4.13.2 2. Log Rotation

Log rotation is a critical strategy in file-based logging to manage log files over time. As your application generates more log data, log files can become large and unwieldy. Log rotation involves creating new log files periodically or based on certain conditions, and optionally archiving or deleting older log files. This helps keep the log files at a manageable size, ensures easier log analysis, and prevents running out of disk space.

4.13.3 3. Asynchronous Logging

In network based apps, especially those with high levels of concurrency or those that involve asynchronous operations, implementing asynchronous logging can be beneficial, and that's the only reason we're using `node:fs/promises` instead of `node:fs`. Asynchronous logging ensures that the act of writing logs doesn't block or slow down the main execution thread of your application. This can prevent performance bottlenecks and maintain the responsiveness of your application.

When logging is performed synchronously, each log entry is written immediately to the log file or output, which can introduce delays and impact the overall performance of your application. Asynchronous logging, on the other hand, involves buffering log messages and writing them to the log file in batches or on a separate thread or process.

> We'll need to do more optimization than just using asynchronous file writing. Specifically, we should store the entire log contents in memory and write them periodically. This will make the process extremely fast and ensure that it doesn't consume too much memory.

By decoupling the logging process from the main application logic, we can achieve several advantages:

- **Improved Performance:** Asynchronous logging allows the main application thread to continue executing without waiting for log writes to complete. This can be crucial for applications that require high responsiveness and throughput.
- **Reduced I/O Overhead:** Writing log messages to disk can be an I/O-intensive operation. By batching multiple log messages together and writing them in one go, you reduce the frequency of disk I/O operations, which can improve efficiency.
- **Better Resource Utilization:** Asynchronous logging allows you to optimize resource utilization, such as managing concurrent log writes, handling errors without disrupting the main flow, and efficiently managing file handles.
- **Enhanced Scalability:** Applications with multiple threads or processes benefit from asynchronous logging because it minimizes contention for resources like the log file. This is particularly valuable in

scenarios where multiple components are concurrently generating log messages

4.13.4 4. Getting Caller Information (Module and Line Number)

Including caller information, such as the file name and line number from which a log message originated, can significantly enhance the effectiveness of our logging library. This feature provides contextual insight into where specific events occurred in the codebase, making it easier to identify the source of issues and troubleshoot them.

When an application encounters an error or unexpected behavior, having access to the module and line number associated with the log message allows developers to:

- Quickly locate the exact location in the code where the event occurred.
- Understand the sequence of events leading up to the issue.
- Make precise code adjustments to fix problems.

Implementing this feature might involve using techniques from the programming language's stack trace or introspection mechanisms. Here's a high-level overview of how you could achieve this:

1. **Capture Caller Information:** When a log message is generated, our logging library will retrieve the caller information, including the module and line number. We'll learn it in a bit that how can we do that.
2. **Format the Log Message:** Combine the captured caller information with the log message and other relevant details like timestamp and log level.
3. **Output the Log Message:** Write the formatted log message to the desired output destinations, ensuring that the caller information is included.

Enough of the theory, let's start writing logs.

4.13.5 Testing our current API

Let's start with building small features that we need to have on our `Logger` class. Before that, we're going to do some testing on whatever we've built till now.

To do the testing, we'll create a new file `test.js` and a `config.json`. `test.js` will hold our code that we write while making use of our `logtar` library. The `config.json` will used to store our config for `LogConfig` as a `json` object.

```
// test.js

const { Logger, LogConfig } = require("./index");
```

And in test.js call these methods

```
// file: test.js
const { Logger, LogConfig } = require('./index')

const logger = Logger.with_config(LogConfig.from_file('./config.json'));

console.log(logger.file_prefix);
console.log(logger.size_threshold);
console.log(logger.time_threshold);
console.log(logger.level);

logger.debug('Hello debug');
logger.info('Hello info');
logger.warn('Hello warning');
logger.error('Hello error');
logger.critical('Hello critical');


// outputs
LogTar_
1024000
86400
3
Debug: Hello debug
Info: Hello info
Warn: Hello warning
Error: Hello error
Critical: Hello critical
```

4.13.7 DRY (Don't Repeat Yourself)

The ”Don't Repeat Yourself” (DRY) principle is a basic concept in software development that promotes code reusability and maintainability. The idea behind DRY is to avoid duplicating code or logic in multiple places within your codebase. Instead, you aim to create a single source for a particular piece of functionality, and whenever you need that functionality, you refer to that source.

