**Java Serialization (Serializable-Externalizable)-2024-Precise**

**What is Serialization, Why Serialization ?**

Serialization is the process of storing the object graph in the file.

1. Instead of recreating the object again and again, we can serialize once and deserialize to get the object in case of Garbage collection.
2. **Persistence** Object can be serialized into byte array to store in DB for later use.
3. **Deep Copy**: simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
4. **Caching**: Really just an application of the above, but sometimes an object takes some time to build, but would only take few seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed.
5. **Cross JVM Synchronization**: Serialization works across different JVMs that may be running on different architectures.

**Significance of Serial Version UID**

private static final long *serialVesionUID* = 1L;

So everytime an object is serialized, the java serialization mechanism automatically computes a hash value using **ObjectStreamClass’s computeSerialVersionUID()** method.

**long serialVersionUID = ObjectStreamClass.lookup(YourObject.getClass()).getSerialVersionUID();**

**long lookupID = ObjectStreamClass.*lookup*(Emp.class).getSerialVersionUID();**

**How serialization works**

**Now when the serialized object is retrieved, the JVM first evaluates the serialVersionUID of the serialized class and compares the serialVersionUID value with the one of the objects**. **If the serialVersionUID 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.** It is always recommended to generate and use the serial version uid to avoid any conflict or issues for the system running in another JVM like IBM, Oracle etc. **How serialization happens? JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal method. If the object does not support Externalizable but implement Serializable, then the object is saved using ObjectOutputStream**.

**private final static long serialVersionUID = <integer value> .**

**Why is it recommended to specify your own serialVersionUID ?**

**Serial Version UID is useful in the following case.**

1. If you want to forbid the deserialization of already existing serialized object, in that case you can

change the serial version UID.

2. If you do not provide the serial version UID, other JVMs like Jikes(IBM), JRockit may calculate in a different manner. It is always better to provide the UID.

3. In case of Network, if you are transfering the object, the destination may calculate the UID in a different manner.

4. If you do not provide the serial version UID, java will automatically calculate the serial version uid everytime at the time of serialization and deserialization which may create performance overhead.

Always remember that the **object's class file and methods are not saved; only the object's state is saved**.

**How to get or calculate the serial version UID of a class ?**

There are two ways you can obtain the serial version UID of a class.

1. You can get from the **ObjectStreamClass** using the methods **lookupAny() and lookup()**.

2. You can obtain by calculating the serial version UID of a class.

**lookupAny() and lookup()**

**ObjectStreamClass.lookupAny(Emp.class).getSerialVersionUID();**

**ObjectStreamClass.lookup(Emp.class).getSerialVersionUID();**

**lookupAny() method is used to get the class descriptor for any class, regardless of whether it implements Serializable or not.** **If the class does not implement Serializable interface, the default**

**serial version UID will be 0**. ObjectStreamClass.lookupAny(Emp.class).getSerialVersionUID(); will be 0 in case of non-serilizable class.

**lookup()** method is used to find the descriptor for a class that can be serialized. It creates an ObjectStreamClass instance if one does not exist yet for class. Null is returned if the specified class does not implement java.io.Serializable or java.io.Externalizable. If the class does not implement Serializable interface, the invocation of the following line will throw NullPointerException.

ObjectStreamClass.lookup(Emp.class).getSerialVersionUID(); will throw NullPointerException in case of non-serializable class.

**How to calculate the serial version UID using Reflection**

The code is given below.

**Method method = ObjectStreamClass.class.getDeclaredMethod("computeDefaultSUID", Class.class);**

**method.setAccessible(true);**

**Long serialVerUID = (Long)method.invoke(null, Emp.class);**

**System.out.println("Calculated Serial Version UID ->"+serialVerUID.longValue());**

**Best way to find out Serial Version UID**

Integer i = Integer.*valueOf*(9);  
**long serialVersionID = ObjectStreamClass.*lookup*(i.getClass()).getSerialVersionUID();**  
System.*out*.println(serialVersionID);

**If the class does not implement Serializable interface, the calculated serial version UID will be 0.**

**Can we serialize static variables and why ?**

Since static variables are not part of Object, they belong to Class so static variables can not be serialized. We know that static field can not be serialized but in case of serial version id is an exception.

