# Closing Resources

Closing `Closeable` resources properly when finished with them is important for ensuring that socket connections are closed, file descriptors are not leaked, etc. as well as for ensuring correct program behavior. Pre-JDK7, it's extremely difficult to do properly, with many pitfalls that may not be immediately obvious. It's notoriously *almost never* done correctly, even within the JDK itself.

Here's an example of some typical code to open, use and close an `InputStream` in JDK6:

```
InputStream in = null;
try {
  in = openInputStream();
  // do something with in
} finally {
  if (in != null) {
    in.close();
  }
}
```

This code, while far more complicated than we'd like, is about as simple as you can get using standard JDK APIs. Unfortunately, even it has a problem: if an exception is thrown inside the `try` block and then an exception is thrown when calling `in.close()` in the `finally` block, the exception from the `try` block will be swallowed by the exception thrown in `finally` and you won't see the details of that error or even get any indication that it occurred at all!

It gets much, much worse if there are two streams that must be open at the same time:

```
InputStream in = null;
try {
  in = openInputStream();
  OutputStream out = null;
  try {
    out = openOutputStream();
    // do something with in and out
  } finally {
    if (out != null) {
      out.close();
    }
  }
} finally {
  if (in != null) {
    in.close();
  }
}
```

The nested `try` / `finally` blocks here are extremely unreadable, and the code still has the same issues with exceptions as above.

## Input vs. output streams

There's an important difference between input and output streams when it comes to closing resources.

With *input* stream, an error that occurs when attempting to close the stream is unlikely to be significant to your program: you're done with it, you've read everything you need.

With *output* streams, exceptions thrown when closing the stream must be treated as just as significant as exceptions thrown when actually using the stream. The reason for this is that output streams may buffer data that is written to them and need to flush buffered data to the underlying output sink when `close()` is called. In other words, the `close()` call may be triggering an actual write to a file or other sink, so an exception thrown from that call may indicate actual failure to complete the intended write operations.

# Solutions

## try-with-resources

*If you're using JDK7+*, this is easy: use try-with-resources. Here's an example of opening and closing multiple resources using try-with-resources:

```
try (InputStream in = openInputStream();
     OutputStream out = openOutputStream()) {
  // do stuff with in and out
}
```

Both streams will automatically be closed at the end of the `try` block, and if multiple exceptions are thrown, the first exception thrown will *suppress* the other two, and the stack traces of all the exceptions will be present in the single exception thrown from the `try` block. It's easy, it fixes all the problems, use it!

## Sources and Sinks

`ByteSource` and `CharSource` represent readable *sources* of binary and character data, respectively. `ByteSink` and `CharSink` represent writable *sinks* for binary and character data, respectively. All should be able to open multiple *independent* streams (e.g. `InputStream` or `Writer`) for reading from or writing to them.

When possible, creating an implementation of one of these types (or using a provided implementation such as `Files.asByteSource(File)`) allows you to circumvent the problem of opening and closing streams entirely by allowing you to use methods on those classes that handle opening and closing the resource for you. For example, you can copy the contents of a `File` to some `ByteSink` using the following:

```
ByteSink sink = ...
Files.asByteSource(file).copyTo(sink);
```

## Closer

`Closer` is a class added to Guava in release 14.0. It's intended as a poor man's try-with-resources block for code that must compile and run under JDK6. Here's an example of using it:

```
Closer closer = Closer.create();
try {
```

```
  InputStream in = closer.register(openInputStream());
  OutputStream out = closer.register(openOutputStream());
  // do stuff with in and out
} catch (Throwable e) { // must catch Throwable
  throw closer.rethrow(e);
} finally {
  closer.close();
}
```

Calling `close()` on the `Closer` will safely close all `Closeable` objects that have been registered with it. When *running* under Java 7, it even uses the same mechanism for suppressing additional exceptions that try-with-resources uses, producing behavior that should be identical. Do note that when using `Closer`, it's very important to follow the prescribed usage pattern: any `Throwable` thrown from the block where the `Closeable` resources are used must be caught and rethrown through `Closer`'s `rethrow` method before calling `closer.close()`. Also note that if you wish to catch any exception thrown from this whole block of code, you should wrap the entire thing in *another* `try` block.