

Q1) What is Serialization?

Serializable is a marker interface. When an object has to be transferred over a network (typically through rmi or EJB) or to persist the state of an object to a file, the object Class needs to implement Serializable interface. Implementing this interface will allow the object converted into bytestream and transfer over a network.

So...what is serialVersionUID?

The JVM associates a version (*long*) number with each serializable class. It is used to verify that the saved and loaded objects have the same attributes and thus are compatible on serialization.

This number can be generated automatically by most IDEs and is based on the class name, its attributes and associated access modifiers. Any changes result in a different number and can cause an *InvalidClassException*.

If a serializable class doesn't declare a *serialVersionUID*, the JVM will generate one automatically at run-time. However, it is highly recommended that each class declares its *serialVersionUID* as the generated one is compiler dependent and thus may result in unexpected *InvalidClassExceptions*.

Q2) What is use of serialVersionUID?

Ans) 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. This class definition is stored as a part of the serialized object. This stored metadata enables the deserialization process to reconstitute the objects and map the stream data into the class attributes with the appropriate type

Everytime an object is serialized the java serialization mechanism automatically computes a hash value. ObjectOutputStream's *computeSerialVersionUID()* method passes the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value. The serialVersionUID is also called *suid*.

So when the serilaize object is retrieved , the JVM first evaluates the *suid* of the serialized class and compares the *suid* value with the one of the object. If the *suid* 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.

Changes to a serializable class can be compatible or incompatible. Following is the list of changes which are compatible:

- Add fields
- Change a field from static to non-static
- Change a field from transient to non-transient
- Add classes to the object tree

List of incompatible changes:

- Delete fields
- Change class hierarchy
- Change non-static to static
- Change non-transient to transient
- Change type of a primitive field

So, if no *serialVersionUID* is present, inspite of making compatible changes, jvm generates new *serialVersionUID* thus resulting in an exception if prior release version object is used .

The only way to get rid of the exception is to recompile and deploy the application again.

If we explicitly mention the *serialVersionUID* using the statement:

```
private final static long serialVersionUID = <integer value>
```

then if any of the mentioned compatible changes are made the class need not to be recompiled. But for incompatible changes there is no other way than to compile again.

2.1 Same serialVersionUID

Same *serialVersionUID* , there is no problem during the deserialization process

Bash

```
javac Address.java
```

```
javac WriteObject.java
```

```
javac ReadObject.java
```

```
java WriteObject
```

```
java ReadObject
```

Street : wall street Country : united states

2.2 Different serialVersionUID

In Address.java, **change the serialVersionUID to 2L** (it was 1L), and compile it again.

Bash

```
javac Address.java

java ReadObject

java.io.InvalidClassException: Address; local class incompatible:
stream classdesc serialVersionUID = 1, local class serialVersionUID = 2
...

    at ReadObject.main(ReadObject.java:14)
```

The “**InvalidClassException**” will raise, because you write a serialization class with **serialVersionUID “1L”** but try to retrieve it back with updated serialization class, **serialVersionUID “2L”**.

Inheritance and Composition

When a class implements the *java.io.Serializable* interface, all its sub-classes are serializable as well. On the contrary, when an object has a reference to another object, these objects must implement the *Serializable* interface separately, or else a *NotSerializableException* will be thrown:

```
1 public class Person implements Serializable {
2     private int age;
3     private String name;
4     private Address country; // must be serializable too
5 }
```

If one of the fields in a serializable object consists of an array of objects, then all these objects must be serializable as well, or else a *NotSerializableException* will be thrown.

Q3) What is the need of Serialization?

Ans) The serialization is used :-

- To send state of one or more object's state over the network through a socket.
- To save the state of an object in a file.
- An object's state needs to be manipulated as a stream of bytes.

Q4) Other than Serialization what are the different approach to make object Serializable?

Ans) Besides the Serializable interface, at least three alternate approaches can serialize Java objects:

- For object serialization, instead of implementing the Serializable interface, a developer can implement the **Externalizable** interface, which extends Serializable. By implementing Externalizable, a developer is responsible for implementing the writeExternal() and readExternal() methods. As a result, a developer has sole control over reading and writing the serialized objects.
- **XML serialization** is an often-used approach for data interchange. This approach lags runtime performance when compared with Java serialization, both in terms of the size of the object and the processing time. With a speedier XML parser, the performance gap with respect to the processing time narrows. Nonetheless, XML serialization provides a more malleable solution when faced with changes in the serializable object.
- Finally, consider a "roll-your-own" serialization approach. You can write an object's content directly via either the ObjectOutputStream or the DataOutputStream. While this approach is more involved in its initial implementation, it offers the greatest flexibility and extensibility. In addition, this approach provides a performance advantage over Java serialization.

Q5) Do we need to implement any method of Serializable interface to make an object serializable?

