom deserialization in case the migration is not so simple. In either case, you need to know which changes are compatible and which are not:

Incompatible changes

- Deleting fields
- Moving classes up or down the hierarchy
- Changing a nonstatic field to static or a nontransient field to transient
- Changing the declared type of primitive field
- Changing the writeObject or readObject method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not.
- Changing a class from Serializable to Externalizable or vice versa
- Changing a class from a non-enum type to an enum type or vice versa
- Removing either Serializable or Externalizable
- Adding the writeReplace or readResolve method

Compatible changes

- Adding fields
- Adding classes
- Removing classes

- Removing writeObject/readObject methods
- Adding java.io.Serializable
- Changing the access to a field
- Changing a field from static to nonstatic or transient to nontransient

For further details, see the Java Object Serialization Specification.

Making sure nothing breaks

When you use explicit version number you need to make sure you properly update it when necessary or make sure the class is backwards compatible. The best approach would be to build a habit of always checking whether the class you are just changing is seralizable and to manage the version number appropriately. As a safeguard, it is handy to have serialization unit tests when aiming for backwards compatibility. Keep serialized objects of the previous versions of the class (with the same version id), which are supposed to be compatible with current implementation. Then in the unit tests make sure that you can still deserialize those without problems and that the state of such objects is as expected. It is well worth the effort as it allows you to reveal serialization problems before it is too late.

Conclusion

Serialization has many pitfalls including maintaining backwards compatibility. The first consideration should be whether you should be using serialization at all. You should consider whether for your specific scenario it is not better to use different approach. Maybe something JVM independent, so it is easier to integrate with other non-jvm systems. You may consider using XML or JSON and OXM or JSON-object mapping (such as GSON) instead of serialization.

If you decide to use serialization, it is generally recommended to declare serialVersionUID. However, you need to be very careful. Every time you make change to a serializable class, you need to make sure you make the changes backwards-compatible or update the version number.

Exception may be if your framework requires you to have serialized classes (or inherit from them), but they are not actually stored or sent to a remote environment. In such case it may be better just to omit the version number altogether.

What is your approach to serialVersionUID? Do you declare it always or never? Do you have any personal experience of failures due to autogenerated version numbers? Or because you forgot to update the declared number? Please share your opinions in the comments bellow.

Filed Under: Java
