

JavaScript Objects – From Basics to Advanced (With Practical Examples)

Think of this as your mini-book about JavaScript objects: from zero → interview-ready.

1. What is JavaScript? (Very short intro)

JavaScript is a programming language that runs mainly in the browser (and also on servers with Node.js). It is used to:

- Add behavior to web pages (clicking a button, showing/hiding things)
- Talk to servers (APIs)
- Build full applications (web, mobile, backend)

One of the most important building blocks in JavaScript is the **object**.

2. What is an Object?

2.1 Plain English definition

An **object** in JavaScript is a collection of **key-value pairs**.

- **Key** → usually a string (like `"name"`, `"age"`)
- **Value** → can be *anything*: number, string, boolean, function, array, even another object

Example (real-life idea): - You (a person) have properties: name, age, email, address - We can represent you as an object.

```
const user = {  
  name: "Rahul",  
  age: 25,  
  email: "rahul@example.com"  
};
```

Here: - Keys: `"name"`, `"age"`, `"email"` - Values: `"Rahul"`, `25`, `"rahul@example.com"`

This is a **JavaScript object**.

2.2 What is a key-value pair?

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

```
const car = {
  brand: "Toyota",
  model: "Camry",
  year: 2022
};
```

- brand: "Toyota" → key brand, value "Toyota"
- model: "Camry" → key model, value "Camry"
- year: 2022 → key year, value 2022

Order does not matter inside an object. Keys are unique.

3. How to Create an Object (Different Ways)

We will start from the most common to more advanced.

3.1 Object literal (most common)

```
const user = {
  name: "Anita",
  age: 30
};
```

- `const` creates a **constant reference** to the object.
- You **can change properties** of the object.
- You **cannot reassign** `user` to a completely new object.

```
user.age = 31;           // ✓ allowed (modify property)
user.city = "Pune";     // ✓ allowed (add property)

// user = {}           // ✗ not allowed (reassigning a const reference)
```

3.2 `let` vs `const` vs `var` with objects

Using `const`

```
const person = { name: "Kiran" };
person.name = "Kiran Kumar"; // ✓ allowed
// person = { name: "Someone Else" }; // ✗ TypeError in strict mode
```

- `const` **freezes the variable reference, not the internal data.**
- We can **modify** the inside (properties), but cannot point `person` to a new object.

Using `let`

```
let settings = { theme: "light" };
settings.theme = "dark";           // ✓ allowed
settings = { theme: "blue" };      // ✓ allowed (reassignment allowed)
```

- `let` allows **reassignment** and **modification**.
- Common in code where the entire object may be replaced.

Using `var` (old style, avoid in modern code)

```
var config = { debug: true };
config.debug = false;    // ✓ allowed
config = {};              // ✓ allowed
```

- `var` has function scope and hoisting issues.
- For interview and modern code, prefer `const` **for most objects**, `let` when you need to reassign.

Interview note:

Q: If I create an object with `const`, can I change its properties?

A: Yes, you can add, update, or delete properties. You just cannot reassign the variable to a new object.

3.3 `new Object()` constructor (less used nowadays)

```
const user = new Object();
user.name = "Priya";
user.age = 27;
```

This is equivalent to:

```
const user = { };
user.name = "Priya";
user.age = 27;
```

Use **object literals** `{}` instead. They're shorter and more common.

3.4 `Object.create()` (for prototypal inheritance)

```
const personPrototype = {
  greet() {
    console.log("Hello, I am " + this.name);
  }
}
```

```
};

const person = Object.create(personPrototype);
person.name = "Suresh";
person.greet(); // "Hello, I am Suresh"
```

- `Object.create(proto)` creates a new object whose internal prototype is `proto`.
- Used in more advanced patterns and libraries.

4. Accessing Object Properties (Dot vs Bracket)

Given:

```
const user = {
  name: "Asha",
  age: 28,
  "phone-number": "9999999999"
};
```

4.1 Dot notation

```
console.log(user.name); // "Asha"
console.log(user.age);  // 28
```

Use when: - The key is a valid identifier (no spaces, no `-`, doesn't start with a number).

4.2 Bracket notation

```
console.log(user["name"]);           // "Asha"
console.log(user["phone-number"]);  // "9999999999"

const key = "age";
console.log(user[key]);              // 28 (dynamic key)
```

Use when: - Key has special characters or spaces: `"phone-number"`, `"full name"` - Key is stored in a variable (dynamic)

Interview note:

Q: When must you use bracket notation instead of dot notation?

