JavaScript Object-Oriented Programming (OOP) Complete Guide

📑 Table of Contents

📘 What is OOP?

Object-Oriented Programming (OOP) uses objects and classes to structure code around real-world entities, enabling better code reusability, scalability, and maintainability.

Why OOP is Used in JavaScript?

- ✅ To manage large applications
- ✅ To reuse code
- ✅ To reduce duplicate code

- ✅ To make code easy to understand & maintain
- ✅ Used heavily in React, Node.js, backend systems

🏛️ JavaScript OOP - 4 Main Pillars

1. **Object**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**

1️⃣ OBJECT

Object → Collection of key-value pairs

- ✓ **Properties** → Attributes / data of an object
- ✓ **Methods** → Functions inside an object
- ✓ **this** → Refers to the current object whose method is being executed

Creating an Object

```
let person = {
  name: "Sushan",
  age: 23,
  IsStudent: true
};
```

Adding Methods

```
let person = {
  name: "Sushan",
  age: 23,
  greet: function () {
    console.log("Hello, my name is " + this.name);
  }
};
```

IMPORTANT: `this` = current object
Here → `this.name` means `person.name`

🏗️ Constructor Functions

Rules:

- Function name starts with **Capital letter**

- Use `this`
- Use `new` keyword

Basic Constructor

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}

let s1 = new Student("Sushan", 23);
let s2 = new Student("Rahul", 22);
```

What Happens Internally?

1. `new` creates empty object → `{ }`
2. `this` points to that empty object
3. `this.name = "Sushan"` → stored
4. `this.age = 23` → stored
5. Object returned automatically

Adding Method to Constructor

```
function Student(name, age) {
  this.name = name;
  this.age = age;
  this.introduce = function () {
    console.log("Hi, I am " + this.name);
  };
}

s1.introduce();
```

⚠️ **Problem (IMPORTANT CONCEPT)**

Every object gets its own copy of `introduce()`

👉 **Wastes memory** ❌

Solution 👉 **Prototype** (explained next)

Key Point:

- Constructor → used to create multiple / dynamic objects
- ✓ `new` keyword → creates a new object and assigns it to `this`

🔗 PROTOTYPE

1️⃣ WHAT is Prototype?

Prototype is a shared object where JavaScript stores methods so all objects can use them.

2️⃣ WHERE is Prototype?

> 🧠 **Important Truth:**
> 👉 Every JavaScript function automatically has a prototype object

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}

Student.prototype.introduce = function () {
  console.log("Hi I am " + this.name);
};
```

📍 Where is it stored?

- ❌ Not inside `s1`
- ❌ Not inside `s2`
- ✅ It is stored in `Student.prototype`

3️⃣ WHY do we need Prototype?

❌ Problem without Prototype

```
function Student(name, age) {
  this.name = name;
  this.age = age;
  this.introduce = function () {
    console.log(this.name);
  };
}
```

- Every object gets its own copy ❌
- Waste of memory ❌

✅ Solution using Prototype

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}

Student.prototype.introduce = function () {
  console.log(this.name);
};
```

> - ✓ Only ONE function
> - ✓ Shared by all objects
> - ✓ Memory efficient
> - ✓ Faster

How JS Finds Methods (Prototype Chain)

1. Look inside `s1` → ❌ not found
2. Go to `Student.prototype` → ✓ found
3. Execute function
4. `this` → `s1`

🧠 VISUAL (REMEMBER THIS)

```
s1
 ↓
Student.prototype
 ↓
Object.prototype
 ↓
null
```

This path is called → Prototype Chain

🧠 INTERVIEW GOLD (1 LINE EACH)

- **What?** Prototype is an object used for inheritance in JavaScript.
- **Where?** Stored as `ConstructorFunction.prototype`.
- **Why?** To share methods among objects and save memory.

🧪 Small Proof (Very Important)

```
console.log(s1.hasOwnProperty("name"));      // true
console.log(s1.hasOwnProperty("introduce")); // false
```

Because:

- `name` → own property
- `introduce` → prototype property

> ### ✅ CORRECT CONCEPT (VERY IMPORTANT)
>
> ❗ All objects do **NOT** have their own prototype object
> ❗ All objects **SHARE** the SAME prototype

Example:

```
let s1 = new Student("Sushan", 23);
let s2 = new Student("Rahul", 22);
```

Memory looks like this:

