## Java Serialization

#### Chapter 01 - Introduction

Suppose I want to save the state of one or more objects. If Java didn’t have serialization, I would have to use one of the IO classes to write out the state of the instance variables of all the objects I wanted to save - say to a csv, xml, json or just plain text.

Then I would need to reconstruct the objects that had been saved in the same order of instance fields as it was saved. This is error-prone as we are doing lots of stuff manually here.

Also, if the objects are huge - containing reference to other objects (Object Graphs) and many instance fields, then the manual processing of serialization and deserialization would be very complex and error-prone.

Java’s object serialization allows us to take any object that implements the Serializable  
interface and turn it into a **sequence of bytes** that can later be fully restored to regenerate the original object. This is even true across a network, which means that the serialization  
mechanism automatically compensates for differences in operating systems. That is, I can  
create an object on a Windows machine, serialize it, and send it across the network to a Unix  
machine, where it will be correctly reconstructed. I don’t have to worry about the data  
representations on the different machines, the byte ordering, or any other details.



The magic of basic serialization happens with just 2 methods:

* one to serialize objects and write them to a stream
* second to read the stream and deserialize objects

**ObjectOutputStream.writeObject()** // serialize and write

**ObjectInputStream.readObject()** // read and deserialize

#### Chapter 02 - Serialization with memory buffer

The first example is using memory as buffer to where the serialized streams of bytes will be written to and then retrieved from using desrialization.

**Java Object => Serialize to byte array => Memory**

**Memory => Deserialize from byte array => Java Object**

**ByteArrayOutputStream** class is used to serialize to byte array.

This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using toByteArray() and toString().

**ByteArrayInputStream** class is used to deserialize from byte array.

This class contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the read method.

Drawback using memory buffer is that once the JVM shuts down - the serialized data in memory is erased and can not be used after application restart.

#### Chapter 03 - Serialization with file

It would be incredibly useful if an object could exist and hold its information even while the program wasn’t running. Then, the next time we started the program, the object would be there and it would have the same information or state it had the previous time the program was running.

Object serialization allows us to implement persistence. Persistence means that an object’s lifetime is not determined by whether a program is executing; the object lives in between invocations of the program. By taking a serializable object and writing it to disk (via file), then restoring that object when the program is reinvoked, we’re able to produce the effect of persistence.

**Java Object => Stream of bytes => File**

**File => Deserialize from stream of bytes => Java Object**

**FileOutputStream** class is used to write the serialized stream of bytes to file on the disk.

This class is an output stream for writing data to a File and is meant for writing streams of raw bytes such as image data.

**FileInputStream** class is used to read the stream of bytes from the file.

This class obtains input bytes from a file in a file system and is meant for reading streams of raw bytes such as image data.

#### Chapter 04 - Serialization versioning

Suppose we have a class and we have serialized its object to a file on the disk, and due to some new requirements, we added/removed one field from our class. Or just added a new utility method to it. Now, if we try to deserialize the already serialized object, we will get **InvalidClassException**.

When we serialize a class, each class has a unique identification number associated with it.  
Its also called stream unique identifiers, more commonly known as **serial versionUIDs**.  
  
If we do not specify this number by declaring a static final long field named **serialVersionUID**,   
the system automatically generates it at runtime by applying a cryptographic hash function (SHA-1) to the structure of the class.  
  
This value is affected by the names of the class, the interfaces it implements, and most of its members, including synthetic members generated by the compiler.  
  
If we change any of these things, for example, by adding a convenience method, the generated serial version UID **changes**.

If we fail to declare a serial version UID, compatibility will be broken, resulting in an **InvalidClassException** at runtime.  
  
The basic idea is a class could have been serialized with an older version of the class and deserialized with a newer version of the class.  
  
The **serialVersionUID** helps inform the JVM that the stored data may not match the new class definition.   
  
Thus, if an older version of the class is encountered during deserialization, a **java.io.InvalidClassException** is thrown.  
  
***Conclusion***: It's a good practice to declare a static **serialVersionUID** variable in every class that implements Serializable.

In **IntelliJ IDE**, we can generate **serialVersionUID** using the following action:

Ctrl + Shift + A (will launch *Actions* tab menu) => type ‘**Serializable class without**’ and select the item to toggle it on/off.

OR

File -> Settings -> Editor -> Inspections -> Java -> Serialization issues:

Find ‘**Serializable class without** **serialVersionUID**’ and check it.

