## 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.