Coding Standards

1. Naming Conventions

1.1 Variables & Functions

- Use camelCase (lowerCamelCase) for variable names and function names.

  Example:
  let studentCount = 25;
  function calculateGrade(score) { … }

- Functions should typically use verbs to indicate action (e.g. `fetchStudents`, `enrollCourse`).
- Avoid underscores in variable/function names (i.e. don’t use `get_user_info`).
- Do not start variable/function names with uppercase (unless it’s a class).

1.2 Constants

- Use UPPERCASE_WITH_UNDERSCORES for constants (i.e. fixed values).

  Example:
  const MAX_COURSES = 10;
  const API_BASE_URL = 'https://api.ocms.com';

1.3 Classes

- Use PascalCase for class / constructor names (UpperCamelCase).

  Example:
  ```
  class CourseManager { … }
  class StudentProfile { … }
  ```

1.4 Private Members / Internal Fields

- Prefix private or internal fields / methods with an underscore `_`.

  Example:
  ```
  class Enrollment {
   _validatePrerequisites() { … }
  _maxSeats = 50;
  }
  ```

1.5 Boolean Variables

- Prefix boolean variables with `is`, `has`, `should`, etc.

  Example:
  ```
  let isActive = true;
  let hasPaid = false;
  let shouldNotify = true;
  ```

2. Layout & Formatting Conventions

2.1 Indentation

- Use 2 spaces per indentation level.

2.2 Line Length

- Limit lines to 80–100 characters (i.e. don’t make extremely long lines).

2.3 Semicolons

- Always end statements with semicolons.

  Example:
  ```
  const x = 10;
  function foo() { … }
  ```

2.4 Curly Braces / Block Style

- Opening braces on same line; closing brace on next line.
- Always use braces `{}` even for single-line statements.

  Example:
  ```
   if (isValid) {
  doSomething();
  } else {
  handleError();
  }
  ```

(Do not write `if (isValid) doSomething();` without braces.)

2.5 Spaces

- Place spaces around operators and after commas.

  Example:
  let sum = a + b;
  function foo(x, y) { … }

- Avoid no-space style like `a+b` or `foo(x,y)`.

2.6 Newlines & Spacing Between Blocks

- Insert a blank line between logically separate blocks or functions.
- Ensure file ends with a newline (i.e. final line break).

3. Member (Class) Order

Within a class, maintain a consistent ordering:

1. Static properties

2. Instance (non-static) properties

3. `constructor(...)`

4. Static methods

5. Public (non-private) methods

6. Private / internal methods (those prefixed with `_`)

```
}
```

Example:

```
class CourseManager {
  // 1. static properties
  static MAX_COURSES = 50;

  // 2. instance properties
  name;
  code;

  // 3. constructor
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }

  // 4. static methods
  static getMaxCourses() {
    return CourseManager.MAX_COURSES;
  }

  // 5. public methods
  enrollStudent(student) {
    // …
  }
  dropStudent(studentId) {
    // …
  }

  // 6. private / internal methods
  _validateStudent(student) {
    // …
  }
}
```

This ordering helps readability and consistency.

4. Comments & Documentation

4.1 Single-line Comments

- Use `//` with a space after, for short notes.
- Place comment above the code it refers to (not at end-of-line).

  Example:
  ```
  // Initialize the course modules
  initializeModules();
  ```

4.2 Multi-line Comments

- Use block style with `/* … */`, aligned asterisks.

  Example:

```
/*
 * This function handles batch enrollment of students.
 * It checks prerequisites, updates databases, and returns
 * a status object indicating success/failure for each student.
 */
function batchEnroll(students) {
  // …
}
```

4.3 JSDoc / API Documentation

- Use JSDoc style to document functions, parameters, return types, etc.

  Example:

```
/**
 * Enrolls a student into a course.
 *
 * @param {string} studentId - Unique ID of student.
 * @param {string} courseId - Unique ID of course.
 * @returns {boolean} True if enrollment was successful, else false.
 */
function enroll(studentId, courseId) {
  // …
 }
```

- Always specify parameter names, types, descriptions and return types.
- Use a JSDoc linter / plugin to enforce JSDoc correctness.

