factory for `RawComparator` instances (that `Writable` implementations have registered). For example, to obtain a comparator for `IntWritable`, we just use:

```
RawComparator<IntWritable> comparator = WritableComparator.get(IntWritable.class);
```

The comparator can be used to compare two `IntWritable` objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
    greaterThan(0));
```

Writable Classes

Hadoop comes with a large selection of `Writable` classes in the `org.apache.hadoop.io` package. They form the class hierarchy shown in Figure 4-1.

Writable wrappers for Java primitives

There are `Writable` wrappers for all the Java primitive types (see Table 4-7) except `char` (which can be stored in an `IntWritable`). All have a `get()` and `set()` method for retrieving and storing the wrapped value.

Table 4-7. Writable wrapper classes for Java primitives

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

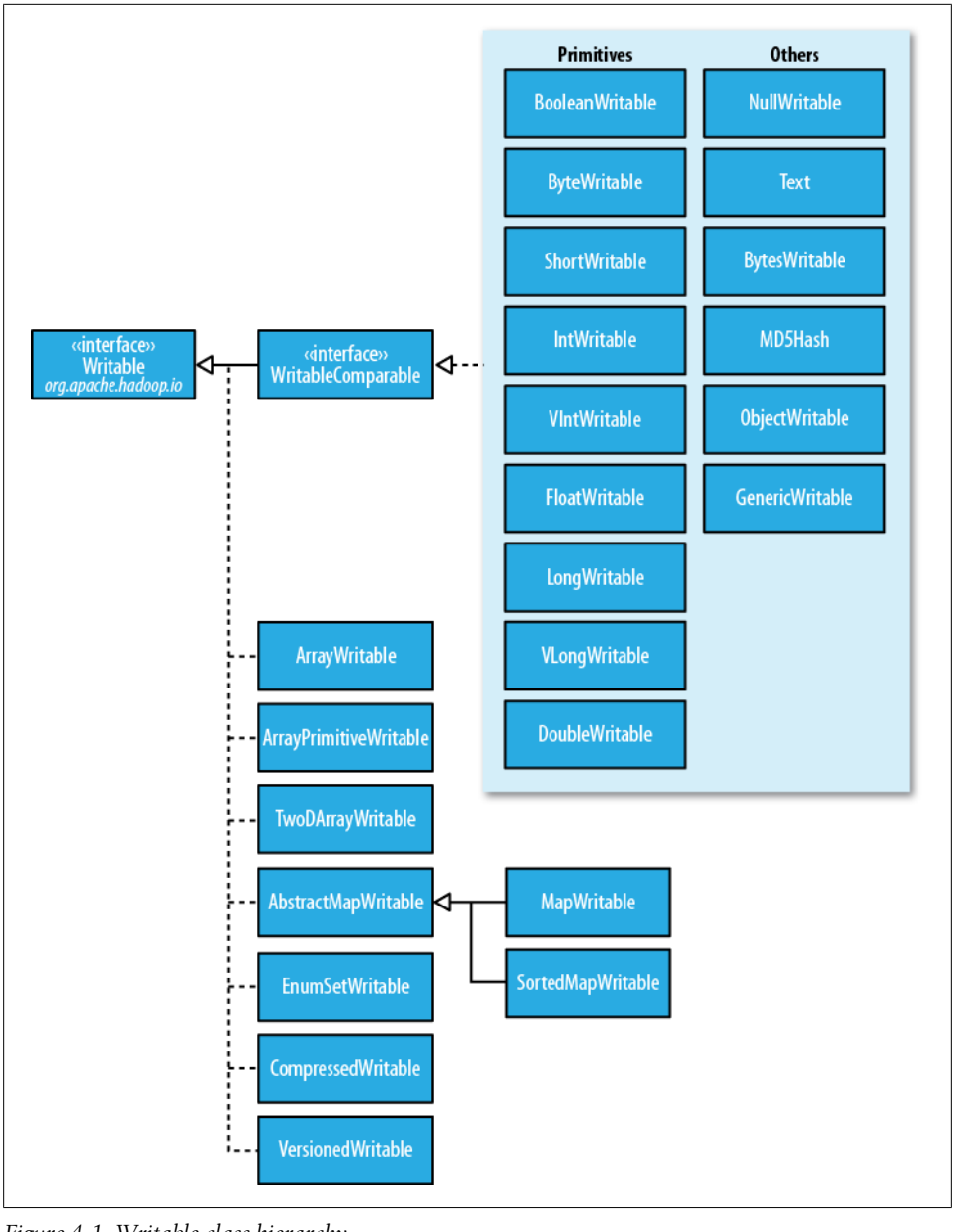


Figure 4-1. Writable class hierarchy

When it comes to 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 –112 and 127, inclusive); otherwise, they use the first

byte to indicate whether the value is positive or negative, and how many bytes follow. For example, 163 requires two bytes:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

How do you choose between a fixed-length and a variable-length encoding? Fixed-length encodings are good when the distribution of values is fairly uniform across the whole value space, such as a (well-designed) hash function. Most numeric variables tend to have nonuniform distributions, and on average the variable-length encoding will save space. Another advantage of variable-length encodings is that you can switch from `VIntWritable` to `VLongWritable`, because their encodings are actually the same. So by choosing a variable-length representation, you have room to grow without committing to an 8-byte `long` representation from the beginning.

Text

`Text` is a `Writable` for UTF-8 sequences. It can be thought of as the `Writable` equivalent of `java.lang.String`. `Text` is a replacement for the `UTF8` class, which was deprecated because it didn't support strings whose encoding was over 32,767 bytes and because it used Java's modified UTF-8.

The `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. Furthermore, `Text` uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

Indexing. Because of its emphasis on using standard UTF-8, there are some differences between `Text` and the Java `String` class. Indexing for the `Text` class is in terms of position in the encoded byte sequence, not the Unicode character in the string or the Java `char` code unit (as it is for `String`). For ASCII strings, these three concepts of index position coincide. Here is an example to demonstrate the use of the `charAt()` method:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Notice that `charAt()` returns an `int` representing a Unicode code point, unlike the `String` variant that returns a `char`. `Text` also has a `find()` method, which is analogous to `String`'s `indexOf()`:

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

`ArrayWritable` and `TwoDArrayWritable` both have `get()` and `set()` methods, as well as a `toArray()` method, which creates a shallow copy of the array (or 2D 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. The type of each key and value field is a part of the serialization format for that field. The type is stored as a single byte that acts as an index into an array of types. The array is populated with the standard types in the `org.apache.hadoop.io` package, but custom `Writable` types are accommodated, too, by writing a header that encodes the type array for nonstandard types. As they are implemented, `MapWritable` and `SortedMapWritable` use positive `byte` values for custom types, so a maximum of 127 distinct nonstandard `Writable` classes can be used in any particular `MapWritable` or `SortedMapWritable` instance. Here's a demonstration of using a `MapWritable` with different types for keys and values:

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)), is(new
  LongWritable(163)));
```

