Day 06: JavaScript Objects & OOP Basics (Project: Simple Product Catalog)

Welcome to Day 06 of my 30 Days of JavaScript Challenge! Today, we're diving into Objects and the foundational principles of Object-Oriented Programming (OOP) in JavaScript. Objects are powerful data structures that allow you to store collections of related data (properties) and functions (methods) in a single unit, mimicking real-world entities.For our project, we'll build a Simple Product Catalog Viewer, which is a perfect way to see how objects can be used to represent structured data like products.

Core Concepts Covered:

1. What are Objects?

In JavaScript, almost everything is an object or can behave like one. An object is an independent entity, with properties and type. It's a collection of key-value pairs.Key (or Property Name):

A string(or Symbol) that uniquely identifies a piece of data within the object.Value: Any JavaScript data type (primitive, array, function, or even another object).

2.Why use objects?

To represent real-world entities (e.g., a person, a car, a product).To group related data and functionality together.To organize and structure complex data.

3. Creating Objects.

A Object Literal (Most Common and Recommended)This is the simplest and most widely used way to create objects.

Syntax:

```
const objectName = {
  key1: value1,
  key2: value2,
  // ...
};
```

Example:

```
const person = {
  firstName: "Alice",
  lastName: "Smith",
  age: 30,
  isStudent: false,
};
```

```
console.log(person);
// Output: { firstName: 'Alice', lastName: 'Smith', age: 30, isStudent: false }
```

- b. Using the new `Object()` Constructor (Less Common)You can create an empty object and then add properties to it.

Syntax:

```
const objectName = new Object();
objectName.key1 = value1;
objectName.key2 = value2;
```

Example:

```
const car = new Object();
car.make = "Toyota";
car.model = "Camry";
car.year = 2022;
console.log(car);
// Output: { make: 'Toyota', model: 'Camry', year: 2022 }
```

- **c.** Constructor Functions (Traditional OOP)Before ES6 Classes, constructor functions were the primary way to create multiple objects of the same "type" or "blueprint." **Syntax:**

```
function ConstructorName(param1, param2) {
  this.prop1 = param1;
  this.prop2 = param2;
  this.method = function () {
    // object method
    // ...
  };
}
```

- **const instance = new ConstructorName(arg1, arg2);**

Example:

```
 function Book(title, author, pages) {
  this.title = title;
  this.author = author;
  this.pages = pages;
  this.getInfo = function () {
    return `${this.title} by ${this.author}, ${this.pages} pages.`;
  };
```

```
}
const book1 = new Book("The Hobbit", "J.R.R. Tolkien", 310);
console.log(book1.getInfo()); // Output: The Hobbit by J.R.R. Tolkien, 310 pages.
```

4. Accessing Object Propertiesa.

- Dot Notation (Most Common)Use when you know the property name beforehand and it's a valid JavaScript identifier (no spaces, starts with letter/underscore/$).

Syntax:

```
  objectName.propertyNameExample:const user = { name: "John", email:
"john@example.com" };
console.log(user.name);  // Output: John
console.log(user.email); // Output: john@example.com
```

- b. Bracket Notation Use when:

The property name contains spaces or special characters.The property name is stored in a variable (dynamic access).The property name is a number (though not common for object keys).

Syntax:

```
 objectName['propertyName'] or
objectName[variableHoldingPropertyName]Example:const product = { "product name":
"Laptop", price: 1200 };
console.log(product["product name"]); // Output: Laptop

const prop = "price";
console.log(product[prop]); // Output: 1200 (dynamic access)
```

5. Modifying, Adding, and Deleting Properties:

- **Modifying:** Assign a new value to an existing property.

```
const person = { name: "Alice", age: 30 };
person.age = 31; // Modify age
console.log(person); // Output: { name: 'Alice', age: 31 }
```

- ***Adding:** Assign a value to a new property name.

```
const person = { name: "Alice" };
person.city = "New York"; // Add new property
```

```
console.log(person); // Output: { name: 'Alice', city: 'New York' }
```

- **Deleting:** Use the delete operator.

```
const person = { name: "Alice", age: 30 };
delete person.age; // Delete age property
console.log(person); // Output: { name: 'Alice' }
```

6. Nested Objects

Object values can themselves be other objects, creating nested structures.

Example:

```
const student = {
  id: 101,
  name: "Bob",
  address: {
    street: "123 Main St",
    city: "Anytown",
    zip: "12345",
  },
  grades: {
    math: 90,
    science: 85,
  },
};
console.log(student.address.city); // Output: Anytown
console.log(student.grades.math); // Output: 90
```

7. Object Methods (this Keyword)

Functions stored as object properties are called methods. Inside a method, the this keyword refers to the object that the method belongs to. **Example:**

```
const dog = {
  name: "Buddy",
  breed: "Golden Retriever",
  bark: function () {
    console.log(this.name + " says Woof!"); // 'this' refers to the 'dog' object
  },
  // Arrow functions behave differently with 'this', capturing 'this' from their
surrounding scope.
  // For methods directly on objects, traditional function syntax is often clearer
for 'this'.
  introduce: () => {
    console.log(`Hi, I'm ${dog.name} the ${dog.breed}.`); // Explicitly using
```

```
dog.name
    // console.log(`Hi, I'm ${this.name} the ${this.breed}.`); // 'this' here
would NOT refer to 'dog'
  },
};
dog.bark(); // Output: Buddy says Woof!
dog.introduce(); // Output: Hi, I'm Buddy the Golden Retriever.
```

