

# Javascript Interview Questions Deep Dive(1-15 QA)

## 1. What is the difference between 'Pass by Value' and 'Pass by Reference'? Relation with shallow and deep copy

The **difference between pass by value and pass by reference in JavaScript**, and how that relates to **shallow copy vs deep copy**.

---

### 1. Pass by Value vs Pass by Reference

Concept	Description
<b>Pass by Value</b>	When you pass a <b>primitive type</b> (e.g., <b>number</b> , <b>string</b> , <b>boolean</b> ), JavaScript passes a <b>copy</b> of the value. Changing the parameter does <b>not</b> affect the original.
<b>Pass by Reference</b>	When you pass a <b>non-primitive type</b> (e.g., <b>object</b> , <b>array</b> ), JavaScript passes a <b>reference to the memory location</b> . Changing the parameter <b>can</b> affect the original.

---

#### ✓ Example – Pass by Value:

```
let x = 10;

function update(val) {
  val = val + 5;
}

update(x);
console.log(x); // ♦ Output: 10 (unchanged)
```

**x** is a **primitive**, so a copy was passed.

#### ✓ Example – Pass by Reference:

```
let obj = { count: 10 };

function update(o) {
  o.count += 5;
}

update(obj);
console.log(obj.count); // ♦ Output: 15 (modified)
```

**obj** is an **object**, so a reference was passed.

## 🧠 2. Shallow Copy vs Deep Copy

Concept	Description
Shallow Copy	Copies <b>only the top-level</b> properties. If those are objects/arrays, they are <b>still references</b> to the same memory.
Deep Copy	Copies <b>everything recursively</b> . Entirely new nested objects/arrays are created, no shared references.

---

### ✅ Shallow Copy Example:

```
let original = { a: 1, b: { c: 2 } };
let copy = { ...original }; // or Object.assign({}, original)

copy.b.c = 99;

console.log(original.b.c); // ♦ Output: 99 (original affected!)
```

Because **b** is a reference — **shallow copy** didn't clone deeply.

### ✅ Deep Copy Example:

```
let original = { a: 1, b: { c: 2 } };

// Deep copy using JSON (not ideal for functions, dates, etc.)
let copy = JSON.parse(JSON.stringify(original));

copy.b.c = 99;

console.log(original.b.c); // ♦ Output: 2 (original safe)
```

Now **copy.b** is a **new object**, not shared.

### 🔄 Comparing All Together

Concept	Example	Original Affected?	Depth	Notes
Pass by Value	<code>function(x) { x = 2 }</code>	❌	N/A	For primitives
Pass by Ref	<code>function(obj) { obj.x = 2 }</code>	✅	N/A	For objects/arrays
Shallow Copy	<code>{...obj}</code> or <code>Object.assign</code>	✅ (if nested)	One	Only top-level cloned
Deep Copy	<code>JSON.parse(JSON.stringify())</code>	❌	Full	All levels cloned

## 2. What is the difference between map and filter, reduce ?

### ✓ 1. **map()** – Transform each element

#### 📌 Purpose:

Transforms **each element** in an array and returns a **new array** of the same length.

#### 🧠 Key Points:

- **Does not mutate** the original array.
- Always returns an array.
- Each item is processed **individually**.

#### ✓ Syntax:

```
const result = array.map((element, index, array) => {  
  |   return transformedValue;  
});
```

#### 🖋 Example:

```
const nums = [1, 2, 3];  
const doubled = nums.map(n => n * 2);  
console.log(doubled); // [2, 4, 6]
```

#### 🎯 Use Cases:

- UI rendering (e.g., `users.map(u => <User {...u} />)`)
  - Data transformation
  - Adding fields to objects
- 

### ✓ 2. **filter()** – Select specific elements

#### 📌 Purpose:

Returns a **subset** of the array based on a condition (returns only those that pass).

### 🧠 Key Points:

- Does **not mutate** the original array.
- Returns a **new array** with **zero or more elements**.
- Only keeps elements where the **callback returns true**.

