



JavaScript Foundation – Training Day-5



Course content and duration:

Duration: 16 hours | Schedule: 8 days @ 2 hours/day

S. No	Day	Module	Topics
1	WED (10/12/2025)	Module 1: Introduction to JavaScript and Basics	JavaScript Overview, Syntax, and Variables
2	FRI (12/12/2025)	Module 2: Control Flow and Loops	Conditional Statements and Iteration
3	MON (15/12/2025)	Module 3: Functions and Scope	Function Declaration, Expressions, and Scope
4	TUE (16/12/2025)	Module 4: Arrays and Array Methods	Array Manipulation and Higher-Order Functions
5	WED (18/12/2025)	Module 5: Objects and Object-Oriented Programming	Objects, Properties, Methods, and Prototypes
6	THU (18/12/2025)	Module 6: DOM Manipulation and Events	Document Object Model and Event Handling
7	FRI (22/12/2025)	Module 7: Asynchronous JavaScript	Callbacks, Promises, and Async/Await
8	MON (23/12/2025)	Module 8: ES6+ Features and Best Practices	Modern JavaScript Features and Code Quality



Day5: Work with objects and understand OOP concepts (JavaScript)

Contents
Creating objects (literal and constructor)
Object properties and methods
Accessing and modifying properties
Object destructuring
this keyword and context
Constructor functions and prototypes
ES6 classes and inheritance
Object methods: Object.keys, Object.values, Object.entries



JavaScript Objects

What is an Object in JavaScript?

An **object** is a collection of **key–value pairs** where:

- Keys are **properties**
- Values can be **data or functions (methods)**

// key : value

Objects are used to represent **real-world entities** like users, products, cars, students, etc.

Objects help us **group related data together**.

Creating Objects using Object Literal

Why Object Literals Are Preferred

- Quick to write
- Easy to read
- No need for new
- No risk of incorrect this

Object literals are like **filled forms** – everything is written once and ready to use.

Simplest and most common way

Syntax

```
const objectName = {  
  key1: value1,  
  key2: value2,  
  method() {  
    // code  
  }  
};
```

Example 1: Simple Object Literal

```
const user = {  
  name: "Srikanth",  
  age: 30,  
  isTrainer: true  
};  
  
console.log(user.name); // Srikanth  
console.log(user.age);  // 30
```

Example 2: Object with Method

```
const car = {  
  brand: "Toyota",  
  model: "Innova",  
  start() {  
    console.log("Car started");  
  }  
};  
car.start(); // Car started
```

Example 3: Using this inside Object Literal (this)

```
const student = {  
  name: "Ravi",  
  marks: 85,  
  getResult() {  
    return `${this.name} scored ${this.marks}`;  
  }  
};  
  
console.log(student.getResult());  
// Ravi scored 85
```

this refers to the **current object**



When to Use Object Literal

- You need **only one object**
 - Object structure is **simple**
 - No need to create multiple similar objects
-

Advantages

- ✓ Easy to write
 - ✓ Readable
 - ✓ No boilerplate
-

Creating Objects using Constructor Function

Used when we need **multiple objects with the same structure**

What is a Constructor Function?

- A **regular function**
- Used with the **new keyword**
- Acts as a **blueprint** for objects

Naming convention: **PascalCase**

Syntax

```
function ConstructorName(param1, param2) {  
  this.param1 = param1;  
  this.param2 = param2;  
}
```

Example 1: Constructor Function

```
function User(name, age) {  
  this.name = name;  
  this.age = age;  
}  
  
const user1 = new User("Srikanth", 30);  
const user2 = new User("Ravi", 25);  
  
console.log(user1.name); // Srikanth  
console.log(user2.name); // Ravi
```

What Happens Internally with new?

1. Creates an empty object { }
 2. Sets this to that object
 3. Links prototype
 4. Returns the object automatically
-

Example 2: Constructor with Method (Not Recommended)

```
function Car(brand) {  
  this.brand = brand;  
  this.start = function () {  
    console.log("Car started");  
  };  
}
```



Problem:

Each object gets its **own copy of the function** → memory waste

What is happening?

