Topic Javascript

🔷 1. `var`, `let`, and `const`

🔸 `var`

- **Function-scoped**: Accessible within the function where it's declared.
- **Hoisted**: Declaration is moved to the top, but not the value.
- **Can be re-declared and updated**.

```
function exampleVar() {
    console.log(a); // undefined (due to hoisting)
    var a = 10;
    console.log(a); // 10
}
exampleVar();

var a = 5;
var a = 6; // No error
console.log(a); // 6
```

🔸 `let`

- **Block-scoped**: Limited to the `{}` block.
- **Not hoisted** (or in "temporal dead zone").
- **Can be updated, not re-declared in same scope**.

```
let b = 10;
b = 20; // OK
// let b = 30; // ❌ SyntaxError in same scope

{
    let b = 40; // Different scope
    console.log(b); // 40
}
console.log(b); // 20
```

🔸 `const`

- **Block-scoped**.
- **Cannot be re-assigned**.
- **Must be initialized when declared**.

```
const c = 100;
// c = 200; // ❌ TypeError

const obj = { name: "Alice" };
obj.name = "Bob"; // ✅ Allowed (object properties can be changed)
```

```
console.log(obj.name); // Bob
```

🔷 2. JavaScript Data Types

🔸 Primitive Types

Immutable and compared by **value**:

Type	Example
`string`	`"Hello"`
`number`	`42`, `3.14`
`boolean`	`true`, `false`
`null`	`null`
`undefined`	`undefined`
`symbol`	`Symbol("id")`
`bigint`	`1234567890123456789012345678901234567890n`

```
let s = "text";         // string
let n = 42;             // number
let isReady = false;    // boolean
let u;                  // undefined
let empty = null;       // null
let sym = Symbol("id"); // symbol
let big = 123456789012345678901234567890n; // bigint
```

🔸 Non-Primitive (Reference) Types

Mutable and compared by **reference**:

Type	Example
`object`	`{ name: "Alice" }`
`array`	`[1, 2, 3]`
`function`	`function() {}`
`date`	`new Date()`

```
let obj = { key: "value" };
let arr = [1, 2, 3];
let greet = function(name) {
    return "Hello, " + name;
};
let today = new Date();
```

🔹 3. `typeof` Operator

Used to determine the type of a value.

```
console.log(typeof "text");          // string
console.log(typeof 42);              // number
console.log(typeof true);            // boolean
console.log(typeof undefined);       // undefined
console.log(typeof null);            // object (known quirk)
console.log(typeof Symbol("id"));    // symbol
console.log(typeof 123n);            // bigint

console.log(typeof {});              // object
console.log(typeof []);              // object (arrays are objects)
console.log(typeof function(){});    // function
```

> 🔍 Note: `typeof null === "object"` is a historical bug in JavaScript.

🧠 Summary Table

Feature	`var`	`let`	`const`
Scope	Function	Block	Block
Hoisting	Yes	Temporal Dead Zone	Temporal Dead Zone
Reassignable	Yes	Yes	❌ No
Redeclarable	Yes	❌ No	❌ No

Here's an **in-depth study material** for your requested JavaScript topics with detailed explanations and examples.

🔹 Conditionals: `if`, `else if`, `else`, `switch`

🔸 `if`, `else if`, `else`

Used to execute code blocks based on conditions.

```
let score = 85;

if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 75) {
    console.log("Grade: B");
} else {
    console.log("Grade: C or below");
}
```

🔸 `switch`

Used when you want to compare the same variable to multiple values.

```
let day = 3;
switch (day) {
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    default:
        console.log("Another day");
}
```

🔷 Loops

🔸 `for`

Standard counting loop.

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

🔸 `while`

Repeats while the condition is true.

```
let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

🔸 `do...while`

Executes at least once before checking condition.

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 5);
```

🔸 `for...in`

Iterates over **keys (property names)** of an object.

```
let user = { name: "Alice", age: 25 };
for (let key in user) {
    console.log(key, user[key]);
}
```

🔸 `for...of`

Iterates over **values** in an iterable like an array or string.

```
let fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
```

🔷 Break and Continue

🔸 `break`

Exits the loop immediately.

```
for (let i = 0; i < 10; i++) {
    if (i === 5) break;
    console.log(i);
}
// Output: 0 1 2 3 4
```

🔸 `continue`

Skips the current iteration.

```
for (let i = 0; i < 5; i++) {
    if (i === 2) continue;
    console.log(i);
}
// Output: 0 1 3 4
```

🔷 Browser JavaScript

Runs in browsers and manipulates HTML/CSS using the **DOM (Document Object Model)**.