### ✅ Syntax:

```
const result = array.filter((element, index, array) => {  
  |   return condition;  
  | });
```

### 🔧 Example:

```
const nums = [1, 2, 3, 4];  
const evens = nums.filter(n => n % 2 === 0);  
console.log(evens); // [2, 4]
```

### 🎯 Use Cases:

- Filtering active users
  - Removing invalid data
  - Searching items with conditions
- 

## ✅ 3. **reduce()** – Accumulate into a single value

### 📌 Purpose:

Reduces an array to a **single value** (e.g., sum, object, string, etc.).

### 🧠 Key Points:

- Can return **anything**: number, string, object, array, etc.
- Iterates left-to-right.
- Can use an **initial value**.

## ✓ Syntax:

```
const result = array.reduce((accumulator, current, index, array) => {  
  return updatedAccumulator;  
}, initialValue);
```

## ✏ Example – Sum:

```
const nums = [1, 2, 3];  
const sum = nums.reduce((acc, curr) => acc + curr, 0);  
console.log(sum); // 6
```

## ✏ Example – Group by:

```
const users = [  
  { role: "admin", name: "Alice" },  
  { role: "user", name: "Bob" },  
  { role: "admin", name: "Eve" },  
];  
  
const grouped = users.reduce((acc, curr) => {  
  acc[curr.role] = acc[curr.role] || [];  
  acc[curr.role].push(curr.name);  
  return acc;  
}, {});  
  
console.log(grouped);  
// { admin: ['Alice', 'Eve'], user: ['Bob'] }
```

## 🎯 Use Cases:

- Summing numbers
- Grouping by fields
- Transforming to different structures

---

## 🚀 Performance Tips

- `map` and `filter` are **non-destructive** and chainable.

- **reduce** is more **powerful**, but can be **overkill** if you're just mapping/filtering.
  - If you're doing multiple passes (e.g., filter → map → reduce), consider combining them into a single **reduce** for better performance in large arrays.
- 

### ✂ Real-world Example: filter + map + reduce

```
const users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 35 },
  { name: "Eve", age: 45 },
];

// Get names of users over 30, then uppercase them, and count total characters
const totalLength = users
  .filter(u => u.age > 30) /* return [ { name: "Bob", age: 35 }, { name: "Eve", age: 45 }, ] */
  .map(u => u.name.toUpperCase()) /* ['Bob', 'Eve'] */
  .reduce((acc, name) => acc + name.length, 0); // count 3+3 = 6

console.log(totalLength); // 6
```

### 3. What is the difference between map() and forEach()

- ♦ **forEach()** — used for executing side effects

```
const numbers = [1, 2, 3];
numbers.forEach(num => {
  console.log(num * 2);
});
```

- ☒ Executes a function for each element
  - ☒ Does not return anything useful (**undefined**)
  - ☒ Used for side effects (e.g., logging, DOM updates, API calls)
-

- ♦ `map()` — used for transforming data

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

- ☒ Returns a new array with transformed items
- ☒ Doesn't modify the original array
- ☒ Preferred when you want to build a new result array

---

#### Summary Table

Feature	<code>map()</code>	<code>forEach()</code>
Returns	New array	<code>undefined</code>
Mutates data?	No	No
Use for	Data transformation	Side effects only
Chainable	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No

---

#### Common Mistake

Using `map()` without using the returned array:

```
items.map(item => doSomething(item)); //  Wrong: use forEach() instead
```

Use `forEach()` here, since you're not collecting a result.

☒ In short:


- ♦ Use `map()` when you need a new array.
- ♦ Use `forEach()` when you're just doing something with each item.

## 4. What is the difference between Pure and Impure functions?

☒ **Pure Function**

A **pure function** is a function that:

1. **Given the same inputs, always returns the same output**
2. **Has no side effects** (doesn't modify external state, variables, DOM, API calls, etc.)

 **Example:**

```
function add(a, b) {  
  return a + b;  
}
```

`add(2, 3);` // always returns 5 — no matter when or where it's called


✓ This is a **pure function** — it is predictable and safe to test or reuse.

---

## ✗ Impure Function

An **impure function** is a function that:

1. **May return different results for the same inputs**
2. **Has side effects**, such as modifying external variables, making API calls, changing the DOM, etc.