**If no serialVersionUID is declare, JVM will used its own algorithm to generate a default SerialVersionUID.**

**Serialization in case of Inheritance and Composition**

**Use case 1**: - Parent class is not serializable but child class is serializable.

In this case, we will be able to serialize the child class but parent class data members will not be persisted. Here it will not throw any Exception, Child class will work fine, but Parent object will null.

**Use Case 2** : - Parent class is Serializable but child class is not.

In this case Serialization happens, it means you will be able to persist the data members from child as well as parent. In this case you will be successfully serialize and de-serialize the data memebers of the super class by using the methods "**writeObject( ObjectOutputStream oout )**" and "**readObject( ObjectInputStream oin )**" in the child class.

**Use Case 3** : - In case of composition, Parent class is not Serializable but child class is serializable.

In case of hasA relationship, if the a class does not implement Serializable interface, that class object

not be persisted. It will throw exception "java.io.NotSerializableException".

**Use Case 4** : In case of Inheritance, Parent class is serializable and child class also, but you do not want the child class to be serialized because child class contains sensitive information.

In this particular situation, declare the following line in the child class.

**private final static ObjectStreamField[] serialPersistentFields = {};**

**Use Case 5 :- A class is final, does not implement Serializable, How to serialize the class**

Think about a situation where there is a final class and not serializable and you do not have access to the source code of this class. If you want to use this class in another class as a composition, you will be able to persist the data members of the Parent class using **defaultWriteObject()** and **defaultReadObject()** method. It is required when you are using some third party library.

**Note: Always remember that if the class is not serializable and if you want to persist the data members you have to declare the class as transient in another class where you want to use it. Otherwise it will throw NotSerializableException just like below.**

Exception in thread "main" java.io.NotSerializableException: com.ddlab.rnd.type1.Parent

Complete Example is given below.

public final class Parent {  
 private String parentName;  
 private String adrs;  
 get()/set() Methods

}

import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.io.Serializable;  
  
public class Child implements Serializable {  
 private static final long *serialVersionUID* = 5189322249391804225L;  
 protected String childName;  
 private transient Parent parent;  
  
 public Parent getParent() {  
 return parent;  
 }  
  
 public void setParent(Parent parent) {  
 this.parent = parent;  
 }  
  
 private void writeObject(ObjectOutputStream oos) throws Exception {  
 oos.defaultWriteObject();  
 oos.writeObject(parent.getParentName());  
 oos.writeObject(parent.getAdrs());  
 }  
  
 private void readObject(ObjectInputStream ois) throws Exception {  
 ois.defaultReadObject();  
 String s = (String) ois.readObject();  
 String adrs = (String) ois.readObject();  
 *//In this case you have to reform the object* parent = new Parent();  
 parent.setParentName(s);  
 parent.setAdrs(adrs);  
 }  
}

public class Test {  
 public static void main(String[] args) throws Exception {  
 Child child = new Child();  
 child.childName = "child";  
 Parent parent = new Parent();  
 parent.setParentName("Parent");  
 parent.setAdrs("Bangalore");  
 child.setParent(parent);  
   
 OutputStream out = new FileOutputStream("data/obj1.ser");  
 InputStream in = new FileInputStream("data/obj1.ser");  
 ObjectOutputStream oos = new ObjectOutputStream(out);  
 oos.writeObject(child);  
   
 oos.flush();  
 oos.close();  
 out.flush();  
 out.close();  
   
 ObjectInputStream oin = new ObjectInputStream(in);  
 Child child1 = (Child)oin.readObject();  
 System.*out*.println("Child Name : "+child1.childName);  
 System.*out*.println("Parent Name : "+child1.getParent().getParentName());  
 System.*out*.println("Parent Address : "+child1.getParent().getAdrs());  
   
 oin.close();  
 in.close();  
 }  
  
}

Note: **While storing/persisting the object graph, it is necessary to store the values in a particular order and also while retrieving the same order should be followed.**

private void writeObject( ObjectOutputStream oos ) throws Exception {

oos.defaultWriteObject();

oos.writeObject(parent.getParentName());//Sequence 1

oos.writeObject(parent.getAdrs());//Sequence 2

}

If you just change the sequence, you will get weird result ie ParentName will be bangalore and address

will be ParentName. You can also see that we have written defaultWriteObject() and defaultReadObject(), Why ? It means by using these method, it is an indication that , you do the normal serialization for the serializable class.

**What is the use of transient variable in Java?**

1. If a variable is declared as transient, that value will not be persisted during serialization.

2. If a class is final and not serializable and if you want to persist the data members of this class

in another class in case of composition, you have to declare the former as transient.