A: When the property name is not a valid identifier (e.g., contains `-` or spaces) or when the property name is stored in a variable.

5. Adding, Updating, and Deleting Properties

Given:

```
const user = { name: "Meera" };
```

5.1 Adding a property

```
user.age = 26;           // using dot
user["city"] = "Chennai"; // using bracket

console.log(user);
// { name: "Meera", age: 26, city: "Chennai" }
```

5.2 Updating a property

```
user.age = 27; // simply assign again
```

5.3 Deleting a property – different ways

1) Using `delete` operator

```
delete user.city;
console.log(user);
// { name: "Meera", age: 27 }
```

- Actually removes the key from the object.
- Returns `true` / `false` depending on success.

2) Setting to `undefined` or `null`

```
user.age = undefined; // key is still there, value is undefined
user.age = null;      // key is there, value is null
```

- Not truly deleted; just the value changes.
- Often, `delete` is clearer when you really want to remove the property.

3) Creating a new object without a key (using spread – advanced-ish)

```
const { age, ...userWithoutAge } = user;
console.log(userWithoutAge);
```

This is more functional-style (don't mutate the original object).

5.4 Merging objects / copying properties

`Object.assign()`

```
const defaults = { theme: "light", showSidebar: true };
const overrides = { theme: "dark" };

const settings = Object.assign({}, defaults, overrides);
console.log(settings);
// { theme: "dark", showSidebar: true }
```

Spread syntax `{ ...obj }`

```
const baseUser = { name: "Ravi", age: 30 };
const extendedUser = { ...baseUser, city: "Hyderabad" };

console.log(extendedUser);
// { name: "Ravi", age: 30, city: "Hyderabad" }
```

6. Nested Objects

6.1 What is a nested object?

A **nested object** is an object **inside another object**.

Real-world example: user with address.

```
const user = {
  name: "Pooja",
  age: 24,
  address: {
    street: "MG Road",
    city: "Bengaluru",
    pinCode: "560001"
  }
};
```

Here: - `user` is an object - `user.address` is a nested object

6.2 Accessing nested properties

```
console.log(user.address.city);    // "Bengaluru"
console.log(user["address"]["pinCode"]); // "560001"
```

6.3 Creating nested objects step by step

```
const user = {}; // Step 1: empty object
user.name = "Pooja"; // Step 2: add simple property
user.address = {}; // Step 3: create nested object
user.address.city = "Pune"; // Step 4: add properties to nested
```

6.4 Optional chaining (advanced but important)

Sometimes nested objects may not exist. Accessing them can throw an error:

```
// TypeError if profile is undefined
// console.log(user.profile.username);
```

Use **optional chaining**:

```
console.log(user.profile?.username); // returns undefined instead of error
```

7. Looping Over Object Keys and Values (Dynamic Access)

We often need to: - Go through all keys of an object - Read their values - Maybe build something using them

Given this object:

```
const user = {
  name: "Varun",
  age: 29,
  city: "Delhi"
};
```

We'll see multiple ways to loop over it.

7.1 Using `for...in` loop (classic for objects)

```
for (const key in user) {
  if (Object.hasOwn(user, key)) { // or user.hasOwnProperty(key) in older code
    console.log("Key:", key, "Value:", user[key]);
  }
}
```

Step-by-step explanation

1. **Find keys**
2. `for (const key in user)` automatically finds **all enumerable keys** in `user`.
3. **Loop each key**
4. For each iteration, `key` will be one of: `"name"`, `"age"`, `"city"`.
5. **Check own property** (important in some interview questions)
6. `Object.hasOwnProperty(user, key)` checks if the key really belongs to `user` (not from prototype).
7. **Read value using key**
8. `user[key]` gives the value.

Output:

```
Key: name Value: Varun
Key: age Value: 29
Key: city Value: Delhi
```

Interview note:

Q: Why do we use `hasOwnProperty` or `Object.hasOwnProperty` inside `for...in`?

A: Because `for...in` iterates over **inherited** properties too. The check ensures we only use properties that belong directly to the object.

