Background Story

It was a humid afternoon in Makati when Odessa realized her code was starting to look like scribbles on a jeepney window during rush hour—messy and hard to read. She was building an e-commerce dashboard for her logistics firm, defining products, user profiles, and inventory items. Every time she needed a new kind of object, she'd copy-paste code, change a few names, and pray nothing else broke.

Her mentor, Kuya Ronnie, dropped a pull request with a comment: "Why not use **classes**? Think of them as blueprints—architectural plans for your data." He showed her how one `class Product` could hold all product logic—constructor, methods like `applyDiscount()`, and shared defaults.

Odessa remembered her college days in Quezon City, sketching building blueprints in her IT drawing class—lines, measurements, labels. Classes felt the same: define a template once, then **instantiate** (`new`) as many as needed. No more repetitive code!

She opened her editor and wrote:

```
class Product {
  constructor(id, name, price) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

  applyDiscount(percent) {
    this.price = this.price * (1 - percent / 100);
  }
}
```

She created two products, applied discounts, and saw the correct prices log out. Next, Kuya Ronnie introduced inheritance. A `class Admin extends UserProfile` could reuse user logic and add permissions. Her codebase shrank—no more copy-paste monsters.

By evening, Odessa had transformed her spaghetti functions into neat blueprints. She felt like a systems architect designing reusable components for her firm's growing codebase. She imagined scaling this pattern to orders, shipments, and customer reviews—all with clean, testable classes.

With blueprints in place, Odessa was ready to build robust, scalable apps. Tomorrow, she'd learn about event-driven patterns—how to fire and listen for events, making her UIs and data models interact seamlessly. But for now, she savored the clarity that **classes** bring: one blueprint at a time. 🏗️✨

Theory & Lecture Content

JavaScript classes (ES6) let you define blueprints for objects. They wrap constructor logic, methods, and inheritance in a clear syntax.

1. Defining a Class

```
class Product {
  constructor(id, name, price) {
    this.id = id; // instance property
    this.name = name;
    this.price = price;
  }

  applyDiscount(percent) {
    // instance method
    this.price *= 1 - percent / 100;
  }

  toString() {
    return `${this.name} (₱${this.price.toFixed(2)})`;
  }
}
```

- `constructor(...)`
  Called when you do `new Product(...)`.
- `this`
  Refers to the new instance.
- **Instance methods**
  Defined inside class body.

Reference:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

2. Instantiation

```
const p1 = new Product("P001", "Rice Cooker", 2500);
const p2 = new Product("P002", "Kettle", 1200);
p1.applyDiscount(10);
console.log(p1.toString()); // "Rice Cooker (₱2250.00)"
```

3. Static Members

Static methods or properties belong to the class, not instances.

```
class InventoryItem {
  static count = 0;

  constructor(name, qty) {
    this.name = name;
    this.qty = qty;
    InventoryItem.count++;
  }
  dispose() {
    InventoryItem.count--;
  }

  static getCount() {
    return InventoryItem.count;
  }
}

console.log(InventoryItem.getCount()); // 0
new InventoryItem("Box", 5);
console.log(InventoryItem.getCount()); // 1
```

4. Inheritance (`extends` & `super`)

```
class UserProfile {
  constructor(username, email) {
    this.username = username;
    this.email = email;
  }

  greet() {
    return `Hello, ${this.username}!`;
  }
}

class Admin extends UserProfile {
  constructor(username, email, permissions = []) {
    super(username, email); // call parent constructor
    this.permissions = permissions;
  }

  addPermission(perm) {
    this.permissions.push(perm);
  }
}

const admin = new Admin("ods", "ods@example.com");
console.log(admin.greet()); // "Hello, ods!"
admin.addPermission("EDIT_USERS");
```

Exercises

Exercise 1: Product Class

Problem Statement
Create a `Product` class in `product.js` that:

1. Takes `id`, `name`, `price` in constructor.

2. Has method `applyDiscount(percent)` that updates `price`.
3. Has method `toString()` that returns `"<name> (₱<price>)"` with two decimals.

TODOs

- Define `class Product`.
- Implement `constructor`, `applyDiscount`, `toString`.
- Export `Product`.

Starter Code (product.js)

```
// product.js

/**
 * @param {string} id
 * @param {string} name
 * @param {number} price
 */
export class Product {
  constructor(id, name, price) {
    // TODO
  }

  applyDiscount(percent) {
    // TODO
  }

  toString() {
    // TODO
  }
}
```

Full Solution (product.js)

```
export class Product {
  constructor(id, name, price) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

  applyDiscount(percent) {
    this.price *= 1 - percent / 100;
  }

  toString() {
    return `${this.name} (₱${this.price.toFixed(2)})`;
  }
}
```

Exercise 2: UserProfile Class

Problem Statement

In `userProfile.js`, build a `UserProfile` class that:

1. Accepts `username`, `email` in constructor.
2. Method `greet()` returns `"Hello, <username>!"`.
3. Method `updateEmail(newEmail)` updates the email property.

TODOs

- Define `class UserProfile`.
- Implement `constructor`, `greet`, `updateEmail`.
- Export `UserProfile`.