Example: Changing a paragraph's text in HTML

```
<p id="demo">Original Text</p>
<button onclick="changeText()">Click Me</button>

<script>
function changeText() {
    document.getElementById("demo").innerText = "Text changed!";
}
</script>
```

DOM Methods

- `document.getElementById()`
- `document.querySelector()`
- `element.innerText` / `element.innerHTML`
- `element.style`

🔹 Node.js

- **Server-side JavaScript runtime** built on Chrome's V8 engine.
- Can run outside browsers — in servers, scripts, and backend apps.

Example: Simple HTTP server in Node.js

```
// Save this as server.js and run using: node server.js

const http = require('http');

const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello from Node.js!");
});

server.listen(3000, () => {
    console.log("Server running at http://localhost:3000");
});
```

> 💡 Node.js uses modules like `fs`, `http`, `path`, and you can install more using `npm`.

Here’s a deep dive into the **DOM (Document Object Model)** and the **BOM (Browser Object Model)**, with examples to illustrate how you can traverse, manipulate, and interact with the page and the browser itself.

🔹 1. The DOM (Document Object Model)

What is the DOM?

- A hierarchical, tree-like representation of an HTML (or XML) document.
- Every HTML element, attribute, and piece of text becomes a **Node**.
- JavaScript can traverse and change this tree at runtime.

```
<html>
  <body>
    <h1>Hello</h1>
    <p>World</p>
  </body>
</html>
```

becomes:

```
Document
└─ html
   └─ head
   └─ body
      ├─ h1  (text node: "Hello")
      └─ p   (text node: "World")
```

1.1 Accessing Elements

Method	**Returns**	**Use case**
`getElementById(id)`	Single `Element`	Quick lookup by unique `id`.
`getElementsByClassName(className)`	HTMLCollection	Live list of all matching classes.
`getElementsByTagName(tag)`	HTMLCollection	Live list of all matching tags.
`querySelector(selector)`	First matching `Element`	CSS-style selector (e.g. `"#myId .a b"`).
`querySelectorAll(selector)`	`NodeList`	Static list of all matches.

```
const box   = document.getElementById("box");
const items = document.getElementsByClassName("item");
const paras = document.querySelectorAll("article p");
const first = document.querySelector(".item.highlighted");
```

1.2 Reading & Writing Content

- **`.innerText`** Gets/sets the *rendered* text content (ignores hidden elements).

- **`.textContent`** Gets/sets *all* text, including hidden.

- **`.innerHTML`** Gets/sets a string of HTML markup.

```
<div id="greeting">
  Hello, <span style="display:none">secret</span>World!
</div>
<script>
  let el = document.getElementById("greeting");
  console.log(el.innerText);    // "Hello, World!"
  console.log(el.textContent);  // "Hello, secretWorld!"
  console.log(el.innerHTML);    // "Hello, <span…>secret</span>World!"

  // Replace with new HTML:
  el.innerHTML = "<strong>Hi!</strong>";
</script>
```

1.3 Attributes & Classes

- **`.getAttribute(name)`** / **`.setAttribute(name, value)`**
- **`.removeAttribute(name)`**
- **`.classList`**: add/remove/toggle CSS classes.

```
const img = document.querySelector("img");
let src = img.getAttribute("src");
img.setAttribute("alt", "A descriptive text");
img.classList.add("responsive");
img.classList.remove("thumbnail");
img.classList.toggle("hidden");
```

1.4 Creating & Inserting Nodes

- **`document.createElement(tagName)`**: makes a new element.
- **`node.appendChild(child)`**: adds at end.
- **`node.insertBefore(newNode, referenceNode)`**: inserts before.
- **`node.removeChild(child)`** / **`.replaceChild(newChild, oldChild)`**.

```
// Create a new list item and append it:
const ul = document.querySelector("ul");
let li = document.createElement("li");
li.innerText = "New Item";
ul.appendChild(li);

// Insert at beginning:
let first = ul.firstElementChild;
ul.insertBefore(li.cloneNode(true), first);

// Remove or replace:
ul.removeChild(ul.lastElementChild);
ul.replaceChild(li, ul.children[1]);
```

