Java Serialization Example Tutorial, Serializable, serialVersionUID

Pankaj December 27, 2013 Java

Java Serialization was introduced in JDK 1.1 and it is one of the important feature of Core Java.

Java Serialization API allows us to convert an Object to stream that we can send over the network or save it as file or store in DB for later usage. Deserialization is the process of converting Object stream to actual Java Object to be used in our program. Java Synchronization process seems very easy to use at first but it comes with some trivial security and integrity issues that we will look in the later part of this article. We will look into following topics in this tutorial.

Java Serializable Interface

If you want a class object to be serializable, all you need to do it implement the `java.io.Serializable` interface. Serializable is a marker interface and has no fields or methods to implement. It's like an Opt-In process through which we make our classes serializable.

Serialization process is implemented by `ObjectInputStream` and `ObjectOutputStream`, so all we need is a wrapper over them to either save it to file or send it over the network. Let's see a simple Serialization example.

Employee.java

```java
package com.journaldev.serialization;

import java.io.Serializable;

public class Employee implements Serializable {

//	private static final long serialVersionUID = -6470090944414208496L;

	private String name;
	private int id;
	transient private int salary;
//	private String password;

	@Override
	public String toString(){
		return "Employee{name="+name+",id="+id+",salary="+salary+"}";
```

```java
    }

    //getter and setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

//  public String getPassword() {
//      return password;
//  }
//
//  public void setPassword(String password) {
//      this.password = password;
//  }

```

```
}
```

Notice that it’s a simple java bean with some properties and getter-setter methods. If you want an object property to be not serialized to stream, you can use **transient** keyword like I have done with salary variable.

Now suppose we want to write our objects to file and then deserialize it from the same file. So we need utility methods that will use `ObjectInputStream` and `ObjectOutputStream` for serialization purposes.

SerializationUtil.java

```java
package com.journaldev.serialization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * A simple class with generic serialize and deserialize method implementatior
 *
 * @author pankaj
 *
 */
public class SerializationUtil {

    // deserialize to Object from given file
    public static Object deserialize(String fileName) throws IOException,
            ClassNotFoundException {
        FileInputStream fis = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Object obj = ois.readObject();
```

```java
            ois.close();
            return obj;
        }

        // serialize the given object and save it to file
        public static void serialize(Object obj, String fileName)
                throws IOException {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(obj);

            fos.close();
        }

}
```

Notice that the method arguments work with Object that is the base class of any java object. It's written in this way to be generic in nature.

Now let's write a test program to see Java Serialization in action.

SerializationTest.java

```java
package com.journaldev.serialization;

import java.io.IOException;

public class SerializationTest {

        public static void main(String[] args) {
            String fileName="employee.ser";
            Employee emp = new Employee();
            emp.setId(100);
```

```java
		emp.setName("Pankaj");
		emp.setSalary(5000);
		
		//serialize to file
		try {
			SerializationUtil.serialize(emp, fileName);
		} catch (IOException e) {
			e.printStackTrace();
			return;
		}
		
		Employee empNew = null;
		try {
			empNew = (Employee) SerializationUtil.deserialize(fileName);
		} catch (ClassNotFoundException | IOException e) {
			e.printStackTrace();
		}
		
		System.out.println("emp Object::"+emp);
		System.out.println("empNew Object::"+empNew);
	}
	
	
}
```

When we run above test program, we get following output.

```
emp Object::Employee{name=Pankaj,id=100,salary=5000}
empNew Object::Employee{name=Pankaj,id=100,salary=0}
```

Since salary is a transient variable, it's value was not saved to file and hence not retrieved in the new object. Similarly **static** variable values are also not serialized since they belongs to class and not object.

Class Refactoring with Serialization and serialVersionUID

Java Serialization permits some changes in the java class if they can be ignored. Some of the changes in class that will not affect the deserialization process are:

- Adding new variables to the class
- Changing the variables from transient to non-transient, for serialization it's like having a new field.
- Changing the variable from static to non-static, for serialization it's like having a new field.

