

* Writable Interface!→

Writable is interface mechanism to Serialise and de-Serialise your data. It is an interface which has write method and read fields method.

• Write field!→

Write field is to writing your data into the output stream or network.

• Read field!→

Read field is to read data from the input stream.

The Hadoop is used for MapReduce Computations. It uses the Writable interface based classes as the data types. These data types from Writable are used throughout the MapReduce Data Flow Structure. It starts from reading input data, transferring intermediate data between Map & Reduce and then writing output data.

* Writable Interface Functions!→

- A data type must implements the `org.apache.hadoop.io.Writable` interface in order to be used as a Value data type of a MapReduce Computation.
- It is only one interface will define how a Value should be Serialized and de-Serialized in Hadoop for ~~transf~~ transmitting and storing the data.

Coding :-

```
package org.apache.hadoop.io;
```

```
import java.io.DataInput;
```

```
import java.io.DataOutput;
```

```
public interface Writable
```

```
{
```

```
    void write (DataOutput out) throws IOException;
```

```
    void readFields (DataInput in) throws IOException;
```

```
}
```

* Writable Comparable and Comparators :->

IntWritable implements the ~~int~~ WritableComparable interface, which is just a subinterface of the Writable and java.lang.Comparable interface.
~~java.lang.Comparable~~

```
package org.apache.hadoop.io;
```

```
public interface WritableComparable <T> extends Writable, Comparable <T>
```

```
{
```

```
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. one optimization that Hadoop provides is the RowComparator extension of Java's Comparator.

