Hadoop I/O

Learning Objectives

- Data Integrity
- Compression
- Serialization
- File-based Data Structures

Data Integrity – 1

- The chance of data corruption is high because of the large volumes of data handled by Hadoop.
- Checksum
    - Usual way of detecting corrupted data.
    - Technique for only error detection (cannot fix the corrupted data).
    - CRC-32 (cyclic redundancy check) - compute a 32-bit integer checksum for input of any size.

Data Integrity – 2

- HDFS transparently checksums all data written to it and by default verifies checksums when reading data.
  - Compute checksums for every 512 bytes by default.
- Datanodes are responsible for verifying the data they receive before storing the data and its checksum.
  - Client receives a ChecksumException, if it detects an error.
- When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanode.

Data Integrity – 3

- Checksum verification log
  - Each datanode keeps a persistent log to know the last time each of its blocks was verified.
  - When a client successfully verifies a block, it tells the datanode who sends the block.
  - Then, the datanode updates its log.
- DataBlockScanner
  - Background thread that periodically verifies all the blocks stored on the datanode.
  - Guard against corruption due to “bit rot” in the physical storage media.

Data Integrity – 4

- Healing corrupted blocks
  - If a client detects an error when reading a block, it reports the bad block and the datanode to the namenode.
  - Namenode marks the block replica as corrupt.
  - Namenode schedules a copy of the block to be replicated on another datanode.
  - The corrupt replica is deleted.

Data Integrity – 5

- Disabling verification of checksum
  - Pass false to the **setVerifyCheckSum()** method on FileSystem.
  - **-ignoreCrc** option with the **-get** or the equivalent **-copyToLocal** command.
  - This feature is useful if you have a corrupt file that you want to inspect and decide what to do.

Data Integrity – 6

- LocalFileSystem
  - Performs client-side checksumming;
  - When you write a file called *filename*, the FS client transparently creates a hidden file, .*filename.crc*, in the same directory containing the checksums for each chunk of the file.
- ChecksumFileSystem
  - LocalFileSystem uses ChecksumFileSystem to do its work.
  - Make it easy to add checksumming to other (nonchecksummed) filesystems.

Compression

- Two major benefits of file compression
  - Reduce the space needed to store files;
  - Speed up data transfer across the network.
- When dealing with large volumes of data, both of these savings can be significant.
- So it pays to carefully consider how to use compression in Hadoop.

Hadoop Compression Codecs

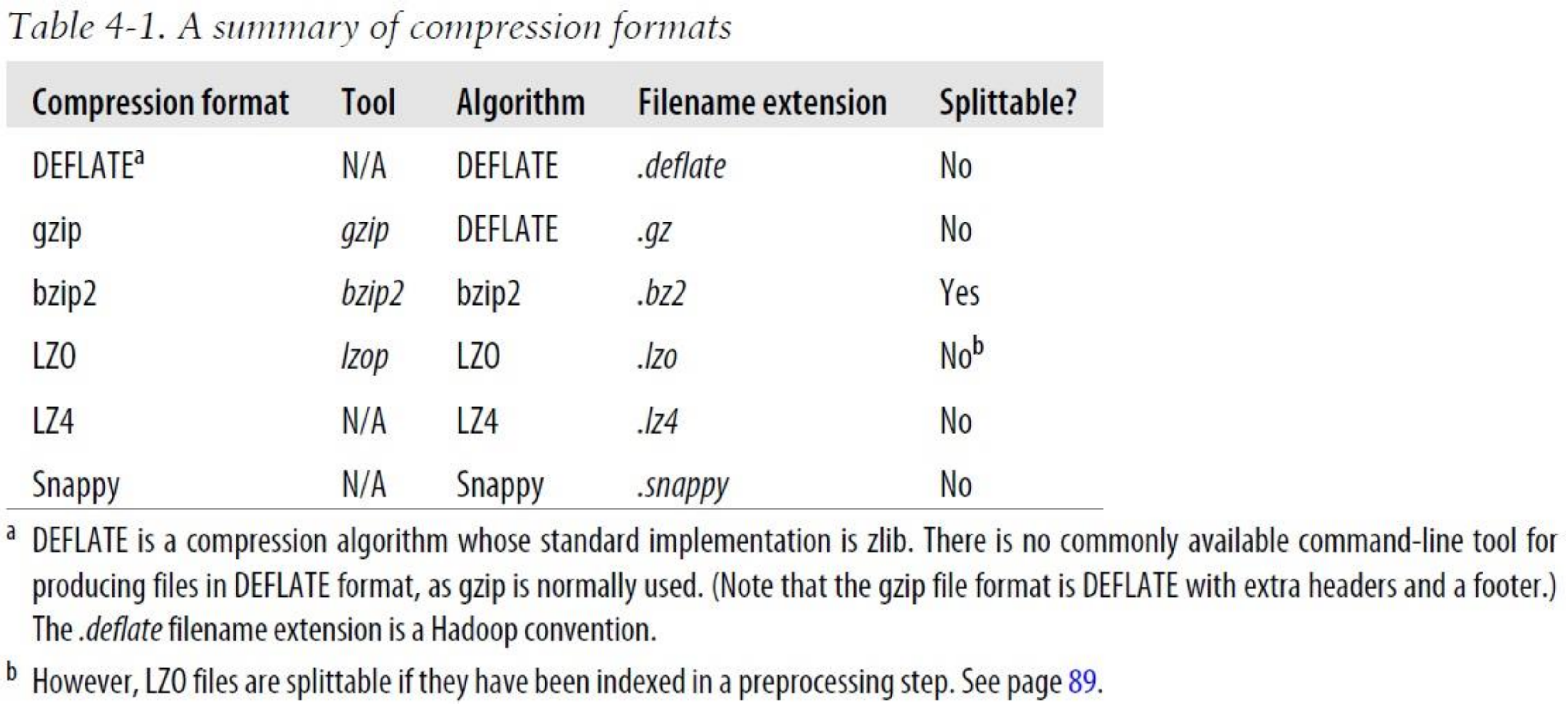


Table 4-1. A summary of compression formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

[a] DEFLATE is a compression algorithm whose standard implementation is zlib. There is no commonly available command-line tool for producing files in DEFLATE format, as gzip is normally used. (Note that the gzip file format is DEFLATE with extra headers and a footer.) The *.deflate* filename extension is a Hadoop convention.

[b] However, LZO files are splittable if they have been indexed in a preprocessing step. See page 89.

“Splittable” Column

- “Splittable” column
  - Indicates whether the compression format supports splitting;
  - Whether you can seek to any point in the stream and start reading from some point further on.
  - Splittable compression formats are especially suitable for MapReduce.