 **Example:**

```
let count = 0;  
  
function increment() {  
  count += 1;  
  return count;  
}
```

✗ This is an **impure function** — it relies on (and changes) an external variable.

---

 **Another Impure Example:**

```
function getTime() {  
  return new Date().toLocaleTimeString();  
}
```

✗ Impure: output changes every second even with same input (no input here!).



## 5. What is the difference between for-in and for-of ?

### for...in vs for...of

Feature	for...in	for...of
Iterates over	<b>Keys (property names)</b>	<b>Values (iterable items)</b>
Works on	Objects, Arrays, Inherited Properties	Arrays, Strings, Maps, Sets, etc.
Use case	Looping through <b>object properties</b>	Looping through <b>iterable values</b>
Output	<b>String keys</b> (for objects or array indices)	<b>Actual values</b> (not keys)
Can include inherited props	✅ Yes	❌ No

---

### ✅ Example: for...in with an object

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

for (const key in user) {
  console.log(key, user[key]);
}
// Output:
// name Alice
// age 25
```

- ♦ Use for...in for objects — it iterates over **keys** (property names).

### ✅ Example: for...of with an array

```
const nums = [10, 20, 30];

for (const num of nums) {
  console.log(num);
}
// Output:
// 10
// 20
// 30
```

- ♦ Use for...of for arrays — it iterates over **values**.

⚠ **Example: Using `for...in` on an array (not recommended)**

```
const nums = [10, 20, 30];
```

```
for (const index in nums) {  
  console.log(index, nums[index]);  
}
```

// Output:

// 0 10

// 1 20

// 2 30

`for...in` gives **indexes (as strings)** — can lead to bugs if the array is extended via prototype.

---

✅ **`for...of` supports:**

- Arrays
- Strings
- Maps
- Sets
- DOM NodeLists
- Generators

```
for (const char of "hello") {  
  console.log(char);  
}  
// Output: h e l l o
```

🧠 **Summary:**

Use case	Recommended Loop
Object keys/properties	<code>for...in</code>
Array values	<code>for...of</code>
Iterable collections	<code>for...of</code>
Avoid for array keys	Don't use <code>for...in</code> on arrays unless needed

### 💡 Bonus: `Object.keys()` + `for...of`

If you want to iterate over object **keys and values** with `for...of`:

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

for (const key of Object.keys(user)) {
  console.log(key, user[key]);
}
```

---

## 6. What are the differences between `call()`, `apply()` and `bind()` ?

✅ `call()` - Call a function with a specific `this` and arguments (comma-separated)

### ♦ Description:

Calls the function **immediately**, with `this` and arguments **passed one by one**.

### 🔧 Example:

```
function greet(greeting) {
  console.log(greeting + ', ' + this.name);
}

const user = { name: 'Alice' };

greet.call(user, 'Hello'); // ♦ Hello, Alice
```

---

✅ `apply()` - Call a function with `this` and arguments as an array

### ♦ Description:

Like `call()`, but arguments are passed in a **single array**.

### 🔧 Example:

```
function greet(greeting) {
  console.log(greeting + ', ' + this.name);
}

const user = { name: 'Alice' };

greet.apply(user, ['Hi']); // ♦ Hi, Alice
```

---

## ✅ **bind()** – Create a new function with **this** and optional preset arguments

### ♦ Description:

Returns a **new function** with fixed **this** and optional arguments. Does **not** call immediately.

### 🔧 Example:

```
function greet(greeting) {  
  console.log(greeting + ', ' + this.name);  
}  
  
const user = { name: 'Alice' };  
  
const sayHello = greet.bind(user, 'Hey');  
sayHello(); // ♦ Hey, Alice
```

### 🧠 Summary Table

Method	Calls Function?	Arguments Format	Returns
call	✅ Immediately	arg1, arg2, ...	result
apply	✅ Immediately	[arg1, arg2, ...]	result
bind	❌ Not immediately	arg1, arg2, ...	new function

```
function greet(greeting) {  
  console.log(greeting + ', ' + this.name);  
}  
  
const user = { name: 'Alice' };  
  
greet.call(user, 'Hello'); // ♦ Hello, Alice  
greet.apply(user, ['Hi']); // ♦ Hi, Alice  
  
const sayHello = greet.bind(user, 'Hey');  
sayHello(); // ♦ Hey, Alice
```