But for all these changes to work, the java class should have **serialVersionUID** defined for the class. Let's write a test class just for deserialization of the already serialized file from previous test class.

DeserializationTest.java

```java
package com.journaldev.serialization;

import java.io.IOException;

public class DeserializationTest {

    public static void main(String[] args) {

        String fileName="employee.ser";
        Employee empNew = null;

        try {
            empNew = (Employee) SerializationUtil.deserialize(fileName);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }

        System.out.println("empNew Object::"+empNew);

    }

}
```

Now uncomment the “password” variable and it’s getter-setter methods from Employee class and run it. You will get below exception;

```
java.io.InvalidClassException: com.journaldev.serialization.Employee; local cl
    at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:604)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1601)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1514)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:175
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1347)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
    at com.journaldev.serialization.SerializationUtil.deserialize(Serializatic
    at com.journaldev.serialization.DeserializationTest.main(DeserializationTe
empNew Object::null
```

The reason is clear that serialVersionUID of the previous class and new class are different. Actually if the class doesn’t define serialVersionUID, it’s getting calculated automatically and assigned to the class. Java uses class variables, methods, class name, package etc to generate this unique long number. If you are working with any IDE, you will automatically get a warning that “The serializable class Employee does not declare a static final serialVersionUID field of type long”.

We can use java utility “serialver” to generate the class serialVersionUID, for Employee class we can run it with below command.

```
SerializationExample/bin$serialver -classpath . com.journaldev.serialization.Em
```

Note that it’s not required that the serial version is generated from this program itself, we can assign this value as we want. It just need to be there to let deserialization process know that the new class is the new version of the same class and should be deserialized of possible.

For example, uncomment only the serialVersionUID field from the `Employee` class and run `SerializationTest` program. Now uncomment the password field from Employee class and run the `DeserializationTest` program and you will see that the object stream is deserialized successfully because the change in Employee class is compatible with serialization process.

Java Externalizable Interface

If you notice the serialization process, it’s done automatically. Sometimes we want to obscure the object data to maintain it’s integrity. We can do this by implementing `java.io.Externalizable` interface and provide implementation of *writeExternal()* and *readExternal()* methods to be used in serialization process.

Person.java

```java
package com.journaldev.externalization;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Person implements Externalizable{

    private int id;
    private String name;
    private String gender;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(name+"xyz");
        out.writeObject("abc"+gender);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        id=in.readInt();
        //read in the same order as written
        name=(String) in.readObject();
        if(!name.endsWith("xyz")) throw new IOException("corrupted data");
        name=name.substring(0, name.length()-3);
        gender=(String) in.readObject();
        if(!gender.startsWith("abc")) throw new IOException("corrupted data");
        gender=gender.substring(3);
    }

    @Override
    public String toString(){
        return "Person{id="+id+",name="+name+",gender="+gender+"}";
    }
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

}
```

Notice that I have changed the field values before converting it to Stream and then while reading reversed the changes. In this way, we can maintain data integrity of some sorts. We can throw exception if after reading the stream data, the integrity checks fail. Let’s write a test program to see it in action.

ExternalizationTest.java

```java
package com.journaldev.externalization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ExternalizationTest {

    public static void main(String[] args) {

        String fileName = "person.ser";
        Person person = new Person();
        person.setId(1);
        person.setName("Pankaj");
        person.setGender("Male");

        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(person);
            oos.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        FileInputStream fis;
        try {
            fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Person p = (Person)ois.readObject();
            ois.close();
            System.out.println("Person Object Read="+p);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }

        }

}
```

When we run above program, we get following output.

```
Person Object Read=Person{id=1,name=Pankaj,gender=Male}
```

So which one is better to be used for serialization purpose. Actually it’s better to use Serializable interface and by the time we reach at the end of article, you will know why.

Java Serialization Methods

We have seen that java serialization is automatic and all we need is implementing Serializable interface. The implementation is present in the ObjectInputStream and ObjectOutputStream classes. But what if we want to change the way we are saving data, for example we have some sensitive information in the object and before saving/retrieving we want to encrypt/decrypt it. That’s why there are four methods that we can provide in the class to change the serialization behavior.