7.2 Using `Object.keys()` + normal `for` loop

`Object.keys(obj)` returns an **array of keys**.

```
const keys = Object.keys(user); // ["name", "age", "city"]

for (let i = 0; i < keys.length; i++) {
  const key = keys[i];
  const value = user[key];
  console.log("Key:", key, "Value:", value);
}
```

Step-by-step explanation

1. **Get all keys as an array**
2. `keys` becomes `["name", "age", "city"]`.
3. **Loop through the array with normal `for`**
4. `i` starts at `0` and goes until `keys.length - 1`.
5. **Read key dynamically**
6. `const key = keys[i];`
7. **Read value using bracket notation**
8. `const value = user[key];`
9. **Use key-value**
10. Log it or use it in logic.

This is very common in modern JavaScript.

7.3 Using `Object.keys()` + `forEach`

```
Object.keys(user).forEach((key) => {  
  const value = user[key];  
  console.log("Key:", key, "Value:", value);  
});
```

Step-by-step: 1. `Object.keys(user)` → `["name", "age", "city"]` 2. `forEach` runs the callback for each key 3. Inside callback: - `key` is current key - `value = user[key]` is value

7.4 Using `Object.entries()` + `for...of`

`Object.entries(obj)` returns an **array of** `[key, value]` **pairs**.

```
for (const [key, value] of Object.entries(user)) {  
  console.log("Key:", key, "Value:", value);  
}
```

Step-by-step: 1. `Object.entries(user)` →

```
[  
  ["name", "Varun"],  
  ["age", 29],  
  ["city", "Delhi"]  
]
```

2. `for...of` iterates over each inner array. 3. `[key, value]` syntax is **array destructuring**. 4. Inside the loop, you directly get both `key` and `value`.

This is very clean and often preferred.

7.5 Dynamic access example (real scenario)

Imagine you have a configuration object, and you want to print all settings:

```
const config = {  
  apiBaseUrl: "https://api.example.com",  
  retries: 3,  
  timeoutMs: 5000  
};  
  
for (const [key, value] of Object.entries(config)) {
```

```
console.log(`Config ${key} = ${value}`);
}
```

Output:

```
Config apiUrl = https://api.example.com
Config retries = 3
Config timeoutMs = 5000
```

8. Important Object Methods: `Object.keys`, `Object.values`, `Object.entries`

Given:

```
const user = {
  name: "Rina",
  age: 32,
  city: "Mumbai"
};
```

8.1 `Object.keys(obj)`

- Returns an **array of keys**.

```
const keys = Object.keys(user);
console.log(keys); // ["name", "age", "city"]
```

Use cases: - Loop over keys - Count number of properties: `Object.keys(user).length`

8.2 `Object.values(obj)`

- Returns an **array of values**.

```
const values = Object.values(user);
console.log(values); // ["Rina", 32, "Mumbai"]
```

Use cases: - Sum numeric values - Check if some value exists

Example: sum of values

```
const scores = { math: 90, english: 85, science: 95 };
```

```
const total = Object.values(scores).reduce((sum, score) => sum + score, 0);
console.log(total); // 270
```

8.3 Object.entries(obj)

- Returns an **array of** [key, value] **pairs**.

```
const entries = Object.entries(user);
console.log(entries);
// [ ["name", "Rina"], ["age", 32], ["city", "Mumbai"] ]
```

Use cases: - Convert object to array - Loop with both key and value

Example: convert to array of strings

```
const pairs = Object.entries(user).map(([key, value]) => `${key}: ${value}`);
console.log(pairs);
// ["name: Rina", "age: 32", "city: Mumbai"]
```

9. Object Methods (Functions Inside Objects)

A **method** is just a function that is a property of an object.

```
const calculator = {
  a: 10,
  b: 5,
  sum() {
    return this.a + this.b;
  },
  difference: function () {
    return this.a - this.b;
  }
};

console.log(calculator.sum()); // 15
console.log(calculator.difference()); // 5
```

Here: - `sum` and `difference` are **methods**. - `this` refers to the **object itself** (`calculator`).

9.1 Arrow functions as methods (be careful)

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

```
const obj = {
  value: 42,
  badMethod: () => {
    console.log(this.value); // usually undefined
  }
};

obj.badMethod();
```

Use **normal function syntax** for methods that use `this`.

Interview note:

Q: Why shouldn't we use arrow functions for object methods that use `this`?

A: Because arrow functions capture `this` from the outer scope and do not bind their own `this`, which often leads to `undefined` or unexpected values.