---

## 7. List out some key features of ES6 ?

### Top ES6 Features

#### 1. **let and const**

- Block-scoped variable declarations (instead of `var`)

```
let count = 0;  
const name = "Alice";
```

---

#### 2. **Arrow Functions**

- Shorter function syntax with lexical `this`

```
const add = (a, b) => a + b;
```

---

#### 3. **Template Literals**

- Multi-line strings and interpolation with backticks

```
const name = "Bob";  
console.log(`Hello, ${name}!`);
```

---

#### 4. **Destructuring**

- Unpack values from arrays or objects

```
const [x, y] = [1, 2];  
const { name, age } = { name: "Alice", age: 25 };
```

---

#### 5. **Default Parameters**

- Function parameters with default values

```
function greet(name = "Guest") {  
  console.log(`Hello, ${name}`);  
}
```

---

## 6. Rest & Spread Operators

- Rest: collect remaining args; Spread: expand iterable

```
const sum = (...nums) => nums.reduce((a, b) => a + b, 0);
```

```
const arr1 = [1, 2];  
const arr2 = [...arr1, 3, 4];
```

---

## 7. Enhanced Object Literals

- Shorthand for object properties and methods

```
const name = "Tom";  
  
const user = {  
  name,  
  greet() {  
    console.log("Hi");  
  }  
};
```

---

## 8. Modules (import/export)

- Modular code with native support

```
// export  
  
export const pi = 3.14;  
  
// import  
import { pi } from './math.js';
```

---

## 9. Classes

- Syntactic sugar for constructor functions

```
class Person {  
  
  constructor(name) {  
    this.name = name;  
  }  
  greet() {  
    console.log(`Hello, ${this.name}`);  
  }  
}
```

---

## 10. Promises

- Handle async code without callbacks

```
fetch('/data.json')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

---

## 11. for...of Loop

- Iterate over iterable values (arrays, strings, etc.)

```
for (const char of "Hi") {
  console.log(char);
}
```

---

## 12. Symbol

- Unique, immutable primitive useful for object keys

```
const sym = Symbol('id');
```

what is symbol

### What is a **Symbol**?

A **Symbol** is a **primitive data type** introduced in **ES6**. Every **Symbol** is **unique and immutable**, even if they have the same description.

Used mostly as **unique property keys** to avoid name conflicts.

---

### Create a Symbol

```
const sym1 = Symbol("id");
const sym2 = Symbol("id");
```

```
console.log(sym1 === sym2); // ✖ false (each symbol is unique)
```

---

### Using Symbol as a Unique Object Key

```
const user = {
  name: "Alice",
  [Symbol("id")]: 101 // hidden-like unique key
```

```
};
```

```
console.log(user); // { name: "Alice", [Symbol(id)]: 101 }
```

🔒 Symbol keys don't show up in:

- `for...in`
  - `Object.keys()`
  - `JSON.stringify()`
- 

### ✅ Accessing Symbol Key

```
const id = Symbol("id");
const user = {
  name: "Bob",
  [id]: 123
};
```

```
console.log(user[id]); // 💎 123
```

---

### 🧠 Why Use Symbol?

- Avoid property name clashes in objects
  - Create hidden-like private properties (not truly private, but protected)
  - Often used in libraries/frameworks (e.g., `Symbol.iterator`, `Symbol.toStringTag`)
- 

### ✅ Built-in Symbols (Advanced Usage)

javascript

```
const obj = {
  [Symbol.toPrimitive]() {
    return 42;
  }
};
```

```
console.log(+obj); // 💎 42
```

---



## Summary

Feature	Symbol
Type	Primitive
Unique	✓ Always unique
Use case	Hidden keys, safe extensions
Shown in loops	✗ Hidden

## 8. What's the spread and rest operator in javascript ?

### ✓ Spread Operator ...

#### ◆ Description:

"Spreads" the elements of an array/object into individual values.

#### Examples:

##### ◆ Spread in Arrays:

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];

console.log(arr2); // [1, 2, 3, 4]
```

