Diagram Practice and Unit Testing 101

Have out Paper and Pencil *or an iPad/Tablet+Stylus* - Laptops placed away! Bags under seats!

Only use Zones ABC. No DEF!

Kris Jordan / September 4, 2024 / COMP423 / Class 06

Diagram the following code listing

Assume that new Date().toISOString() returns a string like "2024-09-04T16:28:04.818Z"

```
1   export const main = () => {
2     logError("TODO: Implement main.");
3   };
4
5   const logger = (level: string) => {
6     return (message: string) => {
7       const dateTime: string = new Date().toISOString();
8       console.log(`${dateTime} [${level}] ${message}`);
9     };
10  };
11
12  export const logError = logger("ERROR");
13  export const logInfo = logger("INFO");
14
15  main();
```

```ts
export const main = () => {
  logError("TODO: Implement main.");
};

const logger = (level: string) => {
  return (message: string) => {
    const dateTime: string = new Date().toISOString();
    console.log(`${dateTime} [${level}] ${message}`);
  };
};

export const logError = logger("ERROR");
export const logInfo = logger("INFO");

main();
```

ISO Date: 2024-09-04T16:28:04.818Z

Compare *functional* versus *object-oriented* approaches

```
const logger = (level: string) => {
  return (message: string) => {
    const dateTime: string = new Date().toISOString();
    console.log(`${dateTime} [${level}] ${message}`);
  };
};
```

For each approach, write example usage code that "constructs" a logger and logs "Hello, world". Assume a log level of "INFO".

```
export class Logger {
  private level: string;

  constructor(level: string) {
    this.level = level;
  }

  log(message: string) {
    const dateTime: string = new Date().toISOString();
    console.log(`${dateTime} [${this.level}] ${message}`);
  }
}
```

Verification is inherent to Engineering

Testing *well* results in more reliable, more maintainable products.

- In Software Engineering, there are *many* testing and verification strategies
  - Each has trade-offs and generally a diverse testing strategy is encouraged
- Unit Testing - Test individual units of code in isolation
- Integration Testing - Test multiple units of code integrating with one another
- Functional Testing - Verifies software functionality meets expectations
- End-to-End Testing - Tests complete flows of application across whole stack
- Other kinds of tests: performance, security, acceptance, exploratory, compatibility, mutation, chaos, recovery, and more...

Unit Testing

Testing "Unit" of Code in Isolation

- Unit Testing emphasizes *isolating* the code "under test" to single "units", typically individual functions, methods, and single classes
- Generally used in Early Stages of Development, especially initial implementation
- Test-Driven ("Test-first") Development pairs nicely with unit tests:
  - Write a test for the behavior you want first, *with a test that fails*
  - Then go correctly implement the unit under test *to pass the test*
  - Repeat until fully implemented
- Isolating a unit under test can be a real chore for functions with dependencies

Imagine unit testing `logger`.
What dependencies does it have?

```
const logger = (level: string) => {
  return (message: string) => {
    const dateTime: string = new Date().toISOString();
    console.log(`${dateTime} [${level}] ${message}`);
  };
};
```

How do you isolate dependencies in unit tests?

The Dark Arts of Mocks, Fakes, and Stubs

- Lots of isolation strategies and best practices are often language specific
  - One upside in dynamic languages like JavaScript/Python/Ruby is that testing frameworks can easily exploit the ability to hot swap implementations of objects, methods, functions, and so on, at run time
- General idea:
  1. Before a test runs, swap out dependencies with instrumented "fakes"
  2. During a test, use instrumentation to confirm expected behavior
  3. After a test runs, swap back in the real dependencies / undo mutation

Case Study: jest Spying and Mocking

jest is a JavaScript/TypeScript testing framework from Facebook/Meta

- The syntax of jest tests leverages arrow functions for simple readability:

```
describe('logger function', () => {
 it('should log the correct message with the given level', () => {
    const level = 'INFO';
    const message = 'This is a log message';

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // TODO: Verify expected... but how?!?
  });
});
```

Spying on and mocking in jest

- Establishing a **spy** in jest **instruments** a function/method so that you can test whether the function/method was called, what arguments it was called with, and so on. Spying alone *does not alter behavior*!
  - When tests need spying capabilities, you need a variable to refer to the spy.
- Establishing a **mock** in jest replaces a function/method's implementation with mocked implementation. The mocked implementation typically either does nothing or returns an expected value.
  - This is very handy for functions that involve slow input/output side-effects like saving files to storage or loading data from the network.
- These two concepts can be combined! You can *spy* on a method *and mock* it.