Ans) No. Serializable is a Marker Interface. It does not have any methods.

Q6) What happens if the object to be serialized includes the references to other serializable objects?

Ans) If the object to be serialized includes references to the other objects, then all those object's state also will be saved as the part of the serialized state of the object in question. The whole

object graph of the object to be serialized will be saved during serialization automatically provided all the objects included in the object's graph are serializable.

Q7) What happens if an object is serializable but it includes a reference to a non-serializable object?

Ans- If you try to serialize an object of a class which implements serializable, but the object includes a reference to an non-serializable class then a 'NotSerializableException' will be thrown at runtime.

```
public class NonSerial {
    //This is a non-serializable class
}

public class MyClass implements Serializable {
    private static final long serialVersionUID = 1L;
    private NonSerial nonSerial;
    MyClass(NonSerial nonSerial){
        this.nonSerial = nonSerial;
    }
    public static void main(String [] args) {
        NonSerial nonSer = new NonSerial();
        MyClass c = new MyClass(nonSer);
        try {
            FileOutputStream fs = new FileOutputStream("test1.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c);
            os.close();
        } catch (Exception e) { e.printStackTrace(); }
        try {
            FileInputStream fis = new FileInputStream("test1.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
```

```

        c = (MyClass) ois.readObject();
        ois.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

On execution of above code following exception will be thrown;

java.io.NotSerializableException: NonSerial

at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java)

Q8) Are the static variables saved as the part of serialization?

Ans) No. The static variables belong to the class are not the part of the state of the object so they are not saved as the part of serialized object.

Q9)What is a transient variable?

Ans) These variables are not included in the process of serialization and are not the part of the object's serialized state.

Q10) What will be the value of transient variable after de-serialization?

Ans) It's default value.

e.g. if the transient variable in question is an int, it's value after deserialization will be zero.

```

public class TestTransientVal implements Serializable {
    private static final long serialVersionUID = -22L;
    private String name;
    transient private int age;
    TestTransientVal(int age, String name) {
        this.age = age;
        this.name = name;
    }
    public static void main(String [] args) {

```

```

TestTransientVal c = new TestTransientVal(1,"ONE");
System.out.println("Before serialization:" + c.name + " " + c.age);
try {
    FileOutputStream fs =new FileOutputStream("testTransient.ser");
    ObjectOutputStream os = new ObjectOutputStream(fs);
    os.writeObject(c);
    os.close();
} catch (Exception e) { e.printStackTrace(); }
try {
    FileInputStream fis =new FileInputStream("testTransient.ser");
    ObjectInputStream ois =new ObjectInputStream(fis);
    c = (TestTransientVal) ois.readObject();
    ois.close();
} catch (Exception e) { e.printStackTrace(); }
System.out.println("After de-serialization:" + c.name + " " + c.age);
}
}

```

Result of executing above piece of code –

Before serialization: - Value of non-transient variable ONE Value of transient variable 1

After de-serialization:- Value of non-transient variable ONE Value of transient variable 0

Explanation –

The transient variable is not saved as the part of the state of the serailized variable, it's value after de-serialization is it's default value.

Q11) Does the order in which the value of the transient variables and the state of the object using the defaultWriteObject() method are saved during serialization matter?

Ans) Yes, while restoring the object's state the transient variables and the serializable variables that are stored must be restored in the same order in which they were saved.

Q12) How can one customize the Serialization process? or What is the purpose of implementing the writeObject() and readObject() method?

Ans) When you want to store the transient variables state as a part of the serialized object at the time of serialization the class must implement the following methods –

```
private void writeObject(ObjectOutputStream outStream) {
    //code to save the transient variables state
    //as a part of serialized object
}
private void readObject(ObjectInputStream inStream) {
    //code to read the transient variables state
    //and assign it to the de-serialized object
}

public class TestCustomizedSerialization implements Serializable {
    private static final long serialVersionUID = -22L;
    private String noOfSerVar;
    transient private int noOfTranVar;
    TestCustomizedSerialization(int noOfTranVar, String noOfSerVar) {
        this.noOfTranVar = noOfTranVar;
        this.noOfSerVar = noOfSerVar;
    }
    private void writeObject(ObjectOutputStream os) {
        try {
            os.defaultWriteObject();
            os.writeInt(noOfTranVar);
        } catch (Exception e) { e.printStackTrace(); }
    }
    private void readObject(ObjectInputStream is) {
        try {
            is.defaultReadObject();
            int noOfTransients = (is.readInt());
```



```

    } catch (Exception e) {
        e.printStackTrace(); }
    }
    public int getNoOfTranVar() {
        return noOfTranVar;
    }

```

The value of transient variable 'noOfTranVar' is saved as part of the serialized object manually by implementing writeObject() and restored by implementing readObject().

The normal serializable variables are saved and restored by calling defaultWriteObject() and defaultReadObject() respectively. These methods perform the normal serialization and de-serialization process for the object to be saved or restored respectively.

Q13) If a class is serializable but its superclass is not, what will be the state of the instance variables inherited from super class after deserialization?

Ans) The values of the instance variables inherited from superclass will be reset to the values they were given during the original construction of the object as the non-serializable super-class constructor will run.

E.g.

