Background Story

It was Odess a's first week as a summer intern at **StoreStream PH**, a small startup in Marikina that helps sari-sari store owners digitize their inventories. 🌴 The hot Metro Manila sun beat down on her as she rode the jeepney from Katipunan to Cubao. Her heart raced—this was her chance to apply what she learned in class, plus impress her new teammates.

On Day 2, her mentor, Ate Marife, handed Odessa a spreadsheet of products from a sari-sari store in Pasig: items like *taho*, *kwek-kwek*, *choc-nut*, and *sachet shampoos*. But the spreadsheet was static and hard to analyze: which products were running low? Which items made the most money? Odessa needed to transform and summarize that data in seconds.

Inside StoreStream's cozy office, with walls painted a warm *senyales green* and a wall-mural of the Mayon Volcano, Odessa opened Chrome DevTools. She remembered her JavaScript professor's tip from JRU: "Use the right array methods—`.map()`, `.filter()`, and `.reduce()`—to handle lists like a pro." 🥷

Odessa's first task: list all the product names in uppercase. Next, identify items with stock less than 10 so they could be reordered. Finally, calculate the total inventory value so the sari-sari store owner knows how much money is tied up in stock.

At 4 PM, she typed furiously:

```
// TODO: Use map, filter, reduce…
```

She ran her code. It worked! The console printed:

```
["TAHO", "KWEK-KWEK", "CHOCO-NUT", "SACHET SHAMPOO"]
[ { name: 'Kwek-kwek', stock: 5 }, … ]
₱3,750
```

Ate Marife beamed. "Galing mo, Odessa! You just automated what used to take me an hour of manual work."

That afternoon over merienda of **halo-halo** at a nearby store, Odessa realized how powerful JavaScript array methods could be in real business logic. She wasn't just writing code—she was solving real problems for Filipino sari-sari store owners. PH As the sun set, she prepared for tomorrow's challenge: chaining these methods to build dashboards and reports.

Theory

JavaScript provides three powerful array methods for transforming and summarizing data:

1. `.map()`
    - Transforms each element in an array and returns a new array.
    - Signature:

```
const newArr = oldArr.map((currentValue, index, array) => {
  /* return transformedValue */
});
```

- Best Practice: Pure functions (no side effects) and always return a value.
- MDN Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Example:

```
const products = [
  { name: "Taho", price: 10 },
  { name: "Choc-nut", price: 20 },
];
const namesUpper = products.map((p) => p.name.toUpperCase());
console.log(namesUpper); // ["TAHO", "CHOC-NUT"]
```

2. `.filter()`

- Selects a subset of elements based on a condition. Returns a new array.
- Signature:

```
const filteredArr = oldArr.filter((currentValue, index, array) => {
  /* return true or false */
});
```

- Edge Case: If callback doesn't return `true` or `false`, the element is dropped.
- MDN Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Example:

```
const lowStock = products.filter((p) => p.stock < 10);
console.log(lowStock);
// [ { name: 'Choc-nut', price: 20, stock: 5 } ]
```

3. `.reduce()`

- Reduces array to a single value. Great for sums, totals, or building objects.
- Signature:

```
const result = oldArr.reduce((accumulator, currentValue, index, array)
=> {
```

```
    /* return updatedAccumulator */
  }, initialValue);
```

- Always provide `initialValue` to avoid errors on empty arrays.
- MDN Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Example:

```
// Calculate total inventory value = sum of price * stock
const totalValue = products.reduce((sum, p) => sum + p.price * p.stock, 0);
console.log(totalValue); // e.g. 750
```

4. **Chaining**
   You can combine them in one pipeline:

```
const reorderValue = products
  .filter((p) => p.stock < 10) // pick low stock
  .map((p) => p.price * p.stock) // value per item
  .reduce((sum, val) => sum + val, 0); // total reorder value
console.log(reorderValue);
```

Exercises

Exercise 1: Product Names Extractor

Problem Statement
Create a function that returns an array of product names in uppercase using `.map()`.

Todo:

- Implement `getProductNames(products)` using `.map()`.
- Return a new array of strings.

Starter Code (`productNames.js`):

```
// productNames.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function getProductNames(products) {
  // TODO: Use .map() to extract product names in uppercase
}
```

```
module.exports = { products, getProductNames };
```

Solution (`productNames.js`):

```
// productNames.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function getProductNames(products) {
  return products.map((p) => p.name.toUpperCase());
}

module.exports = { products, getProductNames };
```

Exercise 2: Low Stock Filter

Problem Statement
Create a function that returns products with stock less than a given threshold using `.filter()`.

Todo:

- Implement `findLowStock(products, threshold)` using `.filter()`.
- Return an array of product objects.

Starter Code (`lowStock.js`):

```
// lowStock.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function findLowStock(products, threshold) {
  // TODO: Use .filter() to get products with stock < threshold
}

module.exports = { products, findLowStock };
```

Solution (`lowStock.js`):

```
// lowStock.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function findLowStock(products, threshold) {
  return products.filter((p) => p.stock < threshold);
}

module.exports = { products, findLowStock };
```

Exercise 3: Total Inventory Value

Problem Statement

Calculate the total value of all products in inventory using `.reduce()`. Value of each item is `price * stock`.