1.5 Events & Delegation

Attach behavior to elements:

- **`.addEventListener(event, handler)`**
- **`.removeEventListener(...)`**

```
const btn = document.querySelector("button#save");
btn.addEventListener("click", e => {
  alert("Saved!");
});

// Event delegation (one listener on parent):
const list = document.querySelector("ul");
list.addEventListener("click", e => {
  if (e.target.tagName === "LI") {
    console.log("You clicked", e.target.innerText);
  }
});
```

◆ 2. The BOM (Browser Object Model)

While the DOM models the **document**, the BOM models the **browser window** and its environment. All BOM properties are accessed via the global `window` object.

2.1 `window` (Global Object)

- Implicit in every script: `window.alert()` ↔ `alert()`
- Useful methods/properties:
  - `window.location`
  - `window.history`
  - `window.navigator`
  - `window.screen`
  - Timers: `setTimeout()`, `setInterval()`

```
window.alert("Hello!");        // modal alert
setTimeout(() => window.alert("Delayed"), 1000);
```

2.2 `location` — URL Management

Property	Example Use
`location.href`	get/set full URL
`location.assign(url)`	navigate to URL
`location.reload()`	reload page
`location.protocol`	"https:"
`location.host`	"example.com:8080"

Property	Example Use
`location.pathname`	"/path/page.html"
`location.search`	"?q=js&lang=en"
`location.hash`	"#section2"

```
console.log(location.href);
location.assign("https://news.example.com");
location.reload();
```

2.3 `history` — Session Navigation

Method	Effect
`history.back()`	Like browser “back” button
`history.forward()`	“forward” button
`history.go(n)`	move n steps (±)
`history.pushState(state, title, url)`	add entry
`history.replaceState(state, title, url)`	replace current

```
history.back();
history.go(-2);   // two pages back
history.pushState({page:2}, "Page 2", "/page2.html");
```

2.4 `navigator` — Browser Info

- `navigator.userAgent`
- `navigator.platform`
- `navigator.language`
- `navigator.onLine`

```
console.log(navigator.userAgent);
if (!navigator.onLine) {
  alert("You are offline");
}
```

2.5 `screen` — Screen Dimensions

- `screen.width` / `screen.height`
- `screen.availWidth` / `screen.availHeight`
- `screen.pixelDepth`

```
console.log(`Your screen is ${screen.width}×${screen.height}`);
```

📝 Key Takeaways

1. **DOM** lets you **read** and **modify** the page content and structure.
2. **BOM** lets you **interact** with the browser environment (navigation, alerts, info).
3. Use **selectors** (`getElementById`, `querySelector`…) to grab nodes, then **properties** and **methods** (`innerText`, `setAttribute`, `appendChild`) to alter them.
4. Leverage **events** to make dynamic, interactive pages.
5. Remember the global `window` object: it’s the gateway to your page’s document and to the browser itself.

Below is an in-depth guide on **JavaScript functions** and **object creation**, complete with explanations and examples.

🔹 1. Functions

JavaScript treats functions as **first-class citizens**: they can be declared, assigned to variables, passed as arguments, and returned from other functions.

1.1 Function Declaration

- **Hoisted**: Available before its definition in code.
- Syntax:

  ```
  function greet(name) {
    return "Hello " + name;
  }

  console.log(greet("Alice")); // "Hello Alice"
  ```

1.2 Function Expression

- Not hoisted: only available after the assignment.
- Can be anonymous or named.

  ```
  const greet = function(name) {
    return "Hello " + name;
  };

  console.log(greet("Bob")); // "Hello Bob"
  ```

1.3 Arrow Functions (ES6+)

- **Concise syntax**, lexical `this`.