DRY encourages developers to write clean, efficient, and modular code by:

```
            return levelMap[log_level];
        }

        throw new Error(`Unsupported log level ${log_level}`);
    }
    ...
}
```

Change the code inside the `log()` method of `Logger` class.

```
// lib/logger.js

const { LogLevel } = require("./utils/log-level");

class Logger {
    ...

    #log(message, log_level) {
        console.log('%s: %s', message, LogLevel.to_string(log_level))
    }
    ...
}
```

This outputs

```
Hello debug: DEBUG
Hello info: INFO
Hello warning: WARN
Hello error: ERROR
Hello critical: CRITICAL
```

Everything looks good.

4.13.9 Considering the `log_level` member variable

Notice that in our `config.json` we specified that the log level should be 3 that is `LogLevel.Error`. Specifying the log level means that we should only write logs that are equal to or above the specified level.

Imagine a production application, which is usually under a very heavy load. We'd like to specify the `level` as `LogLevel.Warn` or even `LogLevel.Info`. We don't care about the `LogLevel.Debug` logs at all. They might

4.13.10.1 A small primer on Regular Expressions

1. The `init` method is responsible for initializing the logger, which includes creating or opening a log file with a dynamically generated name based on the current date and time.
2. `this.#config.file_prefix` is used to prefix the name of the log file, which is actually set in the `config.json` file or can be passed as a json object or utilizing the `Builder` pattern.
3. `new Date().toISOString()` generates a string representation of the current date and time in the ISO 8601 format, such as "2023-08-18T15:30:00.000Z".
4. `.replace(/[\.:]+/, "")` is a regular expression operation. Let's break down the regex:
   - the square brackets `[]` are used to define a character class. A character class is a way to specify a set of characters from which a single character can match. For example:
     - [abc] matches any single character that is either 'a', 'b', or 'c'.
     - [0-9] matches any single digit from 0 to 9.
     - [a-zA-Z] matches any single alphabetical character, regardless of case.

     You can also use special characters within character classes to match certain character types, like `\d` for digits, `\w` for word characters, `\s` for whitespace, etc.

     In this case we are looking for all the dot (`.`) characters and colon (`:`) characters in the string.
   - `\.` matches a literal dot character. Since the dot (`.`) is a special character in regular expression, known as a wildcard. It matches any single character except for a newline character (`\n`). This is useful when you want to match any character, which is often used in pattern matching.

     However, in this case, we want to match a literal dot character in the string (the dot in the date-time format "00.000Z"). To achieve this, we need to escape the dot character by preceding it with a backslash (\). When you place a backslash before a special character, it indicates that you want to match the literal character itself, rather than its special meaning.
   - `+` matches one or more of any character except newline. We match for all the characters following the dot (.) and the (:) character.
   - `/g` is the global flag, indicating that the replacement should be applied to all occurrences of the pattern.
   - So, the regex `[\.:]+` matches the dot or colon and all the repeated occurences of these 2 characters.
   - The replacement "" removes the dot and all characters after it.
5. The result of the `replace` operation is a modified version of the ISO string, which now includes only the date and time portion, without the fractional seconds.

The name of the log file is

`LogTar_2023-08-18T17:20:23.log`

`LogTar_` is the prefix that I specified in the `config.json` file. Following it is the timestamp of when the file was created. We also add a `.log` extension at the end. All working as expected.

4.13.11 Another gotcha

If you're paying attention, you'll have already figured out that the way we're providing path to a file is not a nice way. If we wanted to run the `test.js` file from a different directory, say `lib/config` we'll get an error.

```
$ cd lib/config && node ../../test.js

// outputs
Error: ENOENT: no such file or directory, open './config.json'
```

Let's fix this by using the `__dirname` global variable that we used earlier.

```
// file: test.js

const path = require("node:path");
const { Logger, LogConfig } = require("./index");

async function main() {
    const logger = Logger.with_config(LogConfig.from_file(path.join(__dirname, "config.json")));
    await logger.init();
    console.log("End of the file");
}

main();
```