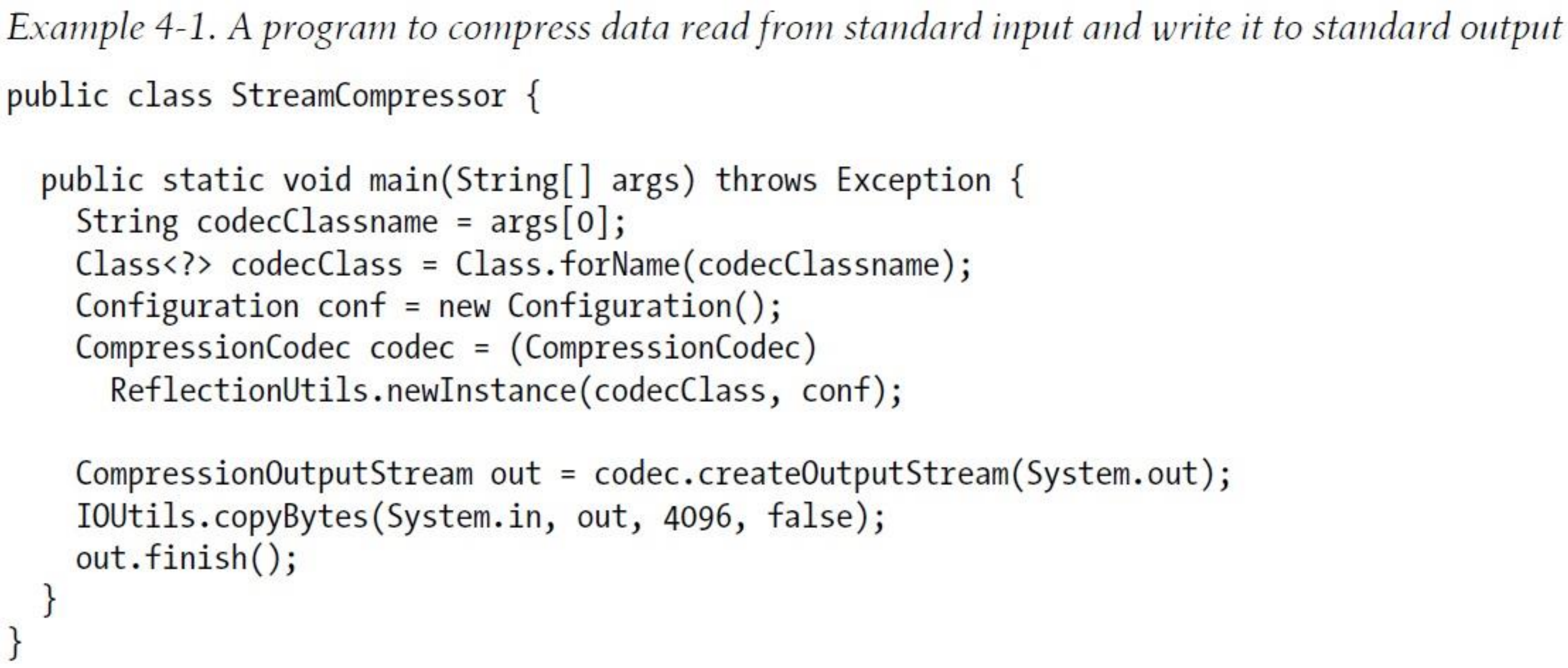
Input Splits

- It is important whether the compression format supports splitting.
- Example of not-splitable compression problem:
  - A gzip-compressed file is 1 GB;
  - Creating a split for each block won’t work since it is impossible to start reading at an arbitrary point in the gzip stream;
  - Therefore impossible for a map task to read its split independently of the others.

CompressionCodec

- CompressionCodec
  - **createOutputStream(OutputStream out)**: create a CompressionOutputStream to which you write your uncompressed data to have it written in compressed form to the underlying stream.
  - **createInputStream(InputStream in)**: obtain a CompressionInputStream, which allows you to read uncompressed data from the underlying stream.

Example



Example 4-1. A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \
| gunzip -
Text
```

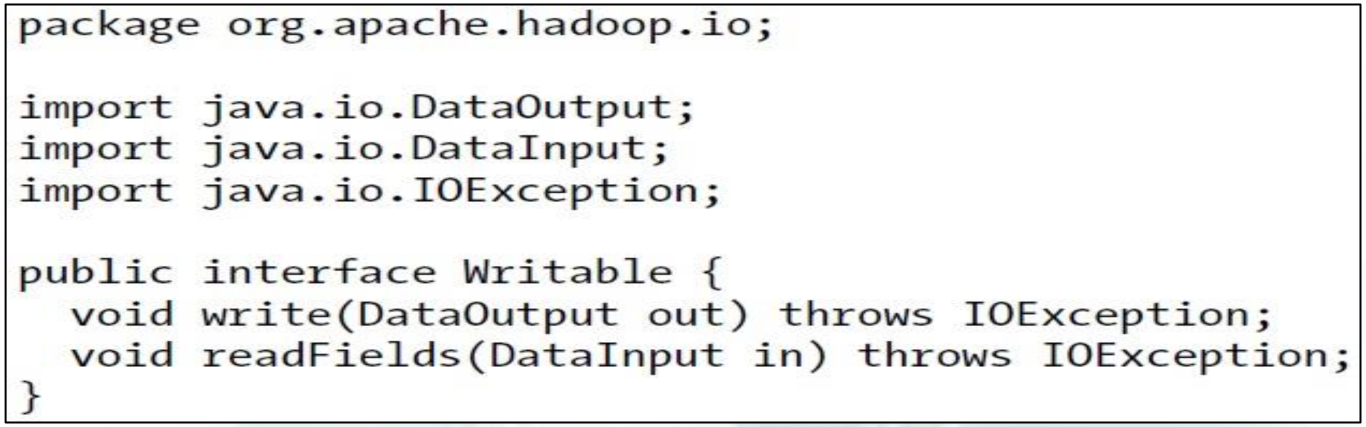
Serialization

- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.
- Deserialization is the reverse process of serialization.
- Serialization appears in two quite distinct areas of distributed data processes: for interprocess communication and for persistent storage.

Remote Procedure Calls (RPCs) Serialization

- In Hadoop, interprocess communication between nodes is implemented using RPCs. Requirements for RPCs:
  - Compact – to make efficient use of storage space;
  - Fast – the overhead in reading and writing of data is minimal;
  - Extensible – we can transparently read data written in an older format;
  - Interoperable – we can read or write persistent data using different language.

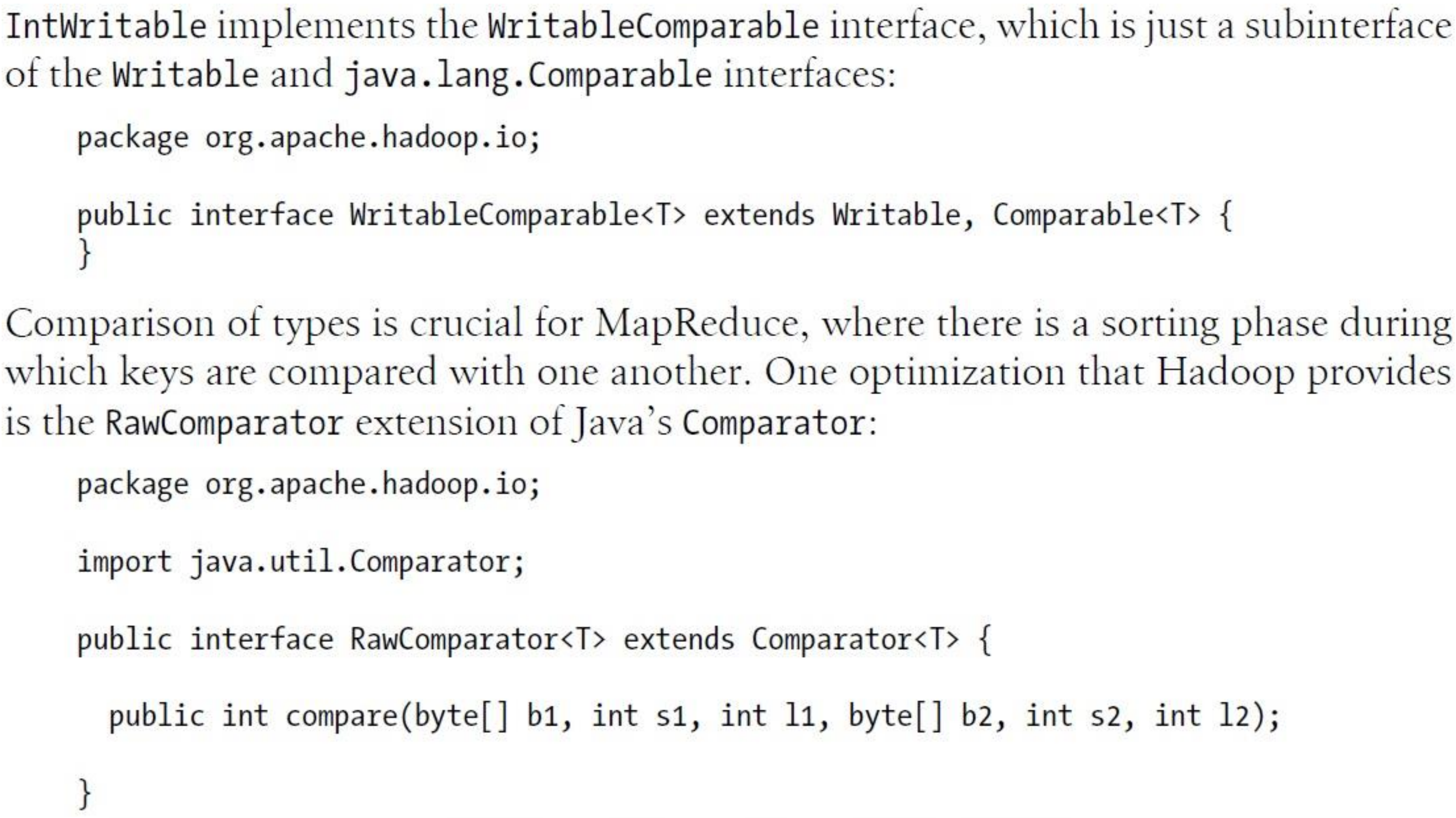
Writable Interface



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

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
  void readFields(DataInput in) throws IOException;
}
```

```
public static byte[] serialize(Writable writable) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(out);
    writable.write(dataOut);
    dataOut.close();
    return out.toByteArray();
}
```