If these methods are present in the class, they are used for serialization purposes.

1. **readObject(ObjectInputStream ois)**: If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from stream.
2. **writeObject(ObjectOutputStream oos)**: If this method is present in the class, ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usage is to obscure the object variables to maintain data integrity.
3. **Object writeReplace()**: If this method is present, then after serialization process this method is called and the object returned is serialized to the stream.
4. **Object readResolve()**: If this method is present, then after deserialization process, this method is called to return the final object to the caller program. One of the usage of this method is to implement Singleton pattern with Serialized classes. Read more at Serialization and Singleton.

Usually while implementing above methods, it's kept as private so that subclasses can't override them. They are meant for serialization purpose only and keeping them private avoids any security issue.

Serialization with Inheritance

Sometimes we need to extend a class that doesn't implement Serializable interface. If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.

This is one place, where readObject() and writeObject() methods really help. By providing their implementation, we can save the super class state to the stream and then retrieve it later on. Let's see this in action.

SuperClass.java

```java
package com.journaldev.serialization.inheritance;

public class SuperClass {

    private int id;
    private String value;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }


}
```

SuperClass is a simple java bean but it’s not implementing Serializable interface.

SubClass.java

```java
package com.journaldev.serialization.inheritance;

import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class SubClass extends SuperClass implements Serializable, ObjectInputV

    private static final long serialVersionUID = -1322322139926390329L;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString(){
        return "SubClass{id="+getId()+",value="+getValue()+",name="+getName()+
    }

    //adding helper method for serialization to save/initialize super class st
    private void readObject(ObjectInputStream ois) throws ClassNotFoundExcepti
        ois.defaultReadObject();

        //notice the order of read and write should be same
        setId(ois.readInt());
        setValue((String) ois.readObject());

    }

    private void writeObject(ObjectOutputStream oos) throws IOException{
        oos.defaultWriteObject();

        oos.writeInt(getId());
        oos.writeObject(getValue());
    }

    @Override
    public void validateObject() throws InvalidObjectException {
        //validate the object here
        if(name == null || "".equals(name)) throw new InvalidObjectException('
        if(getId() <=0) throw new InvalidObjectException("ID can't be negative
    }

}
```

Notice that order of writing and reading the extra data to the stream should be same. We can put some logic in reading and writing data to make it secure.

Also notice that the class is implementing `ObjectInputValidation` interface. By implementing *validateObject()* method, we can put some business validations to make sure that the data integrity is not harmed.

Let's write a test class and see if we can retrieve the super class state from serialized data or not.

InheritanceSerializationTest.java

```
package com.journaldev.serialization.inheritance;

import java.io.IOException;

import com.journaldev.serialization.SerializationUtil;

```

```java
public class InheritanceSerializationTest {

    public static void main(String[] args) {
        String fileName = "subclass.ser";

        SubClass subClass = new SubClass();
        subClass.setId(10);
        subClass.setValue("Data");
        subClass.setName("Pankaj");

        try {
            SerializationUtil.serialize(subClass, fileName);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        try {
            SubClass subNew = (SubClass) SerializationUtil.deserialize(fileNam
            System.out.println("SubClass read = "+subNew);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

When we run above class, we get following output.

```
SubClass read = SubClass{id=10,value=Data,name=Pankaj}
```

So in this way, we can serialize super class state even though it’s not implementing Serializable interface. This strategy comes handy when the super class is a third-party class that we can’t change.

Serialization Proxy Pattern

Java Serialization comes with some serious pitfalls such as;

- The class structure can't be changed a lot without breaking the serialization process. So even though we don't need some variables later on, we need to keep them just for backward compatibility.
- Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, user role is serialized and an attacker change the stream value to make it admin and run malicious code.

Serialization Proxy pattern is a way to achieve greater security with Serialization. In this pattern, an inner private static class is used as a proxy class for serialization purpose. This class is designed in the way to maintain the state of the main class. This pattern is implemented by properly implementing *readResolve()* and *writeReplace()* methods.

Let us first write a class which implements serialization proxy pattern and then we will analyze it for better understanding.

Data.java