Todo:

- Implement `calculateInventoryValue(products)` using `.reduce()`.
- Return a number.

Starter Code (`inventoryValue.js`):

```
// inventoryValue.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function calculateInventoryValue(products) {
  // TODO: Use .reduce() to sum (price * stock)
}

module.exports = { products, calculateInventoryValue };
```

Solution (`inventoryValue.js`):

```
// inventoryValue.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];
```

```
function calculateInventoryValue(products) {
  return products.reduce((total, p) => total + p.price * p.stock, 0);
}

module.exports = { products, calculateInventoryValue };
```

Exercise 4: Reorder Value Calculation

Problem Statement
Using chaining, calculate the total value of products that need to be reordered (stock < threshold).

Todo:

- Implement `calculateReorderValue(products, threshold)` with `.filter()`, `.map()`, and `.reduce()`.
- Return a number.

Starter Code (`reorderValue.js`):

```
// reorderValue.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function calculateReorderValue(products, threshold) {
  // TODO: Chain filter, map, and reduce
}

module.exports = { products, calculateReorderValue };
```

Solution (`reorderValue.js`):

```
// reorderValue.js
const products = [
  { id: 1, name: "Sabong", price: 25, stock: 20 },
  { id: 2, name: "Kwek-kwek", price: 15, stock: 5 },
  { id: 3, name: "Taho", price: 10, stock: 50 },
];

function calculateReorderValue(products, threshold) {
  return products
    .filter((p) => p.stock < threshold)
    .map((p) => p.price * p.stock)
    .reduce((sum, val) => sum + val, 0);
}
```

```
module.exports = { products, calculateReorderValue };
```

Test Cases

Below are Jest test scripts for each exercise. Organize them in a `__tests__` folder.

1. **`__tests__/productNames.test.js`**

```
const { getProductNames } = require("../productNames");

describe("getProductNames()", () => {
  test("returns uppercase names", () => {
    const input = [{ name: "Taho" }, { name: "Choc-nut" }];
    expect(getProductNames(input)).toEqual(["TAHO", "CHOC-NUT"]);
  });

  test("empty array returns empty", () => {
    expect(getProductNames([])).toEqual([]);
  });
});
```

2. **`__tests__/lowStock.test.js`**

```
const { findLowStock } = require("../lowStock");

describe("findLowStock()", () => {
  const sample = [
    { name: "A", stock: 2 },
    { name: "B", stock: 12 },
    { name: "C", stock: 5 },
  ];

  test("filters stock < threshold", () => {
    expect(findLowStock(sample, 10)).toEqual([
      { name: "A", stock: 2 },
      { name: "C", stock: 5 },
    ]);
  });

  test("no matches returns empty array", () => {
    expect(findLowStock(sample, 1)).toEqual([]);
  });
});
```

3. **`__tests__/inventoryValue.test.js`**

```
const { calculateInventoryValue } = require("../inventoryValue");

describe("calculateInventoryValue()", () => {
  test("calculates total inventory value", () => {
    const sample = [
      { price: 10, stock: 2 },
      { price: 5, stock: 5 },
    ];
    // 10*2 + 5*5 = 20 + 25 = 45
    expect(calculateInventoryValue(sample)).toBe(45);
  });

  test("empty array returns 0", () => {
    expect(calculateInventoryValue([])).toBe(0);
  });
});
```

4. `__tests__/reorderValue.test.js`

```
const { calculateReorderValue } = require("../reorderValue");

describe("calculateReorderValue()", () => {
  const sample = [
    { price: 10, stock: 2 },
    { price: 5, stock: 12 },
    { price: 20, stock: 3 },
  ];

  test("calculates reorder value when stock < threshold", () => {
    // Items to reorder: stock 2 & 3 => 10*2 + 20*3 = 20 + 60 = 80
    expect(calculateReorderValue(sample, 5)).toBe(80);
  });

  test("no items to reorder returns 0", () => {
    expect(calculateReorderValue(sample, 1)).toBe(0);
  });
});
```

Closing Story

As the sun dipped behind Marikina's skyline, Odessa leaned back in her chair. She watched the console print out the results of her map, filter, and reduce functions—data transformed in milliseconds. In that moment, she realized these methods were more than "just code." They were the building blocks for real dashboards, reorder alerts, and even automated order forms for sari-sari store owners across the Philippines.

She sipped her buko juice, excitement bubbling inside. Tomorrow, she would explore **asynchronous JavaScript**—fetching live data from APIs, handling promises, and updating the UI in real-time. Just as map,

filter, and reduce empowered her to process arrays, async code would let her connect StoreStream PH's frontend to real databases and cloud services.

Odessa closed her laptop with a confident smile. Her journey from student to startup developer was shining brighter than ever. 💡 Next stop: **"Async Adventures"**—where callbacks, promises, and `async/await` await. Are you ready to code with her?

Happy coding! 🚀