Notes - JavaScript - Design Principles - Foundational - Encapsulation

Dr Nick Hayward

A brief introduction to encapsulation as a design principle in JavaScript/TypeScript.

Contents

Intro

Design principles help us build maintainable, scalable, and robust applications. In JavaScript ecosystems (Node.js, browsers, React/React Native), we often isolate things that change and hide them behind stable contracts so the rest of the code stays simple.

Related ideas:

- encapsulate what varies
- favor composition over inheritance
- program to interfaces (contracts), not implementations
- loose coupling

Principle — encapsulate what varies

Apps evolve: requirements change, libraries are swapped, and behavior differs across environments. Encapsulation isolates the parts most likely to change and hides them behind a stable API. You can then change the inside without rewriting the outside.

Benefits:

- maintenance: modify only the encapsulated part
- flexibility: swap or remove behaviors safely
- readability: clear boundaries make code easier to reason about

Techniques for encapsulation (JavaScript)

1) Polymorphism / duck typing

- Use a shared contract (method names and shapes) across implementations. In JavaScript this is typically duck typing or, with TypeScript, interfaces.

2) Getters and setters

- JavaScript supports `get` / `set` accessors to validate, compute, or log around property access.

3) Private fields and closures

- Class private fields (`#name`) and closures hide internal state from external mutation.

4) Dependency injection and strategy

- Pass in collaborators (strategies) so behavior can vary without `if/else` branches.

5) Module boundaries

- Export a small public API and keep internals file-local (or unexported) to prevent tight coupling.

Example — encapsulation with polymorphism

We'll model writing to different databases with a shared contract. Here we use a base class to illustrate polymorphism. Note: in many JS apps, a strategy object without inheritance is even simpler (see Bonus).

```
// Base class with a contract; subclasses must implement writeQuery
class DbBase {
    constructor(data) {
        this.data = data;
    }
    writeQuery() {
        throw new Error("writeQuery() must be implemented by subclass");
    }
}

class NoSQLDB extends DbBase {
    writeQuery() {
        console.log(`NoSQL update: ${this.data}`);
    }
}

class SqlDB extends DbBase {
    writeQuery() {
        console.log(`SQL update: ${this.data}`);
    }
}

// usage
const updates = [new NoSQLDB(250), new SqlDB(500)];
for (const u of updates) {
    u.writeQuery();
}
```

Why it works: Callers depend on the stable `writeQuery()` contract, not the concrete class. You can add `GraphDB` without changing the loop.

Example — encapsulation with properties (get/set and private fields)

Use getters/setters to control access and private fields to protect state.

```
class Circle {
    #radius; // private field
    constructor(radius) {
        this.#radius = radius;
    }
    get radius() {
        return this.#radius;
    }
    set radius(value) {
        if (value < 0) throw new Error("Radius cannot be negative");
        this.#radius = value;
    }
    get area() {
```

```
        return Math.PI * this.#radius * this.#radius; // computed property (read-only)
    }
}

// usage
const c = new Circle(10);
console.log("Initial radius:", c.radius);
c.radius = 15;
console.log("New radius:", c.radius, "area:", c.area);
```

Note: Private fields are enforced by the language; outside code cannot access `#radius` .

Alternative with closures (no classes):

```
function createCircle(radius) {
    let r = radius; // closed over private state
    return {
        get radius() { return r; },
        set radius(value) {
            if (value < 0) throw new Error("Radius cannot be negative");
            r = value;
        },
        get area() { return Math.PI * r * r; }
    };
}

const cc = createCircle(12);
console.log(cc.area);
```

Bonus — encapsulation via strategy (composition)

Rather than subclassing, pass a strategy that implements the changing behavior. This keeps types shallow and dependencies explicit.

```
// strategies
const NoSQLWriter = () => ({
    writeQuery: (data) => console.log(`NoSQL update: ${data}`),
});

const SQLWriter = () => ({
    writeQuery: (data) => console.log(`SQL update: ${data}`),
});

// service composed with a writer strategy
class DataUpdater {
    constructor({ writer }) {
        this.writer = writer;
    }
    update(data) {
        this.writer.writeQuery(data);
    }
}

// usage
const updater1 = new DataUpdater({ writer: NoSQLWriter() });
```

```
const updater2 = new DataUpdater({ writer: SQLWriter() });

updater1.update(250);
updater2.update(500);
```

Benefits: You can inject a mock for tests, add new writers without touching `DataUpdater`, and avoid deep inheritance.

Edge cases and trade-offs

When inheritance is reasonable:

- clear subtype relationships with stable base contracts (LSP holds)
- framework constraints that require extending a base class

Pitfalls to avoid:

- over-generalizing too early — don't abstract until variation is real
- leaky encapsulation — exposing internal fields or returning mutable internals
- hidden coupling — shared mutable state across "encapsulated" parts

Testing benefits:

- strategies can be stubbed/mocked via constructor injection
- private fields reduce unintended state changes
- pure functions (via closures/modules) are trivial to test

Try it

Paste the examples into a Node 18+ REPL or `*.mjs` file and run with `node`.

Summary: Find the parts that change and hide them behind a small, stable API using getters/setters, private fields, closures, or strategy objects. Keep the rest of the system simple and insulated from change.