1. Test and Benchmark FileCopy

The starter code for `FileCopy.java` copies a source file to a destination file.

Usage: `java CopyFile` *`sourcefile`* *`destinationfile`*

Copies sourcefile to destinationfile. The source file must already exist and the destination file must not exist.

1.1 **Test** this code and verify it works correctly. Things you should test are:

- it makes an exact copy of the file, whether file contains text or binary data (e.g. MP3 or image file)
- if wrong number of arguments on the command line, it prints a message and exits.
- if source file does not exist or is not a plain file (e.g. directory), it prints a message and exits.
- if destination file *already* exists, it prints a message and exits (does not overwrite the file)

1.2 **Benchmark** the code and write the results in `README.md`. In `README.md` create a table including a description of the file , size in bytes, and time needed to copy it.

Benchmark at least 2 cases (OK to add others if you like):

a) time to copy a small text file, approximately 1 KB (larger is OK)

b) time to copy an MP3, JPEG, or PNG file of size several megabytes

Hint: you can modify the main method to use your Stopwatch before/after calling copyfile() to get the elapsed time. On Linux (and maybe MacOS) the builtin `time` command also does this.

2. Modify CopyFile to Read Using an Array.

2.1 Modify the `copyfile( )` method so that instead of copying the file one byte at a time, it uses an **array of byte**. See the slides on "Input-Output" for how to.
Use an array size that is a multiple of 1024 (1KB), e.g. 1024, 4096, etc., as explained in the slides.
Optional: If you are interested in performance then try a few sizes and see what works best.

2.2 Repeat the benchmarks in Problem 1.2 and add the results to `README.md`

2.3 In README.md also record what array size (called a "buffer size") you use in `copyfile`.

3. Use Try-with-Resources

The code for the try - catch - finally block in copyfile is a bit complex.

```
// Must define InputStream (in) and OutputStream (out) here so that
// their "scope" includes the finally block
InputStream in = null;
OutputStream out = null;
try {
   in = new FileInputStream(sourcefile);
   out = new FileOutputStream(destfile);
   int c = 0;
   while( (c=in.read()) >= 0) out.write(c);
}
catch (FileNotFoundException fne) {
   error( fne.getMessage() );
}
catch (IOException ioe) {
   error( ioe.getMessage() );
}
finally {
   // Close the input and output files.
   // If you use "try with resources" (problem 3) this is not needed.
   try { in.close(); } catch (Exception ex) { /* nothing to do */ }
   try { out.close(); } catch (Exception ex) { /* nothing to do */ }
}
```

Modify the **try - catch** block to use "try with resources". The "resourses" are automatically closed when the `try` block exits, so you don't need a `finally` block to close them yourself.

```
try ( create objects used in try block ) // create "resources"
{
    // body of the try block
}
catch (SomeException ex) {
     ...
}
```

See this article for how to use Try-With-Resources:

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Modify the code so that you define and open the FileInputStream and FileOutputStream inside the parenthesis after the "try" keyword. Separate the statements with semi-colon (;) but no semi-colon after the last statement.

The syntax is like this:

```
// Must define InputStream and OutputStream here so that their
// "scope" includes the finally block

try (
    // Create input and output streams here.
    // They are automatically closed when the try block exits.
     InputStream in = ...;
     OutputStream out = ... )
{
   int c = 0;
   while( (c=in.read()) >= 0) out.write(c); //TODO Problem 2- use array
}
catch (FileNotFoundException fne) {
   error( fne.getMessage() );
}
catch (IOException ioe) {
   error( ioe.getMessage() );
}
// no finally block -- resources are closed automatically
```

4. Write a Compare Files Program

Write a runnable Java class named Compare that compares two files. This is like the Linux `cmp` command.

Usage: `java Compare file1 file2`

a. If the two files are the same, print "Same" and exit with exit code 0.

b. If the files differ at some point, print "Differ at byte nnn" and exit with code 1.

c. If one file is longer than the other, but *all* the shorter file matches the start of the longer file, print "EOF *shorterfile* after byte nnn" (print name of shorter file and index of first byte that didn't exist) and exit with code 1. "EOF" means "End Of File".

d. If wrong number of command line arguments, print a usage message and exit with exit code 0.

e. If either file does not exist or is not a readable ordinary file, print a message stating the problem and exit with code 1.

f. Always catch and handle exceptions. Program should never print a stack trace.

5. Word Count Application

Usage: `java WordCount file [ file2 ... ]`

For each file on the command line, print the number of lines, words, and characters (not bytes!) in the file. Print them together with the filename (one per line).

A "word" is any sequence of non-whitespace characters, delimited by whitespace or end-of-line.

The character count should include the line-end characters (newline) in the file.

The file (or files) must be a text format (not binary data).

Example: `Alice-in-Wonderland.txt` contains 1207 lines, 9660 words, 52,540 characters.

```
java WordCount Alice-in-Wonderland.txt
  1207  9660 52540 Alice-in-Wonderland.txt
```

Hint: this is easy to do if you exmime each character in sequence and keep track of the current "state" of the input. The "states" are:

whitespace - previous character read was whitespace or newline. This is also the initial state.

not-space - previous character read was not whitespace or newline.

Since there are only 2 states you can use a boolean variable (`isspace`) to keep track.

For each character, check if it is a newline, whitespace, or something else (not whitespace) and update the state (isspace) and the counts of characters and lines.

Whenever the input changes state from the "whitespace" state to "not space", count one word. When you see whitespace or newline, switch to "whitespace" state.

Using this approach, you never need to look ahead or look back at a previous character. The "state" variable and the current character in the input contain all the information you need.

Java API Reference

Please see the Java API docs for details. You should install the Java API docs on your own computer for quick access.

InputStream

`int read(byte[] b)` - read InputStream data into an array of bytes. The return value is the number of bytes actually read, which may be less than the array size. Returns -1 at end of input.

OutputStream

`void write(byte[] b, int startindex, int length)` - write bytes from array to the output stream, starting at index `startindex`, and wrting `length` bytes.