Spying and Mocking console.log

```
describe('logger function', () => {
  let logSpy: jest.SpyInstance;

  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
  });

  afterEach(() => { jest.restoreAllMocks(); });

  it('should log the correct message with the given level', () => {
    const level = 'INFO';
    const message = 'This is a log message';

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // Verify expected output
    const expectedLog = `${new Date().toISOString()} [${level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```

Spying and Mocking console.log

```
describe('logger function', () => {
  let logSpy: jest.SpyInstance;

  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
  });

  afterEach(() => { jest.restoreAllMocks(); });

  it('should log the correct message with the given                {
    const level = 'INFO';
    const message = 'This is a log message';

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // Verify expected output
    const expectedLog = `${new Date().toISOString()}                level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```

Handle on the spy.

Establish spy and mock before each test is run.

Restores all mocks after each test completes.

The `expect` function in jest is like a fluent assertion. On a spy, you can test usage.

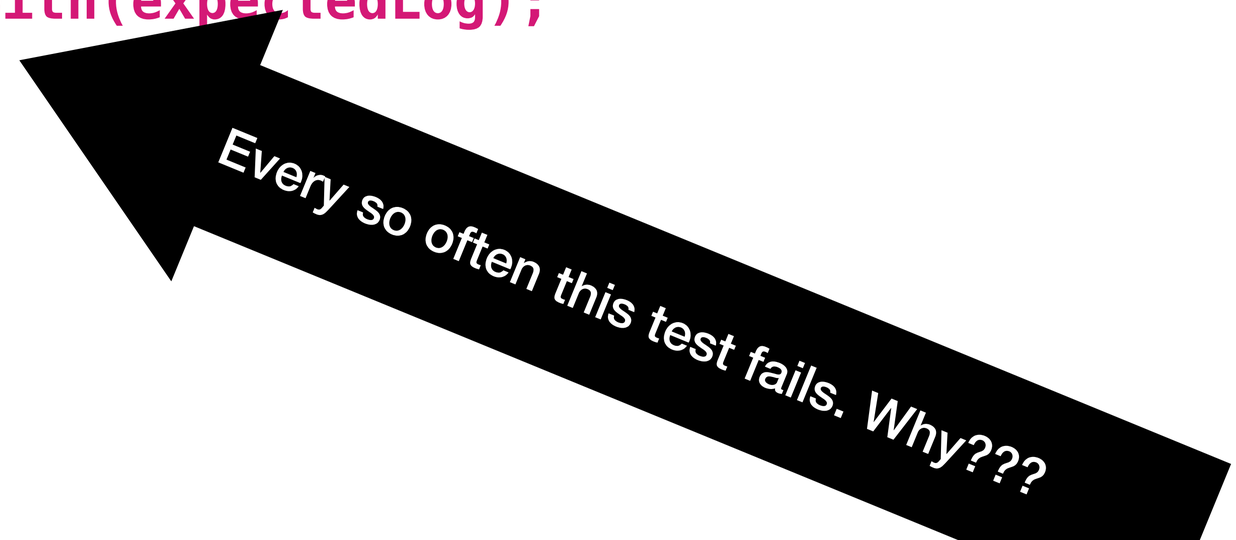
```
  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
  });

  afterEach(() => { jest.restoreAllMocks(); });

  it('should log the correct message with the given level', () => {
    const level = 'INFO';
    const message = 'This is a log message';

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // Verify expected output
    const expectedLog = `${new Date().toISOString()} [${level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```



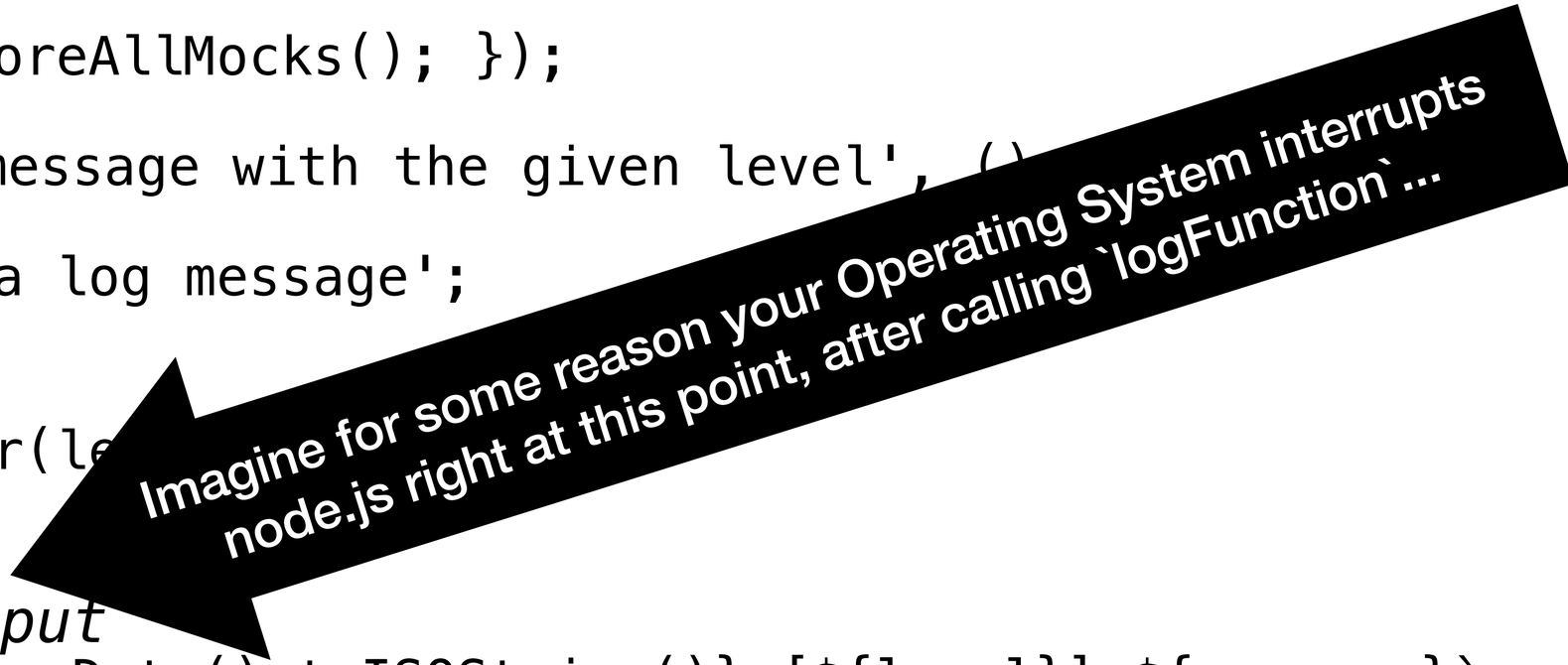
```
  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
  });

