AVRO - SERIALIZATION

What is Serialization?

Serialization is the process of translating data structures or objects state into binary or textual form to transport the data over network or to store on some persisten storage. Once the data is transported over network or retrieved from the persistent storage, it needs to be deserialized again. Serialization is termed as **marshalling** and deserialization is termed as **unmarshalling**.

Serialization in Java

Java provides a mechanism, called **object serialization** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object is written into a file, it can be read from the file and deserialized. That is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

ObjectInputStream and **ObjectOutputStream** classes are used to serialize and deserialize an object respectively in Java.

Serialization in Hadoop

Generally in distributed systems like Hadoop, the concept of serialization is used for **Interprocess Communication** and **Persistent Storage**.

Interprocess Communication

- To establish the interprocess communication between the nodes connected in a network, RPC technique was used.
- RPC used internal serialization to convert the message into binary format before sending it to the remote node via network. At the other end the remote system deserializes the binary stream into the original message.
- The RPC serialization format is required to be as follows −
    - **Compact** − To make the best use of network bandwidth, which is the most scarce resource in a data center.
    - **Fast** − Since the communication between the nodes is crucial in distributed systems, the serialization and deserialization process should be quick, producing less overhead.
    - **Extensible** − Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers.
    - **Interoperable** − The message format should support the nodes that are written in different languages.

Persistent Storage

Persistent Storage is a digital storage facility that does not lose its data with the loss of power supply. For example - Magnetic disks and Hard Disk Drives.

Writable Interface

This is the interface in Hadoop which provides methods for serialization and deserialization. The following table describes the methods −

S.No.	Methods and Description

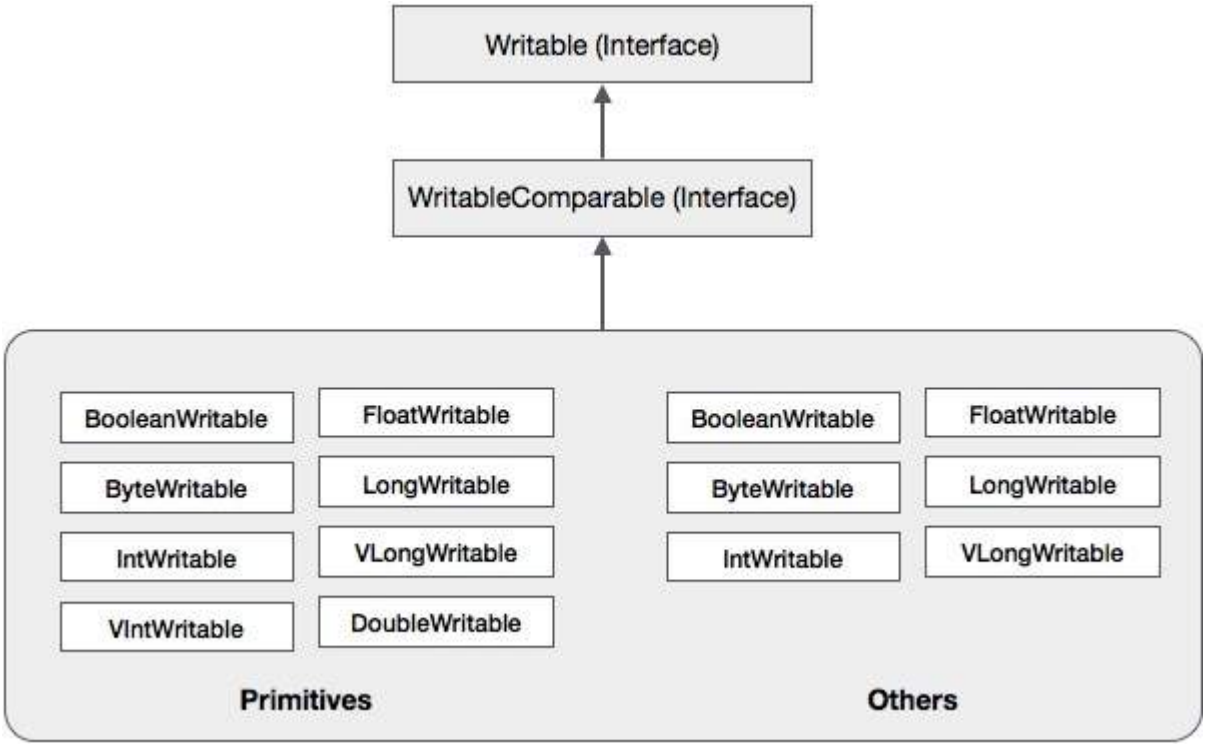
1	**void readFields***DataInputin* This method is used to deserialize the fields of the given object.
2	**void write***DataOutputout* This method is used to serialize the fields of the given object.

