````
## Background Story

It was a warm Thursday afternoon in Quezon City, and Odessa was sipping her iced coffee from Jollibee while studying in the campus garden. 🌳☕ As an IT student at UP Diliman, she'd been collecting her classmates' grades in Excel sheets for her "Database Fundamentals" class. Every time she opened that spreadsheet, she felt the numbers were just floating—detached from real meaning.

One evening, while riding the MRT home, an idea struck her: "What if I treat each record as an object in JavaScript?" Suddenly, her mind painted a vivid street scene: jeepneys filled with students, tricycles zooming by, and every passenger carrying their own "object" of data—name, age, course, year. She imagined mapping those passengers into objects, each with nested details like contact info and grades. 🚍👩‍🎓

The next day in the lab, she opened VS Code and wrote her first object literal:

```js
const passenger = {
  name: "Juan Dela Cruz",
  age: 20,
  course: "BSIT",
  year: 2,
};
```
````

She saw data transform into relationships: `passenger.name` was no longer a cell in Excel, but a property she could access and manipulate. This shift opened her eyes to the power of objects.

That same week, the president of her student developers' org approached her with an exciting challenge: build a mini internal tool to manage project assignments and member details. It would be Odessa's first mini client project! Her heart raced—this was her chance to apply object literals, dot vs bracket notation, and nested objects to a real-world scenario.

Under the soft glow of her table lamp that night, Odessa sketched a data model on her notebook:

- Project object with title, deadline, and members array
- Member objects nested with name, role, and tasks
- Task objects with description and status

She could almost hear the clack-clack of her keyboard as she planned out functions to add members, update statuses, and print summaries. The project didn't have a fancy UI yet, but the data structure was solid. 🔧✨

Inspired by the busy scenes of Maginhawa Street and the warmth of her community, Odessa took a deep breath. "Let's turn spreadsheet chaos into clean JavaScript objects!" she whispered. With that, she embarked on Lesson 6: **Objects in the Mirror**.

Theory & Lecture Content

1. Object Literals

In JavaScript, an **object literal** is a comma-separated list of key–value pairs wrapped in curly braces.

```
const student = {
  name: "Odessa",
  course: "BSIT",
  year: 3,
};
```

- Keys (properties) are strings or identifiers.
- Values can be primitives, arrays, functions, or even other objects (nested).

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

2. Dot Notation vs. Bracket Notation

Dot notation is the most common:

```
console.log(student.name); // "Odessa"
student.year = 4;
```

Bracket notation lets you use dynamic keys or keys with spaces:

```
const prop = "course";
console.log(student[prop]); // "BSIT"

const weird = {
  "full name": "Odessa Santos",
};
console.log(weird["full name"]);
```

When to use bracket notation:

- Property names with spaces or special characters
- Dynamic property access (variables)

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

3. Nested Objects

Objects can contain other objects or arrays. This models real-world relationships:

```
const project = {
  title: "Campus App",
  deadline: "2024-05-30",
  members: [
    { name: "Li", role: "Frontend" },
    { name: "Arnel", role: "Backend" },
  ],
  tasks: {
    design: { status: "done" },
    develop: { status: "in-progress" },
  },
};
```

Access nested values:

```
console.log(project.members[1].name);
// "Arnel"

console.log(project.tasks.develop.status);
// "in-progress"
```

Edge Cases & Best Practices

- Avoid deep nesting beyond 3–4 levels. Consider flattening or using separate modules.
- Use `Object.freeze()` for read-only configs.
- Validate that keys exist: `if (obj && obj.subObj) {...}` to prevent runtime errors.

Exercises

Exercise 1: Build a Student Object

Problem Statement

Create a `studentOdessa` object with properties: `name`, `age`, `course`, `year`. Then write `getStudentInfo(obj)` that returns a string:
"Odessa, age 20, is taking BSIT in year 3."

Todo

- Define `studentOdessa` as an object literal.
- Implement `getStudentInfo` using dot notation.

Starter Code (starter.js)

```
// TODO: 1. Create studentOdessa object literal
const studentOdessa = {
  // name: ___,
  // age: ___,
```

```
  // course: ___,
  // year: ___
};

// TODO: 2. Implement getStudentInfo(obj)
function getStudentInfo(student) {
  // return a formatted string
}
```

Solution (solution.js)

```
const studentOdessa = {
  name: "Odessa Santos",
  age: 20,
  course: "BSIT",
  year: 3,
};

function getStudentInfo(student) {
  return `${student.name}, age ${student.age}, is taking ${student.course} in year
${student.year}.`;
}

module.exports = { studentOdessa, getStudentInfo };
```

Exercise 2: Dynamic Scores with Bracket Notation

Problem Statement

Given two arrays, `modules` and `scores`, build an object `scoreBoard` where each key is the module name and each value is the corresponding score.