```

public class ChildSerializable extends ParentNonSerializable implements Serializable {
    private static final long serialVersionUID = 1L;
    String color;
    ChildSerializable() {
        this.noOfWheels = 8;
        this.color = "blue";
    }
}

```

```

public class SubSerialSuperNotSerial {
    public static void main(String [] args) {
        ChildSerializable c = new ChildSerializable();
        System.out.println("Before : - " + c.noOfWheels + " " + c.color);
    }
}

```

```

try {
    FileOutputStream fs = new FileOutputStream("superNotSerail.ser");
    ObjectOutputStream os = new ObjectOutputStream(fs);
    os.writeObject(c);
    os.close();
} catch (Exception e) { e.printStackTrace(); }

try {
    FileInputStream fis = new FileInputStream("superNotSerail.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    c = (ChildSerializable) ois.readObject();
    ois.close();
} catch (Exception e) { e.printStackTrace(); }

System.out.println("After :- " + c.noOfWheels + " " + c.color);
}
}

```

Result on executing above code –

Before : - 8 blue

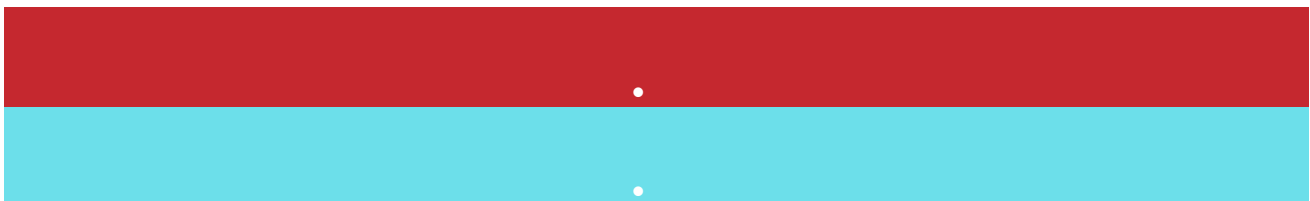
After :- 4 blue

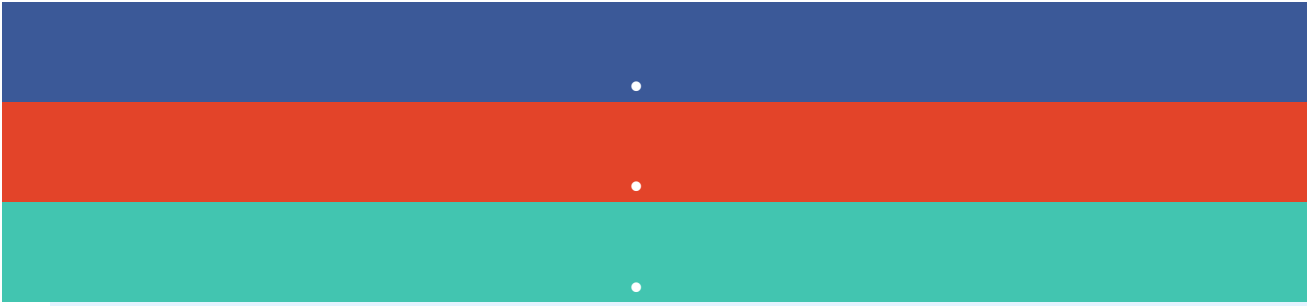
The instance variable 'noOfWheels' is inherited from superclass which is not serializable. Therefore while restoring it the non-serializable superclass constructor runs and its value is set to 8 and is not same as the value saved during serialization which is 4.

Transient vs Static variable java

[Dec 29, 2013](#) [35 Comments](#) [MBA Gautam](#) 4 min read

SHARE





This entry is part 30 of 34 in the series [Core Java Course](#)

Here i will show you what is the difference between Static and Transient. In any way these two things are completely different with different scope but people use to ask me this question every time so i am mentioning it here. For previous Serialization articles click [here\(Part I\)](#) and [here\(Part II\)](#).

Static Variable

Static variables belong to a class and not to any individual instance. The concept of serialization is concerned with the object's current state. Only data associated with a specific instance of a class is serialized, therefore static member fields are ignored during serialization.

Transient Variable

While serialization if you don't want to save state of a variable. You have to mark that variable as Transient. Environment will know that this variable should be ignored and will not save the value of same.

Note*: *Even concept is completely different and reason behind not saving is different still people get confused about the existence of both. As in both the case variables value will not get saved.*