**Examples about to serialize and deserialize object**

public static void serialize(String filePath) throws Exception {  
 Person p = new Person();  
 p.setName("John");  
 p.setSal(1000);  
  
 FileOutputStream fout = new FileOutputStream(filePath);  
 ObjectOutputStream oos = new ObjectOutputStream(fout);  
 oos.writeObject(p); *// serialize* **long fileSize = fout.getChannel().size();** *//Serialization: 84 bytes, Externalization: 50 bytes* System.*out*.println("Now File Size: " + fileSize);  
 System.*out*.println("---- Serialization Complete ----");  
}  
  
public static void deSerialize(String filePath) throws Exception {  
 FileInputStream fin = new FileInputStream(filePath);  
 ObjectInputStream oin = new ObjectInputStream(fin);  
 Person p = (Person) oin.readObject();  
 System.*out*.println("Complete Object : " + p);  
}

**Serialize and Deserialize to and from byte[] array**

public static byte[] toByteArray() {  
 byte[] objectBytes = null;  
 Person p = new Person();  
 p.setName("John");  
 p.setSal(1000);  
  
 ByteArrayOutputStream bos = null;  
 ObjectOutputStream oos = null;  
 try {  
 bos = new ByteArrayOutputStream();  
 oos = new ObjectOutputStream(bos);  
 oos.writeObject(p);  
 **objectBytes = bos.toByteArray();  
 int len = objectBytes.length;** System.*out*.println("Total By Array Size: " + len); *// 50 bytes* } catch (IOException e) {  
 throw new RuntimeException(e);  
 } finally {  
 try {  
 oos.close();  
 bos.close();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 return objectBytes;  
}  
  
public static void fromByteArray(byte[] objBytes) {  
 ByteArrayInputStream bin = null;  
 ObjectInputStream oin = null;  
 try {  
 bin = new ByteArrayInputStream(objBytes);  
 oin = new ObjectInputStream(bin);  
 Person p = (Person) oin.readObject();  
 System.*out*.println("Value : "+p);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 } catch (ClassNotFoundException e) {  
 throw new RuntimeException(e);  
 } finally {  
 try {  
 oin.close();  
 bin.close();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

**Which are the objects that cannot be serialized**

* **Streams, Threads, Runtime** and those which are related to OS are never serialized.
* It means thread class object, **FileInputStream** and **FileOutputStream** can not be serialized.
* There are some GUI classes that cannot be serialized.

**If you do not want to serialize a class what will you do ?**

What if you create a class whose superclass is serializable but you do not want that new class to be serializable? You cannot unimplement an interface, so if your superclass does implement Serializable, your new class implements it, too (assuming both rules listed above are met). To stop the automatic serialization, **you can once again use the private methods to just throw the NotSerializableException. Here is how that would be done**:

**private void writeObject(ObjectOutputStream out) throws IOException {**

**throw new NotSerializableException("Not today!");**

**}**

**private void readObject(ObjectInputStream in) throws IOException {**

**throw new NotSerializableException("Not today!");**

**}**

**An alternative approach to use of transient variable and readObject() and writeObject() methods**

One common reason to override **readObject** and **writeObject** is to serialize the data for a superclass that is not Serializable itself. There are two ways to define what fields get streamed when an object is serialized. By default, every non-static and non-transient field is preserved. However, if your class defines an array of **ObjectStreamField** objects named **serialPersistentFields** (**that happens to be private, static, and final**), then you can explicitly declare the specific fields saved. The order you place fields in the array is the order in which they are written. For instance, in the following class, only the username and counter fields are serialized, not the password.

public class MyClass implements Serializable {

private String username;

private int counter;

private String password;

**private final static ObjectStreamField[]**

**serialPersistentFields = { new ObjectStreamField("username", String.class),**

**new ObjectStreamField("counter", int.class)**

**};**

}

**By default, no customization of readObject and writeObject is necessary when you provide a serialPersistentFields setting.**

**What about Singleton Class**

A singleton class can also be serialized.

public class Singleton implements Serializable {  
 private static final long *serialVersionUID* = 6147365679111551115L;  
 private Singleton() {  
 super();  
 }  
  
 **private static class Holder {  
 private static Singleton *INSTANCE* = new Singleton();  
 }  
  
 public static Singleton getInstance() {  
 return Holder.*INSTANCE*;  
 }**}

However, if you serialize and deserialize the above singleton object, you will be get multiple instances of the Singleton class. To resolve the above issue and to maintain the singleton design pattern, you have to use the method