Todo

- Loop through `modules` array.
- Use bracket notation to assign scores.

Starter Code (starter.js)

```
function createScoreBoard(modules, scores) {
  const scoreBoard = {};
  for (let i = 0; i < modules.length; i++) {
    // TODO: assign scoreBoard[modules[i]] = scores[i]
  }
  return scoreBoard;
}

module.exports = createScoreBoard;
```

Solution (solution.js)

```
function createScoreBoard(modules, scores) {
  const scoreBoard = {};
  for (let i = 0; i < modules.length; i++) {
    scoreBoard[modules[i]] = scores[i];
  }
  return scoreBoard;
}

module.exports = createScoreBoard;
```

Exercise 3: Nested Class Record

Problem Statement

Create a `classRecord` object with these properties:

- `course`: string
- `teacher`: object with `name` and `email`
- `students`: array of student objects `{ name, grade }`

Then implement `getStudentGrade(record, studentName)` to return the grade or `"Not found"`.

Todo

- Build `classRecord` with nested objects.
- Implement `getStudentGrade` using nested access.

Starter Code (starter.js)

```
const classRecord = {
  course: "Web Development",
  teacher: {
    // name: ___,
    // email: ___
  },
  students: [
    // { name: ___, grade: ___ },
    // { name: ___, grade: ___ }
  ],
};

function getStudentGrade(record, studentName) {
  // TODO: find student by name and return grade or "Not found"
}

module.exports = { classRecord, getStudentGrade };
```

Solution (solution.js)

```
const classRecord = {
  course: "Web Development",
  teacher: {
    name: "Ma’am Liza",
    email: "liza@university.edu.ph",
  },
  students: [
    { name: "Rico", grade: 88 },
    { name: "Ana", grade: 92 },
  ],
};

function getStudentGrade(record, studentName) {
  const found = record.students.find((s) => s.name === studentName);
  return found ? found.grade : "Not found";
}

module.exports = { classRecord, getStudentGrade };
```

Test Cases

Create a `__tests__` folder and add `objects.test.js`:

```
// __tests__/objects.test.js
const { studentOdessa, getStudentInfo } = require("../exercise1/solution");
const createScoreBoard = require("../exercise2/solution");
const { classRecord, getStudentGrade } = require("../exercise3/solution");

describe("Exercise 1: studentOdessa & getStudentInfo", () => {
  test("studentOdessa has correct properties", () => {
    expect(studentOdessa).toEqual({
      name: "Odessa Santos",
      age: 20,
      course: "BSIT",
      year: 3,
    });
  });

  test("getStudentInfo returns formatted string", () => {
    const info = getStudentInfo(studentOdessa);
    expect(info).toBe("Odessa Santos, age 20, is taking BSIT in year 3.");
  });
});

describe("Exercise 2: createScoreBoard", () => {
  test("correctly creates scoreBoard object", () => {
    const modules = ["HTML", "CSS", "JS"];
```

```
    const scores = [80, 85, 90];
    const sb = createScoreBoard(modules, scores);
    expect(sb).toEqual({ HTML: 80, CSS: 85, JS: 90 });
  });
});

describe("Exercise 3: classRecord & getStudentGrade", () => {
  test("classRecord nested structure is correct", () => {
    expect(classRecord.course).toBe("Web Development");
    expect(classRecord.teacher).toEqual({
      name: "Ma'am Liza",
      email: "liza@university.edu.ph",
    });
    expect(classRecord.students.length).toBe(2);
  });

  test("getStudentGrade returns correct grade", () => {
    expect(getStudentGrade(classRecord, "Ana")).toBe(92);
  });

  test("getStudentGrade returns 'Not found' for unknown student", () => {
    expect(getStudentGrade(classRecord, "Bob")).toBe("Not found");
  });
});
```

Run tests with:

```
npm install jest --save-dev
npx jest
```

Closing Story

Odessa hit **CTRL+S** with a big smile. She tested the internal tool, saw her objects connect seamlessly, and even impressed the org president during their Monday briefing. 🎉 Her simple console-based dashboard listed projects, members, and tasks—all powered by her object models.

That evening, as she toasted a **halo-halo** with friends in Cubao, she realized how object literals transformed raw data into living models, just like how stories transform simple facts into memories. Next up, she'll dive into **Arrays & Iteration** to handle lists of data more efficiently—because every great app needs both objects *and* arrays working in harmony.

Stay tuned for Lesson 7: **"Array of Sunshine"**! 😇

Good luck, future full-stack dev! 🚀