- Implicit return when no braces.

```
const greet = name => "Hello " + name;
console.log(greet("Carol")); // "Hello Carol"

// With multiple params or multiline:
const sum = (a, b) => {
  const result = a + b;
  return result;
};
```

1.4 Higher-Order Functions

- **Pass functions as arguments** or **return functions**.

```
// 1. Passing as argument
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}
repeat(3, i => console.log("Iteration", i));
// → "Iteration 0", "Iteration 1", "Iteration 2"

// 2. Returning a function (closure)
function makeMultiplier(x) {
  return function(y) {
    return x * y;
  };
}
const double = makeMultiplier(2);
console.log(double(5)); // 10
```

🔹 2. Object Creation

JavaScript objects are collections of key–value pairs. There are several ways to create them:

2.1 Object Literal

- **Simplest**, most common.
- Ideal for one-off objects.

```
const person1 = {
  name: "Alice",
  age: 25,
  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
};

person1.greet(); // "Hi, I'm Alice"
```

2.2 `new Object()` Constructor

- Equivalent to literal but more verbose.
- Rarely used today.

```
const person2 = new Object();
person2.name = "Bob";
person2.age = 30;
person2.greet = function() {
  console.log(`Hi, I'm ${this.name}`);
};

person2.greet(); // "Hi, I'm Bob"
```

2.3 Constructor Functions (Pre-ES6 “Classes”)

- Blueprint for multiple similar objects.
- Use `new` to instantiate.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.greet = function() {
  console.log(`Hello, I'm ${this.name}`);
};

const p1 = new Person("Carol", 28);
p1.greet(); // "Hello, I'm Carol"
```

2.4 ES6 `class` Syntax

- Syntactic sugar over prototype-based constructors.
- Clearer, more familiar OOP style.

```
class PersonClass {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hey there, I'm ${this.name}`);
  }
  static species() {
    return "Homo sapiens";
  }
}

const p2 = new PersonClass("Dave", 32);
p2.greet();              // "Hey there, I'm Dave"
console.log(PersonClass.species()); // "Homo sapiens"
```

📝 Key Differences & When to Use

Creation Method	Pros	Cons
Literal	Concise, clear	No blueprint for many objects
`new Object()`	Explicit constructor	Verbose, less idiomatic
Constructor Func	Pre-ES6, flexible	Verbose prototypes
`class`	Clear OOP syntax, methods in prototype	Slight learning curve for prototype

---

✅ Next Steps / Practice

1. **Try writing**:
   - A higher-order function like `map`, `filter`, or `reduce`.
   - A small class hierarchy (e.g., `Animal` base class, `Dog` subclass).
2. **Exercises**:
   - Implement a `debounce` or `throttle` function.
   - Create an object representing a shopping cart with methods to add/remove items and calculate totals.
3. **Mini-project**:
   - Build a simple **to-do list** where tasks are objects; add methods to mark complete, delete, and render in the DOM.

Below is an in-depth look at **Constructor Functions** and **ES6 Classes**, covering how they work, how inheritance is handled, and best practices.

🔹 3. Constructor Functions

3.1 What They Are

- **Pre-ES6 “classes”** in JavaScript.
- Plain functions used as **blueprints** for objects of the same shape.
- Invoked with the `new` keyword.

3.2 How They Work

- When you call `new Person(...)`, JavaScript:
  1. Creates a new empty object.
  2. Sets that object’s internal `[[Prototype]]` to `Person.prototype`.
  3. Binds `this` inside `Person` to the new object.
  4. Executes the constructor body.
  5. Returns the object (unless you explicitly return another object).

```
function Person(name, age) {
  // `this` refers to the new object
  this.name = name;
  this.age  = age;
}

// Add shared methods on the prototype:
Person.prototype.greet = function() {
  console.log(`Hello, I’m ${this.name}`);
};

// Instantiate
const p1 = new Person("Charlie", 28);
p1.greet();                // "Hello, I’m Charlie"
console.log(p1 instanceof Person); // true
```

3.3 Prototype Chain

- **Shared methods** live on `Person.prototype`.
- Saves memory: every instance doesn’t carry its own copy of `greet`.
- You can augment or inspect the prototype:

```
console.log(Object.getPrototypeOf(p1) === Person.prototype); // true

// Add another method later:
Person.prototype.sayAge = function() {
  console.log(`I’m ${this.age} years old`);
};
p1.sayAge();               // "I’m 28 years old"
```

◆ 4. ES6 `class` Syntax

4.1 Overview

- **Syntactic sugar** over constructor functions & prototypes.
- Cleaner, more familiar “classical” OOP style.
- Supports `constructor`, instance methods, `static` methods, and inheritance via `extends`.

4.2 Defining a Class