##### ◆ Spread in Objects:

```
const user = { name: "Alice", age: 25 };
const updated = { ...user, age: 30 };

console.log(updated); // { name: 'Alice', age: 30 }
```

##### ◆ Spread in Function Calls:

```
const nums = [1, 2, 3];

function add(a, b, c) {
  return a + b + c;
}

console.log(add(...nums)); // 6
```

---

## ✓ Rest Operator ...

### ◆ Description:

"Collects" the remaining values into an array or object. Used in **function parameters** or **destructuring**.

### 🔧 Examples:

#### ◆ Rest in Functions:

javascript

```
function sum(...numbers) {  
  return numbers.reduce((a, b) => a + b, 0);  
}
```

```
console.log(sum(1, 2, 3)); // 6
```

#### ◆ Rest in Destructuring:

javascript

```
const [first, ...rest] = [10, 20, 30, 40];
```

```
console.log(first); // 10  
console.log(rest); // [20, 30, 40]
```

#### ◆ Rest in Object Destructuring:

javascript

```
const user = { name: "Bob", age: 30, city: "Delhi" };  
const { name, ...other } = user;
```

```
console.log(name); // Bob  
console.log(other); // { age: 30, city: "Delhi" }
```

---

## 📄 Summary Table

Operator	Purpose	Used In	Example
... (spread)	Expand	Arrays, Objects, Functions	<code>func(...arr)</code> or <code>{...obj}</code>
... (rest)	Collect	Function params, Destructuring	<code>function(...args)</code> or <code>[a, ...b]</code>

## 9. What is rest operator in javascript ?

see above  question 8 for answer.

## 10. What are DRY, KISS, YAGNI, SOLID Principles ?

### 1. DRY — Don't Repeat Yourself

Avoid duplicating code. Reuse logic through functions or abstraction.

#### Bad (Repeating logic)

```
console.log("Welcome, John");  
console.log("Welcome, Alice");  
console.log("Welcome, Bob");
```

#### Good (DRY)

```
function greet(name) {  
  console.log(`Welcome, ${name}`);  
}  
  
greet("John");  
greet("Alice");  
greet("Bob");
```

---

### 2. KISS — Keep It Simple, Stupid

Prefer simple, readable code. Don't overcomplicate.

#### Bad (over-engineered for a sum)

```
function complexSum(arr) {  
  return arr.reduce((acc, val) => {  
    if (typeof val === 'number') {  
      return acc + val;  
    } else {  
      return acc;  
    }  
  }, 0);  
}
```

#### Good (simple and clear)

```
function sum(arr) {  
  return arr.reduce((a, b) => a + b, 0);  
}
```

---

### 3. YAGNI — You Aren't Gonna Need It

Don't build features until they're actually needed.

✗ Bad (extra functionality never used)

```
function fetchData() {  
  // Fetching and also preparing for pagination, filtering, sorting  
  // But currently only need to get all users  
  let currentPage = 1;  
  let sortBy = 'name';  
  let filter = null;  
  
  // Unnecessary complexity  
  console.log("Fetching users with filters...");  
}
```

✓ Good (only what's needed now)

```
function fetchUsers() {  
  console.log("Fetching all users...");  
}
```

---

✓ Summary

Principle	Meaning	Tip
DRY	Don't repeat yourself	Extract functions and avoid duplicate logic
KISS	Keep it simple, stupid	Favor clarity over cleverness
YAGNI	You aren't gonna need it	Don't code features unless required

## 🧠 4. SOLID Principles

Letter	Full Form	Meaning
S	<b>Single Responsibility Principle</b>	A class or function should have <b>one reason to change</b> (do one thing well).
O	<b>Open/Closed Principle</b>	Software entities should be <b>open for extension</b> , but <b>closed for modification</b> .
L	<b>Liskov Substitution Principle</b>	Subclasses should be <b>substitutable</b> for their base classes <b>without breaking the app</b> .
I	<b>Interface Segregation Principle</b>	Don't force a class to implement <b>methods it doesn't use</b> . Prefer smaller, focused interfaces.
D	<b>Dependency Inversion Principle</b>	Depend on <b>abstractions</b> , not concrete implementations. High-level modules shouldn't rely on low-level ones.