Conspicuous by their absence are `Writable` collection implementations for sets and lists. A general set can be emulated by using a `MapWritable` (or a `SortedMapWritable` for a sorted set) with `NullWritable` values. There is also `EnumSetWritable` for sets of enum types. For lists of a single type of `Writable`, `ArrayWritable` is adequate, but to store different types of `Writable` in a single list, you can use `GenericWritable` to wrap the elements in an `ArrayWritable`. Alternatively, you could write a general `ListWritable` using the ideas from `MapWritable`.

Implementing a Custom Writable

Hadoop comes with a useful set of `Writable` implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. With a custom `Writable`, you have full control over the binary representation and the sort order. Because `Writable`s are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. The stock `Writable` implementations that come with Hadoop are well-tuned, but for more elaborate structures, it is often better to create a new `Writable` type rather than compose the stock types.

To demonstrate how to create a custom `Writable`, we shall write an implementation that represents a pair of strings, called `TextPair`. The basic implementation is shown in Example 4-7.

Example 4-7. A Writable implementation that stores a pair of Text objects

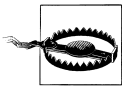
```
import java.io.*;

import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<TextPair> {

  private Text first;
  private Text second;

  public TextPair() {
    set(new Text(), new Text());
  }

  public TextPair(String first, String second) {
    set(new Text(first), new Text(second));
  }

  public TextPair(Text first, Text second) {
    set(first, second);
  }

  public void set(Text first, Text second) {
    this.first = first;
    this.second = second;
  }

  public Text getFirst() {
    return first;
  }

  public Text getSecond() {
    return second;
  }

  @Override
  public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException {
    first.readFields(in);
    second.readFields(in);
  }

  @Override
  public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
```

```
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof TextPair) {
      TextPair tp = (TextPair) o;
      return first.equals(tp.first) && second.equals(tp.second);
    }
    return false;
  }

  @Override
  public String toString() {
    return first + "\t" + second;
  }

  @Override
  public int compareTo(TextPair tp) {
    int cmp = first.compareTo(tp.first);
    if (cmp != 0) {
      return cmp;
    }
    return second.compareTo(tp.second);
  }
}
```

The first part of the implementation is straightforward: there are two `Text` instance variables, `first` and `second`, and associated constructors, getters, and setters. All `Writable` implementations must have a default constructor so that the MapReduce framework can instantiate them, then populate their fields by calling `readFields()`. Writable instances are mutable and often reused, so you should take care to avoid allocating objects in the `write()` or `readFields()` methods.

`TextPair`'s `write()` method serializes each `Text` object in turn to the output stream by delegating to the `Text` objects themselves. Similarly, `readFields()` deserializes the bytes from the input stream by delegating to each `Text` object. The `DataOutput` and `DataInput` interfaces have a rich set of methods for serializing and deserializing Java primitives, so, in general, you have complete control over the wire format of your `Writable` object.

Just as you would for any value object you write in Java, you should override the `hashCode()`, `equals()`, and `toString()` methods from `java.lang.Object`. The `hashCode()` method is used by the `HashPartitioner` (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.



If you ever plan to use your custom `Writable` with `TextOutputFormat`, then you must implement its `toString()` method. `TextOutputFormat` calls `toString()` on keys and values for their output representation. For `TextPair`, we write the underlying `Text` objects as strings separated by a tab character.