- Every time new Car() is called:
 - A **new object** is created
 - A **new function** start() is created **again**

So:

- car1.start ❌
- car2.start ❌
➡ **Two separate function copies in memory**

Why is This a Memory Waste?

Inefficient

- Functions take **memory**
- If you create **1000 objects**, you create **1000 identical functions**
- Unnecessary duplication

Correct Way: Using prototype

Methods should be shared using **prototype**

```
function Car(brand) {  
  this.brand = brand;  
}  
  
Car.prototype.start = function () {  
  console.log(this.brand + " car started");  
};  
  
const car1 = new Car("Toyota");  
const car2 = new Car("Honda");  
  
car1.start(); // Toyota car started  
car2.start(); // Honda car started
```



- ✓ Method shared by all instances
- ✓ Better performance
- ✓ Only **one function** exists in memory
- ✓ All objects **reuse it**

How Prototype Helps

- Saves memory
- Improves performance

Object literals are used for simple, single objects, whereas constructor functions are used to create multiple objects with the same structure using the new keyword.

ES6 Class (Same Concept, Cleaner Syntax)

```
class Car {  
  constructor(brand) {  
    this.brand = brand;  
  }  
  start() {  
    console.log(this.brand + " car started");  
  }  
}
```

Under the hood:

- start() is placed on Car.prototype
- Same memory efficiency as prototype



Object Properties and Methods in JavaScript

What is an Object?

An **object** is a collection of **properties** (data) and **methods** (behavior).

```
const person = {  
  name: "Srikanth",  
  age: 30,  
  greet() {  
    console.log("Hello");  
  }  
};
```

Object Properties

What is a Property?

A **property** is a **key-value pair** inside an object.

key : value

Example

```
const user = {  
  username: "admin",  
  isActive: true,  
  loginCount: 5  
};
```

- username, isActive, loginCount → properties
- Values can be **any data type**

Accessing Object Properties

Dot Notation (most common)

```
const user = {  
  username: "admin",  
  isActive: true,  
  loginCount: 5  
};  
console.log(user.username);
```

✓ Simple & readable

Bracket Notation

```
const user = {  
  username: "admin",  
  isActive: true,  
  loginCount: 5  
};  
  
console.log(user["username"]);
```

✓ Required when:

- Property name has spaces
- Property name is dynamic

```
const user = {  
  username: "admin",  
  isActive: true,  
  loginCount: 5  
};  
  
const key = "loginCount";  
console.log(user[key]); // 5
```

Adding, Updating, Deleting Properties

+ Add

```
const user = {  
  username: "admin",  
  isActive: true,  
  loginCount: 5  
};  
  
user.role = "admin";  
console.log(user);
```

Update

```
const user = {
  username: "admin",
  isActive: true,
  loginCount: 5
};

user.loginCount = 10;
console.log(user);
```

✗ Delete

```
const user = {
  username: "admin",
  isActive: true,
  loginCount: 5
};

delete user.isActive;
console.log(user);
```

Object Methods

What is a Method?

A **method** is a **function stored as a property** of an object.

```
const car = {
  brand: "Toyota",
  start() {
    console.log("Car started");
  }
};
```

- start → method
 - Represents **behavior**
-

Method Syntax Styles

Old Syntax

```
start: function () {  
  console.log("Car started");  
}
```

ES6 Shorthand (Recommended)

```
start() {  
  console.log("Car started");  
}
```

- ✓ Cleaner
- ✓ Widely used

Using this in Methods

What is this?

this refers to the **current object**

```
const student = {  
  name: "Anita",  
  marks: 90,  
  getResult() {  
    return `${this.name} scored ${this.marks}`;  
  }  
};  
student.getResult(); // Anita scored 90
```

Without this, JS looks outside the object

Arrow Functions as Methods

Arrow functions **do not have their own this**

```
const user = {  
  name: "Ravi",  
  greet: () => {  
    console.log(this.name);  
  }  
};  
user.greet();
```