"**public Object readResolve()**".

import java.io.Serializable;  
public class Singleton implements Serializable {  
 private static final long *serialVersionUID* = 6147365679111551115L;  
  
 private Singleton() {  
 super();  
 }  
 private static class Holder {  
 **private static Singleton *INSTANCE* = new Singleton();**  
 }  
  
 public static Singleton getInstance() {  
 return Holder.*INSTANCE*;  
 }  
  
 **private Object readResolve() {🡸*//This method is important* return Holder.*INSTANCE*;  
 }**}

**Note: enum can be serialized, but instance will be one**.

**What is the use of readResolve() and writeReplace() in Serialization ?**

**ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;**

For Serializable and Externalizable classes, the **readResolve** method allows a class **to replace/resolve**

**the object read from the stream before it is returned to the caller.** By implementing the readResolve

method, a class can directly control the types and instances of its own instances being deserialized.

**ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;**

For Serializable and Externalizable classes, the **writeReplace** method allows a class of an object to

nominate its own replacement in the stream before the object is written. By implementing the

writeReplace method, a class can directly control the types and instances of its own instances being serialized.

**readResolve is called after readObject** has returned (conversely writeReplace is called before writeObject and probably on a different object). The object the method returns replaces this object returned to the user of ObjectInputStream.readObject. readObject() is an existing method in ObjectInputStream class. **At the time of deserialization readObject() method internally checks whether the object that is being deserialized has readResolve() method implemented**. If readResolve() method exists then it will be invoked.

However both the method readResolve() and writeReplace() are used to change the behaviour of the object graph during serialization.

An example is given below.

import java.io.Serializable;  
  
public final class Employee implements Serializable {  
  
 private static final long *serialVersionUID* = 7127244578320585835L;  
 private String name;  
 private String pwd;  
  
 public Employee() {  
  
 }  
  
 public Employee(String name, String pwd) {  
 this.name = name;  
 this.pwd = pwd;  
 }  
  
 get()/set() Methods

}

import java.io.Serializable;  
  
public class Organization implements Serializable {  
  
 private static final long *serialVersionUID* = 4440396810435081170L;  
 private String name;  
 private Employee emp;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Employee getEmp() {  
 return emp;  
 }  
  
 public void setEmp(Employee emp) {  
 this.emp = emp;  
 }  
  
 private Object writeReplace() throws Exception {  
 Employee emp = new Employee(this.emp.getName(), "\*\*\*\*\*");  
 this.setEmp(emp);  
 return this;  
 }  
  
 *//OR* private Object readResolve() throws Exception {  
 Employee emp = new Employee(this.emp.getName(), "\*\*\*\*\*");  
 this.setEmp(emp);  
 return this;  
 }  
  
}

public class Test1 {  
 public static void main(String[] args) throws Exception {  
 Employee emp = new Employee("John", "ABCD");  
 Organization org = new Organization();  
 org.setName("Org-Name");  
 org.setEmp(emp);  
  
 OutputStream out = new FileOutputStream("data/s1.dat");  
 ObjectOutputStream oout = new ObjectOutputStream( out );  
 oout.writeObject(org);  
 out.flush();  
 oout.flush();  
 out.close();  
 oout.close();  
  
 InputStream in = new FileInputStream("data/s1.dat");  
 ObjectInputStream oin = new ObjectInputStream(in);  
 Organization or = (Organization)oin.readObject();  
 System.*out*.println("organization Name: "+or.getName());  
 System.*out*.println("Employee Name :::"+or.getEmp().getName());  
 System.*out*.println("Employee Password :::"+or.getEmp().getPwd());  
  
 }  
}

Here you can see that password is retrieved as “\*\*\*\*\*” during desrialization process.

Externalization in Java

**Why Externalization ?**

**Serialization Responsibility:**

In case of Serialization JVM takes full responsibility for serializing the class instance, in case of Externalizable, it’s the programmer takes care of the whole serialization and also deserialization process.

**Performance:**

The java.io.Serializable interface uses reflection and metadata which causes relatively slow performance. By comparison, the Externalizable interface gives you full control over the serialization process. During serialization the JVM will always first check if the class is Externalizable. If that's the case then it will use the read/writeExternal methods. Hence Externalization is bit faster.