WritableComparable and Comparator – 1



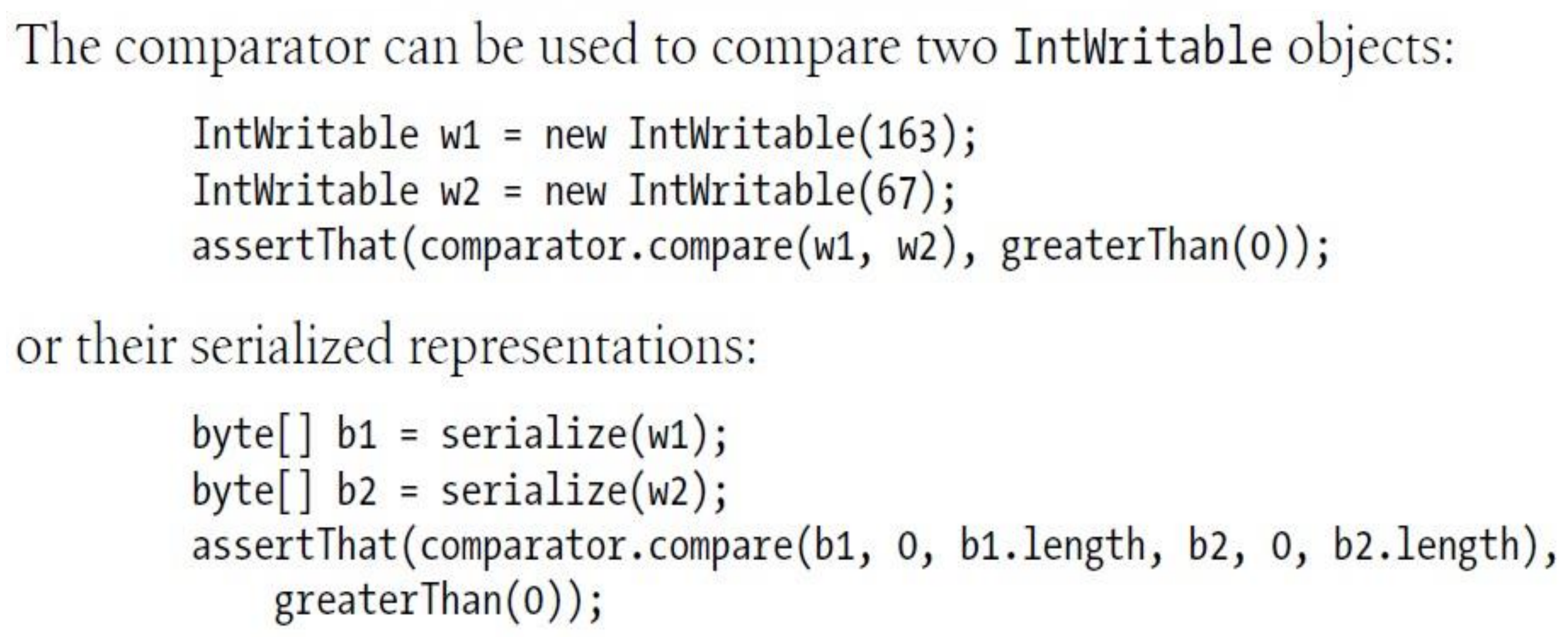
IntWritable implements the WritableComparable interface, which is just a subinterface of the Writable and java.lang.Comparable interfaces:

```
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 RawComparator 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);

}
```

WritableComparable and Comparator – 2



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));
```

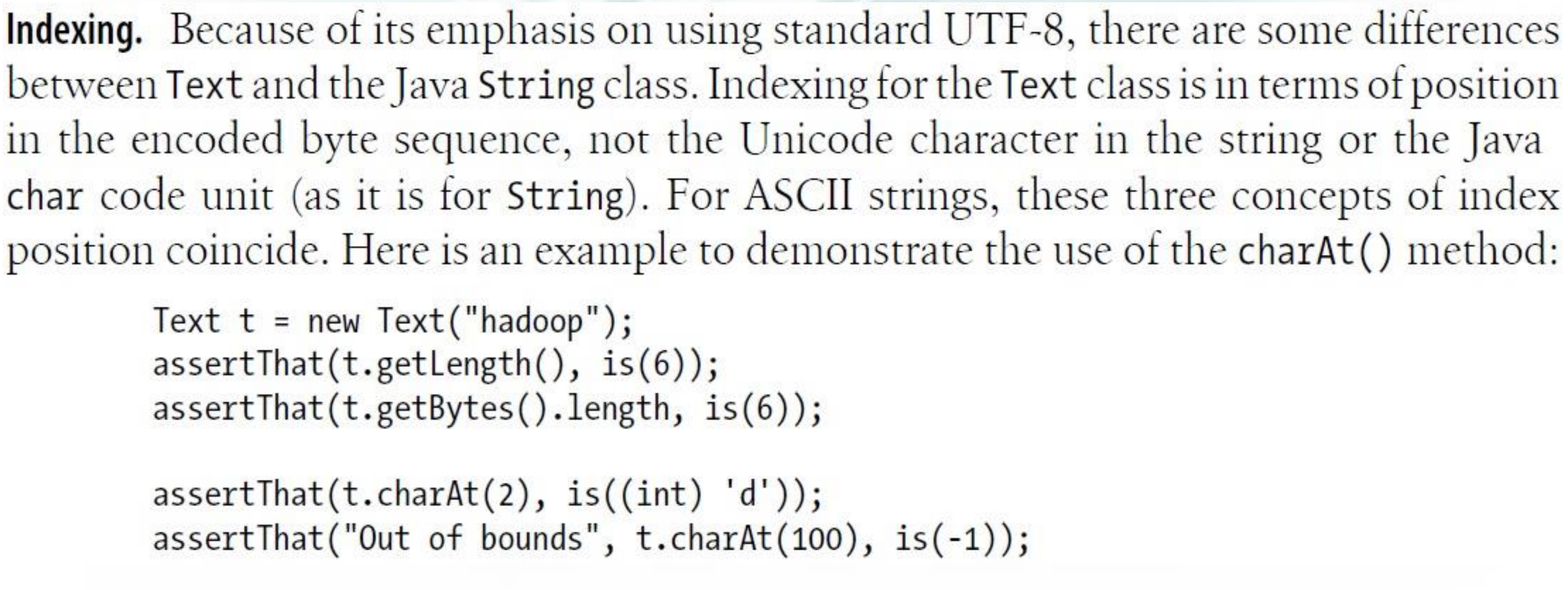
Writable Wrappers for Java Primitives

- There are Writable wrappers for all the Java primitive types except char(which can be stored in an IntWritable).
- get() for retrieving and set() for storing the wrapped value.

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

Text – 1

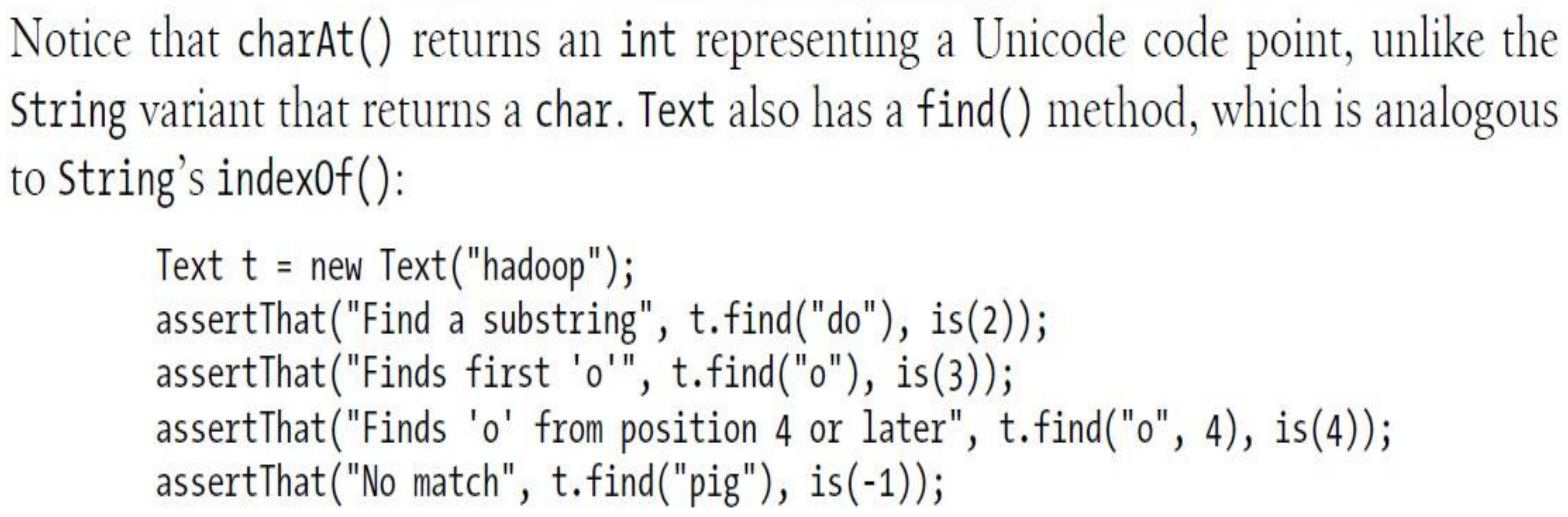
- Text is a Writable for UTF-8 sequences; can be seen as the Writable equivalent of java.lang.String.
- Text is a replacement for the org.apache.hadoop.io.UTF8 class (deprecated).



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));
```