- ♦ S – Single Responsibility Principle (SRP)

A module/class/function should have one responsibility

```
class UserRepository {
  save(user) {
    console.log(`Saving user ${user.name} to DB`);
  }
}

class EmailService {
  sendWelcome(email) {
    console.log(`Sending welcome email to ${email}`);
  }
}

const userRepo = new UserRepository();
const emailer = new EmailService();

const user = { name: "Alice", email: "alice@example.com" };
userRepo.save(user);
emailer.sendWelcome(user.email);
|
```

- ♦ O – Open/Closed Principle (OCP)

Code should be open for extension but closed for modification

```
class Discount {
  get(user) {
    return 0;
  }
}

class PremiumDiscount extends Discount {
  get(user) {
    return 20;
  }
}

function calculatePrice(user, discountStrategy) {
  const discount = discountStrategy.get(user);
  return user.total - discount;
}

const regularUser = { name: "Tom", total: 100 };
const premiumUser = { name: "Jane", total: 100 };

console.log(calculatePrice(regularUser, new Discount())); // 100
console.log(calculatePrice(premiumUser, new PremiumDiscount())); // 80
```

- ♦ L – Liskov Substitution Principle (LSP)

Subclasses should be replaceable for their base class

```
class Bird {
  move() {
    console.log("Moving...");
  }
}

class FlyingBird extends Bird {
  fly() {
    console.log("Flying...");
  }
}

class Penguin extends Bird {
  swim() {
    console.log("Swimming...");
  }
}

const pigeon = new FlyingBird();
pigeon.move();
pigeon.fly();

const penguin = new Penguin();
penguin.move();
penguin.swim();
```

- ♦ I – Interface Segregation Principle (ISP)

Clients should not be forced to implement unused interfaces (split roles)

```
const Flyer = {
  fly() {
    console.log("I can fly");
  }
};

const Swimmer = {
  swim() {
    console.log("I can swim");
  }
};

const duck = Object.assign({}, Flyer, Swimmer);
duck.fly();
duck.swim();
```

- ♦ D – Dependency Inversion Principle (DIP)

High-level modules should depend on abstractions, not low-level ones

```
// Low-level service
class FetchAPI {
  get(url) {
    return fetch(url).then(res => res.json());
  }
}

// High-level logic depends on abstraction
class UserService {
  constructor(api) {
    this.api = api;
  }

  fetchUser() {
    return this.api.get("/api/user");
  }
}

const fetchApi = new FetchAPI();
const userService = new UserService(fetchApi);
userService.fetchUser().then(console.log);
```

## 11. What is temporal dead zone ?

**TDZ is the time between** the start of a block scope and the point where a `let` or `const` variable is declared.

During this time, the variable **exists but cannot be accessed** — any attempt to do so will throw a **ReferenceError**.

### ✗ Example: Accessing before declaration

javascript

```
console.log(a); // ✗ ReferenceError: Cannot access 'a' before initialization
let a = 10;
```

Even though `a` is declared later in the code, **JavaScript knows about it during the creation phase** (hoisting) — but **doesn't initialize it until the actual line**.

### ✅ With **var** (no TDZ)

javascript

```
console.log(b); // ✅ undefined  
var b = 20;
```

**var** is **hoisted** and **initialized** as **undefined**, so no error.

---

### ✅ Accessing after declaration

javascript

```
let x = 5;  
console.log(x); // ✅ 5
```

### 🔍 Example Inside Block

javascript

```
{  
  console.log(myVar); // ❌ ReferenceError  
  let myVar = "hello";  
}
```

## 12. Different ways to create object in javascript ?

### ✅ 1. Object Literal (Most Common)

javascript

```
const person = {  
  name: "Alice",  
  age: 25,  
};
```

### ✅ 2. Using **new Object()**

javascript

```
const person = new Object();  
person.name = "Bob";  
person.age = 30;
```