```
class Person {
  // `constructor` runs when new instances are created
  constructor(name, age) {
    this.name = name;
    this.age  = age;
  }

  // Instance method (on prototype)
  greet() {
    console.log(`Hello, I’m ${this.name}`);
  }

  // Static method (on the class itself)
  static species() {
    return "Homo sapiens";
  }
}

const p2 = new Person("Dana", 22);
p2.greet();                   // "Hello, I’m Dana"
console.log(Person.species()); // "Homo sapiens"
```

4.3 Inheritance with `extends`

```
// Subclass of Person:
class Employee extends Person {
  constructor(name, age, role) {
    // call parent constructor
    super(name, age);
    this.role = role;
  }

  // Override or add methods:
  greet() {
    console.log(`Hi, I’m ${this.name}, the ${this.role}`);
  }
}

const e1 = new Employee("Eve", 30, "Engineer");
e1.greet();                   // "Hi, I’m Eve, the Engineer"
console.log(e1 instanceof Person);   // true
console.log(e1 instanceof Employee); // true
```

4.4 Private & Public Fields (ES2022+)

- **Public fields** declared directly in the class body.
- **Private fields** start with `#` and are inaccessible outside.

```
class Counter {
  #count = 0;        // private field
  label  = "Count";  // public field

  increment() {
    this.#count++;
    console.log(`${this.label}: ${this.#count}`);
  }
}

const c = new Counter();
c.increment();      // "Count: 1"
// console.log(c.#count); // SyntaxError
```

---

5. Comparing Approaches

Feature	Constructor Function	ES6 Class
Syntax	Function + prototype	`class` keyword

Feature	Constructor Function	ES6 Class
Inheritance	`Child.prototype = Object.create(Parent.prototype)` + `Child.prototype.constructor`	`extends` + `super()`
Static methods	`Constructor.myStatic = fn`	`static` keyword
Private fields	Simulated via closures or Symbols	`#privateField` (native support)
Readability	Less intuitive for OOP developers	Clear, concise

◆ 6. Best Practices

1. **Prefer ES6 Classes** for new code:
    - Clearer inheritance.
    - Built-in support for static methods and private fields.
2. **Use Prototype Methods** for behavior shared across instances:
    - Avoid defining methods inside the constructor; they'll be recreated per instance.
3. **Keep Constructors Lightweight**:
    - Only assign properties. Heavy logic belongs elsewhere.
4. **Use `super()` correctly**:
    - Always call `super(...)` before using `this` in subclasses.
5. **Leverage Private Fields** when you need true encapsulation.

Below is an **in-depth exploration** of **Factory Functions** and **`Object.create()`**, showing how they work, when to use them, and advanced patterns you can build on top of them.

◆ 5. Factory Functions

5.1 What Is a Factory Function?

A **factory function** is any function that **creates and returns** a new object, without requiring the `new` keyword. It's simply a function whose job is object creation.

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

const p3 = createPerson("Eva", 35);
p3.greet(); // Hi, I’m Eva
```

5.2 Why Use Factory Functions?

1. **No `new` required** You don’t have to worry about forgetting `new` (and getting `this` bound to `window` or `undefined`).

2. **Encapsulation via Closures** Variables declared in the factory scope but not put on the returned object stay *private*:

   ```
   function createCounter() {
     let count = 0;         // private
     return {
       increment() { count++; },
       get value() { return count; }
     };
   }

   const counter = createCounter();
   console.log(counter.value); // 0
   counter.increment();
   console.log(counter.value); // 1
   // `count` is not accessible from outside
   ```

3. **Flexible Composition** You can combine multiple behaviors by merging objects or copying methods at creation time (a rudimentary “mixin”):

   ```
   function withLogging(obj) {
     return {
       ...obj,
       log() {
         console.log("Logging:", obj);
       }
     };
   }

   const basic = { x: 1, y: 2 };
   const enhanced = withLogging(basic);
   enhanced.log(); // Logging: { x: 1, y: 2 }
   ```

5.3 Performance Considerations

- **Per-instance copies**: Methods declared inside the factory (like `greet` above) are recreated for every object. For many instances, consider sharing methods via a separate prototype object:

  ```
  const personMethods = {
    greet() { console.log(`Hi, I’m ${this.name}`); }
  };

  function createPersonShared(name, age) {
  ```

```
  const person = Object.create(personMethods);
  person.name = name;
  person.age  = age;
  return person;
}

const p4 = createPersonShared("Gina", 29);
p4.greet(); // Hi, I’m Gina
```

5.4 When to Choose Factory Functions

- You want **encapsulation** without classes.
- You need **multiple independent objects** that may carry private state.
- You prefer a **functional style**, composing behaviors rather than deep inheritance chains.