Text – 2



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));
```

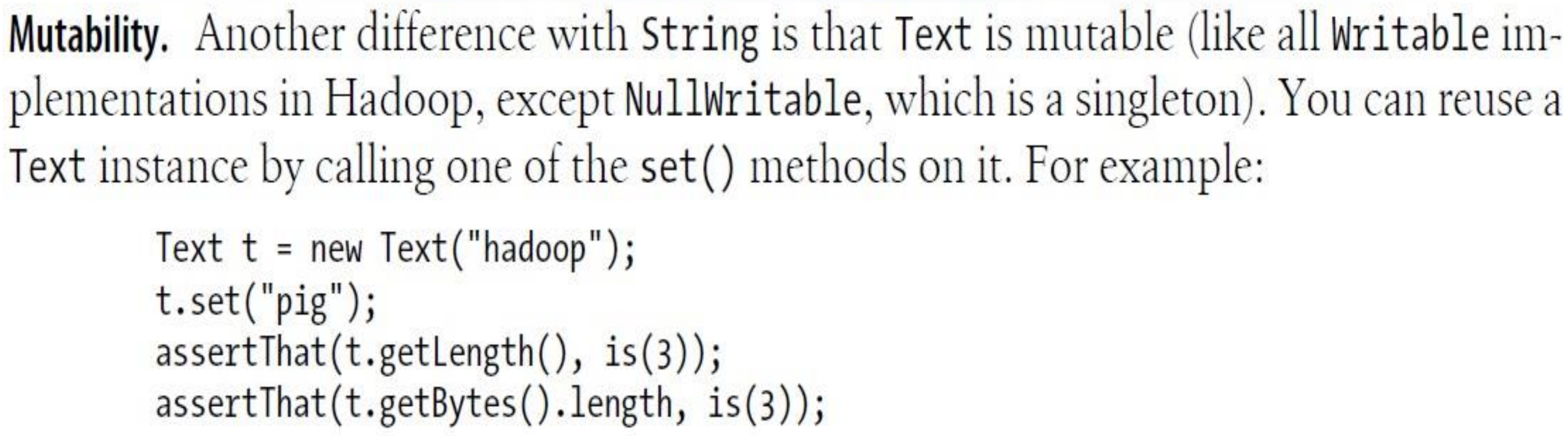
Iteration

Iteration. Iterating over the Unicode characters in `Text` is complicated by the use of byte offsets for indexing, since you can't just increment the index. The idiom for iteration is a little obscure (see Example 4-6): turn the `Text` object into a `java.nio.ByteBuffer`, then repeatedly call the `bytesToCodePoint()` static method on `Text` with the buffer. This method extracts the next code point as an `int` and updates the position in the buffer. The end of the string is detected when `bytesToCodePoint()` returns –1.

Example 4-6. Iterating over the characters in a Text object

```
public class TextIterator {

  public static void main(String[] args) {
    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");

    ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
    int cp;
    while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {
      System.out.println(Integer.toHexString(cp));
    }
  }
}
```

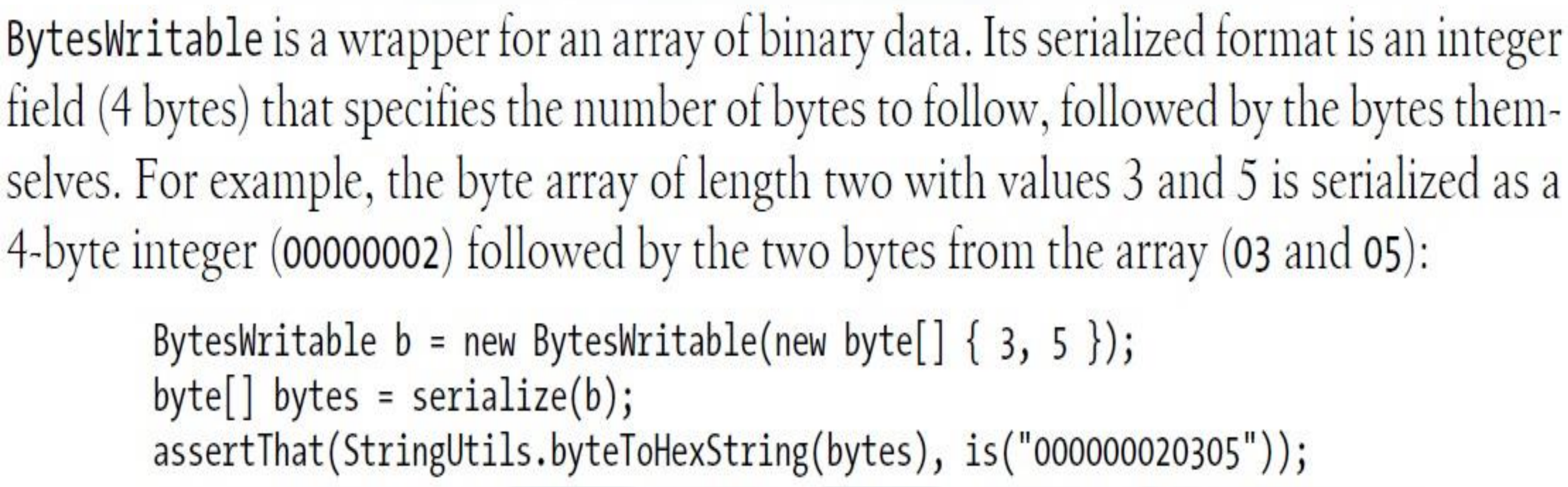
Mutability



Mutability. Another difference with `String` is that `Text` is mutable (like all `Writable` implementations in Hadoop, except `NullWritable`, which is a singleton). You can reuse a `Text` instance by calling one of the `set()` methods on it. For example:

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

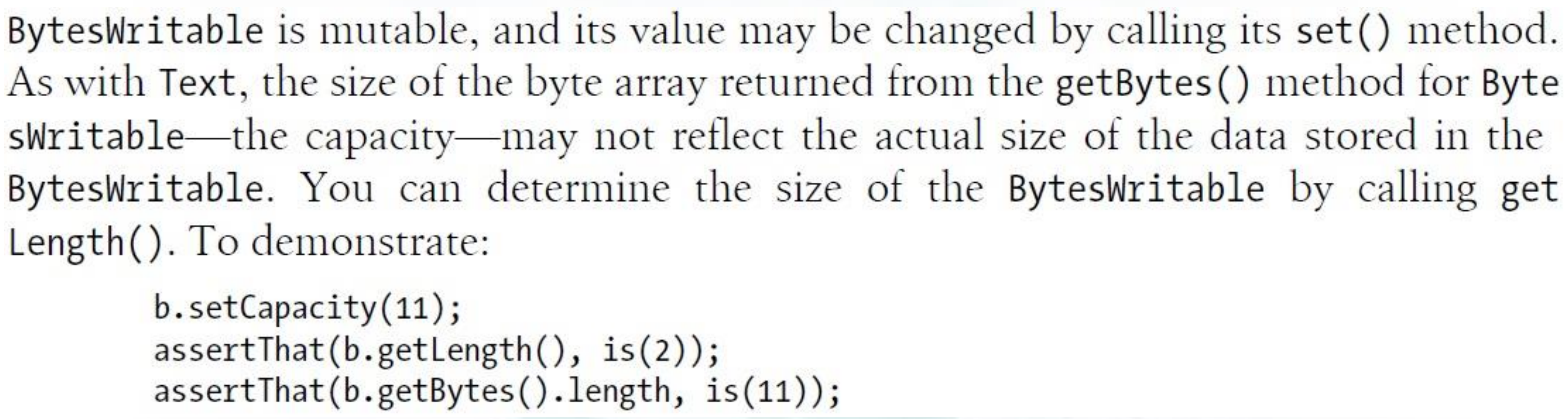
BytesWritable – 1



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 values 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(StringUtils.byteToHexString(bytes), is("000000020305"));
```