Now, for each class implementing **Serializable** interface will show warning if no **serialVersionUID** is declared.

Press “Alt + Enter” and Click on “Create constant field serialVersionUID in …” and **serialVersionUID** will be generated and declared in the class.

OR, we can just declare any random number like 1L or 42L and declare in the class.

For example:

private static final long serialVersionUID = 1L; // 1L or 2L or 3L - any number can be chosen

#### Chapter 05 - Serialization with array field members

Suppose we have a class which has array of primitives or array of Objects as its field members. In this scenario, we need to ensure that every element in the array is Serializable, otherwise the serialization will fail.

#### Chapter 06 - Serialization with Java Objects as array

If we have an array of a serializable class objects, the whole array can be serialized in one go by just calling **writeObject()** method from **ObjectOutputStream** class. We need to ensure that every element in the array is Serializable, otherwise the serialization will fail.

Similarly, while deserialization, the whole array object can be read by calling **readObject()** method from **ObjectInputStream** class.

#### Chapter 07 - Serialization with Collection field members

Suppose we have a class which has collection of objects say **List** as its field members. In this scenario, we need to ensure that every element in the collection is Serializable, otherwise the serialization will fail.

Please note that while the collection interfaces like **List**, **Set**, etc. are **NOT** serializable, the concrete collection classes like **ArrayList**, **HashSet** etc. **ARE** serializable.

#### Chapter 08 - Serialization with Java Objects as Collection

If we have a collection (List, Set, etc) of a serializable class objects, the whole collection can be serialized in one go by just calling **writeObject()** method from **ObjectOutputStream** class. We need to ensure that every element in the collection is Serializable, otherwise the serialization will fail.

Similarly, while deserialization, the whole collection object can be read by calling **readObject()** method from **ObjectInputStream** class.

#### Chapter 09 - Serialization with Enum Constants

As per Java specifications, Enum constants are serialized differently than ordinary serializable objects.

The serialized form of an enum constant consists solely of its **name**; field values of the constant are not present in the form.

To **serialize** an enum constant, **ObjectOutputStream**writes the value returned by the enum constant’s **name()**method.

For reference,

* **name()** is an instance method which returns the name of the instance.
* **valueOf()** is a static method taking a String and returning the enum instance with that name.
* **values()** is a static method returning all the enum instances.

To **deserialize** an enum constant, **ObjectInputStream**reads the constant name from the stream; the deserialized constant is then obtained by calling the **valueOf()** method, passing the constant’s enum type along with the received constant name as arguments.

Few more points to take note of:

* enum types have a fixed **serialVersionUID**of **0L** and cannot be changed
* process by which enum constants are serialized **cannot** be customized: any class-specific **writeObject()**, **readObject()**, **readObjectNoData()**, **writeReplace()**, and **readResolve()**methods defined by enum types are ignored during serialization and deserialization

#### Chapter 10 - Serialization with static fields

Static fields are NEVER saved as part of the object’s state.

We should think of static variables purely as CLASS variables. They have nothing to do with individual instances. And serialization applies only to OBJECTS.

Suppose we have 3 instances of a Serializable class => all of which were serialized at **different** times, and thus all of which were saved when the value of a static variable in the class was different.

Now, if static variables were serializable, which of the 3 instances would win ? OR, Which instance’s static value would be used to replace the one currently in the one and only class that’s currently loaded ? => this is a problem… that’s why, static fields are never part of serialization / deserialization process.

#### Chapter 11 - Object graphs

Java's default serialization process is fully **recursive**, so whenever we try to serialize one object, the serialization process try to serialize all the fields (primitive and reference) with our class (except **static**and **transient**fields).

The reference to objects fields are serialized and Java serialization takes care of saving that object’s entire “object graph”. That means a deep copy of everything the saved objects needs to be stored.

Just remember to implement the Serialization interface for all the objects in the “object graph” - otherwise we will get **NotSerializableException**.

#### Chapter 12 - Using transient keyword

Suppose one of the objects in the “object graph” is not accessible (meaning we can’t modify it due to permission issues) and not marked as Serializable. In that case, are we blocked to Serialize our main class which has reference to that read-only object?

One option is to subclass that read-only class, but it may be marked as **final**.

OR, we don’t know if the read-only class object has other similar objects in its own “object graph”.

There are other private members fields (primitive and reference) of a class which contains sensitive data like passwords, or it doesn’t make sense to serialize it - like the state of an in-memory Thread.

