Day 4 — `this`, Prototypes & Object Model

---

1. Understanding `this` in JavaScript

`this` represents the **execution context** — the object that is calling the function. The value of `this` depends on **how the function is called**.

1.1 Implicit Binding (`object.method()`)

```
const user = {
  name: "AbdelSalam",
  sayName() {
    console.log(this.name);
  }
};

user.sayName(); // "AbdelSalam"
```

When a function is called as a method of an object, `this = object`.

1.2 Explicit Binding (`call`, `apply`, `bind`)

`call()`

```
function greet(g) {
  console.log(g + ", " + this.name);
}

const person = { name: "Ali" };
greet.call(person, "Hello");
```

`apply()`

```
greet.apply(person, ["Hi"]);
```

`bind()`

```
const greetAli = greet.bind(person, "Hey");
greetAli();
```

`bind` returns a new function with `this` permanently set.

1.3 `new` Binding (Constructor Functions)

```
function User(name) {
  this.name = name;
}

const u1 = new User("Ali");
console.log(u1.name);
```

Using `new` creates a new object and sets `this` to that object.

1.4 Arrow Functions (Lexical `this`)

Arrow functions **do not** have their own `this`. They take `this` from the outer scope.

```
const obj = {
  name: "Ali",
  normal() {
    console.log("normal:", this.name);
  },
  arrow: () => {
    console.log("arrow:", this.name);
  }
};
```

---

2. Prototypes & Prototype Chain

Every JavaScript object has a hidden reference to another object called its **prototype**. If a property is not found on the object, JavaScript looks for it in the prototype.

```
const user = { name: "Ali" };
console.log(user.toString); // from Object.prototype
```

2.1 Constructor Function + Prototype

```
function User(name) {
  this.name = name;
}

User.prototype.sayHi = function () {
  console.log("Hi, I'm " + this.name);
};

const u1 = new User("Ali");
u1.sayHi();
```

Methods in the prototype are **shared** between all instances.

2.2 Prototype Chain Visualization

```
u1
 ↑
User.prototype
 ↑
Object.prototype
 ↑
null
```

---

3. Constructor Pattern

```
function Car(model, year) {
  this.model = model;
  this.year = year;
}

Car.prototype.getInfo = function () {
  return `${this.model} (${this.year})`;
};
```

```
Car.prototype.honk = function () {
  console.log(this.model + " says: Beep!");
};

const c1 = new Car("BMW", 2020);
console.log(c1.getInfo());
c1.honk();
```

---

4. Method Borrowing with `call`

```
const person1 = {
  name: "Ali",
  sayName() {
    console.log("My name is " + this.name);
  }
};

const person2 = { name: "Sara" };

person1.sayName.call(person2); // "My name is Sara"
```

Borrowing methods allows reusing logic across objects.