```
s1 ─┐
    ├──► Student.prototype (ONE shared object)
s2 ─┘
```

- One prototype ✓
- Many objects pointing to it ✓
- Memory efficient

Important Terminology (Interview Level)

- `Student.prototype` → Prototype object (shared)
- `s1.__proto__` → Internal reference (pointer)
- `s1.__proto__ === Student.prototype` // true

> 📌 **Objects don't own prototype**
> 📌 **They reference prototype**

🧠 Final Correct Answers (EXAM READY)

1. **Prototype is stored where?**
   👉 In `ConstructorFunction.prototype`
2. **Why use prototype?**
   👉 To share methods and save memory
3. **What happens if method not found?**
   👉 JS looks into the prototype via prototype chain
4. **Does every object have its own prototype?**
   👉 ❌ No. Objects share the same prototype

🎓 ES6 CLASSES

1️⃣ What is a Class? (Simple words)

A class is a blueprint to create objects.
Just like constructor + prototype combined in clean syntax.

Basic Class Syntax

```
class Student {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

- `constructor()` → runs automatically when object is created
- `this` → refers to the new object

Create Objects from Class

```
let s1 = new Student("Sushan", 23);
let s2 = new Student("Rahul", 22);
```

Result:

```
s1 = { name: "Sushan", age: 23 }
s2 = { name: "Rahul", age: 22 }
```

- ✓ Same as constructor function
- ✓ Cleaner syntax

Add Method to Class (IMPORTANT)

```
class Student {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  introduce() {
    console.log("Hi, I am " + this.name);
  }
}

s1.introduce(); // Hi, I am Sushan
s2.introduce(); // Hi, I am Rahul
```

VERY IMPORTANT TRUTH (INTERVIEW GOLD)

❓ **Where is `introduce()` stored?**

👉 **NOT** inside object

It is stored in: `Student.prototype`

Proof:

```
console.log(s1.hasOwnProperty("introduce")); // false
```

- ✓ Same prototype behavior
- ✓ Same memory optimization

Compare: Constructor vs Class

Constructor + Prototype

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}

Student.prototype.introduce = function () {
  console.log(this.name);
};
```

ES6 Class (Same thing)

```
class Student {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  introduce() {
    console.log(this.name);
  }
}
```

> 🧠 **Both are SAME internally**

typeof Surprise (Interview Question)

```
typeof Student; // "function"
```

😳 Yes! **Classes are functions internally**

2️⃣ ENCAPSULATION

❓ What is Encapsulation? (Simple)

Encapsulation = hiding data + controlled access

Real life: ATM card → you can withdraw money
You cannot access bank server directly

🟢 Example WITHOUT Encapsulation (Problem)

```
class BankAccount {
  constructor(balance) {
    this.balance = balance;
  }
}

let acc = new BankAccount(1000);
acc.balance = -5000; // 😱 allowed ❌ Dangerous
```

🟢 Example WITH Encapsulation (Modern JS)

🔐 Private Field (#)

```
class BankAccount {
  #balance;

  constructor(balance) {
    this.#balance = balance;
  }

  getBalance() {
    return this.#balance;
  }

  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount;
    }
  }
}
```

Usage

```
let acc = new BankAccount(1000);
console.log(acc.getBalance()); // 1000

acc.deposit(500);
console.log(acc.getBalance()); // 1500

acc.#balance = 9999; // ❌ Error
```

🧠 Why Encapsulation?

- ✓ Prevents invalid data
- ✓ Improves security
- ✓ Makes code safe
- ✓ Easier to maintain

🔐 Understanding the # Symbol

🔐 What is # in JavaScript Classes?

🧠 **Simple answer:** `#` is used to create **PRIVATE** variables and methods inside a class.

❓ Why was # introduced?

Before `#`, everything was public 😟
Anyone could change your data.

Example (OLD problem):

```
class User {
  constructor(password) {
    this.password = password;
  }
}

let u = new User("1234");
u.password = "hack"; // 😱 allowed
```

✅ With # (Modern JS)

```
class User {
  #password;

  constructor(password) {
    this.#password = password;
  }

  showPassword() {
    return this.#password;
  }
}
```

🔍 What does # really mean?

Without #	With #
Public	Private
Accessible outside	❌ Not accessible
Unsafe	Secure

🧪 Proof (VERY IMPORTANT)

```
let u = new User("1234");
console.log(u.#password); // ❌ Error

// ❌ Cannot access
// ❌ Cannot modify
// ✓ Fully private
```

🧠 Where can # be used?

✅ Inside class ONLY

```
class Test {
  #x = 10;

  getX() {
    return this.#x;
  }
}
```

- ❌ Outside class → Error
- ❌ Constructor function → Not allowed

> **🧠 Interview-ready answer**
>
> The `#` symbol in JavaScript is used to define **private class fields and methods** that cannot be accessed outside the class.

3️⃣ INHERITANCE (Parent → Child)

🧠 What is Inheritance? (Simple)

Inheritance means a child class can use properties and methods of a parent class.

Real life example:

- Parent → Vehicle 🚗
- Child → Car, Bike
- Both have: speed, move()
- But car has: AC

🟢 Basic Example (VERY SIMPLE)

🔹 Parent class