Now if we try to run it, it works irrespective of the directory you're running the code from.

```
$ node ../../test.js
### Outputs
Finished creating a file and opening
End of the file
```

```
console.log(my_fun.hey); // there
my_fun(); // 'hi'
```

4.13.14 Adding a new helper to create log directory

Create a new file called `helpers.js` inside the `lib/utils` folder.

```
// file: lib/utils/helpers.js

const fs_sync = require("node:fs");
const path = require("path");

/**
 * @returns {fs_sync.PathLike} The path to the directory.
 */
function check_and_create_dir(path_to_dir) {
    const log_dir = path.resolve(require.main.path, path_to_dir);
    if (!fs_sync.existsSync(log_dir)) {
        fs_sync.mkdirSync(log_dir, { recursive: true });
    }

    return log_dir;
}

module.exports = {
    check_and_create_dir,
};
```

Let's go through the `check_and_create_dir` function's code line by line.

1. The `path.resolve()` function creates an absolute path by combining a sequence of paths or path segments.

   It processes the sequence from right to left, with each subsequent path added to the beginning until an absolute path is created. For example, if the path segments are `/foo`, `/bar`, and `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz` because `'/bar' + '/' + 'baz'` creates an absolute path, but `'baz'` does not.

   If, after processing all the segments in the given path, an absolute path hasn't been created yet, then the current working directory is used instead.

```
    return logger;
}
async function main() {
    let logger = await initialize_logger();
    logger.error("This is an error message");
}

main();
```

In `test.js` we create a new async function `initialize_logger` that creates a logger, initializes it and returns it.

We call the `logger.error()` method to print the log to the file.

Let's run the code

```
$ npm start
```

A new log file will be created inside the `logs` directory. Open and see the contents -

```
This is an error message
```

Perfect! Everything seems to be working now. The logs are being saved, and we can be proud of it! But wait, the logs aren't useful. We don't know what are those logs, when did we write them and what function called the `logger.error()` method?

We'll take care of all this in the next chapter.

> The code for the entire chapter can be found at the code/chapter_04.2 directory

4.14 Capturing metadata

> The code for the entire chapter can be found at the code/chapter_04.3 directory

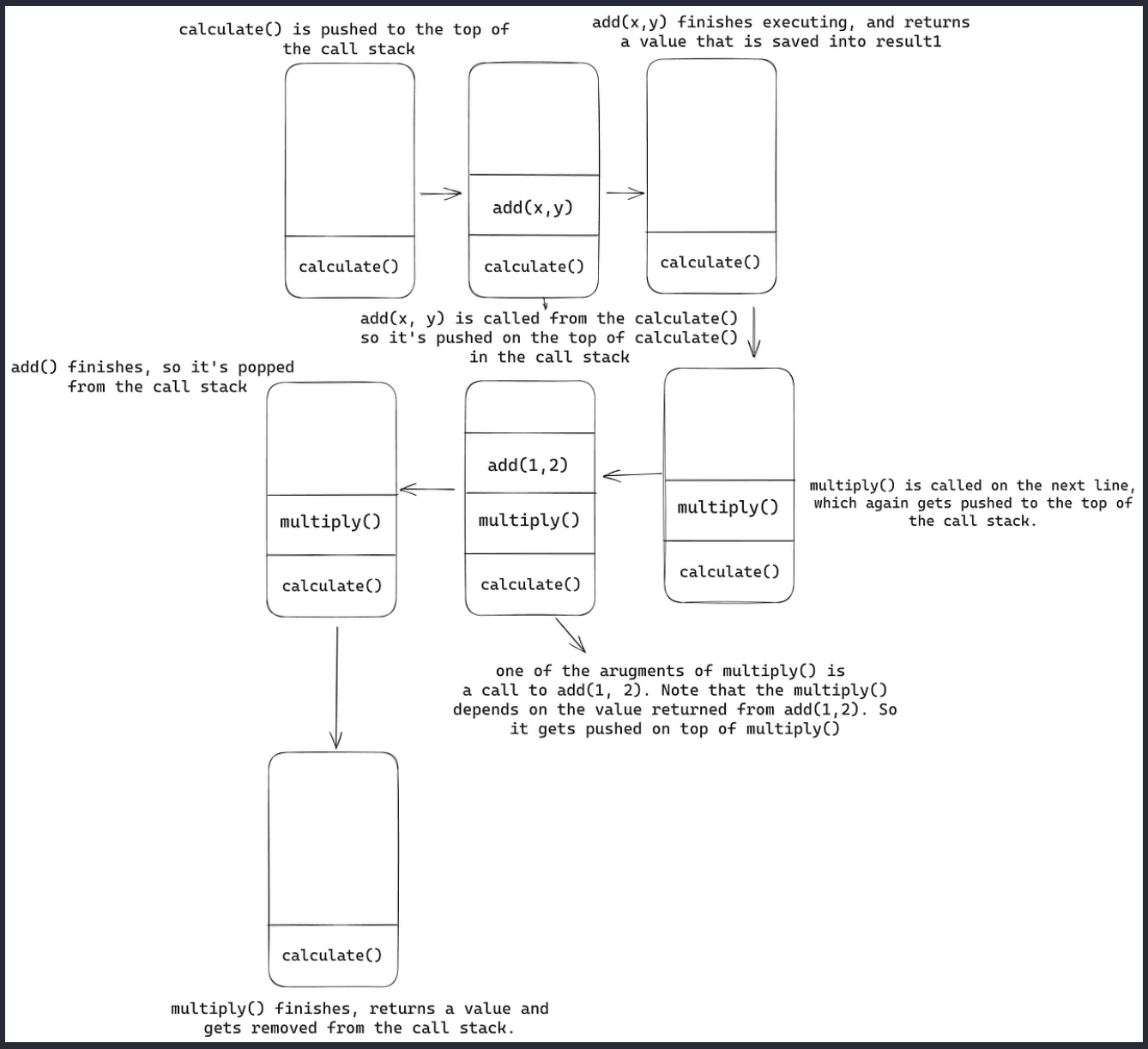
The Logger is writing to the file as expected, but there is an issue: it is not capturing any useful metadata. Here is an example of what our logs currently look like:

```
Hello this is a log of type ERROR
```

```
calculate();
```