**Custom Serialization Strategies:**

Externalization provides developers with the flexibility to implement custom serialization strategies. This could involve intricate logic for **encryption**, **compression**, or **specialized formatting** tailored to unique use cases**.**

**Externalized output is more compact**:

If you would compare the actual output, it would look something like this: The header of the object contains a flag that marks if the class is just Serializable or maybe also Externalizable.

**Security Considerations:**

Externalization excels in terms of security. Developers can implement custom security checks during serialization and deserialization, fortifying applications against potential threats like object injection and data tampering**.**

**Resource Management:**

Externalization facilitates efficient resource management. Through explicit control over serialization and deserialization processes, developers can manage resources such as file handles, network connections, and memory allocations more effectively, reducing the risk of resource leaks and performance bottlenecks**.**

**Concurrency Challenges:**

Serialization might encounter challenges in concurrent environments due to its automatic nature and potential conflicts**.**

**If you want to serialize only part of an object, then Externalization is the best option. You will have to serialize only required fields of an object**.

Unlike Serializable interface, **Externalizable interface is not a marker interface and it provides two methods - writeExternal and readExternal**. These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes.

Now when an Externalizable object is reconstructed, an instance is created first using the public no-arg constructor, then the readExternal method is called. Again if the object does not support Externalizable, then Serializable objects are restored by reading them from an ObjectInputStream.

An example is given below.

import java.io.\*;  
public class Car implements Externalizable {  
 String name;  
 int year;  
  
 */\*  
 \** ***mandatory public no-arg constructor, otherwise it will throw  
 \* java.io.InvalidClassException: com.ddlab.rnd.type1.Car; no valid constructor*** *\*/* public Car() {  
 super();  
 }  
  
 Car(String n, int y) {  
 name = n;  
 year = y;  
 }  
  
 */\*\*  
 \* Mandatory writeExernal method.  
 \*/* public void writeExternal(ObjectOutput out) throws IOException {  
 out.writeObject(name);  
 out.writeInt(year);  
 }  
  
 */\*\*  
 \* Mandatory readExternal method.  
 \*/* public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 name = (String) in.readObject();  
 year = in.readInt();  
 }  
  
 }

import java.io.\*;  
public class ExternExample {  
 public static void main(String args[]) {  
 *// create a Car object* Car car = new Car("Mitsubishi", 2009);  
 Car newCar = null;  
try { *//serialize the car*  
 FileOutputStream fo = new FileOutputStream("data/tmp");  
 ObjectOutputStream so = new ObjectOutputStream(fo);  
 so.writeObject(car);  
 so.flush();  
 } catch (Exception e) { System.*out*.println(e); }  
  
 *// de-serialize the Car* try {  
 FileInputStream fi = new FileInputStream("data/tmp");  
 ObjectInputStream si = new ObjectInputStream(fi);  
 newCar = (Car) si.readObject();  
 } catch (Exception e) { System.*out*.println(e); }

// *Print out the original and new car information* System.*out*.println("The original car is "+car);  
 System.*out*.println("The new car is "+newCar);  
 }  
}

**What will happen when an externalizable class extends a non externalizable super class?**

Then in this case, you need to persist the super class fields also in the sub class that implements Externalizable interface. Look at this example.

class Automobile {  
 String regNo;  
 String mileage;  
public Automobile() {} // *A public no-arg constructor*  
  
 Automobile(String rn, String m) {  
 regNo = rn;  
 mileage = m;  
 }  
}

import java.io.\*;  
public class Car extends Automobile implements Externalizable {  
  
 String name;  
 int year;  
public Car() { super(); } // *mandatory public no-arg constructor*  
 Car(String n, int y) {  
 name = n;  
 year = y;  
 }

*// Mandatory writeExernal method.* public void writeExternal(ObjectOutput out) throws IOException {  
 */\*  
 \* Since the superclass does not implement the Serializable interface  
 \* we explicitly do the saving.  
 \*/* out.writeObject(regNo);  
 out.writeObject(mileage);  
  
 *//Now the subclass fields* out.writeObject(name);  
 out.writeInt(year);  
 }

*// Mandatory readExternal method.* public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 */\*  
 \* Since the superclass does not implement the Serializable interface  
 \* we explicitly do the restoring  
 \*/* regNo = (String) in.readObject();  
 mileage = (String) in.readObject();  
  
 *//Now the subclass fields* name = (String) in.readObject();  
 year = in.readInt();  
 }  
 }

Here the Automobile class does not implement Externalizable interface. So to persist the fields in the automobile class the writeExternal and readExternal methods of Car class are modified to save/restore the super class fields first and then the sub class fields.