◆ 6. `Object.create()`

6.1 What It Does

`Object.create(proto[, descriptors])` returns a brand-new object whose **internal prototype** (`[[Prototype]]`) is set to `proto`. Optionally you can define properties via descriptor objects.

```
const proto = {
  greet() {
    console.log("Hello from prototype");
  }
};

const obj = Object.create(proto);
obj.name = "Frank";
obj.greet(); // Hello from prototype
console.log(Object.getPrototypeOf(obj) === proto); // true
```

6.2 Use Cases

1. **Pure Prototype Inheritance** Create objects that directly delegate to a shared prototype, without constructor functions or classes.

2. **Prototypal “Subclasses”** You can chain multiple levels:

```
const animal = {
  eats: true,
  walk() { console.log("Animal walks"); }
};

const rabbit = Object.create(animal);
rabbit.jump = () => console.log("Rabbit jumps");

const whiteRabbit = Object.create(rabbit);
whiteRabbit.name = "Snowball";
whiteRabbit.walk(); // Animal walks
whiteRabbit.jump(); // Rabbit jumps
```

3. **Fine-grained Property Definition** With the second argument, you can define properties with getters, setters, or control enumerability:

```
const protoDesc = {
  greet: {
    value: function() { console.log(`Hi, I’m ${this.name}`); },
    writable: true,
    enumerable: false
  },
  name: {
    value: "Gus",
    writable: true,
    enumerable: true
  }
};

const p5 = Object.create(Object.prototype, protoDesc);
p5.greet(); // Hi, I'm Gus
console.log(Object.keys(p5)); // ["name"] (greet is non-enumerable)
```

6.3 Comparing to Other Patterns

Feature	**Factory Function**	**`Object.create()`**	**Constructor/`class`**
Syntax	Any function returning an object	Single call, sets prototype	`function` + `new`, or `class`
Prototype linkage	Manual (via `Object.create` inside)	Built-in delegation	Via `.prototype` or `extends`
Encapsulation / private state	Via closures	No closures built-in	Private fields in ES2022+
Per-instance method cost	High (unless you share manually)	Low (methods on proto)	Low (methods on prototype/class)
Defining property descriptors	Must use `Object.defineProperty`	Built-in second arg	Must use defineProperty

---

🔹 7. Putting It All Together

You can **combine** these patterns to get the best of all worlds:

```
// Shared methods prototype
const personProto = {
  greet() { console.log(`Hello, I’m ${this.name}`); }
};

// Factory that uses Object.create for sharing,
// plus closure for private data
```

```
function createSecurePerson(name, age) {
  let secret = "shhh";
  const self = Object.create(personProto);
  self.name = name;
  self.age  = age;
  self.getSecret = () => secret;
  return self;
}

const sp = createSecurePerson("Hank", 40);
sp.greet();             // Hello, I’m Hank
console.log(sp.getSecret()); // shhh
```

📝 Key Takeaways

1. **Factory Functions**
   - Great for simple object creation and **private state** via closures.
   - Watch out for per-instance method duplication; share methods via prototypes if needed.
2. **`Object.create()`**
   - Directly sets up prototype delegation.
   - Ideal for **pure prototypal inheritance** and **property descriptor** control without classes.
3. **Mix and Match**
   - Combine closure-based privacy (factory) with prototype-based sharing (`Object.create`) to craft flexible, efficient objects.

Here’s an in-depth look at **ES6 class inheritance**, covering how to extend base classes, call the parent constructor, override methods, and leverage static and private features.