### ✅ 3. Using a Constructor Function



javascript

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}  
  
const p = new Person("Charlie", 28);
```

---

#### ✓ 4. Using a Class (ES6)

javascript

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
}  
  
const p = new Person("David", 40);
```

---

#### ✓ 5. Using `Object.create()`

javascript

```
const proto = {  
  greet() {  
    console.log("Hello!");  
  }  
};  
  
const person = Object.create(proto);  
person.name = "Eva";
```

- ♦ Creates an object with a given prototype.
- 

#### ✓ 6. Using Factory Function

javascript

```
function createPerson(name, age) {  
  return {  
    name,  
    age,  
    greet() {  
      console.log(`Hi, I'm ${name}`);  
    }  
  };  
}
```

```
    }  
  };  
}  
  
const p = createPerson("Fiona", 22);
```

---

## ✅ 7. Using JSON (for parsing static data)

javascript

```
const json = '{"name":"George","age":35}';  
const person = JSON.parse(json);
```

---

## ✅ Bonus: Using ES6 Spread

javascript

```
const base = { name: "Helen", age: 26 };  
const clone = { ...base };
```

---

## 13. What's the difference between `Object.keys`, `values` and `entries`

### ✅ 1. `Object.keys(obj)`

#### ♦ Returns:

An **array of the object's own enumerable property names** (keys).

javascript

```
const user = { name: "Alice", age: 25 };  
  
console.log(Object.keys(user)); // ♦ ["name", "age"]
```

---

### ✅ 2. `Object.values(obj)`

#### ♦ Returns:

An **array of the object's own enumerable property values**.

javascript

```
console.log(Object.values(user)); // ♦ ["Alice", 25]
```

---

### ✓ 3. `Object.entries(obj)`

#### ◆ Returns:

An **array of key-value pairs** as nested arrays.

javascript

```
console.log(Object.entries(user));  
// ♦ [ ["name", "Alice"], ["age", 25] ]
```

You can loop through with:

javascript

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

---

### 🧠 Summary Table

Method	Returns	Example Output
<code>Object.keys(obj)</code>	Array of keys (strings)	<code>["name", "age"]</code>
<code>Object.values(obj)</code>	Array of values	<code>["Alice", 25]</code>
<code>Object.entries(obj)</code>	Array of <code>[key, value]</code> pairs	<code>[["name", "Alice"], ["age", 25]]</code>

---

## 14. Whats the difference between `Object.freeze()` vs `Object.seal()`

### 📦 `Object.freeze()` – Full Lock 🔒

#### ◆ What it does:

- Prevents adding new properties
- Prevents removing properties
- Prevents modifying existing properties
- Makes object completely immutable

javascript

```
const user = Object.freeze({ name: "Alice", age: 25 });

user.age = 30;      // ❌ Fails silently or throws in strict mode
user.city = "NY";   // ❌ Can't add
delete user.name;    // ❌ Can't delete

console.log(user); // ⬡ { name: "Alice", age: 25 }
```

---

## 🔧 `Object.seal()` - Partial Lock 🗝️

### ♦ What it does:

- ❌ Can't add new properties
- ❌ Can't delete existing properties
- ✅ **Can still modify** existing properties

javascript

```
const user = Object.seal({ name: "Bob", age: 30 });

user.age = 35;      // ✅ Allowed
user.city = "LA";   // ❌ Can't add
delete user.name;    // ❌ Can't delete

console.log(user); // ⬡ { name: "Bob", age: 35 }
```

---

## 15. What is a polyfill in javascript ?

A **polyfill** in JavaScript is a piece of code (usually written in JavaScript) that **implements a feature on older browsers** that **do not natively support** it.

### 🧠 Simple Definition:

A **polyfill is like a fallback** — it "fills in" the gap for **missing browser features**.

---

### ✅ Example: `Array.prototype.includes()` Polyfill

Older browsers like IE don't support `Array.includes()`, so we can write a polyfill:

javascript

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(searchElement) {
```

```
        return this.indexOf(searchElement) !== -1;
    };
}
```

Now, even in older environments:

javascript

```
console.log([1, 2, 3].includes(2)); //  true
```