**What if the super class implements the Externalizable interface?**

**Well, in this case the super class will also have the readExternal and writeExternal methods as in Car class and will persist the respective fields in these methods.**

*// The superclass implements externalizable*class Automobile implements Externalizable {  
 String regNo;  
 String mileage;  
public Automobile() {} // *A public no-arg constructor*  
  
 Automobile(String rn, String m) {  
 regNo = rn;  
 mileage = m;  
 }  
  
 public void writeExternal(ObjectOutput out) throws IOException {  
 out.writeObject(regNo);  
 out.writeObject(mileage);  
 }  
  
 public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 regNo = (String) in.readObject();  
 mileage = (String) in.readObject();  
 }  
}

import java.io.\*;  
public class Car extends Automobile implements Externalizable {  
 String name;  
 int year;  
  
public Car() { // *mandatory public no-arg constructor*  
 super();  
 }  
  
 Car(String n, int y) {  
 name = n;  
 year = y;  
 }

*// Mandatory writeExernal method.* public void writeExternal(ObjectOutput out) throws IOException {  
 *// first we call the writeExternal of the superclass as to write  
 // all the superclass data fields* super.writeExternal(out);  
  
 *//Now the subclass fields* out.writeObject(name);  
 out.writeInt(year);  
 }

*// Mandatory readExternal method.* public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 *// first call the superclass external method* super.readExternal(in);  
  
 *//Now the subclass fields* name = (String) in.readObject();  
 year = in.readInt();  
 }  
  
 }

In this example since the Automobile class stores and restores its fields in its own writeExternal and readExternal methods, you dont need to save/restore the superclass fields in sub class but if you observe closely the writeExternal and readExternal methods of Car class closely, you will find that you still need to first call the super.xxxx() methods that confirms the statement the externalizable object must also coordinate with its supertype to save and restore its state.

Now if you serialize the same by extending Externalizable interface, the size will be reduced drastically and the information saved in the persistant store is also reduced a lot.

**Externalization on the other hand isn't very flexible and requires you to rewrite your marshalling and demarshalling code whenever you change your class definitions.**

As you know a default public no-arg constructor will be called when serializing the objects that implements Externalizable interface. Hence, **Externalizable interface can't be implemented by Inner Classes in Java as all the constructors of an inner class in Java will always accept the instance of the enclosing class as a prepended parameter and therefore you can't have a no-arg constructor for an inner class. Inner classes can achieve object serialization by only implementing Serializable interface.**

If you are subclassing your externalizable class, you have to invoke your superclass’s implementation. So this causes overhead while you subclass your externalizable class.

Complete Example on Externalizable

**package** com.ddlab.rnd.core;

**import** java.io.Externalizable;

**import** java.io.IOException;

**import** java.io.ObjectInput;

**import** java.io.ObjectOutput;

**public** **class** Person1 **implements** Externalizable {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**private** **int** age;

**private** **float** salary;

**get()/set() Methods**

@Override

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeInt(**this**.id);

out.writeObject(**this**.firstName);

out.writeObject(**this**.lastName);

out.writeInt(**this**.age);

out.writeFloat(**this**.salary);

}

@Override

**public** **void** readExternal(ObjectInput in) **throws** IOException,

ClassNotFoundException {

**this**.id = in.readInt();

**this**.firstName = (String) in.readObject();

**this**.lastName = (String) in.readObject();

**this**.age = in.readInt();

**this**.salary = in.readFloat();

}

}

import java.io.File;

import java.io.FileInputStream;

import java.io.ObjectInputStream;

public class Test3 {

public static void main(String[] args) throws Exception {

Person1 person = new Person1();

person.setId(11);

person.setFirstName("Deb, Kumar");

person.setLastName("Mishra");

person.setAge(23);

person.setSalary(1234f);

File file = new File("data/t2.ser");

// ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream(file));

// out.writeObject(person);

// out.flush();

// out.close();

ObjectInputStream oin = new ObjectInputStream( new FileInputStream(file));

Person1 p = (Person1) oin.readObject();

System.out.println(p);

}

}

**To serialize an array or a collection all the members of it must be serializable.** **True**

Will you be able to persist the value of a transient field? **YES**

Example is given below.