BytesWritable – 2



BytesWritable is mutable, and its value may be changed by calling its set() method. As with Text, the size of the byte array returned from the getBytes() method for Byte sWritable—the capacity—may not reflect the actual size of the data stored in the BytesWritable. You can determine the size of the BytesWritable by calling get Length(). To demonstrate:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

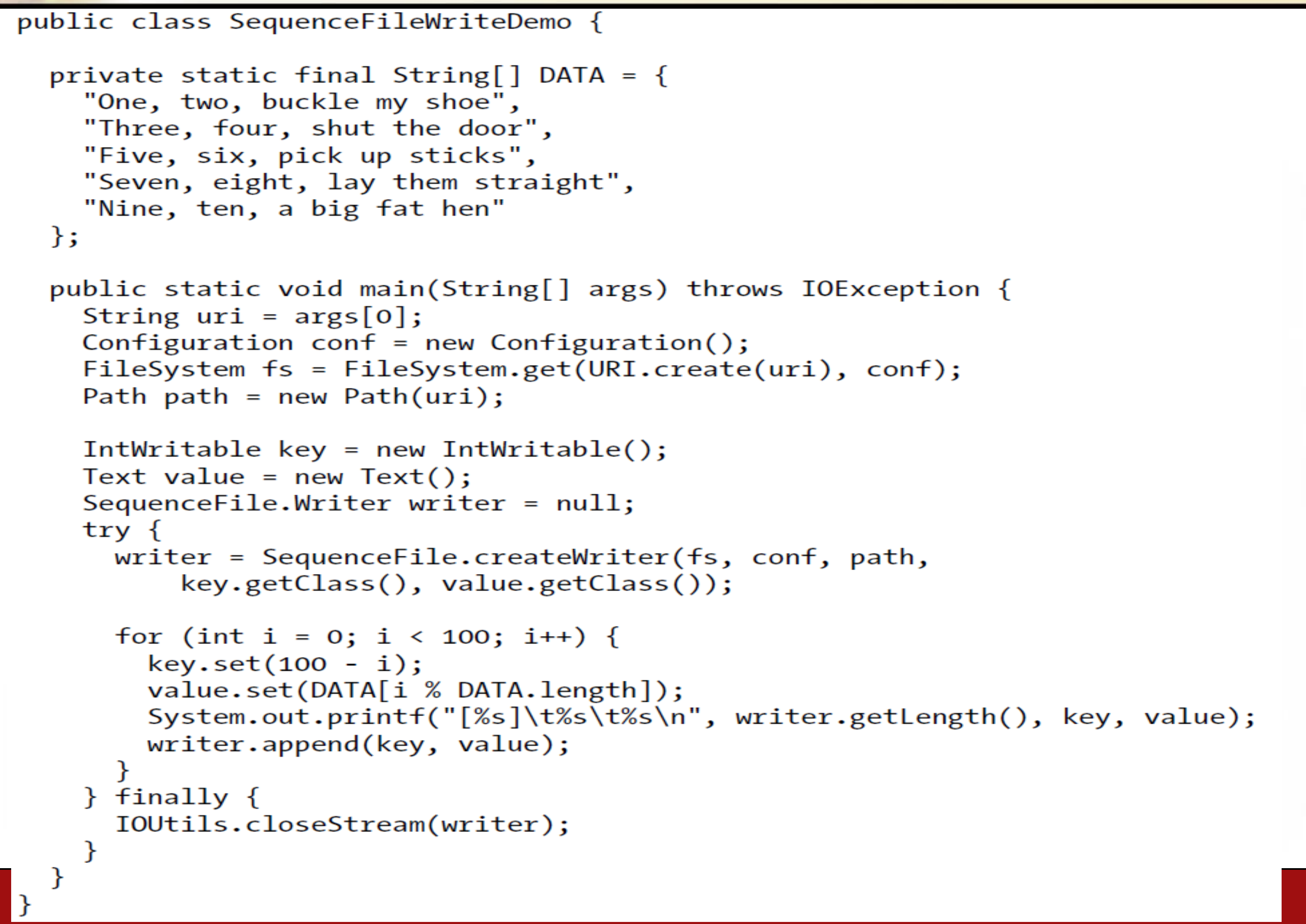
SequenceFile

- Hadoop’s SequenceFile class provides a persistent data structure for binary key-value pairs.
- It is used as a logfile format.
- As containers for smaller files.
- Usage example
  - Key: timestamp represented by a LongWritable.
  - Value: Writable represents the quantity being logged.

Hadoop – The Definitive Guide

Chapter 4: Hadoop I/O

Writing a SequenceFile – p.131

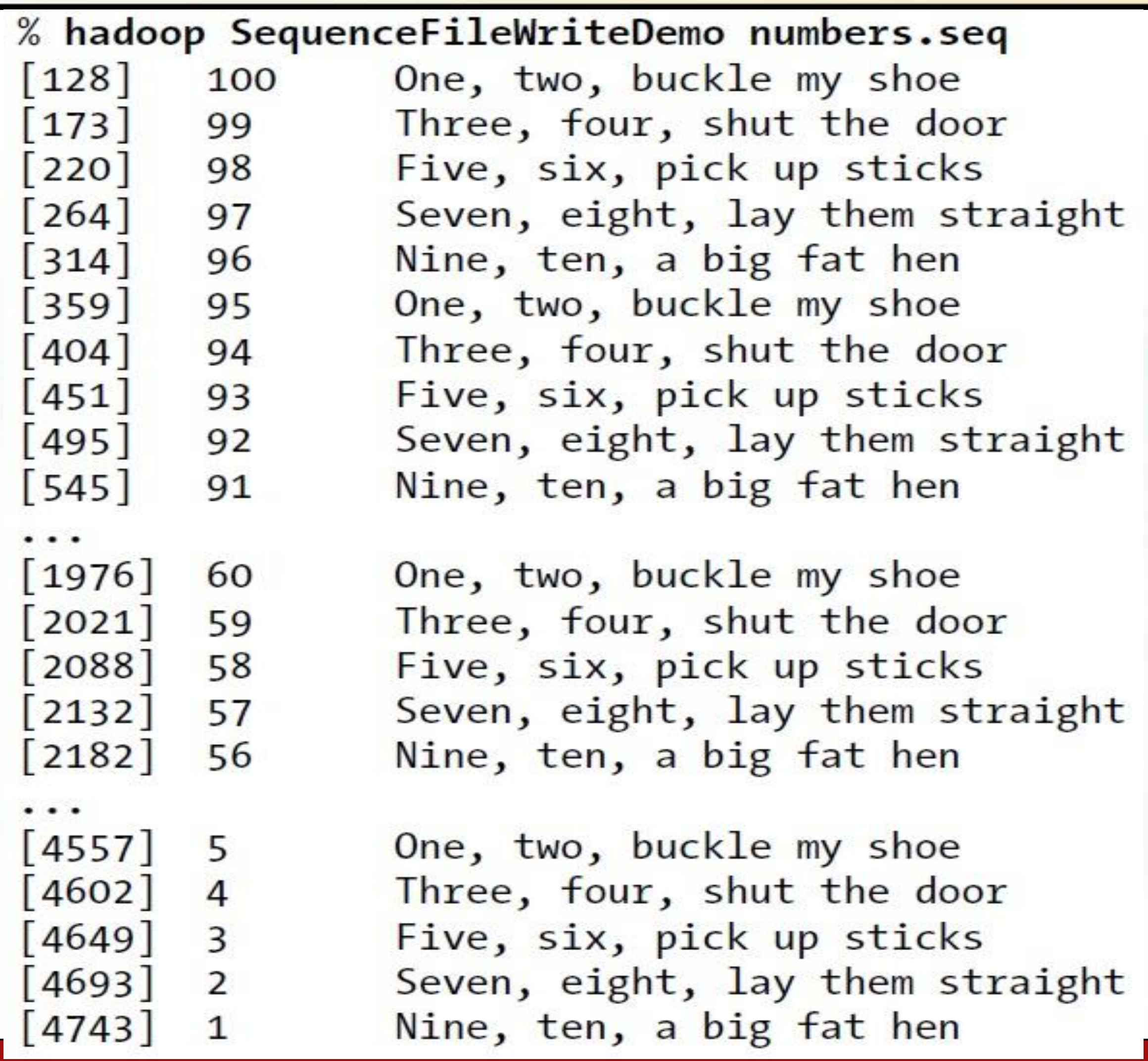


```java
public class SequenceFileWriteDemo {

  private static final String[] DATA = {
    "One, two, buckle my shoe",
    "Three, four, shut the door",
    "Five, six, pick up sticks",
    "Seven, eight, lay them straight",
    "Nine, ten, a big fat hen"
  };

  public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    Path path = new Path(uri);

    IntWritable key = new IntWritable();
    Text value = new Text();
    SequenceFile.Writer writer = null;
    try {
      writer = SequenceFile.createWriter(fs, conf, path,
          key.getClass(), value.getClass());

      for (int i = 0; i < 100; i++) {
        key.set(100 - i);
        value.set(DATA[i % DATA.length]);
        System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
        writer.append(key, value);
      }
    } finally {
      IOUtils.closeStream(writer);
    }
  }
}
```