✅ Correct Way

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

Checking if a Property Exists

in operator

```
const user = {  
  name: "Ravi",  
  greet() {  
    console.log(this.name);  
  }  
};  
"name" in user;
```

hasOwnProperty

```
const user = {  
  name: "Ravi",  
  greet() {  
    console.log(this.name);  
  }  
};  
user.hasOwnProperty("role");
```

Looping Through Object Properties

for...in

```
const emp = { name: "John", role: "Dev", salary: 50000 };  
  
for (let key in emp) {  
  console.log(key, emp[key]);  
}
```

Built-in Object Methods

Object.keys()

```
const emp = { name: "John", role: "Dev", salary: 50000 };

for (let key in emp) {
  console.log(key, emp[key]);
}
Object.keys(emp);
```

Object.values()

```
const emp = { name: "John", role: "Dev", salary: 50000 };

for (let key in emp) {
  console.log(key, emp[key]);
}
Object.values(emp);
```

Object.entries()

```
const emp = { name: "John", role: "Dev", salary: 50000 };

for (let key in emp) {
  console.log(key, emp[key]);
}
Object.entries(emp);
```

✓ Useful for iteration & transformations

Example

```
const bankAccount = {  
  holder: "Srikanth",  
  balance: 10000,  
  
  deposit(amount) {  
    this.balance += amount;  
  },  
  
  withdraw(amount) {  
    this.balance -= amount;  
  }  
};  
  
bankAccount.deposit(5000);  
bankAccount.withdraw(2000);  
  
console.log(bankAccount.balance); // 13000
```

Common Mistakes

- Forgetting this
- Using arrow functions as methods
- accessing properties without checking existence

Key Points

- **Property** → stores data
 - **Method** → performs action
 - Use **dot notation** when possible
 - Use **bracket notation** for dynamic keys
 - Avoid **arrow functions** for object methods
-



Exercise:

// Create an object "product"

// properties: name, price

// method: getFinalPrice(tax)

✅ Solution

```
const product = {  
  name: "Laptop",  
  price: 60000,  
  getFinalPrice(tax) {  
    return this.price + tax;  
  }  
};  
  
product.getFinalPrice(5000);
```



Accessing & Modifying Object Properties in JavaScript

Accessing Nested Properties (Deep Objects)

```
const company = {
  name: "TechCorp",
  address: {
    city: "Hyderabad",
    pincode: 500001
  }
};
console.log(company.address.city);
// Problem: Property may not exist
console.log(company.contact.phone); // Error
```

Optional Chaining (?.)

```
const company = {
  name: "TechCorp",
  address: {
    city: "Hyderabad",
    pincode: 500001
  }
};
console.log(company.address.city);
console.log(company.contact?.phone); // undefined
```

- ✓ Prevents runtime errors
 - ✓ Essential for APIs & real-world apps
-

Modifying Properties Using Bracket Notation

```
const person = {
  name: "Ravi",
  address: {
    city: "Hyderabad",
    pincode: 500001
  }
};
const key = "age";
person[key] = 35;
console.log(person.age); // 35
```

- ✓ Used when keys are dynamic

Computed Properties (Dynamic Keys)

```
const prop = "score";
const student = {
  name: "Anita",
  [prop]: 95
};
console.log(student.score); // 95
```

📌 Common in reducers & dynamic forms

Read-Only & Protected Properties

Object.freeze() (No changes allowed)

```
const settings = {
  theme: "dark"
};
Object.freeze(settings);
settings.theme = "light"; // Ignored
console.log(settings.theme);
```

- ✓ Prevents add/update/delete
 - ✓ Shallow freeze only
-

Object.seal() (Modify only)

```
const config = {
  mode: "prod"
};
Object.seal(config);
config.mode = "dev"; // Allowed
config.version = "1.0"; // ✗ Not allowed
console.log(config);
```

Copying & Updating Objects Safely (Immutability)