8. Useful Object Methods (Static Methods on Object constructor)

These are built-in methods on the global Object constructor that help you work with objects.Object.keys(obj): Returns an array of a given object's own enumerable property names (keys). **Example:**

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.keys(obj)); // Output: ['a', 'b', 'c']
Object.values(obj): Returns an array of a given object's own enumerable property
values.Example:const obj = { a: 1, b: 2, c: 3 };
console.log(Object.values(obj)); // Output: [1, 2, 3]
Object.entries(obj): Returns an array of a given object's own enumerable string-
keyed property [key, value] pairs.Example:const obj = { a: 1, b: 2, c: 3 };
console.log(Object.entries(obj)); // Output: [['a', 1], ['b', 2], ['c', 3]]
```

📝 Project for Day 06: Simple Product Catalog Viewer

Today's project is an interactive Product Catalog Viewer. This application will demonstrate how to represent structured data using JavaScript objects and arrays of objects, and how to display and interact with that data on a web page.Features:Display a list of products with their basic information.Click on a product to view its detailed information in a separate section.Filter products by category.Clear product details. (This project's code is located in index.html and index.js within this Day06 folder.)

💻 Project Logic Breakdown (index.js)

The index.js file manages our product data and controls how it's displayed on the web page. **1. Product Data (products Array)**

```
const products = [
  {
    id: "p001",
    name: "Laptop Pro X",
    category: "Electronics",
    price: 1200,
    description: "High-performance laptop with 16GB RAM and 512GB SSD.",
    image: "https://placehold.co/150x150/007bff/ffffff?text=Laptop",
  },
  // ... more product objects
];
```

- This products array is the central data store. Each element within this array is an object, representing a single product.
- Each product object has several properties (id, name, category, price, description, image) that define its characteristics. This demonstrates how objects group related data.

2. DOM Element References

```
const productListDiv = document.getElementById("productList");
const productDetailDiv = document.getElementById("productDetail");
const categoryFilter = document.getElementById("categoryFilter");
const clearDetailsBtn = document.getElementById("clearDetailsBtn");
// ... and other elements
```

- These variables provide easy access to the HTML elements where we'll display product lists and details.

3. Core Application Functionsa.

```
renderProductList((filterCategory = "All"));
```

- Purpose: Displays the list of products based on the selected category filter.

How it works:

- Clears the productListDiv.
- **Filtering:** If filterCategory is not 'All', it uses products.filter() to create a new array containing only products that match the selected category.
- **Rendering:**

It iterates over the (filtered) products using forEach().

- For each product, it creates a div element (product-card) and populates it with product.name, product.category, and product.price.
- **An addEventListener** is attached to each product card. When clicked, it calls showProductDetails() with the product.id.

b. showProductDetails(productId)

- **Purpose:**

Displays the detailed information of a selected product.

- **How it works:**

1. *Finding the Product:*

It uses products.find() (an array method) to locate the product object in the products array that matches the given productId.

2. **Conditional Display:**

If a product is found, it dynamically creates HTML content (including product.name, product.description, product.price, and product.image) and sets it as the innerHTML of productDetailDiv.

3. If no product is found (e.g., invalid ID), it displays a "Product not found" message.
4. It shows the detail section and hides the initial message.
5. It re-attaches the event listener for the new clear details button.

c. clearProductDetails()

- **Purpose:**
- Clears the content of the product detail section.
- **How it works:** Simply clears the content, hides the detail section, and shows the initial message again.

d. populateCategories()

- **Purpose**

Dynamically populates the category filter dropdown.

- *How it works:*

1. Uses products.map(product => product.category) to get an array of all categories.
2. Uses new Set() to get only unique categories.
3. Iterates over the unique categories and creates
   elements for the categoryFilter dropdown.

4. Event Listeners (Connecting UI to Logic)

```
categoryFilter.addEventListener("change", (event) => {
  renderProductList(event.target.value); // Re-render list with new filter
  clearProductDetails(); // Clear details when filter changes
});

clearDetailsBtn.addEventListener("click", clearProductDetails);

// Initial render on page load
document.addEventListener("DOMContentLoaded", () => {
  populateCategories();
  renderProductList(); // Display all products initially
});
```

Category Filter:

- When the value of the categoryFilter dropdown changes, it calls `renderProductList()` with the newly selected category and also clears any currently displayed product details.
- **Clear Details Button:** When clicked, it calls `clearProductDetails()`.

- **Initial Setup:** DOMContentLoaded ensures that `populateCategories()` and `renderProductList()` are called when the page first loads, populating the filter and displaying all products.

This project provides a hands-on demonstration of how objects are used to structure data, how to access their properties, and how to use array methods (filter, find, forEach, map) to work with collections of objects, all within a visually appealing interface.

☑ Practice Set :

- ▶ 1. Create and Access:
- ▶ 2. Modify and Add Properties:
- ▶ 3. Object Method with this:
- ▶ 4. Nested Object Access:
- ▶ 5. Iterate Object Keys/Values/Entries:
- ▶ 6. Function that Returns an Object:
- ▶ 7. Object in an Array (Find specific object):

💡 Key Takeaways for Day 06:

- `Objects` are fundamental for structuring complex, related data using key-value pairs.
- Use `dot notation` for known, valid property names; `bracket notation` for dynamic access or invalid names.
- Functions within `objects` are called `methods`; this inside a method refers to the object itself.
- `Object.keys()`, `Object.values()`, and `Object.entries()` are useful for iterating over object data.
- Arrays and objects often work together (e.g., an array of objects) to represent collections of structured data.