Hadoop – The Definitive Guide

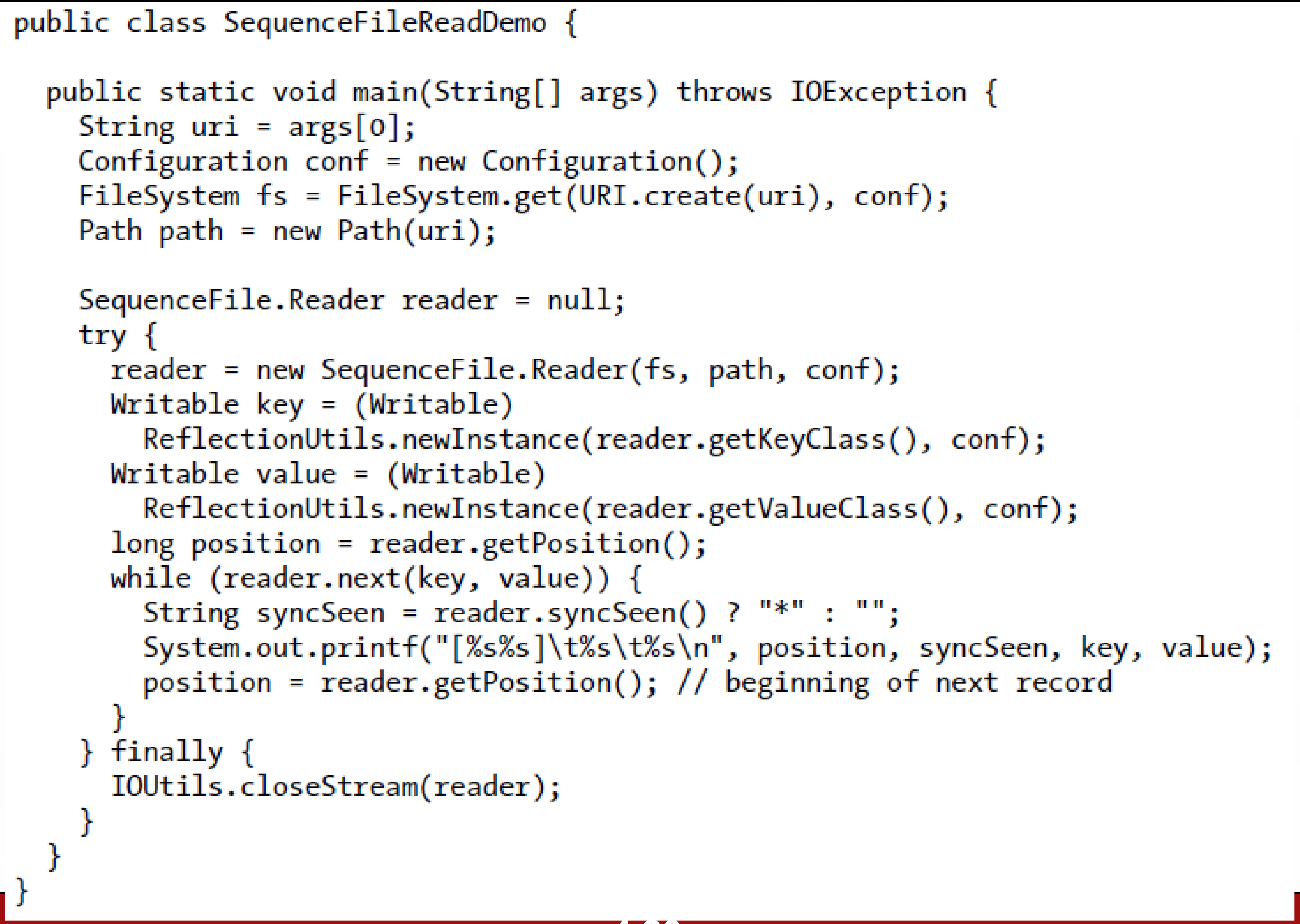
Chapter 4: Hadoop I/O

Writing a SequenceFile – Results



```
% hadoop SequenceFileWriteDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
...
[1976]  60      One, two, buckle my shoe
[2021]  59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

Reading a SequenceFile – p.133

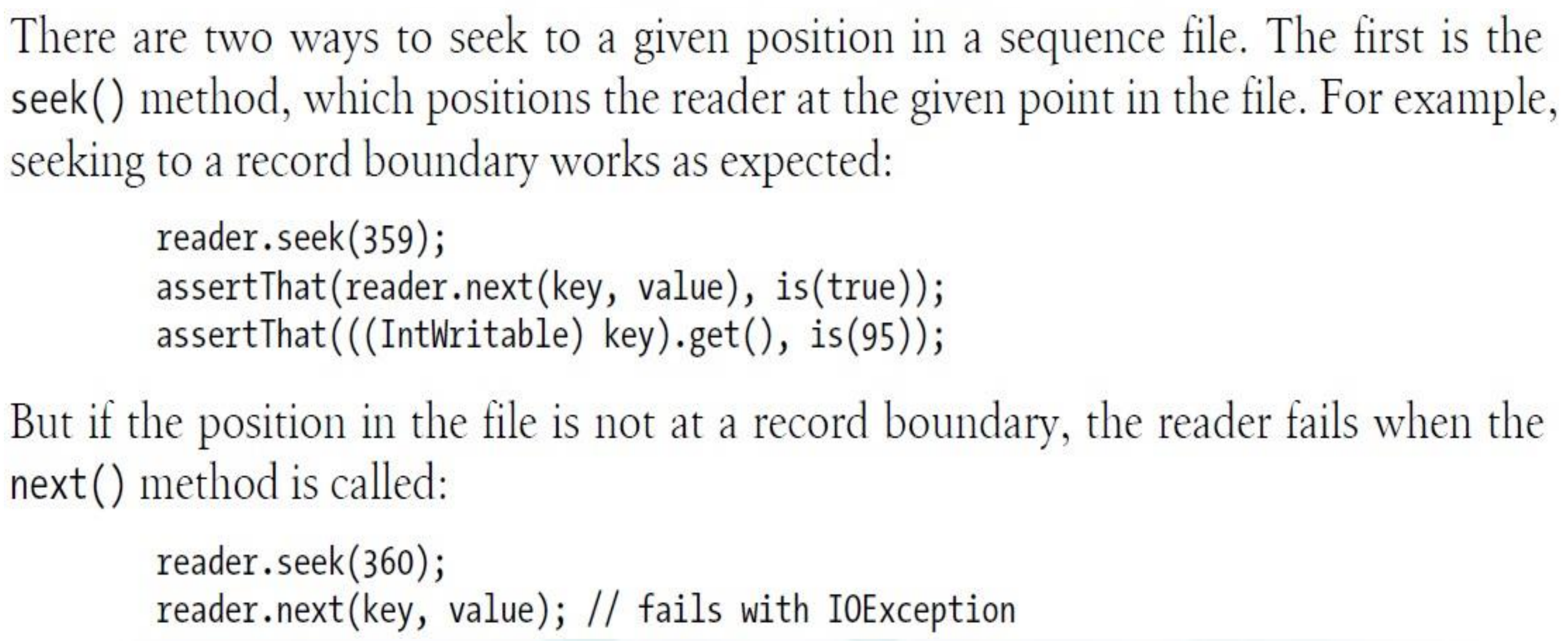


```
public class SequenceFileReadDemo {

  public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    Path path = new Path(uri);