✗ Mutating Original Object

```
const user = {  
  name: "Sita",  
  age: 28  
};  
user.age = 40;  
console.log(user.age);
```

✓ Immutable Update (Recommended)

```
const user = {  
  name: "Sita",  
  age: 28  
};  
const updatedUser = {  
  ...user,  
  age: 40  
};  
console.log(updatedUser.age);
```

✓ Used heavily in React, Redux

Looping While Accessing Properties

```
const user = {  
  name: "Sita",  
  age: 28  
};  
for (let key in user) {  
  console.log(key, user[key]);  
}
```

📌 Always use bracket notation in loops

Common Mistakes ❌

- ❌ Using dot notation with variables
 - ❌ Accessing deep properties without checks
 - ❌ Mutating frozen objects
 - ❌ Confusing undefined vs missing property
-

Practice Exercise

```
const order = {
  id: 101,
  customer: {
    name: "Anita"
  }
};

// 1. Access customer name
// 2. Add property totalAmount
// 3. Update customer name safely
```

✅ Solution

```
const order = {
  id: 101,
  customer: {
    name: "Anita"
  }
};
console.log(order.customer?.name);
const updatedOrder = {
  ...order,
  totalAmount: 2500,
  customer: {
    ...order.customer,
    name: "Ravi"
  }
};
console.log(updatedOrder);
```



Key Takeaways

- ✓ Dot → simple access
 - ✓ Bracket → dynamic access
 - ✓ Optional chaining → safe deep access
 - ✓ Spread → safe updates
 - ✓ Freeze/Seal → control modifications
-

Object Destructuring

What is Object Destructuring?

Object destructuring lets you **extract properties from an object into variables** in a clean, readable way.

Without Destructuring ❌

```
const user = { name: "Srikanth", age: 30, city: "Hyderabad" };
const name = user.name;
const age = user.age;
console.log(name, age);
```

With Destructuring ✅

```
const user = { name: "Srikanth", age: 30, city: "Hyderabad" };
const { name, age } = user;
console.log(name, age);
```

- ✓ Less code
- ✓ Better readability
- ✓ Widely used in modern JS

Basic Syntax

```
const { propertyName } = object;
Example:
const student = {
  name: "Anita",
  marks: 90
};
const { name, marks } = student;
```

📌 Variable names must match **property names**

Renaming Variables (Very Important)

```
const user = { name: "Ravi", age: 25 };

const { name: userName, age: userAge } = user;

console.log(userName); // Ravi
console.log(userAge);  // 25
```

- ✓ Avoids naming conflicts
- ✓ Common in large apps

Default Values

Used when property is **missing or undefined**

```
const config = { theme: "dark" };
const { theme, mode = "light" } = config;
console.log(mode); // light
```

- 📌 Default applies only if value is undefined, not null

Destructuring Nested Objects

```
const company = {
  name: "TechCorp",
  address: {
    city: "Hyderabad",
    pincode: 500001
  }
};

const {
  address: { city, pincode }
} = company;

console.log(city, pincode);
```

- ⚠ address must exist, or it throws error
-

Destructuring in Function Parameters

✗ Without Destructuring

```
const company = {  
  name: "TechCorp",  
  address: {  
    city: "Hyderabad",  
    pincode: 500001  
  }  
};  
  
function printUser(company) {  
  console.log(company.name, company.address.city);  
}  
  
printUser(company);
```

✓ With Destructuring

```
const company = {  
  name: "TechCorp",  
  address: {  
    city: "Hyderabad",  
    pincode: 500001  
  }  
};  
  
function printUser({ name, address: { city } }) {  
  console.log(name, city);  
}  
  
printUser(company);
```

✓ Cleaner

✓ Self-documenting functions

With Defaults in Parameters

```
const company = {  
  
  address: {  
  
    pincode: 500001  
  }  
};  
  
function printUser({ name = "Default", address: { city = "Mumbai" } }) {  
  console.log(name, city);  
}  
  
printUser(company);
```

