Deserialize
of
Java Serialized object

If you want to save an (active) Java instance, you must first serialize the object instance before you can save it to a file. This file is now a serialized Java object file. And if you want to "**revive**" this serialized object, you must first read it from the file and then **deserialize** it. It's a reverse process and is called deserialization. For this Object-saving purpose, JAVA provides you with a tool with 2 APIs:

- **ObjectOutputStream** for serializing an object instance
- **ObjectInputStream** for the reverse process

The serialization process is a straightforward one. An object you want to store just needs to be an “implementation” of the Serializable interface. Example:

```
import java.util.ArrayList;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
public class MyList implements java.io.Serializable {
  public static final long serialVersionUID = 1L;
  ArrayList<String> list;
  public MyList(String[] mylist) {
    list = new ArrayList<>(mylist.length);
    for (String s:mylist) list.add(s);
  }
  public void print() {
    System.out.println(list);
  }
  //
  public static void main(String... a) throws Exception {
    MyList ml = new MyList(a.length == 0? new String[] { "Hello", "World" }:a);
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("serobj/MyList.txt"));
    oos.writeObject(ml);
    oos.flush();
    oos.close();
  }
}
```

And the file MyList.txt looks as following (in plain text)

- Invocation with 3 parameters: **java MyList joe T schwarz**

```
¬í #sr #MyList       ## #L #listt #Ljava/util/ArrayList;xpsr #java.util.ArrayListxÒ#™Ça#□#I #sizexp   #w#   #t #joet #Tt #schwarzx
```

- Invocation without parameter: **java MyList**

```
¬í #sr #MyList       ## #L #listt #Ljava/util/ArrayList;xpsr #java.util.ArrayListxÒ#™Ça#□#I #sizexp   #w#   #t #Hellot #Worldx
```

If you dump these files as hex-content, you will see: (MyList.txt with 3 parameters)

```
0        4        8        C        10
ACED0005 73720006 4D794C69 73740000 00000000 | .... sr.. MyLi st.. .... | 0
00010200 014C0004 6C697374 7400154C 6A617661 | .... .L.. list t..L java | 1
2F757469 6C2F4172 7261794C 6973743B 78707372 | /uti l/Ar rayL ist; xpsr | 2
00136A61 76612E75 74696C2E 41727261 794C6973 | ..ja va.u til. Arra yLis | 3
747881D2 1D99C761 9D030001 49000473 697A6578 | tx.. ...a .... I..s izex | 4
70000000 03770400 00000374 00036A6F 65740001 | p... .w.. ...t ..jo et.. | 5
54740007 73636877 61727A78                    | Tt.. schw arzx           | 6
```

It looks quite confusing. Both processes **ALWAYS** require that the binary bytecode class of the serialized object exists and is known via CLASSPATH. Otherwise a "**ClassNotFoundException**" is thrown. Ugly, isn't it? Now, let's start the reverse process.

```
import java.util.ArrayList;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
public class DeMyList {
  public static void main(String... a) throws Exception {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(a[0]));
    MyList ml = (MyList) ois.readObject( );
    ois.close();
    ml.print();
  }
}
```

If you rename the class MyList.class to any name and run the MyList and this little app, you will get this famous **ClassNotFoundException**. Example:

```
C:\JoeApp\ODB\examples\Test\examples>java MyList joe T schwarz
Error: Could not find or load main class MyList
Caused by: java.lang.ClassNotFoundException: MyList

C:\JoeApp\ODB\examples\Test\examples>java DeMyList serobj/mylist.txt
Exception in thread "main" java.lang.ClassNotFoundException: MyList
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown Source)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Unknown
Source)
```

Let's rename it back to MyList.class and rerun DeMyList.

```
C:\JoeApp\ODB\examples\Test\examples>java DeMyList serobj/mylist.txt
[joe, T, schwarz]
```