```
class Animal {
  speak() {
    console.log("Animal makes sound");
  }
}
```

🔹 Child class

```
class Dog extends Animal {
  bark() {
    console.log("Dog barks");
  }
}
```

🔹 Usage

```
let d = new Dog();
d.speak(); // from Animal
d.bark();  // from Dog
```

- ✓ Dog uses parent method
- ✓ No duplication
- ✓ Clean code

🧠 How extends works

Dog inherits from Animal
Prototype chain is created automatically
`Dog → Animal → Object → null`

🟢 Using constructor + super

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // calls parent constructor
    this.breed = breed;
  }
}

let d = new Dog("Tommy", "Labrador");
```

📌 **`super()` is mandatory before using `this`**

🧠 Interview one-liners

- **Inheritance** → Reusing code using parent-child relationship
- **extends** → creates inheritance
- **super()** → calls parent constructor

4️⃣ POLYMORPHISM

What is Polymorphism? (Simple)

Polymorphism means **one method name** but **different behavior**.

Real life:
Same word "run"

- Run a program
- Run on road
- Run a business

🟢 Example in JavaScript (VERY CLEAR)

🔹 Parent class

```
class Animal {
  speak() {
    console.log("Animal makes a sound");
  }
}
```

🔹 Child class (Overrides method)

```
class Dog extends Animal {
  speak() {
    console.log("Dog barks");
  }
}

class Cat extends Animal {
  speak() {
    console.log("Cat meows");
  }
}
```

🔹 Usage

```
let a1 = new Dog();
let a2 = new Cat();

a1.speak(); // Dog barks
a2.speak(); // Cat meows
```

- 👉 Same method name `speak()`
- 👉 Different output
- 👉 This is **Polymorphism**

🧠 What is Method Overriding?

When child class provides its own implementation of a parent method.

🧠 How JS decides which method to call?

1. JS checks method in child object
2. If found → calls it
3. If not → goes to parent (prototype chain)

🧠 Interview one-liners

- **Polymorphism** → Same method, different behavior
- **Method overriding** → Child replaces parent method
- **JS Polymorphism** → Achieved via inheritance & prototype chain

5️⃣ ABSTRACTION (Hide HOW, Show WHAT)

🧠 What is Abstraction? (Very Simple)

Abstraction means **hiding implementation details** and showing only **essential features**.

Real life:
You drive a bike 🚲

- You use accelerator & brake
- You don't know engine internals
- That's abstraction

🟢 Abstraction in JavaScript (IMPORTANT)

JavaScript does **NOT** have built-in abstract classes like Java

But we simulate abstraction using:

- Methods that must be overridden
- Error-throwing base methods
- Interfaces (by convention)

🟢 Example (Very Clear)

🔹 Abstract-like parent class

```
class Shape {
  area() {
    throw new Error("area() must be implemented");
  }
}
```

🔹 Child class implements it

```
class Rectangle extends Shape {
  constructor(w, h) {
    super();
    this.w = w;
    this.h = h;
  }

  area() {
    return this.w * this.h;
  }
}
```

🔹 Usage

```
let r = new Rectangle(10, 5);
console.log(r.area()); // 50
```

🧠 What is happening?

- Parent defines **what** should exist (`area`)
- Child defines **how** it works
- Parent cannot be used directly

🧠 Interview one-liners

- **Abstraction** → Hide implementation, expose functionality
- **JS abstraction** → Achieved using base classes and method overriding
- **Abstract method** → Method meant to be implemented by child

🧠 FINAL OOP SUMMARY (VERY IMPORTANT)

Pillar	Meaning
Object	Data + behavior
Encapsulation	Hide data
Inheritance	Reuse code
Polymorphism	Same method, different behavior
Abstraction	Hide how, show what

📋 FINAL OOP CHEAT SHEET (SAVE THIS)

Concept	Description
Object	Collection of properties and methods
Constructor	Used to create multiple objects dynamically
Prototype	Shared object used to store methods and save memory
Class	Syntax sugar over constructor + prototype
Encapsulation	Data hiding using private fields (#)
Inheritance	Child class reuses parent class using `extends`
Polymorphism	Same method name, different behavior
Abstraction	Hide implementation, expose functionality

🎯 Key Takeaways

1. **OOP** makes code reusable, scalable, and maintainable
2. **Prototype** saves memory by sharing methods
3. **Classes** are syntactic sugar over constructor functions
4. **#** creates truly private fields
5. **Inheritance** enables code reuse
6. **Polymorphism** allows method overriding
7. **Abstraction** hides complexity

📚 End of Guide ✨

Click the "Save as PDF" button at the top right to download this guide