```
package org.apache.hadoop.io;
```

```
import java.util.Comparator;
```

```
public interface RawComparator<T> extends Comparator<T>
```

```
{
```

```
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int
```

```
    l2)
```

In the above example, the Comparator for IntWritable implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly, from the given start positions (s1 and s2) and lengths (l1 and l2).

* Difference between WritableComparable and WritableComparator :->

	Writable Comparable	Writable Comparator
1.	Writable Comparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.	A Comparator for Writable Comparables. This base implementation uses the natural ordering. To define alternate orderings, override compare (Writable Comparable, Writable Comparable).
2.	org.apache.hadoop.io.Writable Comparable	org.apache.hadoop.io.WritableComparator

For implementing a WritableComparable we must have Compare To method apart from readFields and write methods, as

A Comparator that operates directly on byte representations of object.
Compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)

Compare two objects in binary.
b1[s1:l1] is the first object and b2[s2:l2] is the second object.

* Writable Classes: →

Hadoop comes with a large selection of Writable classes in the org.apache.hadoop.io package.

Hadoop provides classes that wrap the Java primitive types and implement the WritableComparable and Writable Interfaces. They are provided in the org.apache.hadoop.io package.

They form the class hierarchy shown in figure:-

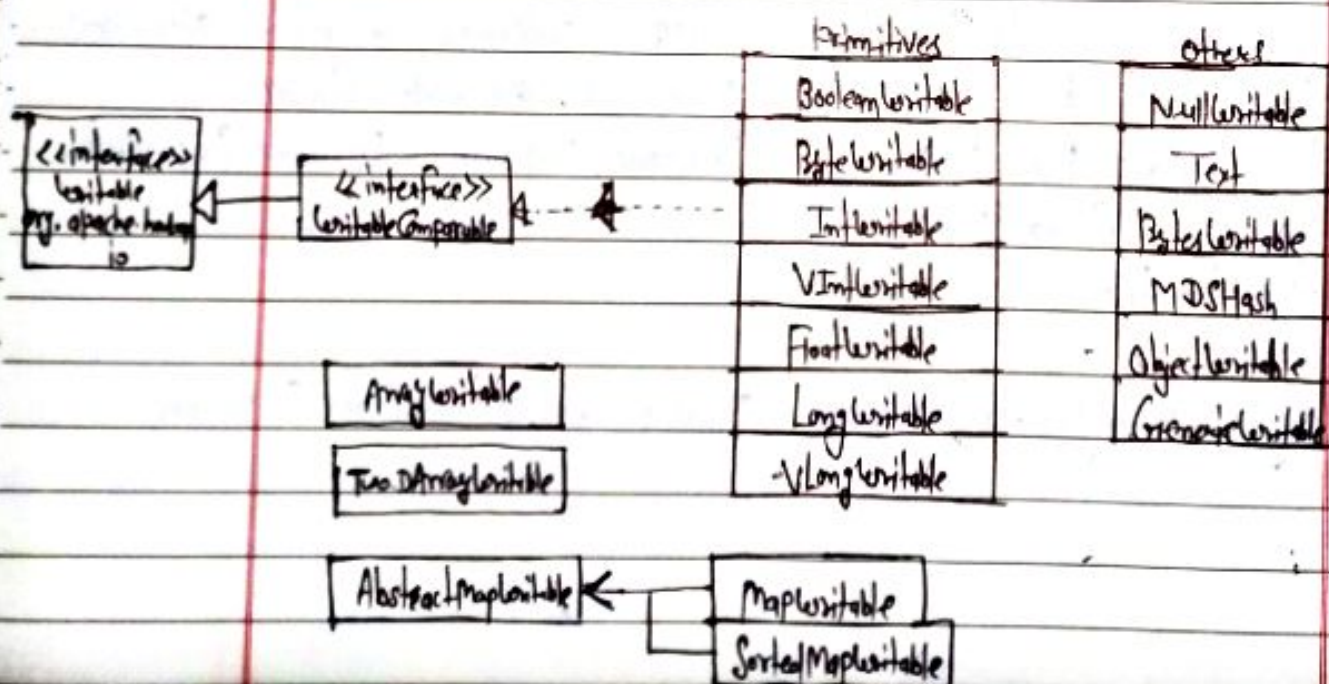


Figure 1: Writable class hierarchy.

⇒ Writable Wrappers for Java Primitives :→

There are Writable Wrappers for all the Java primitive types except char (which can be stored in an IntWritable) as shown in ~~table~~ ^{Table}! All have a get() and a set() method for retrieving and storing the wrapped value.

Java Primitive	Writable Implementation	Serialized Size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1-5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1-9
double	DoubleWritable	8

Table 1:- Writable wrapper classes for Java primitives

When encoding integers, there is a choice between the fixed-length formats (IntWritable and LongWritable) and the Variable-length formats (VIntWritable and VLongWritable). The Variable-length formats use only a single byte to encode the value if it is small enough (between -128 and 127, inclusive); otherwise, they use the first byte to indicate whether the value is positive or negative, and

And 163 ~~requires~~ requires two bytes.

⇒ Text :-

Text is a Writable for UTF-8 sequences. It can be thought of as the Writable Equivalent of `java.lang.String`.

The ~~text~~ Text class uses an `int` (with a variable-length encoding) to store the number of bytes in the string encoding, so the maximum value is 2 GB.

• Indexing :-

Because of its emphasis on using standard UTF-8, ~~there are some~~

• Unicode :-

When we start using characters that are encoded with more than a single byte. Consider the unicode characters shown in table 2.

Unicode Code point	U+0041	U+00DF	U+6771	U+10400
Name	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	N/A (a unified Han ideograph)	DESERET CAPITAL LETTER LONG
UTF-8 Code units	41	C3 9F	E6 9D B1	F0 90 90 80
Java representation	\u0041	\u00DF	\u6771	\u10400

Table-2 :- Unicode characters.

• Iteration :-

Iteration over the Unicode characters in Text is complicated by the use of byte offsets for indexing, since you can't just increment the index.

⇒ BytesWritable :-

BytesWritable is a wrapper for an array of binary data. Its serialized format is an integer field (4 bytes) that specifies the number of bytes to follow, followed by the bytes themselves. For example:- the byte array of length two with value 3 and 5 is serialized as a 4-byte integer (00000002) followed by the two bytes from the array (03 and 05).