What I am showing you here is a trivial work with Java Serializable Object. You might be wondering what all this effort is for? Well, it is probably desirable when you need to send a serialized object instance over a network and there are no binary bytecodes available for that object on the other side. In particular, when there are multiple serialized files (or serialized data stored in the database) without their bytecode class and a request is for a specific serialized file or data, a big problem arises: which file (or data)? The only option is to send ALL files to the requester. And this will overload the network. Trying to do this with **URLClassLoader** or Java **Reflection** ends also with ClassNotFoundException. The only option is to send a bytecode class along with the serialized object instance and make it known via **CLASSPATH**. And that is virtually impossible. Since the serialized object contains valid data (except codes for the methods), the question arises: is there a way to work with the serialized objects (or files)? If you search the internet, you won't find any. The only package I found is the **Github jdeserialize**. This package reads a serialized object file and dumps the layout of the serialized data structure (reverse engineering?) with all the data initialized. Since there is no access to every field, working with this outputted object requires additional hard work.

No chance? Yes, you can! You can reconstruct a serialized object without having its bytecode class. The JAVA serialization protocol is available (HERE) and all you need is to implement an API based on this protocol. And I'll show you below how I did it using an API: SerObjectView in 100% JAVA, without depending on any external package.

SerObjectView API.
Constructors:

- public SerObjectView( )
- public SerObjectView(String fileName) throws Exception
- public SerObjectView(byte[] bb) throws Exception

Methods:

- public void view(byte[] bb) throws Exception
- public void setCharset(Charset charset)
- public long getSerialID()
- public int getSize() -get the number of fields (i.e. variables)
- public String getClassName()
- public ArrayList<String> getFieldNames()
- public String getFieldType(String fieldName) -e.g. double, int, etc.
- public Object getFieldValue(String fName) -e.g. 123.45, 100, etc. Must be cast.

Note:
This API is used to reconstruct a serialized object (via a file or a byte array) without having to know/have its bytecode class. Due to the sheer number of JAVA APIs, this SerObjecrView is limited to:

- Primitives (int, long, short, double, etc.),
- API: String, BigDecimal, BigInteger, Integer, Long, Short, Double, Float, ArrayList (or List)
- Custom serialized object (CSO). No array
- Array with primitives and supported APIs up to 3 dimensions
- Array with CSO or more than three (3) dimensions are also not supported.

When working with Java Object DB, the objects are usually simple POJOs (Plain Old Java Object) and with SerObjectView you can access the data of the variables (fields) and process them outside of the object.

Example: ViewSerObject shows you how to work with SerObjectView

```
import java.util.*;
import java.io.*;
/**
@author Joe T. Schwarz
*/
public class ViewSerObject {
  public static void main(String... a) throws Exception {
    if (a.length == 0) {
      System.out.println("Usage: java ViewSerObject serializedObjectFile");
      System.exit(0);
    }
    ...
    new ViewSerObject(obj);
  }
  /**
  Constructor
  @param sfName   String, serialized file name
  */
```

```
  public ViewSerObject(String sfName) throws Exception {
    show(new SerObjectView(sfName));
  }
  ...
  private static OutputStream output = System.out;
  //
  @SuppressWarnings("unchecked")
  private void show(SerObjectView ov) throws Exception {
    output.write(String.format("Class Name: %s\nSerID: %d\nNumber of Fields: %s\n",
                 ov.getClassName(),ov.getSerialID(), ov.getSize()).getBytes());
    ArrayList<String> fNames = ov.getFieldNames();
    for (String f:fNames) {
      String type = ov.getFieldType(f);
      int a = type.indexOf("][");
      if ("Object".equals(type)) {
        output.write("Embedded ".getBytes());
        show((SerObjectView) ov.getFieldValue(f));
      } else
        output.write(String.format("Field: %s, type: %s, value/Reference: %s\n",
                                   f, type, ov.getFieldValue(f)).getBytes());
      if (type.indexOf("[") > 0) {
        Object obj = ov.getFieldValue(f);
        if (type.indexOf("int") >= 0) {
          if (a > 0) {
            a = type.indexOf("][", a+2);
            if (a < 0) {
              int[][] aa = (int[][]) obj;
              ...
            } else {
              int[][][] aa = (int[][][]) obj;
              ...
            }
          } else {
            int[] aa = (int[]) obj;
            for (int i = 0; i < aa.length; ++i)
              output.write(String.format("    int[%d]: %d\n", i, aa[i]).getBytes());
          }
        } else if (type.indexOf("long") >= 0) {
          ...
        }
      }
  }
}
```