Partial Destructuring

You don't need to extract everything

```
const product = {  
  id: 101,  
  name: "Laptop",  
  price: 60000  
};  
const { name, price } = product;
```

Rest Operator (...) in Destructuring

```
const user = {  
  name: "Srikanth",  
  age: 30,  
  city: "Hyderabad",  
  role: "Trainer"  
};  
  
const { name, ...restDetails } = user;  
console.log(name);  
console.log(restDetails);
```

✓ Useful in forms, API payloads

Destructuring with Dynamic / Computed Properties

```
const key = "score";

const student = {
  name: "Anita",
  score: 95
};

const { [key]: result } = student;

console.log(result); // 95
```

Destructuring + Renaming + Defaults

```
const settings = {
  theme: "dark"
};

const {
  theme,
  layout = "grid",
  mode: appMode = "prod"
} = settings;

console.log(appMode); // prod
```

Common Mistakes

- ✗ Mismatched variable names
 - ✗ Destructuring undefined objects
 - ✗ Forgetting default values
 - ✗ Over-destructuring (hurts readability)
-

Performance Discussion

Object destructuring does **not significantly improve performance** by itself. Its main benefit is **readability and maintainability**.

However, destructuring once and reusing variables avoids repeated property lookups in tight loops.

API Example

```
const response = {
  data: {
    user: {
      id: 1,
      name: "Srikanth"
    }
  }
};

const {
  data: {
    user: { name }
  }
} = response;

console.log(name);
```

Summary ★

- Destructuring extracts properties into variables
 - Supports renaming, defaults, rest operator
 - Very common in function parameters & APIs
 - Improves readability, not raw performance
 - Use carefully with nested objects
-

Practice Exercise

```
const order = {
  id: 101,
  customer: {
    name: "Anita",
    address: {
      city: "Hyderabad"
    }
  }
};

// Extract:
// 1. customer name as custName
// 2. city with default "Unknown"
const {
  customer: {
    name: custName,
    address: { city = "Unknown" }
  }
} = order;
```

Key Takeaways

- ✓ Cleaner variable extraction
- ✓ Essential for modern JS
- ✓ Powerful with defaults & rest
- ✓ Use safely with nested objects

Next topic: this keyword and execution



this keyword and execution

What is this in JavaScript?

this is a **special keyword** that refers to **who is calling the function**, not where it is written.

this is decided at runtime, based on the **execution context**.

Inside Object Method

```
const user = {  
  name: "Srikanth",  
  greet() {  
    console.log(this.name);  
  }  
};  
user.greet(); // Srikanth
```

✓ this → user

✓ Because greet() is called **through the object**

Global Context

```
console.log(this);
```

Browser → window

Node.js → Object {}

Function vs Arrow Function

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



Why Arrow Fails

- Arrow functions **do not bind their own this**
- They borrow this from parent scope

✗ Arrow for object methods

✓ Arrow for callbacks

this in Constructor Functions

```
function User(name) {  
  this.name = name;  
}  
  
const u1 = new User("Srikanth");
```

- this → newly created object

What new Does Internally

1. Creates empty object {}
 2. Binds this to it
 3. Links prototype
 4. Returns object
-

this in Classes (ES6)

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

  greet() {
    console.log(this.name);
  }
}

const p = new Person("Anita");
p.greet(); // Anita
```

- ✓ Same rules as constructor functions
 - ✓ Cleaner syntax
-

Losing this in Callbacks

```
const user = {
  name: "Srikanth",
  greet() {
    setTimeout(function () {
      console.log(this.name);
    }, 1000);
  }
};

user.greet(); // undefined
```



Reason:

- Regular function creates **new this**
 - Not bound to object
-

Fix 1: Arrow Function

```
setTimeout(() => {
  console.log(this.name);
}, 1000);
```

Fix 2: .bind(this)

```
setTimeout(function () {  
  console.log(this.name);  
}.bind(this), 1000);
```

Explicit Binding: call, apply, bind

call → invoke function immediately with this and arguments individually

apply → invoke function immediately with this and arguments as array

bind → returns a new function with permanently bound this