  afterEach(() => { jest.restoreAllMocks(); });

  it('should log the correct message with the given level', ()
    const level = 'INFO';
    const message = 'This is a log message';

    // Unit under test:
    const logFunction = logger(le
    logFunction(message);

    // Verify expected output
    const expectedLog = `${new Date().toISOString()} [${level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```



Spying and Mocking Date

```
describe('logger function', () => {
  let logSpy: jest.SpyInstance;
  const mockDate = new Date('2024-09-04T12:00:00.000Z');

  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
    jest.spyOn(global, "Date").mockImplementation(() => mockDate);
  });

  afterEach(() =>  reAllMocks(); });

  it('should log the corre ith the given level', () =>  {
    const level = 'INFO';
    const message = 'This is a log ;

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // Verify expected output
    const expectedLog = `${mockDate.toISOString()} [${level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```

Notice we are replacing the Date() implementation with a single mock date object.

This way we can test against the same string independent of timing concerns.

```
describe('logger function', () => {
  let logSpy: jest.SpyInstance;
  const mockDate = new Date('2024-09-04T12:00:00.000Z');

  beforeEach(() => {
    logSpy = jest.spyOn(console, 'log').mockImplementation(() => {});
    jest.spyOn(global, "Date").mockImplementation(() => mockDate);
  });

  afterEach(() => { jest.restoreAllMocks(); });

  it('should log the correct message with the given level', () => {
    const level = 'INFO';
    const message = 'This is a log message

    // Unit under test:
    const logFunction = logger(level);
    logFunction(message);

    // Verify expected output
    const expectedLog = `${mockDate.toISOString()} [${level}] ${message}`;
    expect(logSpy).toHaveBeenCalledWith(expectedLog);
  });
});
```

Closing question: what language feature enables this arrow function (in `it`), to read and access `mockDate` and `logSpy`?