And for this SerObj.java

```
public class SerObj implements java.io.Serializable {
  public static final long serialVersionUID = 1L;
  public int i1 = 1;
  public double d2 = 2d;
  public long l9 = 9;
  public float f3 = 3f;
  public short s4 = 4;
  public char c5 = 'J';
  public byte b6 = (byte)'E';
  public boolean boo = true;
  public String T7="XinChao";
  // CUSTOM SERIALIZED OBJECT
  public People p = new People( "Joe", 73, "Kriftel", "Retiree", 18000, "J.jpg");
  public int[] I7 = { 5, 6 };
  public double[] D8 = {7, 8};
  public byte[] bb = {(byte)'j', (byte)'o', (byte)'e'};
  public String[] TT = { "hello", "world"};
  //
  public static void main(String... a) throws Exception {
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("serobj/SerObj.txt"));
    oos.writeObject(new SerObj());
```

```
    oos.flush();
    oos.close();
  }
}
```

And the output of ViewSerObject for SerObj

```
Class Name: SerObj
SerID: 1
Number of Fields: 14
Field: b6, type: byte, value/Reference: 69
Field: boo, type: boolean, value/Reference: true
Field: c5, type: char, value/Reference: J
Field: d2, type: double, value/Reference: 2.0
Field: f3, type: float, value/Reference: 3.0
Field: i1, type: int, value/Reference: 1
Field: l9, type: long, value/Reference: 9
Field: s4, type: short, value/Reference: 4
Field: D8, type: double[2], value/Reference: [D@26f67b76
    double[0]: 7,00
    double[1]: 8,00
Field: I7, type: int[2], value/Reference: [I@7ca48474
    int[0]: 5
    int[1]: 6
Field: T7, type: String, value/Reference: XinChao
Field: TT, type: String[2], value/Reference: [Ljava.lang.String;@337d0578
    String[0]: hello
    String[1]: world
Field: bb, type: byte[3], value/Reference: [B@59e84876
    byte[0]: j (x6A)
    byte[1]: o (x6F)
    byte[2]: e (x65)
Embedded Class Name: People
SerID: 1234
Number of Fields: 6
Field: age, type: int, value/Reference: 73
Field: income, type: double, value/Reference: 18000.0
Field: address, type: String, value/Reference: Kriftel
Field: name, type: String, value/Reference: Joe
Field: profession, type: String, value/Reference: Retiree
Field: url, type: String, value/Reference: J.jpg
```

The Serobj.txt:

```
¬í #sr #SerObj         ## #B #b6Z #booC #c5D #d2F #f3I #i1J #l9S #s4[ #D8t #[D[ #I7t
#[IL #T7t #Ljava/lang/String;[ #TTt #[Ljava/lang/String;[ #bbt #[BL #pt #LPeople;xpE#
J@       @@     #          #ur #[D>¦Œ#«cZ##  xp   #@#      @       ur #[IMº`&vê²¥#  xp
#     #     #t  #XinChaour  #[Ljava.lang.String;ÒVçé#{G#    xp     #t  #hellot  #worldur
#[B¬ó#ø##Tà#  xp    #joesr #People       #Ò# #I #ageD #incomeL #addressq ~ #L #nameq ~
#L
professionq ~ #L #urlq ~ #xp   I@Ñ”     t #Kriftelt #Joet #Retireet #J.jpg
```

Code can be downloaded from Github: click HERE