```java
package com.journaldev.serialization.proxy;

import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.Serializable;
```

```java

public class Data implements Serializable{

    private static final long serialVersionUID = 2087368867376448459L;

    private String data;

    public Data(String d){
        this.data=d;
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    @Override
    public String toString(){
        return "Data{data="+data+"}";
    }

    //serialization proxy class
    private static class DataProxy implements Serializable{

        private static final long serialVersionUID = 8333905273185436744L;

        private String dataProxy;
        private static final String PREFIX = "ABC";
        private static final String SUFFIX = "DEFG";

        public DataProxy(Data d){
            //obscuring data for security
```

```
                this.dataProxy = PREFIX + d.data + SUFFIX;
            }

            private Object readResolve() throws InvalidObjectException {
                if(dataProxy.startsWith(PREFIX) && dataProxy.endsWith(SUFFIX)){
                return new Data(dataProxy.substring(3, dataProxy.length() -4));
                }else throw new InvalidObjectException("data corrupted");
            }

        }

        //replacing serialized object to DataProxy object
        private Object writeReplace(){
            return new DataProxy(this);
        }

        private void readObject(ObjectInputStream ois) throws InvalidObjectExcepti
            throw new InvalidObjectException("Proxy is not used, something fishy")
        }
}
```

- Both `Data` and `DataProxy` class should implement Serializable interface.
- `DataProxy` should be able to maintain the state of Data object.
- `DataProxy` is inner private static class, so that other classes can't access it.
- `DataProxy` should have a single constructor that takes Data as argument.
- `Data` class should provide *writeReplace()* method returning `DataProxy` instance. So when Data object is serialized, the returned stream is of DataProxy class. However DataProxy class is not visible outside, so it can't be used directly.
- `DataProxy` class should implement *readResolve()* method returning `Data` object. So when Data class is

deserialized, internally DataProxy is deserialized and when it’s readResolve() method is called, we get Data object.

- Finally implement *readObject()* method in Data class and throw `InvalidObjectException` to avoid hackers attack trying to fabricate Data object stream and parse it.

Let’s write a small test to check whether implementation works or not.

SerializationProxyTest.java

```java
package com.journaldev.serialization.proxy;

import java.io.IOException;

import com.journaldev.serialization.SerializationUtil;

public class SerializationProxyTest {

    public static void main(String[] args) {
        String fileName = "data.ser";

        Data data = new Data("Pankaj");

        try {
            SerializationUtil.serialize(data, fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            Data newData = (Data) SerializationUtil.deserialize(fileName);
            System.out.println(newData);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

When we run above class, we get below output in console.

```
Data{data=Pankaj}
```

If you will open the data.ser file, you can see that DataProxy object is saved as stream in the file.

Download Java Serialization Project
271 downloads

That's all for Java Serialization, it looks simple but we should use it judiciously and it's always better not to rely on default implementation. Download the project from above link and play around with it to learn more.

Hungry for More? Check out these amazing posts:

- **How to write Object to File in Java**
- **Java Singleton Design Pattern Best Practices with Examples**
- **Java IO Tutorial**
- **Java System (java.lang.System) Class**
- **Java Exception Handling Tutorial with Examples and Best Practices**

- **Java 8 Features for Developers - lambdas, Functional interface, Stream and Time API**
- **40 Java Collections Interview Questions and Answers**

java serializable java serialization

Written by Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

5 Responses to "Java Serialization Example Tutorial, Serializable, serialVersionUID"

Siddheshwar says:

September 15, 2014 at 12:34 am

Its Awesome. Simply great.

Reply

Nikos says:

September 5, 2014 at 2:30 pm

Thank you very much for your help!!!

Reply

Shubham Srivastava says:

July 23, 2014 at 3:42 am

If the class doesn't define serialVersionUID, it's getting calculated automatically by java compiler and assigned to the class as a private variable.

Reply

Swapnil says:

August 26, 2014 at 4:23 am

Really nice fully covered article on java serialization..thanks a lot

Reply

Mufim says:
June 17, 2014 at 6:43 pm

Please explain public vs private vs protected serialversionUID.Since we have to make unique serialversionUID why it is not private by default

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Post Comment

▣ Sign me up for the JournalDev newsletter!