WritableComparable Interface

It is the combination of **Writable** and **Comparable** interfaces. This interface inherits **Writable** interface of Hadoop as well as **Comparable** interface of Java. Therefore it provides methods for data serialization, deserialization, and comparison.

S.No.	Methods and Description
1	**int compareTo***classobj* This method compares current object with the given object obj.

In addition to these classes, Hadoop supports a number of wrapper classes that implement WritableComparable interface. Each class wraps a Java primitive type. The class hierarchy of Hadoop serialization is given below −



These classes are useful to serialize various types of data in Hadoop. For instance, let us consider the **IntWritable** class. Let us see how this class is used to serialize and deserialize the data in Hadoop.

IntWritable Class

This class implements **Writable, Comparable,** and **WritableComparable** interfaces. It wraps an integer data type in it. This class provides methods used to serialize and deserialize integer type of

data.

Constructors

S.No.	Summary
1	**IntWritable**
2	**IntWritable***intvalue*

Methods

S.No.	Summary
1	**int get** Using this method you can get the integer value present in the current object.
2	**void readFields***DataInputin* This method is used to deserialize the data in the given **DataInput** object.
3	**void set***intvalue* This method is used to set the value of the current **IntWritable** object.
4	**void write***DataOutputout* This method is used to serialize the data in the current object to the given **DataOutput** object.

Serializing the Data in Hadoop

The procedure to serialize the integer type of data is discussed below.

- Instantiate **IntWritable** class by wrapping an integer value in it.
- Instantiate **ByteArrayOutputStream** class.
- Instantiate **DataOutputStream** class and pass the object of **ByteArrayOutputStream** class to it.
- Serialize the integer value in **IntWritable** object using **write** method. This method needs an object of DataOutputStream class.
- The serialized data will be stored in the byte array object which is passed as parameter to the **DataOutputStream** class at the time of instantiation. Convert the data in the object to byte array.

Example

The following example shows how to serialize data of integer type in Hadoop −

```
import java.io.ByteArrayOutputStream;
```

```
import java.io.DataOutputStream;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

public class Serialization {
   public byte[] serialize() throws IOException{

      //Instantiating the IntWritable object
      IntWritable intwritable = new IntWritable(12);

      //Instantiating ByteArrayOutputStream object
      ByteArrayOutputStream byteoutputStream = new ByteArrayOutputStream();

      //Instantiating DataOutputStream object
      DataOutputStream dataOutputStream = new
      DataOutputStream(byteoutputStream);

      //Serializing the data
      intwritable.write(dataOutputStream);

      //storing the serialized object in bytearray
      byte[] byteArray = byteoutputStream.toByteArray();

      //Closing the OutputStream
      dataOutputStream.close();
      return(byteArray);
   }

   public static void main(String args[]) throws IOException{
      Serialization serialization= new Serialization();
      serialization.serialize();
      System.out.println();
   }
}
```

Deserializing the Data in Hadoop

The procedure to deserialize the integer type of data is discussed below −

- Instantiate **IntWritable** class by wrapping an integer value in it.
- Instantiate **ByteArrayOutputStream** class.
- Instantiate **DataOutputStream** class and pass the object of **ByteArrayOutputStream** class to it.
- Deserialize the data in the object of **DataInputStream** using **readFields** method of IntWritable class.
- The deserialized data will be stored in the object of IntWritable class. You can retrieve this data using **get** method of this class.

Example

The following example shows how to deserialize the data of integer type in Hadoop −

```
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;

import org.apache.hadoop.io.IntWritable;

public class Deserialization {

   public void deserialize(byte[]byteArray) throws Exception{

      //Instantiating the IntWritable class
```

```
      IntWritable intwritable =new IntWritable();

      //Instantiating ByteArrayInputStream object
      ByteArrayInputStream InputStream = new ByteArrayInputStream(byteArray);

      //Instantiating DataInputStream object
      DataInputStream datainputstream=new DataInputStream(InputStream);

      //deserializing the data in DataInputStream
      intwritable.readFields(datainputstream);

      //printing the serialized data
      System.out.println((intwritable).get());
   }

   public static void main(String args[]) throws Exception {
      Deserialization dese = new Deserialization();
      dese.deserialize(new Serialization().serialize());
   }
}
```

Advantage of Hadoop over Java Serialization

Hadoop's Writable-based serialization is capable of reducing the object-creation overhead by reusing the Writable objects, which is not possible with the Java's native serialization framework.

Disadvantages of Hadoop Serialization

To serialize Hadoop data, there are two ways −

- You can use the **Writable** classes, provided by Hadoop's native library.
- You can also use **Sequence Files** which store the data in binary format.

The main drawback of these two mechanisms is that **Writables** and **SequenceFiles** have only a Java API and they cannot be written or read in any other language.

Therefore any of the files created in Hadoop with above two mechanisms cannot be read by any other third language, which makes Hadoop as a limited box. To address this drawback, Doug Cutting created **Avro**, which is a **language independent data structure**.