I/O utilities

`ByteStreams` and `CharStreams`

Guava uses the term "stream" to refer to a `Closeable` stream for I/O data which has positional state in the underlying resource. The term "`byte` stream" refers to an `InputStream` or `OutputStream`, while "`char` stream" refers to a `Reader` or `Writer` (though their supertypes `Readable` and `Appendable` are often used as method parameter types). Corresponding utilities are divided into the utility classes `ByteStreams` and `CharStreams`.

Most Guava stream-related utilities deal with an entire stream at a time and/or handle buffering themselves for efficiency. Also note that Guava methods that take a stream do *not* close the stream: closing streams is generally the responsibility of the code that opens the stream.

Some of the methods provided by these classes include:

ByteStreams	CharStreams
`byte[] toByteArray(InputStream)`	`String toString(Readable)`
N/A	`List<String> readLines(Readable)`
`long copy(InputStream, OutputStream)`	`long copy(Readable, Appendable)`
`void readFully(InputStream, byte[])`	N/A
`void skipFully(InputStream, long)`	`void skipFully(Reader, long)`
`OutputStream nullOutputStream()`	`Writer nullWriter()`

Sources and sinks

It's common to create I/O utility methods that help you to avoid dealing with streams at all when doing basic operations. For example, Guava has `Files.toByteArray(File)` and `Files.write(File, byte[])`. However, you end up with similar methods scattered all over, each dealing with a different kind of *source* of data or *sink* to which data can be written. For example, Guava has `Resources.toByteArray(URL)` which does the same thing as `Files.toByteArray(File)`, but using a `URL` as the source of data rather than a file.

To address this, Guava has a set of abstractions over different types of data sources and sinks. A source or sink is a resource of some sort that you know how to open a new stream to, such as a `File` or `URL`. Sources are readable, while sinks are writable. Additionally, sources and sinks are broken down according to whether you are dealing with `byte` or `char` data.

Operations	Bytes	Chars
Reading	`ByteSource`	`CharSource`
Writing	`ByteSink`	`CharSink`

The advantage of these APIs is that they provide a common set of operations. Once you've wrapped your data source as a `ByteSource`, for example, you get the same set of methods no matter what that source happens to be.

Creating sources and sinks

Guava provides a number of source and sink implementations:

Bytes	Chars
`Files.asByteSource(File)`	`Files.asCharSource(File, Charset)`
`Files.asByteSink(File, FileWriteMode...)`	`Files.asCharSink(File, Charset, FileWriteMode...)`
`MoreFiles.asByteSource(Path, OpenOption...)`	`MoreFiles.asCharSource(Path, Charset, OpenOption...)`
`MoreFiles.asByteSink(Path, OpenOption...)`	`MoreFiles.asCharSink(Path, Charset, OpenOption...)`
`Resources.asByteSource(URL)`	`Resources.asCharSource(URL, Charset)`
`ByteSource.wrap(byte[])`	`CharSource.wrap(CharSequence)`
`ByteSource.concat(ByteSource...)`	`CharSource.concat(CharSource...)`
`ByteSource.slice(long, long)`	N/A
`CharSource.asByteSource(Charset)`	`ByteSource.asCharSource(Charset)`
N/A	`ByteSink.asCharSink(Charset)`

In addition, you can extend the source and sink classes yourself to create new implementations.

Note: While it can be tempting to create a source or sink that wraps an *open* stream (such as an `InputStream`), this should be avoided. Your source/sink should instead open a new stream each time its `openStream()` method is called. This allows the source or sink to control the full lifecycle of that stream and allows it to be usable multiple times rather that becoming unusable the first time any method on it is called. Additionally, if you're opening the stream before creating the source or sink you may still have to deal with ensuring that the stream is closed correctly if an exception is thrown elsewhere in your code, which defeats many of the advantages of using a source or sink in the first place.

Using Sources and Sinks

Once you have a source or sink instance, you have access to a number of operations for reading or writing.

Common operations

All sources and sinks provide the ability to open a new stream for reading or writing. By default, other operations are all implemented by calling one of these methods to get a stream, doing something, and then ensuring that the stream is closed.

These methods are all named:

- `openStream()` - returns an `InputStream`, `OutputStream`, `Reader` or `Writer` depending on the type of source or sink.
- `openBufferedStream()` - returns an `InputStream`, `OutputStream`, `BufferedReader` or `Writer` depending on the type of source or sink. The returned stream is guaranteed to be buffered if necessary. For example, a source that reads from a byte array has no need for additional buffering in memory. This is why the methods do not return `BufferedInputStream` etc. except in the case of `BufferedReader`, because it defines the `readLine()` method.

Source operations

ByteSource	CharSource
byte[] read()	String read()
N/A	ImmutableList<String> readLines()
N/A	String readFirstLine()
long copyTo(ByteSink)	long copyTo(CharSink)
long copyTo(OutputStream)	long copyTo(Appendable)
Optional<Long> sizeIfKnown()	Optional<Long> lengthIfKnown()
long size()	long length()
boolean isEmpty()	boolean isEmpty()
boolean contentEquals(ByteSource)	N/A
HashCode hash(HashFunction)	N/A

Sink operations

ByteSink	CharSink
void write(byte[])	void write(CharSequence)
long writeFrom(InputStream)	long writeFrom(Readable)
N/A	void writeLines(Iterable<? extends CharSequence>)
N/A	void writeLines(Iterable<? extends CharSequence>, String)

Examples

```
// Read the lines of a UTF-8 text file
ImmutableList<String> lines = Files.asCharSource(file, Charsets.UTF_8)
    .readLines();

// Count distinct word occurrences in a file
Multiset<String> wordOccurrences = HashMultiset.create(
    Splitter.on(CharMatcher.whitespace())
        .trimResults()
        .omitEmptyStrings()
        .split(Files.asCharSource(file, Charsets.UTF_8).read()));

// SHA-1 a file
HashCode hash = Files.asByteSource(file).hash(Hashing.sha1());

// Copy the data from a URL to a file
Resources.asByteSource(url).copyTo(Files.asByteSink(file));
```

`Files`

In addition to methods for creating file sources and sinks, the `Files` class contains a number of convenience methods that you might be interested in.

Method	Description
`createParentDirs(File)`	Creates necessary but nonexistent parent directories of the file.
`getFileExtension(String)`	Gets the file extension of the file described by the path.
`getNameWithoutExtension(String)`	Gets the name of the file with its extension removed
`simplifyPath(String)`	Cleans up the path. Not always consistent with your filesystem; test carefully!
`fileTraverser()`	Returns a `Traverser` that can traverse file trees