Background Story

It was 8 PM at a bustling co-working space in BGC when Odessa leaned back in her chair, rubbing her eyes. Around her, digital nomads were typing furiously, some using phone timers to track Pomodoro sessions, others scribbling on sticky notes. “There must be a better way,” she thought.

Earlier that week, Sir Ramon, the logistics manager, complained: “Ods, our drivers can’t focus on route planning. They need quick breaks and work sprints—like Pomodoro but built into our dashboard.” That evening, Odessa sketched a simple productivity timer on a scrap of grid paper—25 minutes of focus, 5 minutes of break, repeat.

She booted up her editor and wrote:

```
// 25 minutes in milliseconds
setTimeout(() => alert("Time for a break!"), 25 * 60 * 1000);
```

Great for a one-off alert, but she needed a ticking countdown on screen:

```
let seconds = 1500;
const timerId = setInterval(() => {
  seconds--;
  console.log(`Remaining: ${seconds}s`);
  if (seconds <= 0) {
    clearInterval(timerId);
    alert("Focus session over!");
  }
}, 1000);
```

Her console logged “Remaining: 1499s, 1498s, …” like a drumbeat. Next, she added a “Pause” button. Pausing meant capturing the time left, clearing the interval, then restarting later with the remaining seconds. After a few tries, she had a working `CountdownTimer` class with `start()`, `pause()`, and `resume()` methods.

The co-working crowd loved it. People gathered around Odessa’s laptop, cheering when her timer chimed. “This could be a product,” someone said. That applause was the spark of her next startup idea—a built-in focus timer for any web app. 🕒🚀

Now, we’ll learn how to use `setTimeout`, `setInterval`, and how to pause and resume timers in JavaScript—tools that can make your apps come alive on a schedule.

Theory & Lecture Content

1. setTimeout

Schedules a function once after a delay (ms):

```
const timeoutId = setTimeout(() => {
  console.log("Hello after 2 seconds");
}, 2000);

// To cancel before it fires:
clearTimeout(timeoutId);
```

Reference:
https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout

2. setInterval

Calls a function repeatedly at fixed intervals (ms):

```
const intervalId = setInterval(() => {
  console.log("Tick");
}, 1000);

// To stop:
clearInterval(intervalId);
```

Reference:
https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval

3. Pausing & Resuming a Countdown

Browsers don't natively pause timers. You must:

1. Record start time and desired duration.
2. On each tick, compute elapsed time with `Date.now()`.
3. To pause: clear the interval and store remaining time.
4. To resume: start a new interval with the remaining time.

Example skeleton:

```
class CountdownTimer {
  constructor(onTick, onFinish) {
    this.onTick = onTick;
    this.onFinish = onFinish;
    this.remaining = 0;
    this._intervalId = null;
    this._endTime = 0;
  }

  start(seconds) {
    this.remaining = seconds;
    this._endTime = Date.now() + seconds * 1000;
    this.tick(); // immediate update
```

```
    this._intervalId = setInterval(() => this.tick(), 1000);
  }

  tick() {
    const now = Date.now();
    this.remaining = Math.max(0, Math.round((this._endTime - now) / 1000));
    this.onTick(this.remaining);
    if (this.remaining <= 0) {
      clearInterval(this._intervalId);
      this.onFinish();
    }
  }

  pause() {
    clearInterval(this._intervalId);
    this._intervalId = null;
  }

  resume() {
    this._endTime = Date.now() + this.remaining * 1000;
    this._intervalId = setInterval(() => this.tick(), 1000);
  }
}
```

Exercises

Exercise 1: One-Off Delay

Problem Statement

In `exercise1.js`, implement `delay(ms, callback)` that:

- Uses `setTimeout` to call `callback()` after `ms` milliseconds.
- Returns the timeout ID so it can be cancelled.

Starter Code (exercise1.js)

```
// exercise1.js

/**
 * Calls callback after ms milliseconds.
 * @param {number} ms
 * @param {Function} callback
 * @returns {number} timeout ID
 */
export function delay(ms, callback) {
  // TODO
}
```

Full Solution (exercise1.js)

```
export function delay(ms, callback) {
  return setTimeout(callback, ms);
}
```

Exercise 2: Simple Countdown

Problem Statement

In `exercise2.js`, implement `startCountdown(seconds, onTick, onFinish)` that:

1. Every second, calls `onTick(remainingSeconds)`.
2. When `remainingSeconds` reaches 0, stops and calls `onFinish()`.
3. Returns the interval ID.