5. Tooling / Linting Integration

To enforce these rules programmatically, integrate ESLint with required plugins. The source suggests a sample ESLint config:

- Install ESLint and needed plugins:

  bash

  ```
  yarn add eslint eslint-plugin-sort-class-members eslint-plugin-jsdoc --dev
  ```

- Configure ESLint (e.g. in `eslint.config.mjs`) with rules such as:
  1. camelcase rule
  2. no-underscore-dangle (with allowances for private names)
  3. new-cap
  4. indent (2 spaces)
  5. max-len (e.g. 100)
  6. semi
  7. brace-style, curly
  8. spacing rules (space-infix-ops, comma-spacing)
  9. newline-after-var, eol-last
  10. class member sorting via sort-class-members plugin
  11. JSDoc rules via eslint-plugin-jsdoc (e.g. require-param, require-returns, etc.)
- Add a lint script in package.json, e.g.

```
"scripts": {
  "lint": "eslint ."
}
```

- Developers run yarn lint (or npm run lint) and optionally `--fix` to auto-correct issues.

- If using VS Code (or equivalent), integrate ESLint plugin so that linting / warnings are shown in-editor.

6. Additional Guidelines for OCMS Project (Extensions / Suggestions)

6.1 Frontend / UI Code (JS / TS / React / Vue / Angular, etc.)

- If using a framework, adopt its idiomatic style (e.g. for React, hooks naming, component file structure).
- For component file names, use PascalCase (e.g. CourseCard.jsx).
- For CSS / styling, consider a naming scheme (e.g. BEM, CSS Modules) and be consistent.
- Split UI, services, hooks, utils into clear folders.
- Avoid deeply nested props / state; keep components small and modular.
- Write unit / integration tests for components and services.
- Use TypeScript if possible (and add typing rules in lint config).

6.2 Backend / API Code (Node.js, Python, Java, etc.)

- Adopt consistent naming conventions per your language (e.g. in Python, snake_case for variables, PascalCase for classes).

- Use proper layering: controllers / services / models / repositories, as applicable.
- Validate inputs / sanitize data at API boundaries.
- Use consistent status codes / API response structure.
- Document APIs (e.g. with OpenAPI / Swagger / JSDoc).
- Use transactions where needed, handle errors properly and return meaningful error messages.
- Logging: use structured logging, consistent log levels, no sensitive info in logs.

6.3 Database / Schema / Queries

- Use consistent naming for tables, columns, constraints. (E.g. snake_case or camelCase consistently).
- Use plural names for tables (e.g. `students`, `courses`) or choose a style and stick.
- Name foreign keys, indexes, constraints meaningfully (e.g. `fk_enrollment_student_id`).
- Avoid SELECT *; explicitly list columns.
- Use parameterized queries / prepared statements to avoid SQL injection.
- Use migrations (e.g. with a versioning tool) rather than ad-hoc schema changes.

6.4 Testing & Coverage

- Write tests (unit, integration) for all modules / critical logic.
- Enforce minimum code coverage thresholds (e.g. 80%).
- Use descriptive test names.
- Mock external dependencies / DB calls where appropriate.
- Include tests in CI pipeline (see next section).

6.5 Continuous Integration / Deployment (CI/CD)

- Integrate linting, tests, builds in CI (e.g. GitHub Actions, GitLab CI).
- On pull request, enforce that lint and test must pass.
- Optionally auto-format code (e.g. via Prettier) on commit or PR.

- Use feature branching, commit messages should be clear and follow a style (e.g. Conventional Commits).

6.6 Code Review & Pull Requests

- Keep PRs small and focused.
- Use template for PRs: include summary, what changed, how to test.
- Reviewer to check for logic, edge cases, performance, style adherence.
- Peer review mandatory before merging.
- Approve only when lint/tests pass and review done.

6.7 Security / Performance Considerations

- Avoid storing secrets / credentials in code (use environment variables).
- Sanitize / validate all inputs.
- Use CSRF protection, XSS prevention, proper authentication and authorization.
- Use caching for heavy queries.
- Monitor performance (e.g. query times, response times).
- Use pagination / limit for list endpoints.

References

1. Mahiyat. (2020). *Coding Standard*. GitHub. Retrieved September 28, 2025, from https://github.com/Mahiyat/academia-task-management/wiki/Coding-Standard
2. W3Schools. (n.d.). *JavaScript Style Guide and Coding Conventions*. Retrieved from https://www.w3schools.com/js/js_conventions.asp
3. Mozilla Developer Network (MDN). (n.d.). *JavaScript Guide*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
4. ESLint. (n.d.). *ESLint: Pluggable JavaScript Linter*. Retrieved from https://eslint.org