import java.io.Externalizable;  
import java.io.IOException;  
import java.io.ObjectInput;  
import java.io.ObjectOutput;  
  
public class Employee implements Externalizable {  
  
 private static final long *serialVersionUID* = 3638914768807978156L;  
 private String name;  
 private transient String desgn;  
  
 public Employee() {}  
  
 get()/set() Methods  
  
 public void writeExternal(ObjectOutput out) throws IOException {  
 out.writeObject(name);  
 out.writeObject(desgn);  
 }  
  
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 name = (String) in.readObject();  
 desgn = (String) in.readObject();  
 }  
}

public class TestEmp {  
 public static void main(String[] args) throws Exception {  
 Employee emp = new Employee();  
 emp.setName("John");  
 emp.setDesgn("Engineer");  
 OutputStream out = new FileOutputStream("data/hack.ser");  
 ObjectOutputStream oout = new ObjectOutputStream(out);  
 oout.writeObject(emp);  
 oout.flush();  
 oout.close();  
 out.flush();  
 out.close();  
 System.*out*.println("Serialized successfully ....");  
  
 InputStream in = new FileInputStream("data/hack.ser");  
 ObjectInputStream oin = new ObjectInputStream(in);  
 Object obj = oin.readObject();  
 Employee ee1 = (Employee)obj;  
 System.*out*.println("Emp Name :::"+ee1.getName());  
 System.*out*.println("Emp Desgn :::"+ee1.getDesgn());  
  
 }  
}

**Few General Use Cases**

* 1. If there is a generated serial version Id, you add another field, you will be able to deserialize the object. But the new field will be null.
  2. If there is generated serial version id, you serialize the object. Then you delete a field, what will happen? After deleting the field, you will be able to deserialize the object, no exception will be thrown.
  3. There is no serial version id, you serialize the object. You add another field, what will happen? It will throw InvalidClassException.
  4. There is no serial version Id, you serialized the object. You delete a field, What will happen? It will throw InvalidClassException.

**Usage of ObjectStreamField[] *serialPersistentFields***

**simply declare serialPersistenFields with {} only not to persist any field.**

**import** java.io.ObjectStreamField;

**import** java.io.Serializable;

**public** **class** Girl **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 9116375897818728549L;

**private** String name;

**private** **int** age;

**private** **final** **static** ObjectStreamField[] ***serialPersistentFields*** = {}; // Nothing will be persisted

**get()/set() method**

}

In the above case, the object will be serialized but the field values will not be persisted.

If you want to persist few fields and few fields will not be persisted.

**import** java.io.ObjectStreamField;

**import** java.io.Serializable;

**public** **class** Girl **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 9116375897818728549L;

**private** String name;

**private** **int** age;

**private** String pwd;

// Only name and age will be persisted, but password will not be stored

**private** **final** **static** ObjectStreamField[] ***serialPersistentFields*** = {

**new** ObjectStreamField("name", String.**class**),

**new** ObjectStreamField("age", **int**.**class**)

};

get()/set() methods

}

In the above case, password field will not be persisted whereas name and age will be persisted.

**What will happen if you declare writeObject() and readObject(), ie. If the code is written like this.**

**private** **void** writeObject(ObjectOutputStream stream) **throws** IOException {

stream.defaultWriteObject();

stream.writeObject(pwd);

}

**private** **void** readObject(ObjectInputStream stream) **throws** IOException, ClassNotFoundException {

stream.defaultReadObject();

pwd = (String) stream.readObject();

}

In this case serialPersistenFields holds good, it means the above two object has no impact,ie. Pwd field will be null;

Read Later

**Some Situations**

**Situation-1**: A user creates a java bean class by implementing the Serializable interface and generates the serial

version UID. A bad developer serialized the object in the file system and then he modifies the serial version UID in the java bean class, what will happen. In this case, at the time of de-serialization, it will throw "**InvalidClassException**". How will know what is the problem ? How will you resolve this issue ?

If you want to de-serialize it, you will get the following exception. Exception in thread "main" **java.io.InvalidClassException**: Emp; local class incompatible: stream classdesc serialVersionUID = -3612997364053394311, local class serialVersionUID = -4612997364053394311

**To resolve the issue, go to the Emp bean class and comment out the serial version uid and re-run the**

**program, it will work.**

**Situation-2**. Now there is another situation. A developer manually provides the serial version uid without the use

of serialver tool. He serializes the object and then again modifies or tempers the serial version UID.