```
BytesWritable b = new BytesWritable(new byte[] {3, 5});  
byte[] bytes = Serialize(b);  
assertThat(StringUtil.byteToHexString(bytes), is("000000020305"));  
BytesWritable is mutable, and its value may be changed by calling  
its set() method.
```

⇒ NullWritable :-

NullWritable is a special type of Writable, as it has a zero-length serialization. No bytes are written to, or read from, the stream. It is used as a placeholder; for example - in MapReduce, a key or a value can be declared as a NullWritable when you don't need to use that position - it effectively stores a constant empty value. NullWritable can also be useful as a key in SequenceFile when we want to store a list of values, as opposed to key-value pairs.

⇒ ObjectWritable :→

ObjectWritable is a general-purpose wrapper for the following: Java primitives, String, Enum, Writable, ~~and~~ null, or arrays of any of these types.

⇒ GenericWritable :→

GenericWritable is useful when a field can be of more than one type:

for example → if the values in a SequenceFile have multiple types, then we can declare the value type as an GenericWritable and wrap each type in an GenericWritable.

* Writable Collections :→

There are six Writable Collection types in the org.apache.hadoop.io package.

ArrayWritable, ArrayPrimitiveWritable, TwoDArrayWritable, MapWritable, SortedMapWritable and EnumSetWritable.

- ArrayWritable and TwoDArrayWritable are Writable implementations for arrays and Two-dimensional Arrays (array of arrays) of Writable instances. All the elements of an ArrayWritable or a TwoDArrayWritable must be instances of the same class, which is specified at construction, as follows:-

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

ArrayWritable and TwoArrayWritable both have get() and set() methods as well as a toArray() method, which creates a shallow copy of the array.

- ArrayPrimitiveWritable is a wrapper for arrays of Java primitives. The component type is detected when you call set(), so there is no need to subclass to set the type.
- MapWritable and SortedMapWritable are implementations of `java.util.Map<Writable, Writable>` and `java.util.SortedMap<WritableComparable, Writable>`, respectively. Here's demonstration of using a MapWritable with different types for keys and values.

* Implementing a Custom Writable →

Hadoop comes with a useful set of Writable implementations that serve most purposes. You may need to write your own custom implementation. And with a custom Writable we have full control over the binary representation and the sort order.

Because Writables are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. To demonstrate how to create a custom Writable, we shall write an implementation that represents a pair of strings, called TextPair. ~~The~~ to

* Implementing a RawComparator for speed :->

Implementing the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce jobs. As you may recall, a MR (Map/Reduce) job is composed of creating and sending key-value pairs. The process looks like the following:-

$(k_1, v_1) \rightarrow \text{Map} \rightarrow (k_2, v_2)$
 $(k_2, \text{List}[v_2]) \rightarrow \text{Reduce} \rightarrow (k_3, v_3)$

The key-value pairs (k_2, v_2) are called the intermediary key-value pairs. They are ~~for~~ passed from the mapper to the reducer. Before these ~~intermediary~~ intermediary key-value pairs reach the reducer, a shuffle and sort step is performed. The shuffle is the assignment of the intermediary keys (k_2) to reducers and the sort is the sorting of these keys.

Two ways you may compare your keys is by implementing the org.apache.hadoop.io.WritableComparable interface or by implementing the RawComparator interface.

In the former approach, you will compare objects, but in the latter approach, you will compare the keys using their corresponding raw bytes.

* Custom Comparators :->

As we can see with TextPair, writing raw Comparators takes some care, since we have to deal with details at the byte level.

Custom Comparators should also be written to be 'RawComparators', if possible. These are Comparators that implement a different sort order to the natural sort order defined by the default Comparator.

~~we~~ we override the compare () method that takes objects so both compare () methods have the same semantics.