In the example provided, the `calculate` function calls two other functions (`add` and `multiply`). Each time a function is called, its information is added to the call stack. When a function returns, its information is removed from the stack.

To further illustrate this, consider the following graphic:



When the `calculate` function is called, its information is added to the top of the stack. The function then calls the `add` function, which adds its information to the top of the stack. The `add` function then returns, and its information is removed from the stack. The `multiply` function is then called, and its information is added to the top of the stack.

One important thing to note is, when multiply is called, the first argument is a call to `add(1, 2)`. That

We only need to care about the top 5 lines. Rest of those are Node.js's internal mechanism of running and executing the files. Let's go through the error above line by line:

- The first line includes the error message "`This is an error from grand_child()`", which is the custom message we provided when we used the `throw` statement in the `grand_child()` function.
- The second line of the stack trace indicates that the error originated in the `grand_child()` function. The error occurred at line 10, column 11 of the `test.js` file. This is where the `throw` statement is located within the `grand_child()` function. Therefore, `grand_child` was at the top of the stack when the error was encountered.
- The third line shows that the error occurred at line 6, column 5 of the `test.js` file. This line pinpoints where the `grand_child()` function is called within the `child()` function. That means, `child()` was the second top function on the call stack, below `grand_child()`.
- The fourth lines tells us that the `child()` function was called from within the `main()` function. The error occurred at line 2, column 5 of the `test.js` file.
- The line 5th tells that the `main()` function was called from the top-level of the script. This anonymous part of the trace indicates where the script execution starts. The error occurred at line 12, column 1 of the `test.js` file. This is where the `main()` function is called directly from the script.

This is called a stack trace. The throw new Error() statement prints the entire stack trace, which unwinds back through the series of function calls that were made leading up to the point where the error occurred. Each function call is recorded in reverse order, starting from the function that directly caused the error and going back to the initial entry point of the script.

This trace of function calls, along with their corresponding file paths and line numbers, provides developers with a clear trail to follow. It aids in identifying where and how the error originated.

This is exactly what we want to know where was the `logger.error` and the other methods are being called from.

4.14.5 Getting the `callee` name and the line number

How can we use the information above to obtain the line number from the client's code? Can you think about it?

Let's add the following in our `#log` method of the `Logger` class:

```
// file: lib/logger.js

class Logger {
    ...
```