10/25 — Async Programming, Observer Pattern, and Pub/Sub

🔁 Properties of Async Programming with Callbacks

- All functions in async are done as fast as possible (demand-focused).
- Singular: Only one callback is associated with each async call.
- Callbacks are required by the architecture.
- Expectation that the task completes.
- Tight coupling between caller and callback.
- Callbacks can only notify one specific entity that it is finished.

👁️ Properties of Observer Pattern

- Request-oriented (waits for an event).
    - We specify: "If this event occurs in the future, run this function."
    - Different from async which says: "Do this ASAP."
- Can fire multiple times (e.g., pressing the Enter key multiple times).
- No expectation that the event happens (not an error if no event occurs).
- Tight coupling (same as async).
- One-to-many: multiple observers can run their own code when an event is detected.

📣 Pub/Sub Pattern

- No tight coupling between publishers and subscribers.

🧠 Observer Pattern — Key Entities

- **Subjects** (emitters)
- **Observers** (listeners)

🔎 Finding Subjects

```
emit(eventName <string>, [...data]);
```

👂 Observers Register with

```
.on(eventName <string>, listener <function>);
```

📝 Example 01A: `readline-a.js` (Observer Only)

```
const readline = require('readline');
const rl = readline.createInterface({ input: process.stdin });    //
Interact with stdin

rl.on('line', (input) => {   // Triggered when Enter is pressed
    console.log(`Received: ${input}`);
});

console.log("Enter any input and press enter to fire line event");
```

- The function is triggered on a new line and logs “Received: text” after pressing Enter.

🧪 Async Style Version (Not Observer)

```
function captureInput() {
  rl.question(">", (input) => {
    console.log(`Received ${input}`);
  });
}
```

- This example is **not** demand-focused and **not singular**.

🔁 One-to-Many Behavior Example

```
rl.on('line', (input) => { console.log(`Received: ${input}`); });
rl.on('line', (input) => { console.log(`Received: ${input}`); });

console.log("Enter any input and press enter to fire line event");
```

- Outputs input twice, showing one-to-many behavior.

🕒 Listener Delay Problem Example

- If we put a `setTimeout` around listeners, they wouldn't work until timeout is over.
- Only the **last input** would get processed.

🧩 Example 01C: `readline-c.js`

```
const readline = require('readline');
const rl = readline.createInterface({ input: process.stdin });
rl.setMaxListeners(32); // Default is 10

function print(input) {
    console.log(`Received: ${input}`);
    rl.on('line', print);
}

rl.on('line', print);
console.log("If you need more listeners use the method setMaxListeners");
```

❗ Mistake in the Code Above

- Initially, one observer is set up.
- Every time it runs, it **adds another listener**, leading to exponential listener growth.
- Listeners persist until the program exits.

🌄 Example 02A: `DayEmitter.js`

```
const EventEmitter = require('events');

class DayEmitter extends EventEmitter {
    constructor(update_time = 240) {
        super();
        this.day = new Date();
        this.update_time = update_time;        // How many ms represent a
day
    }
    start() {
        this.day.setDate(this.day.getDate() + 1);                // Add 1
to day
        let mm = `${(this.day.getMonth() + 1 + "").padStart(2, "0")}`; //
Month
        let dd = `${(this.day.getDate() + "").padStart(2, "0")}`;     //
Day
        this.emit('newday', { mm_dd: `${mm}/${dd}` });     // Emit event
with data
        this.sleep();
    }
    sleep() {
        setTimeout(() => this.start(), this.update_time);
    }
}

module.exports = DayEmitter;
```

- Defines a class inheriting from `EventEmitter`, so it gets access to `.on()`, `.emit()`, etc.
- Constructor input: number of milliseconds representing a "day" (default: 240ms).
- `start()` updates the date and emits the `'newday'` event.
- `sleep()` uses recursion + `setTimeout()` to loop continuously.

🕹️ Example 02B: `index.js` (Observer File)

```
const DayEmitter = require("./modules/DayEmitter");
const day_emitter = new DayEmitter();

day_emitter.on("newday", function({ mm_dd }) {
    process.stdout.cursorTo(0, 0);
    process.stdout.clearLine();
    process.stdout.write(mm_dd);
    process.stdout.cursorTo(0, 1);
});

console.clear();
day_emitter.start();
```

- A clock-style system: every time new data occurs (`newday` event), it's printed on screen.
- Cursor movement is used to position the output neatly.

🧠 Multiple Observers Example with Additional Data

```
day_emitter.on("newday", function({ mm_dd }) {
    process.stdout.cursorTo(0, 0);
    process.stdout.clearLine();
    process.stdout.write(mm_dd);
    process.stdout.cursorTo(0, 1);
});

day_emitter.on("newday", function({ temp }) {
    process.stdout.cursorTo(5, 0);
    process.stdout.clearLine();
    process.stdout.write(temp);
    process.stdout.cursorTo(0, 1);
});

day_emitter.on("newday", function({ weather }) {
    process.stdout.cursorTo(10, 0);
    process.stdout.clearLine();
    process.stdout.write(weather);
    process.stdout.cursorTo(0, 1);
});
```

- Simulates displaying time, temperature, and weather at different cursor positions.
- Shows one-to-many observer behavior.

🎁 Using Destructuring with Multiple Event Properties

- If we emit multiple fields in one object:

```
this.emit('newday', { mm_dd, weather, temp });
```

- Each observer can destructure only what they need:

```
day_emitter.on("newday", function({ mm_dd }) {
    // Handle just date
});

day_emitter.on("newday", function({ temp }) {
    // Handle just temperature
});
```

- Prevents unused variables and keeps logic clean.

```

```