Starter Code (userProfile.js)

```
// userProfile.js

export class UserProfile {
  constructor(username, email) {
    // TODO
  }

  greet() {
    // TODO
  }

  updateEmail(newEmail) {
    // TODO
  }
}
```

Full Solution (userProfile.js)

```
export class UserProfile {
  constructor(username, email) {
    this.username = username;
    this.email = email;
  }

  greet() {
    return `Hello, ${this.username}!`;
  }

  updateEmail(newEmail) {
    this.email = newEmail;
  }
}
```

Exercise 3: InventoryItem with Static Counter

Problem Statement

Create `inventoryItem.js` with a class `InventoryItem` that:

1. Has static property `count` initialized to `0`.
2. Constructor takes `name` and `qty`, increments `count`.
3. Method `dispose()` decrements `count`.
4. Static method `getCount()` returns current `count`.

TODOs

- Define `class InventoryItem`.
- Add `static count = 0`.
- Implement `constructor`, `dispose`, `static getCount`.
- Export `InventoryItem`.

Starter Code (inventoryItem.js)

```
// inventoryItem.js

export class InventoryItem {
  // TODO: static count

  constructor(name, qty) {
    // TODO: init props and increment count
  }

  dispose() {
    // TODO: decrement count
  }

  static getCount() {
    // TODO: return count
  }
}
```

Full Solution (inventoryItem.js)

```
export class InventoryItem {
  static count = 0;

  constructor(name, qty) {
    this.name = name;
    this.qty = qty;
    InventoryItem.count++;
  }
  dispose() {
    InventoryItem.count--;
  }

  static getCount() {
    return InventoryItem.count;
  }
}
```

Exercise 4: Admin Subclass

Problem Statement

In `admin.js`, extend `UserProfile` to create `Admin`:

1. Constructor takes `username`, `email`, and optional `permissions` array (default `[]`).
2. Calls `super(username, email)`.
3. Has method `addPermission(perm)` to add a permission string.

TODOs

- Import `UserProfile`.
- Define `class Admin extends UserProfile`.
- Implement constructor and `addPermission`.
- Export `Admin`.

Starter Code (admin.js)

```
// admin.js
import { UserProfile } from "./userProfile.js";

export class Admin extends UserProfile {
  constructor(username, email, permissions = []) {
    // TODO
  }

  addPermission(perm) {
    // TODO
  }
}
```

Full Solution (admin.js)

```
import { UserProfile } from "./userProfile.js";

export class Admin extends UserProfile {
  constructor(username, email, permissions = []) {
    super(username, email);
    this.permissions = permissions;
  }

  addPermission(perm) {
    this.permissions.push(perm);
  }
}
```

Test Cases

product.test.js

```
import { Product } from "./product.js";

describe("Product Class", () => {
  test("constructor and toString", () => {
    const p = new Product("P1", "Laptop", 50000);
    expect(p.id).toBe("P1");
    expect(p.name).toBe("Laptop");
    expect(p.price).toBe(50000);
    expect(p.toString()).toBe("Laptop (₱50000.00)");
  });

  test("applyDiscount updates price", () => {
    const p = new Product("P2", "Phone", 20000);
    p.applyDiscount(25);
    expect(p.price).toBeCloseTo(15000);
  });
});
```

userProfile.test.js

```
import { UserProfile } from "./userProfile.js";

describe("UserProfile Class", () => {
  test("greet returns greeting", () => {
    const u = new UserProfile("ods", "ods@mail.com");
    expect(u.greet()).toBe("Hello, ods!");
  });

  test("updateEmail changes email", () => {
    const u = new UserProfile("ods", "old@mail.com");
    u.updateEmail("new@mail.com");
    expect(u.email).toBe("new@mail.com");
  });
});
```

inventoryItem.test.js

```
import { InventoryItem } from "./inventoryItem.js";

describe("InventoryItem Static Counter", () => {
  beforeEach(() => {
    // reset count
    InventoryItem.count = 0;
  });

  test("count increments on creation", () => {
    new InventoryItem("Box", 3);
    new InventoryItem("Bag", 2);
    expect(InventoryItem.getCount()).toBe(2);
  });

  test("dispose decrements count", () => {
    const item = new InventoryItem("Crate", 1);
    expect(InventoryItem.getCount()).toBe(1);
    item.dispose();
    expect(InventoryItem.getCount()).toBe(0);
  });
});
```

admin.test.js

```
import { Admin } from "./admin.js";

describe("Admin Subclass", () => {
  test("inherits greeting and permissions", () => {
    const a = new Admin("ods", "ods@mail.com");
    expect(a.greet()).toBe("Hello, ods!");
    expect(a.permissions).toEqual([]);
  });

  test("addPermission adds to permissions array", () => {
    const a = new Admin("ods", "ods@mail.com", ["READ"]);
    a.addPermission("WRITE");
    expect(a.permissions).toEqual(["READ", "WRITE"]);
  });
});
```

Closing Story

With classes and constructors under her belt, Odessa's code felt as sturdy as a steel-reinforced building. Products, users, inventory items, and admins all had clear blueprints. No more scattered functions—just reusable, testable models. Her logistics dashboard could now scale to thousands of SKUs without rewriting logic.

Kuya Ronnie sent a message: “Next up—event-driven architecture. You’ll learn how to fire and catch events between modules, making your classes interact in real time.”

Odessa closed her laptop with a grin. She had mastered **blueprints with classes**—and was ready to bring her app to life with events. The next chapter awaited, and she couldn’t wait to build it. 🚀