serialVersionUID?

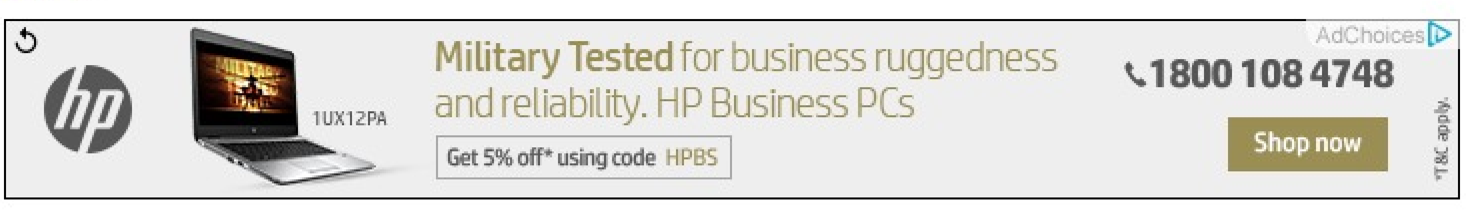
When you serialize an object using Serialization mechanism (by implementing Serializable interface), there is a possibility that you may face versioning issues and because of these versioning issues, you will not be able to deserialize the object. Thats not a good thing. But first, what is this versioning issue that is troubling your serialization process?

Well, lets say you created a class, instantiated it, and wrote it out to an object stream. That flattened object sits in the file system for some time. Meanwhile, you update the class file, perhaps adding a new field. Now try to read the flattened object. hmmmm.. An exception "java.io.InvalidClassException" will be thrown. You dont understand where it went wrong because the changes in the class seem perfectly fine for you.

What is serialVersionUID?

Before we start discussing about the solution for this problem, lets first see what is actually causing this problem? Why should any change in a serialized class throw InvalidClassException? During object serialization, the default Java serialization mechanism writes the metadata about the object, which includes the class name, field names and types, and superclass. All this information is stored as part of the serialized object. When you deserialize the object, this information is read to reconsitute the object. But to perform the deserialization, the object needs to be identified first and this will be done by serialVersionUID. So everytime an object is serialized the java serialization mechanism automatically computes a hash value using ObjectStreamClass's computeSerialVersionUID() method by passing the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value, the serialVersionUID.

Now when the serilaized object is retrieved, the JVM first evaluates the serialVersionUID of the serialized class and compares the serialVersionUID value with the one of the object. If the sserialVersionUID values match then the object is said to be compatible with the class and hence it is de-serialized. If not InvalidClassException exception is thrown.


AdChoices
1UX12PA
Military Tested for business ruggedness and reliability. HP Business PCs
Get 5% off* using code HPBS
1800 108 4748
Shop now
*T&C apply.

The above issue not only occurs when the object is flattened and saved but also when the object is flattened and sent to other JVMs when you implement RMI. Lets assume you have a client/server environment where client is using SUN's JVM in windows while server is using JRockit in Linux. Client sends a serializable class with default generated serialVersionUID (e.g 123L) to server over socket, while server may generate a different serialVersionUID (e.g 124L) during deserialization process, and raise an unexpected InvalidClassExceptions. Since the default serialVersionUID computation is highly sensitive to class details and may vary from different JVM implementation, an unexpected InvalidClassExceptions will result here.

What's the solution for this versioning issue?

The solution is very simple. Instead of relying on the JVM to generate the serialVersionUID, you explicitly mention (generate) the serialVersionUID in your class. The syntax is:

private final static long serialVersionUID = <integer value>

Yes, its a static, private variable in the class. Once you define the serialVersionUID in your class explicitly, you dont need to update it until and unless you make the incompatible changes. Look at the example below that explains the issue and importance of maintaining serialVersionUID.

```
class TestSUID implements Serializable {
    private static final long serialVersionUID = 1L;

    private int someId;

    public TestSUID (int someId) {
        this.someId = someId;
    }

    public int getSomeId() {
        return someId;
    }
}


public class SUIDTester {
    public static void main(String args[]) throws Exception {
        File file = new File("temp.ser");
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        TestSUID writeSUID = new TestSUID(1);
        oos.writeObject(writeSUID);
        oos.close();

        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);

        TestSUID readSUID = (TestSUID) ois.readObject();
        System.out.println("someId : " + readSUID.getSomeId());
        ois.close();
    }
}
```

In this example, we have created a Serializable class with serialVersionUID = 1L and saved the "some id" value in the "temp.ser" file. Now change the serialVersionUID value of "TestSUID" class to 2L and try to just read the "temp.ser" file. It will throw "InvalidClassException". The reason is the version change and exactly this is the reason for maintaining the version.