What will happen and how will you resolve it ?

On running this program, you will get the following exception.

Exception in thread "main" java.io.InvalidClassException: Emp; local class incompatible:

stream classdesc serialVersionUID = 1234567890, local class serialVersionUID = 91234567890

You can not solve this problem by commenting out the serial version UID. **To resolve copy the actual**

**serial version UID from the exception stack trace and paste it in the java bean class's serial version UID**.

**Situation-3**. A class does not contain serial version uid. Class is serialized. Before de-serialization, developer adds one more field called "int age". At the time of de-serialization, it will throw invalid class exception .

Reason : Whenever you add any field, the serial version uid will be changed. If you do not provide

default serial version uid, java will calculate serial version uid at the time of serialization and

de-serialization. If you already have a serialized object and if you add some fields in the class and if you want to avoid "InvalidClassException", generate the serial version uid and put it in the class. So it ovious that if a class does not contain a serial version uid, at the time of serialization java will calculate

the serial version uid of that class and store it in the serialized object. Similrly at the time of

de-serialization, java will check whether the class contains the default serial version uid or not. If the class does not contain, then java will calculate the serial version uid and at the time of deserialization it will compare, if both compare, then it fine otherwise it will throw "InvalidClassException"

complaining about the incompatibilities of the uids of the classes. If a class contains a serial version uid without proper calculation for example "111l" and that object is serialized, java will not calculate the serial version uid, java will take that serial version uid and store it as a part of the object graph. Before serialization if you addd some more fields in the java class, it will not throw exception at the time of serialization.

**Use of Reset()- public void reset() throws IOException**

**Reset will disregard the state of any objects already written to the stream. The state is reset to be the same as a new ObjectOutputStream.** The current point in the stream is marked as reset so the corresponding ObjectInputStream will be reset at the same point. Objects previously written to the stream will not be refered to as already being in the stream. They will be written to the stream again.

/\* prevent using back references \*/

**output.reset();**

**output.writeObject(...);**

Call reset before writing the same object to ensure its updated state is serialized. Otherwise, it will merely use a back reference to the previously written object with its out-dated state. Reset will disregard the state of any objects already written to the stream. The state is reset to be the same as a new ObjectOutputStream. The current point in the stream is marked as reset so the corresponding ObjectInputStream will be reset at the same point. Objects previously written to the stream will not be refered to as already being in the stream. They will be written to the stream again.

**writeUnshared() and readUnshared()**

**Writes an "unshared" object to the ObjectOutputStream. This method is identical to writeObject, except that it always writes the given object as a new, unique object in the stream**.

**Specifically: An object written via writeUnshared is always serialized in the same manner as a newly appearing object** (an object that has not been written to the stream yet), regardless of whether or not the object has been written previously.

If writeObject is used to write an object that has been previously written with writeUnshared, the previous writeUnshared operation is treated as if it were a write of a separate object.While writing an object via writeUnshared does not in itself guarantee a unique reference to the object when it is deserialized, it allows a single object to be defined multiple times in a stream, so that multiple calls to readUnshared by the receiver will not conflict. Note that the rules described above only apply to the base-level object written with writeUnshared, and not to any transitively referenced sub-objects in the object graph to be serialized.

**output.writeUnshared(...);**

Note it's good practice to couple this with ObjectInputStream.readUnshared.

Reads an "unshared" object from the ObjectInputStream. This method is identical to readObject, except that it prevents subsequent calls to readObject and readUnshared from returning additional references to the deserialized instance obtained via this call. Specifically: If readUnshared is called to deserialize a back-reference (the stream representation of an object which has been written previously to the stream), an ObjectStreamException will be thrown

If readUnshared returns successfully, then any subsequent attempts to deserialize back-references to the stream handle deserialized by readUnshared will cause an ObjectStreamException to be thrown.

Deserializing an object via readUnshared invalidates the stream handle associated with the returned object. Note that this in itself does not always guarantee that the reference returned by readUnshared is unique; the deserialized object may define a readResolve method which returns an object visible to other parties, or readUnshared may return a Class object or enum constant obtainable elsewhere in the stream or through external means. If the deserialized object defines a readResolve method and the invocation of that method returns an array, then readUnshared returns a shallow clone of that array; this guarantees that the returned array object is unique and cannot be obtained a second time from an invocation of readObject or readUnshared on the ObjectInputStream, even if the underlying data stream has been manipulated. **obj = input.readUnshared();**