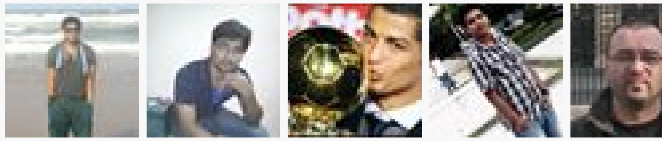
Google™ Custom Search

Be My Friend

JournalDev

Like

8,307 people like JournalDev.



Facebook social plugin

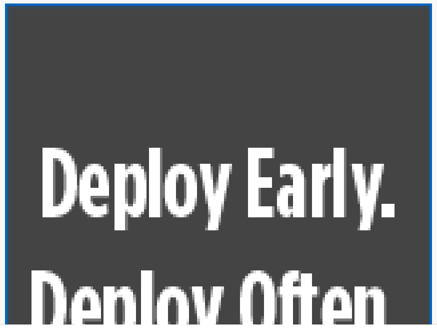
Pankaj Kumar

Follow

Recent Posts

Creating your own jQuery/CSS Thumbnail Image Slider/Image Gallery

10 jQuery Image Slider Plugins for



Categories

Database

MongoDB

Awesome Design

HTML5 Progress Bar Cool Candy Stripe and jQuery Animation Effects

Portlet Lifecycle & Threading Issues

jQuery UI Tabs (Horizontal and Vertical) Example with Java Web Application Integration

5 Worthy CDN Providers to Expedite Your WordPress WebSite

jQuery slideUp(), slideDown() and slideToggle() sliding effects Examples

Portlet Basics & Enhancing Portlets Example Tutorial

jQuery AJAX Servlet JSP Web Application Integration Example Tutorial

How to Deploy Maven WAR Project in Tomcat Automatically

jQueue dequeue() method usage

advertise here

Hibernate

Interview Questions

Java

- Design Patterns
- Java EE

jQuery

JSF

Maven

PHP

Portal and Portlets

PrimeFaces

Random

Resources

- HTML

Scripts

with example

jQuery show, hide and toggle effects for HTML elements

jQuery DOM Elements Manipulation – Get Attribute, Set Attribute, Remove Attribute, Class

jQuery delay function to delay animations

jQuery clearQueue() method to clear queued animations, css changes

Pages

- About
- Advertise
- Download Free eBooks
- Java Interview Questions
- Privacy Policy
- Tutorials
- Write For Us

Softwares

Spring

Struts 2

Web Design

Wordpress

Popular Tags

- hibernate
- hibernate example
- hibernate tutorial
- java Arrays
- java design patterns
- java file
- java interview questions
- java io
- java String
- java Thread
- java xml
- jQuery
- primefaces
- primefaces example
- primefaces tutorial
- spring tutorial
- struts 2
- struts 2 tutorial

Tutorials

Java Tutorials: Java IO Tutorial, Java Regular Expressions Tutorial, Java Multi-Threading Tutorial, Java Logging API Tutorial, Java Annotations Tutorial,Java XML Tutorial, Java Collections Tutorial, Java Generics Tutorial, Java Exception Handling, Java Reflection Tutorial, Java Design Patterns, JDBC Tutorial

Java EE: Servlet JSP Tutorial, Struts2 Tutorial, Spring Tutorial, Hibernate Tutorial

Web Services: Apache Axis 2 Tutorial, Jersey Restful Web Services Tutorial

Misc: Memcached Tutorial

Resources: Free eBooks, My Favorite Web Hosting

Interview Questions

Java String Interview Questions, Java Multi-Threading Interview Questions, Java Programming Interview Questions, Java Interview Questions, Java Collections Interview Questions, Java Exception Interview Questions

Servlet Interview Questions, JSP Interview Questions, Struts2 Interview Questions, JDBC Interview Questions, Spring Interview Questions, Hibernate Interview Questions

Be My Friend

Facebook

Google+

Twitter