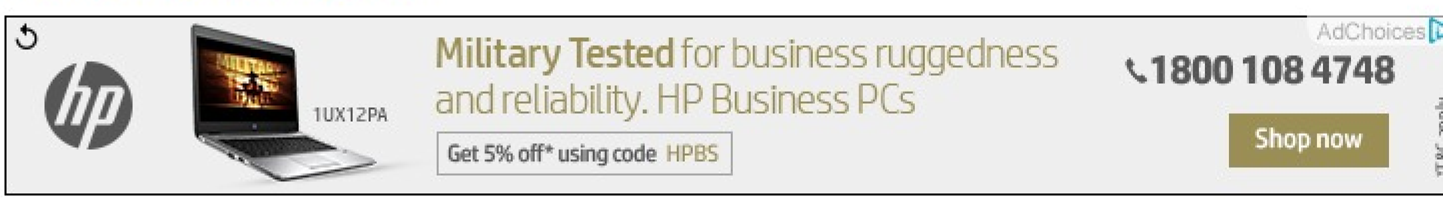
```
Exception in thread "main" java.io.InvalidClassException:
SerializeMe; local class incompatible: stream classdesc
serialVersionUID = 1, local class serialVersionUID = 2
```

When should you update serialVersionUID?

Adding serialVersinUID manually to the class does not mean that it should never be updated and never need not be updated. There is no need to update the serialVersionUID if the change in the class is compatible but it should be updated if the change is incompatible. What are compatible and incompatible changes? A compatible change is a change that does not affect the contract between the class and the callers.

The compatible changes to a class are handled as follows:

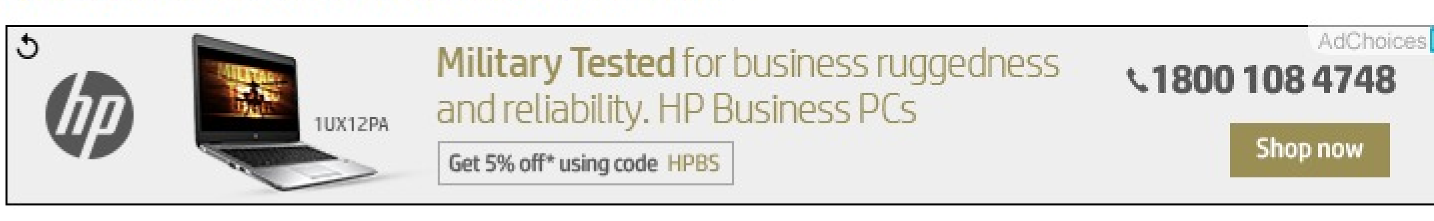
- Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to nondefault values.
- Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
- Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.


AdChoices
1UX12PA
Military Tested for business ruggedness and reliability. HP Business PCs
Get 5% off* using code HPBS
1800 108 4748
Shop now

- Adding writeObject/readObject methods - If the version reading the stream has these methods then readObject is expected, as usual, to read the required data written to the stream by the default serialization. It should call defaultReadObject first before reading any optional data. The writeObject method is expected as usual to call defaultWriteObject to write the required data and then may write optional data.
- Removing writeObject/readObject methods - If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.
- Adding java.io.Serializable - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the InvalidClassException is thrown.
- Changing the access to a field - The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to nonstatic or transient to nontransient - When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained. The incompatible changes that may occur while evolving a class are:

- Deleting fields - If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
- Moving classes up or down the hierarchy - This cannot be allowed since the data in the stream appears in the wrong sequence.
- Changing a nonstatic field to static or a nontransient field to transient - When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
- Changing the declared type of a primitive field - Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the 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. The default field data must consistently either appear or not appear in the stream.
- Changing a class from Serializable to Externalizable or visa-versa is an incompatible change since the stream will contain data that is incompatible with the implementation in the available class.
- Removing either Serializable or Externalizable is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.


AdChoices
1UX12PA
Military Tested for business ruggedness and reliability. HP Business PCs
Get 5% off* using code HPBS
1800 108 4748
Shop now

- Adding the writeReplace or readResolve method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

How to generate a serialVersionUID?

There are two ways to generate the serialVersionUID.

- Go to commanline and type "serialver <>. SerialVersionUID wil be generated. Copy, paste the same into your class.
  In Windows, generate serialVersionUID using the JDK's graphical tool like so : use Control Panel | System | Environment to set the classpath to the correct directory
  run serialver -show from the command line
  point the tool to the class file including the package, for example, finance.stock.Account - without the .class
  (here are the serialver docs for both Win and Unix)
- One way is through Eclipse IDE. After you implement Serializable interface and save the class, eclipse will show a warning asking you to add the serialVersionUID and it provides you the option to generate it or use the default one. Click on the link to generate the serialVersionUID and it will generate it for you and adds it to the class.

Finally few guidelines for serialVersionUID :

- always include it as a field, for example: "private static final long serialVersionUID = 7526472295622776147L; " include this field even in the first version of the class, as a reminder of its importance
- do not change the value of this field in future versions, unless you are knowingly making changes to the class which will render it incompatible with old serialized objects
- new versions of Serializable classes may or may not be able to read old serialized objects; it depends upon the nature of the change; provide a pointer to Sun's guidelines for what constitutes a compatible change, as a convenience to future maintainers