10. Real-Time Scenarios with Objects

10.1 User profile from API

```
const userResponse = {
  id: 101,
  name: "Karthik",
  email: "karthik@example.com",
  settings: {
    darkMode: true,
    language: "en"
  }
};

console.log(userResponse.settings.darkMode); // true
```

10.2 Product catalog item

```
const product = {
  id: "P001",
  name: "Laptop",
  price: 55000,
  specs: {
    ram: "16GB",
    storage: "512GB SSD",
    processor: "i7"
  }
};
```

```
console.log(product.specs.ram); // "16GB"
```

10.3 App configuration

```
const appConfig = {  
  env: "production",  
  loggingEnabled: true,  
  api: {  
    baseUrl: "https://api.myapp.com",  
    timeoutMs: 8000  
  }  
};  
  
console.log(appConfig.api.baseUrl);
```

11. Advanced Topics (Still in Plain English)

11.1 Shallow copy vs deep copy

Shallow copy

```
const original = { name: "Amit", address: { city: "Pune" } };  
  
const copy = { ...original }; // or Object.assign({}, original)  
  
copy.name = "Amit Kumar";           // only changes copy  
copy.address.city = "Mumbai";       // changes both!  
  
console.log(original.address.city); // "Mumbai"
```

- Shallow copy copies **top-level** properties.
- Nested objects are still **shared by reference**.

Deep copy (simple way with JSON – has limitations)

```
const deepCopy = JSON.parse(JSON.stringify(original));
```

- Creates a full copy, but:
- Loses functions, `Date`, `Map`, etc.

11.2 Destructuring objects

```
const user = {  
  name: "Sanjay",
```

```

    age: 33,
    city: "Kolkata"
  };

  const { name, age } = user;
  console.log(name); // "Sanjay"
  console.log(age);  // 33

```

You can also rename:

```

const { city: userCity } = user;
console.log(userCity); // "Kolkata"

```

11.3 Rest properties in objects

```

const { name: firstName, ...rest } = user;

console.log(firstName); // "Sanjay"
console.log(rest);      // { age: 33, city: "Kolkata" }

```

11.4 `Object.freeze` and `Object.seal`

```

const settings = { darkMode: true };

Object.freeze(settings);
settings.darkMode = false; // ✗ silently ignored in non-strict mode

Object.seal(settings);
// You cannot add or remove properties, but can change existing values

```

12. Small Interview Question Set (Objects)

1. What is an object in JavaScript?
2. A collection of key-value pairs where keys are usually strings and values can be any type.
3. Difference between `const` object and `let` object?
4. `const` → cannot reassign the variable to a new object, but can modify properties.
5. `let` → can reassign and modify properties.
6. How do you add a new property to an object?
7. `obj.newKey = value;` or `obj["newKey"] = value;`.

8. **How do you delete a property from an object?**

9. `delete obj.key;`

10. **Difference between `Object.keys` and `Object.entries`?**

11. `Object.keys(obj)` → array of keys.

12. `Object.entries(obj)` → array of `[key, value]` pairs.

13. **How do you loop through an object's properties?**

14. `for...in` loop

15. `Object.keys(obj)` with `for` or `forEach`

16. `Object.entries(obj)` with `for...of`

17. **What is a nested object? How do you access it?**

18. An object inside another object.

19. Access with `obj.inner.key` or `obj["inner"]["key"]`.

20. **What is optional chaining?**

21. Syntax `obj?.inner?.key` to safely access nested properties without throwing errors when something is `undefined` or `null`.

22. **Why is `hasOwnProperty` or `Object.hasOwn` useful in loops?**

23. To ensure that we only work with properties directly belonging to the object, not inherited ones.

24. **When would you use objects in real applications?**

- User data, configuration, API responses, caching data in memory, etc.

13. Summary

- Objects are core building blocks of JavaScript.
- They store data as key-value pairs.
- You can create them with literals, constructors, `Object.create`, etc.
- Access with dot or bracket notation.
- Add, update, and delete properties easily.
- Loop through them using `for...in`, `Object.keys`, `Object.values`, `Object.entries`.
- Nested objects are very common in real-world data (API responses, configs).
- `const` does **not** make contents immutable; it just prevents reassignment of the reference.

You can now revisit each section, run the sample code in the browser console or Node.js, and practice by creating your own objects and loops.