    SequenceFile.Reader reader = null;
    try {
      reader = new SequenceFile.Reader(fs, path, conf);
      Writable key = (Writable)
        ReflectionUtils.newInstance(reader.getKeyClass(), conf);
      Writable value = (Writable)
        ReflectionUtils.newInstance(reader.getValueClass(), conf);
      long position = reader.getPosition();
      while (reader.next(key, value)) {
        String syncSeen = reader.syncSeen() ? "*" : "";
        System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
        position = reader.getPosition(); // beginning of next record
      }
    } finally {
      IOUtils.closeStream(reader);
    }
  }
}
```

Sync Point

- A sync point is a point in the stream that can be used to resynchronize with a record boundary if the reader is "lost"—for example, after seeking to an arbitrary position in the stream.
- Sync points are recorded by SequenceFile.Writer, which inserts a special entry to mark the sync point every few records as a sequence file is being written.
- Sync points always align with record boundaries.

Seek



There are two ways to seek to a given position in a sequence file. The first is the seek() method, which positions the reader at the given point in the file. For example, seeking to a record boundary works as expected:

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

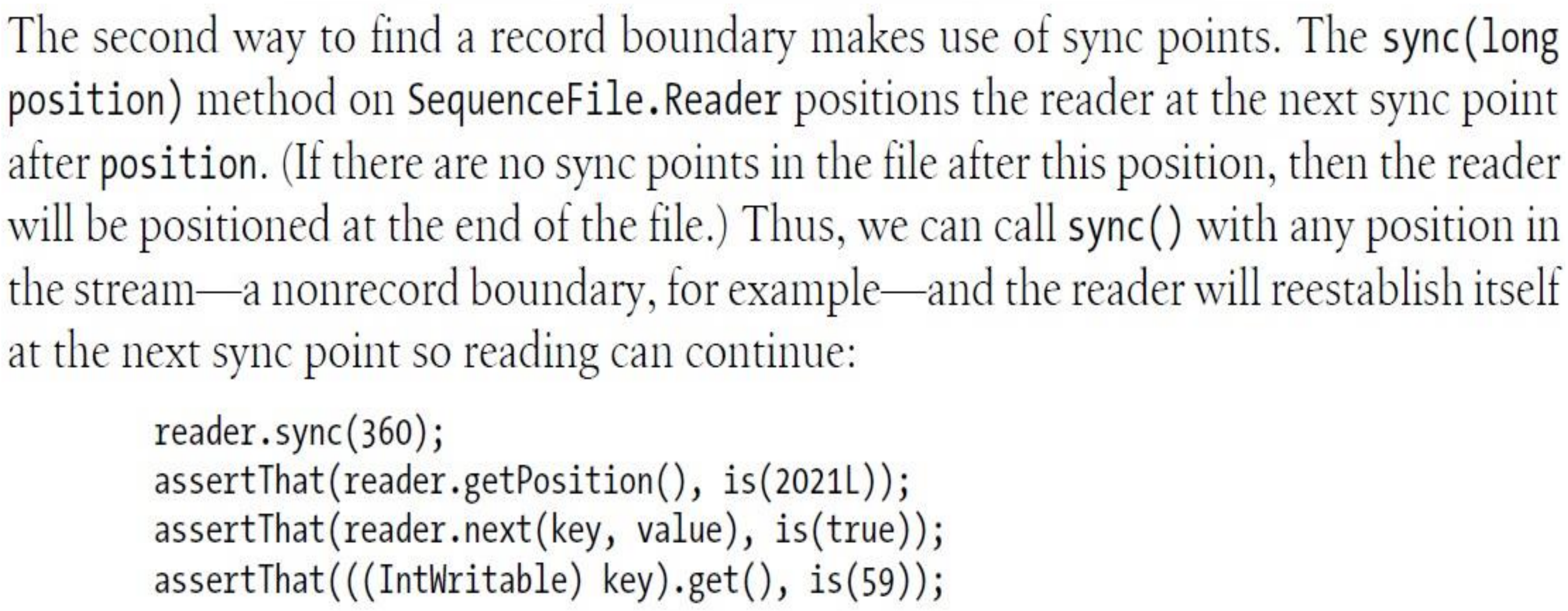
But if the position in the file is not at a record boundary, the reader fails when the next() method is called:

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

Hadoop – The Definitive Guide

Chapter 4: Hadoop I/O

Sync

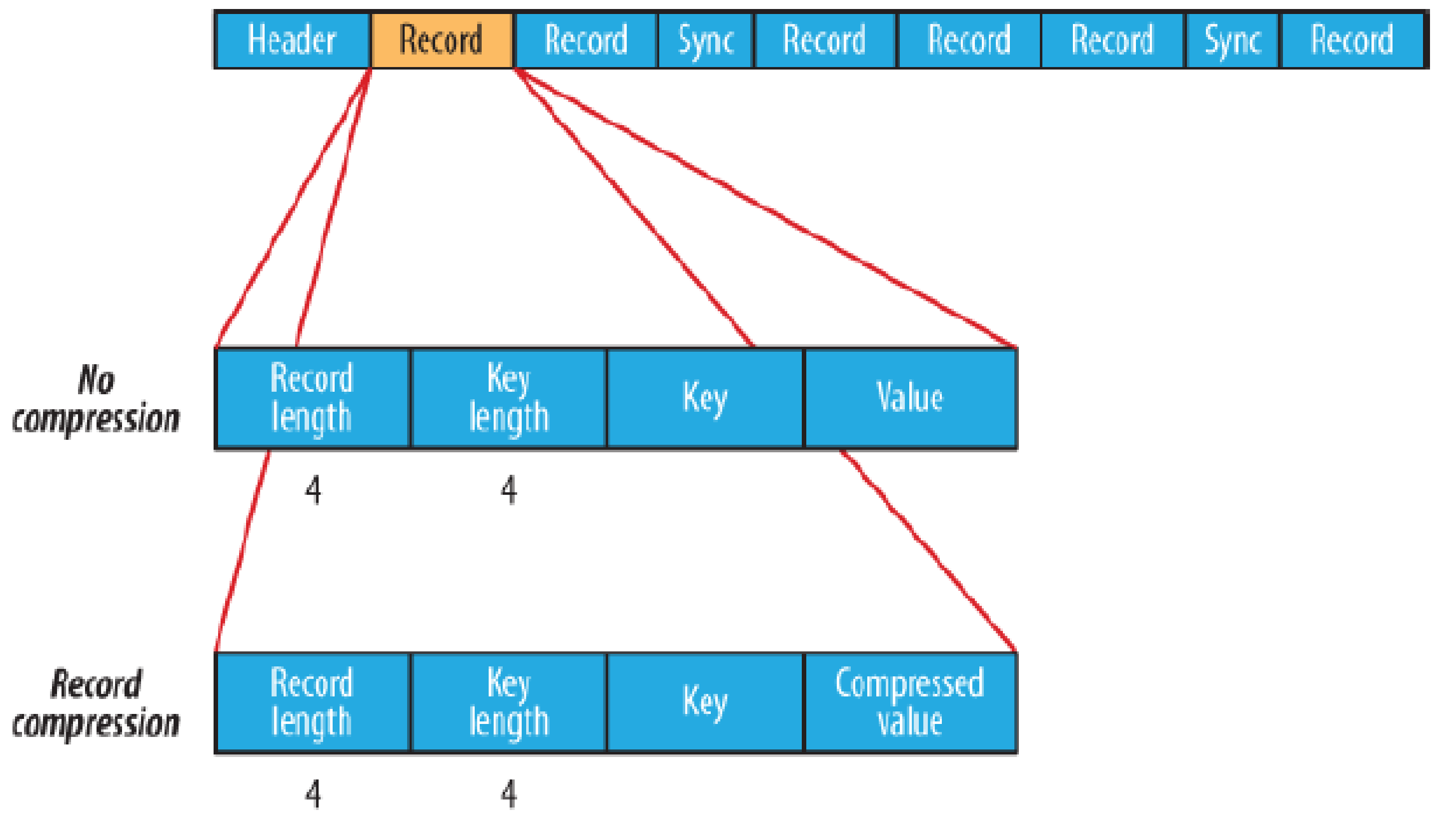


