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

**What is Serialization, Why Serialization / Benefits ?**

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 and can be stored in file or Database for later use. This allows the application to store the object for long-term purposes, such as enterprise applications and data-intensive systems.
3. **Deep Copy**: Simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
4. **Caching**: 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.
6. Interoperability: As we read above, Serialization is JVM-independent, which means it is a platform-independent way of transmitting objects between different systems. This is because serialized objects are represented as a stream of bytes, which can be transmitted over a network and then deserialized on the receiving end to create an identical object.
7. Security: Serialization also gives our program more security as it can be used to implement secure communication between different networks.

**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,**

1. **JVM first evaluates the serialVersionUID of the serialized class**
2. **Compares the serialVersionUID value with the one of the objects**.
3. **If the serialVersionUID values match then the object is said to be compatible with the class and hence it is de-serialized.**
4. **If not InvalidClassException exception is thrown.**

**How serialization happens?**

1. **JVM first checks for the Externalizable interface and**
2. **If object supports Externalizable interface, then serializes the object using writeExternal method.**
3. **If the object does not support Externalizable but implement Serializable, then the object is saved using ObjectOutputStream**.

**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? Y**ou 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. 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.

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

}

public class Organization implements Serializable {  
  
 private static final long *serialVersionUID* = 4440396810435081170L;  
 private String name;  
 private Employee emp;  
  
 get()/set() methods

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.