Starter Code (exercise2.js)

```
// exercise2.js

/**
 * @param {number} seconds
 * @param {Function} onTick  - called with remaining seconds
 * @param {Function} onFinish
 * @returns {number} interval ID
 */
export function startCountdown(seconds, onTick, onFinish) {
  // TODO
}
```

Full Solution (exercise2.js)

```
export function startCountdown(seconds, onTick, onFinish) {
  let remaining = seconds;
  onTick(remaining);
  const id = setInterval(() => {
    remaining--;
    onTick(remaining);
    if (remaining <= 0) {
      clearInterval(id);
      onFinish();
    }
  }, 1000);
  return id;
}
```

Exercise 3: CountdownTimer Class

Problem Statement

In `countdownTimer.js`, implement `CountdownTimer` class with methods:

- `start(seconds)`
- `pause()`
- `resume()`

Constructor signature: `constructor(onTick, onFinish)`.

Starter Code (countdownTimer.js)

```
// countdownTimer.js

export class CountdownTimer {
  constructor(onTick, onFinish) {
    // TODO: initialize properties
  }

  start(seconds) {
    // TODO
  }

  pause() {
    // TODO
  }

  resume() {
    // TODO
  }
}
```

Full Solution (countdownTimer.js)

```
export class CountdownTimer {
  constructor(onTick, onFinish) {
    this.onTick = onTick;
    this.onFinish = onFinish;
    this.remaining = 0;
    this._intervalId = null;
    this._endTime = 0;
  }

  start(seconds) {
    this.remaining = seconds;
    this._endTime = Date.now() + seconds * 1000;
    this.onTick(this.remaining);
    this._intervalId = setInterval(() => this._tick(), 1000);
  }

  _tick() {
```

```
    const now = Date.now();
    this.remaining = Math.max(0, Math.round((this._endTime - now) / 1000));
    this.onTick(this.remaining);
    if (this.remaining <= 0) {
      clearInterval(this._intervalId);
      this.onFinish();
    }
  }

  pause() {
    clearInterval(this._intervalId);
    this._intervalId = null;
  }

  resume() {
    if (this.remaining > 0 && !this._intervalId) {
      this._endTime = Date.now() + this.remaining * 1000;
      this._intervalId = setInterval(() => this._tick(), 1000);
    }
  }
}
```

Test Cases

Use Jest fake timers (`jest.useFakeTimers()`).

exercise1.test.js

```
import { delay } from "./exercise1.js";
jest.useFakeTimers();

describe("delay", () => {
  test("calls callback after ms", () => {
    const cb = jest.fn();
    const id = delay(5000, cb);
    expect(cb).not.toHaveBeenCalled();
    jest.advanceTimersByTime(5000);
    expect(cb).toHaveBeenCalled();
    clearTimeout(id);
  });
});
```

exercise2.test.js

```
import { startCountdown } from "./exercise2.js";
jest.useFakeTimers();

describe("startCountdown", () => {
```

```
  test("ticks down and calls onFinish", () => {
    const ticks = [];
    const onTick = (s) => ticks.push(s);
    const onFinish = jest.fn();
    startCountdown(3, onTick, onFinish);

    // initial tick
    expect(ticks).toEqual([3]);

    jest.advanceTimersByTime(1000);
    expect(ticks).toEqual([3, 2]);

    jest.advanceTimersByTime(2000);
    // ticks: 1 and 0
    expect(ticks).toEqual([3, 2, 1, 0]);
    expect(onFinish).toHaveBeenCalled();
  });
});
```

countdownTimer.test.js

```
import { CountdownTimer } from "./countdownTimer.js";
jest.useFakeTimers();

describe("CountdownTimer", () => {
  let onTick, onFinish, timer;
  beforeEach(() => {
    onTick = jest.fn();
    onFinish = jest.fn();
    timer = new CountdownTimer(onTick, onFinish);
  });

  test("start and finish", () => {
    timer.start(2);
    expect(onTick).toHaveBeenCalledWith(2);

    jest.advanceTimersByTime(1000);
    expect(onTick).toHaveBeenCalledWith(1);

    jest.advanceTimersByTime(1000);
    expect(onTick).toHaveBeenCalledWith(0);
    expect(onFinish).toHaveBeenCalled();
  });

  test("pause and resume", () => {
    timer.start(5);
    jest.advanceTimersByTime(2000); // remaining ≈3
    timer.pause();

    const callsBefore = onTick.mock.calls.length;
```

```
    jest.advanceTimersByTime(5000); // should not tick during pause
    expect(onTick.mock.calls.length).toBe(callsBefore);

    timer.resume();
    jest.advanceTimersByTime(1000);
    // should tick remaining 3 → 2
    expect(onTick).toHaveBeenLastCalledWith(2);
  });
});
```

Closing Story

Odessa's productivity timer became a hit in co-working spaces. Developers tracked their focus sessions right in her web app, and small teams integrated it into project boards. That simple `setInterval` and clever pause/resume logic sparked her first startup seed funding.

As the final "ding!" echoed on her screen, she realized timing is everything—not just in code, but in life. Tomorrow, she'd explore **event handling**: capturing user clicks, bubbling events, and building an event-driven architecture that ties her timers, classes, and state machines together. Her journey was just getting started—one tick at a time. 🚀