That’s where the **transient** modifier comes in. By marking the members fields (primitive and reference) as transient, these fields are NOT serialized just like **static** fields.

What happens to data marked **transient** on **deserialization**? It reverts to  
its default Java values, such as **0.0D** for **double**, **false** for **boolean** or **null** for an **object**.

#### Chapter 13 - Using writeObject() and readObject()

Just based on the above example, if one of the reference object is not accessible and we had to mark it as **transient**. While deserialization, the **transient** reference object will come as default value of null. What can we do to somehow make sure that it matches the same object state when it was saved ?

Java serialization has a special mechanism just for this - a set of private methods we can implement such that it will be invoked automatically during serialization and deserialization.

These 2 special methods we define must have signatures that look EXACTLY like this:

1. For serialization

private void writeObject(ObjectOutputStream stream) {

// code for saving the object

}

1. For deserialization:

private void readObject(ObjectlnputStream stream) {

// code for restoring the same object as it was saved although if its transient

}

By implementing these 2 methods, we can customize or control the default Java serialization process.

Also, we can call the methods like **ObjectOutputStream.defaultWriteObject()** and ObjectInputStream.defaultReadObject() to invoke the default Java serialization process inside **writeObject()** and **readObject()** methods respectively.

#### Chapter 14 - Using ObjectStreamField

In case of **transient** keyword, we chose which member fields (primitive and reference) of a class SHOULD NOT participate in serialization and deserialization process. This is also called as *blacklisting*.

Using **ObjectStreamField** class and **ObjectOutputStream.putFields** and **ObjectInputStream.readFields,** we can choose which member fields (primitive and reference) of a class SHOULD participate in serialization and deserialization process. This is also called as *whitelisting*.

Ideally *whitelisting* should be preferred over *blacklisting* as we have full control on which fields to serialize or not.

#### Chapter 15 - Protecting sensitive information

When developing a class that provides controlled access to resources, care must be taken to protect sensitive information and functions. During deserialization, the private state of the object is restored. To avoid compromising a class, the sensitive state of an object must not be restored from the stream, or it must be re-verified by the class.

The easiest technique is to mark fields that contain sensitive data as private **transient**. Transient fields are not persistent and will not be saved by any persistence mechanism. Marking the field will prevent the state from appearing in the stream and from being restored during deserialization. Since writing and reading (of private fields) cannot be superseded outside of the class, the **transient** fields of the class are safe.

However, if we really want to serialize secured or confidential fields, we should use encryption and decryption of the field.

#### Chapter 16 - How Inheritance Affects Serialization

If a superclass is **Serializable**, then all subclasses are automatically Serializable without having to explicitly mark the subclass as **Serializable**.

If a class does NOT explicitly extend any other class and does NOT implement Serializable, then we can confirm that the class is NOT serializable as class **Object** does NOT implement **Serializable**.

Now suppose a subclass implements **Serializable** but the super class does NOT. How is this going to affect serialization?

When an object is constructed using **new** (as opposed to being **deserialized**), following things happen in this sequence:

1. All **instance variables** are assigned **default** values - like int as 0, double as 0D, boolean as false, String as null, etc.
2. The **constructor** is invoked, which immediately invokes the superclass constructor OR another overloaded constructor, until one of the overloaded constructors invokes the superclass constructor.
3. All **superclass** constructors complete.
4. Instance variables that are initialized as part of their declaration are assigned their initial value overriding the default values they’re given prior to the superclass constructors completing.
5. The constructor completes.

BUT, these things do NOT happen when an object is deserialized.

If the constructor were invoked, and/or variables were assigned the values given in their declarations, the object we are trying to restore would revert to its original state, rather than coming back reflecting the changes in its state that happened sometime after it was created.

In other words, we want only the values saved as part of the serialized state of the object to be reassigned.

If the superclass is not Serializable, the instance variables in the subclass (which implements Serializable) will be serialized and deserialized correctly, but the **inherited variables** from the non-serializable superclass will come back with their default/initially assigned values rather than the values they had at the time of serialization.

Thus, if a subclass implements **Serializable** but the super class does NOT, then any instance variables INHERITED from that superclass will be reset to the values they were given during the original construction of the object. This is because the non-serializable class constructor WILL run.

In fact, every constructor ABOVE the first non-serializable class constructor will also run in the inheritance tree.