HOMEWORK 3

COMMAND LINE. DEBUGGING. ERROR HANDLING / FILE SYSTEM AND STREAMS

Tasks

1. Create directory **utils**. Create util module called `streams.js` inside this directory.
2. This util should be able to work with command line following the next requirements:
    a. Should consist of functions which will be run as actions.
    b. Should receive an action name as a first argument by `--action` option.
    c. Should receive an optional second argument for actions which may require it by `--file` option.
    d. Should process `--help` key. If this option is passed as a first argument, print usage message and ignore other options. Ignore this option if other options were passed before.
    e. Should support shortcuts for options as well (`-a` for `--action`, `-f` for `--file` and `-h` for `--help` respectively). Please note that util should work correctly with any option provided regardless its form (full or shortcuted).

Example:

```
// === streams.js ===


// Main actions to be called

function reverse(str) { /* ... */ }
function transform(str) { /* ... */ }
function outputFile(filePath) { /* ... */ }
function convertFromFile(filePath) { /* ... */ }
function convertToFile(filePath) { /* ... */ }


/*
 *
 * **** CODE WHICH IMPLEMENTS COMMAND LINE INTERACTION ****
 *
 */
```

```
// === Terminal ===

./streams.js --action=outputFile --file=users.csv
./streams.js --action=transformToFile --file=users.csv
./streams.js --action=transform textToTransform
./streams.js -a outputFile -f users.csv
./streams.js --help
./streams.js -h
```

3. If module is called without arguments, notify user about wrong input and print a usage message (equal to calling with `--help` option).
4. Appropriate action passed by `--action` option should be called. If action requires additional argument, it should be called with that argument provided with `--file` option.
5. If `streams.js` util does not contain an action passed or received argument is invalid, appropriate error message should be shown to user. Additionally, util may throw relevant exception.
6. Any number of action functions inside `streams.js` could be implemented but the following ones are mandatory for realization:
    a. `reverse` function to reverse string data from **process.stdin** to **process.stdout**.
    b. `transform` function to convert data from **process.stdin** to upper-cased data on **process.stdout** (e.g. using `through2` module).
    c. `outputFile` function that will use `fs.createReadStream()` to pipe the given file provided by `--file` option to **process.stdout**.
    d. `convertFromFile` function to convert file provided by `--file` option from `csv` to `json` and output data to **process.stdout**. Function should check that the passed file name is valid (*see task 5*).
    e. `convertToFile` function to convert file provided by `--file` option from `csv` to `json` and output data to a result file with the same name but `json` extension. Function should check that the passed file name is valid (*see task 5*) and use `fs.createWriteStream` additionally.
7. Implement `cssBundler` action function which will use an extra parameter `--path` (`-p` as a shortcut). It should do the following:
    a. Grab all css files from the given path provided by `--path` option.
    b. Concat them into one (big) css file.
    c. Add contents of https://epa.ms/nodejs18-hw3-css to the end of the result file.
    d. Save the output in the file called **bundle.css** placed in the same provided path.

Example:

```
./streams.js --action=cssBundler --path=./assets/css
./streams.js --action=cssBundler -p ./assets/css
```

Evaluation criteria:

1. **utils** directory and empty `streams.js` file were created.
2. `streams.js` util is able to read command line and output help usage.
3. Util meets all requirements for command line interaction. Most of mandatory actions are implemented and called when appropriate arguments are passed.
4. All required actions are implemented from *task 6*. Some error handlings are implemented, util validates some of parameters passed to it.
5. All actions are implemented including an extra one from *task 7*. Util handles all possible error cases and validates all required parameters.

Transform flow hint:

```
Transform stream takes input data and applies an operation to the data to
produce the output data.

Create a through stream with `write` and `end` functions:

    const through = require('through2');
    const stream = through(write, end);

The `write` function is called for every buffer of available input:

    function write (buffer, encoding, next) {
        // ...
    }

and the `end` function is called when there is no more data:

    function end () {
        // ...
    }

Inside the write function, call `this.push()` to produce output data and
call `next()` when you're ready to receive the next chunk:

    function write (buffer, encoding, next) {
        this.push('I got some data: ' + buffer + '\n');
        next();
    }

and call `done()` to finish the output:

    function end (done) {
        done();
    }

`write` and `end` are both optional.

If `write` is not specified, the default implementation passes the input
data to the output unmodified.
```

```
If `end` is not specified, the default implementation calls
`this.push(null)` to close the output side when the input side ends.

Make sure to pipe `process.stdin` into your transform stream
and pipe your transform stream into `process.stdout`:

    process.stdin.pipe(tr).pipe(process.stdout);

To convert a buffer to a string call `buffer.toString()`.
```