The second way to find a record boundary makes use of sync points. The `sync(long position)` method on `SequenceFile.Reader` positions the reader at the next sync point after `position`. (If there are no sync points in the file after this position, then the reader will be positioned at the end of the file.) Thus, we can call `sync()` with any position in the stream—a nonrecord boundary, for example—and the reader will reestablish itself at the next sync point so reading can continue:

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```

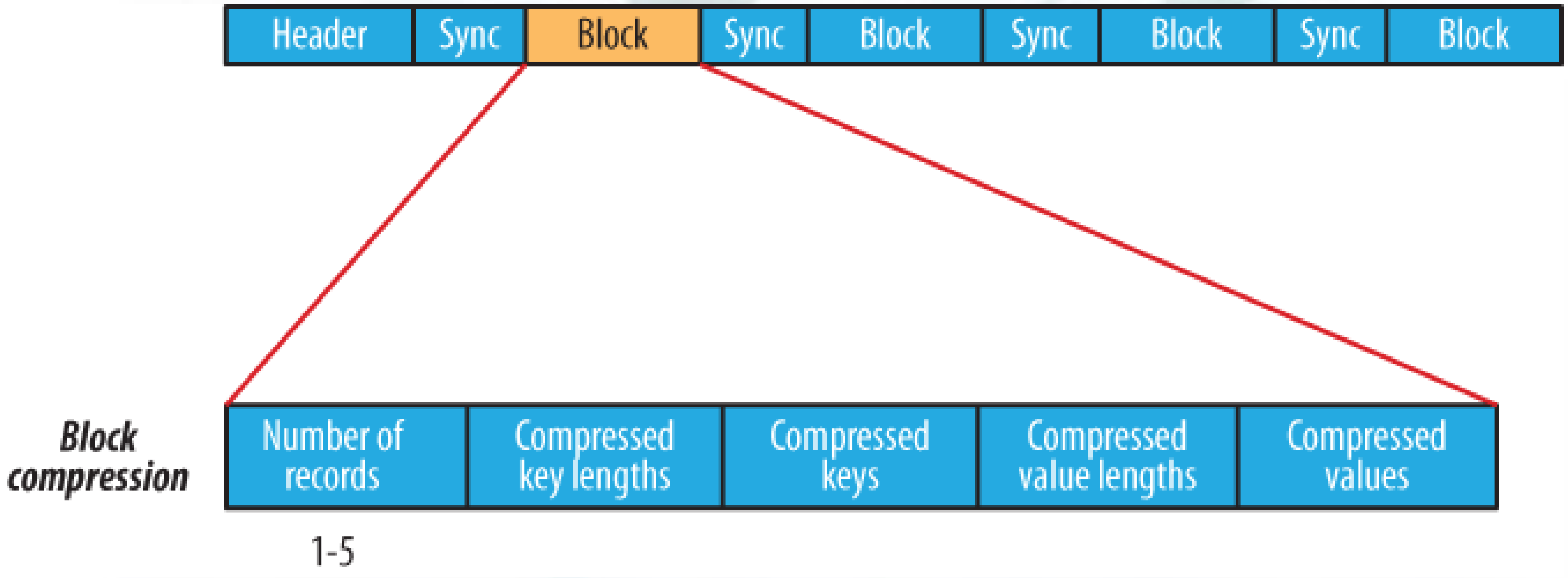
SequenceFile Format – 1

- Header contains the version number, the names of the key and value classes, compression details, user-defined metadata, and the sync marker.



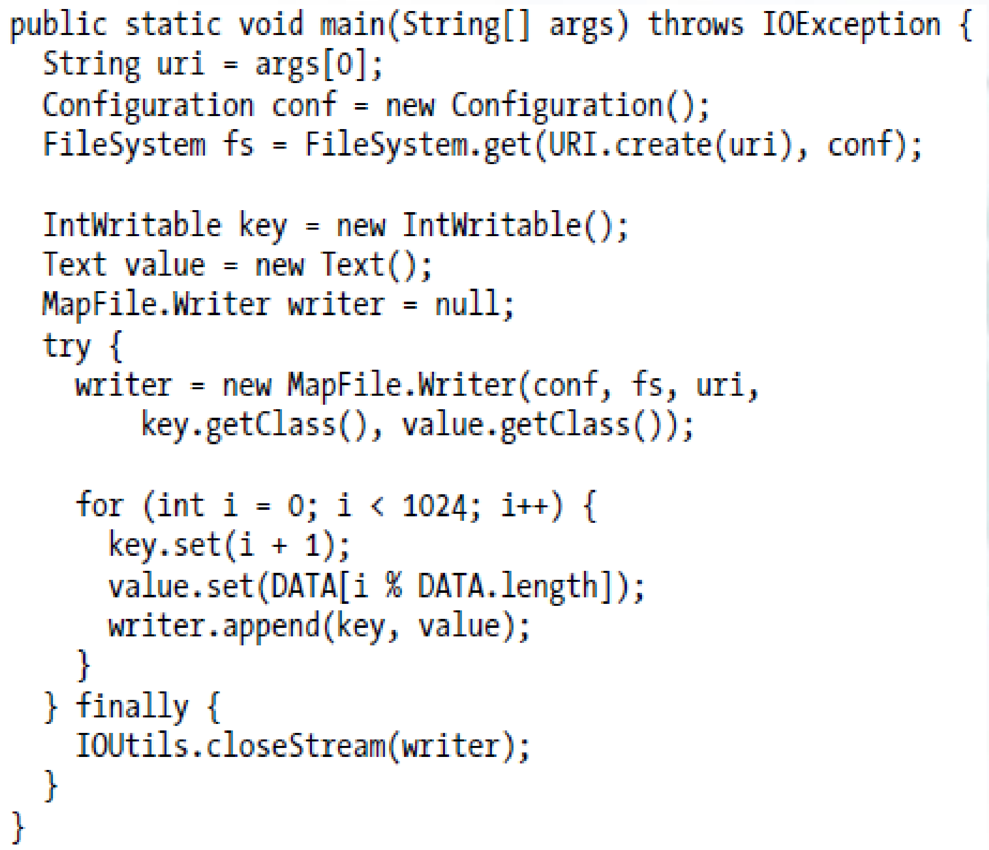
SequenceFile Format – 2

- Block compression compresses multiple records at once; it is therefore more compact.

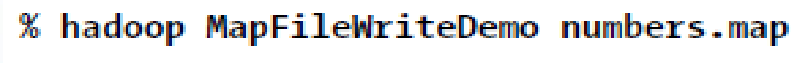


MapFile – p.137

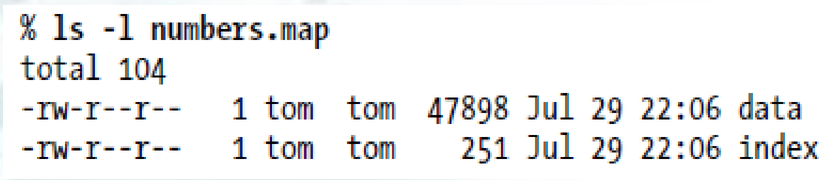
- A MapFile is a sorted SequenceFile with an index to permit lookups by key.
- Keys must be instances of WritableComparable and values must be Writable.



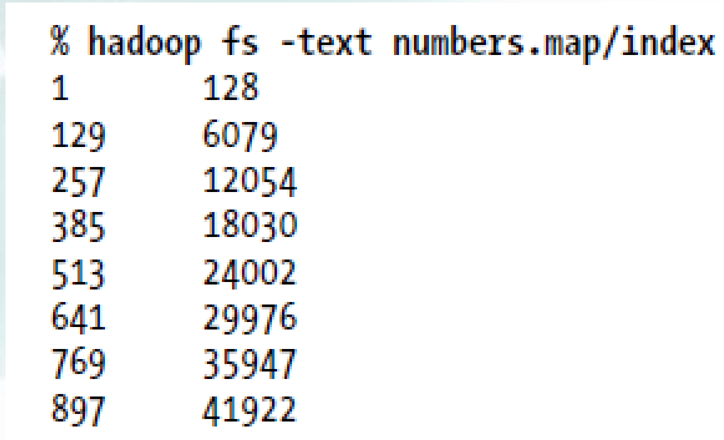
```
public static void main(String[] args) throws IOException {
  String uri = args[0];
  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(URI.create(uri), conf);

  IntWritable key = new IntWritable();
  Text value = new Text();
  MapFile.Writer writer = null;
  try {
    writer = new MapFile.Writer(conf, fs, uri,
        key.getClass(), value.getClass());

    for (int i = 0; i < 1024; i++) {
      key.set(i + 1);
      value.set(DATA[i % DATA.length]);
      writer.append(key, value);
    }
  } finally {
    IOUtils.closeStream(writer);
  }
}
```



```
% hadoop MapFileWriteDemo numbers.map
```



```
% ls -l numbers.map
total 104
-rw-r--r--   1 tom  tom  47898 Jul 29 22:06 data
-rw-r--r--   1 tom  tom    251 Jul 29 22:06 index
```



```
% hadoop fs -text numbers.map/index
1         128
129       6079
257       12054
385       18030
513       24002
641       29976
769       35947
897       41922
```