```
function greet(city) {  
  console.log(this.name, city);  
}  
  
const user = { name: "Ravi" };  
  
greet.call(user, "Hyderabad");  
greet.apply(user, ["Delhi"]);  
  
const boundGreet = greet.bind(user);  
boundGreet("Mumbai");
```

Why call, apply, bind exist

They are used to:

- Explicitly set the value of this
- Reuse functions with different objects
- Fix lost this in callbacks



Constructor Functions & Prototypes in JavaScript

What is a Constructor Function?

A **constructor function** is a **regular function** used with the `new` keyword to create **multiple objects with the same structure**.

Naming Convention

Use **PascalCase**

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

Creating Objects using Constructor Functions

```
const u1 = new User("Srikanth", 30);  
const u2 = new User("Anita", 25);  
  
console.log(u1.name); // Srikanth  
console.log(u2.name); // Anita
```

✓ Each object has its own **data**

Prototypes – The Correct Way

What is a Prototype?

Every function in JavaScript has a **prototype object**.

- Shared place to store **methods**
 - All instances can access them
-

Adding Methods to Prototype

```
function Car(brand) {  
  this.brand = brand;  
}  
  
Car.prototype.start = function () {  
  console.log(this.brand + " started");  
};  
  
const car1 = new Car("Toyota");  
const car2 = new Car("Honda");
```

- ✓ One method in memory
- ✓ Shared across objects

Prototype Chain

car1 → Car.prototype → Object.prototype → null

When JS looks for start():

1. Checks car1
2. Not found → checks Car.prototype
3. Found → executed

Accessing Prototype Methods

```
function Car(brand) {  
  this.brand = brand;  
}  
  
Car.prototype.start = function () {  
  console.log(this.brand + " started");  
};  
  
const car1 = new Car("Toyota");  
const car2 = new Car("Honda");  
  
car1.start(); // Toyota started
```

Even though start() is not on car1, JS finds it via the **prototype chain**.

Adding Multiple Prototype Methods

```
function Car(brand) {
  this.brand = brand;
}

Car.prototype.start = function () {
  console.log(this.brand + " started");
};

const car1 = new Car("Toyota");
const car2 = new Car("Honda");

Car.prototype.stop = function () {
  console.log(this.brand + " stopped");
};

Car.prototype.getBrand = function () {
  return this.brand;
};

car1.start(); // Toyota started
car1.stop(); // Toyota stop
car1.getBrand(); // Toyota getBrand
```

Prototype Chain Explanation

What is the Prototype Chain?

The **prototype chain** is the mechanism JavaScript uses to **look up properties and methods** on objects.

👉 When you access a property:

- JS **does not stop** at the object itself
- It keeps searching **up the chain** until it finds it or reaches null

object → prototype → Object.prototype → null



Why Prototype Chain is Powerful

- ✓ Code reuse
- ✓ Memory efficiency
- ✓ Shared behavior
- ✓ Dynamic lookup

ES6 Classes and Inheritance

What are ES6 Classes?

ES6 classes are a **clean, readable syntax** for creating objects and handling inheritance.

Syntax:

```
class ClassName {  
  constructor() {  
    // initialization  
  }  
  methodName() {  
    // behavior  
  }  
}
```

Example:

```
class User {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
  
  greet() {  
    console.log(`Hello, I am ${this.name}`);  
  }  
}  
  
const u1 = new User("Srikanth", 30);  
u1.greet();
```



Why Classes Were Introduced

- Cleaner syntax
- Familiar to Java/C++ developers
- Same prototype behavior underneath

Key Takeaways

- ✓ Classes are syntax sugar over prototypes
- ✓ Methods live on prototype
- ✓ extends sets prototype chain
- ✓ super() connects parent & child
- ✓ Supports encapsulation and inheritance

Prototype: An object from which other objects inherit properties.

Inheritance: A mechanism where an object can access another object's properties through the prototype chain.