🔹 1. Basic Inheritance with `extends` and `super()`

When you want one class to build on another:

```
// Base class
class Person {
  constructor(name, age) {
    this.name = name;
    this.age  = age;
  }
  greet() {
    console.log(`Hello, I’m ${this.name}`);
  }
}

// Subclass
class Employee extends Person {
  constructor(name, age, role) {
    // MUST call super() before using `this`
    super(name, age);
    this.role = role;
  }
  // New method
  describe() {
    console.log(`${this.name} is a ${this.role}`);
  }
}

const emp = new Employee("Grace", 29, "Designer");
emp.greet();      // Hello, I'm Grace    (inherited)
emp.describe();   // Grace is a Designer
console.log(emp instanceof Person);   // true
console.log(emp instanceof Employee); // true
```

Key Points

- **`extends Person`** sets up the prototype chain: `Employee.prototype.__proto__ === Person.prototype`.
- **`super(name, age)`** calls the parent's constructor, initializing `name` and `age`.
- You **cannot** reference `this` in a subclass before calling `super()`.

◆ 2. Overriding Methods & Calling Parent Methods

You can override and still call the original via `super.method()`:

```
class Employee extends Person {
  constructor(name, age, role) {
    super(name, age);
    this.role = role;
  }
  // Override greet()
  greet() {
    // call Person.greet()
    super.greet();
    console.log(`I work as a ${this.role}`);
  }
}

const emp2 = new Employee("Hank", 35, "Engineer");
emp2.greet();
// → Hello, I'm Hank
// → I work as a Engineer
```

◆ 3. Static Methods & Properties in Inheritance

Static members live on the class itself—not on instances.

```
class Person {
  static species = "Homo sapiens";
  static info() {
    console.log(`Species: ${this.species}`);
  }
}

class Employee extends Person {
  static company = "Acme Corp";
  static info() {
    // `this` here refers to the subclass
    super.info();
    console.log(`Company: ${this.company}`);
  }
}

Person.info();   // Species: Homo sapiens
Employee.info(); // Species: Homo sapiens
                 // Company: Acme Corp
```

4. Private (#) and Protected Patterns

4.1 Private Fields

ES2022 lets you declare true private fields, inherited but inaccessible outside.

```
class Person {
  #ssn;
  constructor(name, age, ssn) {
    this.name = name;
    this.age  = age;
    this.#ssn = ssn;
  }
  getSSN() {
    return this.#ssn;
  }
}

class Employee extends Person {
  constructor(name, age, ssn, role) {
    super(name, age, ssn);
    this.role = role;
  }
  showInfo() {
    console.log(`${this.name}, SSN: ${this.getSSN()}, Role: ${this.role}`);
  }
}

const e = new Employee("Ivy", 40, "123-45-6789", "Manager");
e.showInfo();
// Ivy, SSN: 123-45-6789, Role: Manager
// e.#ssn → SyntaxError (truly private)
```

4.2 “Protected” by Convention

You can prefix with _ to signal “protected” (accessible in subclasses but by convention not outside):

```
class Base {
  constructor() {
    this._internal = 42;
  }
}

class Sub extends Base {
  logInternal() {
    console.log(this._internal);
  }
}
```

🔹 5. Mixins for Multiple Behaviors

JavaScript doesn’t support multiple inheritance directly, but you can compose classes with mixins:

```
const CanEat = Base => class extends Base {
  eat() { console.log(`${this.name} is eating`); }
};
const CanWalk = Base => class extends Base {
  walk() { console.log(`${this.name} is walking`); }
};

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

class Robot {
  constructor(id) { this.id = id; }
}

// Compose Person + eating + walking
class Human extends CanWalk(CanEat(Person)) {}

const h = new Human("Jack");
h.eat();  // Jack is eating
h.walk(); // Jack is walking
```

🔹 6. Checking Prototypes & Inheritance Chains

- **`instanceof`** checks the prototype chain.
- **`Object.getPrototypeOf(obj)`** reveals the immediate prototype.

```
console.log(emp instanceof Employee); // true
console.log(emp instanceof Person);   // true
console.log(Object.getPrototypeOf(emp) === Employee.prototype); // true
console.log(Object.getPrototypeOf(Employee.prototype) ===
Person.prototype); // true
```

📝 Best Practices

1. **Favor ES6 classes** for clear, maintainable OOP patterns.

2. **Always call `super()`** before using `this` in subclasses.
3. **Use private (`#`) fields** for true encapsulation when needed.
4. **Override responsibly**—use `super.method()` if you need the parent behavior too.
5. **Consider mixins** for composing orthogonal behaviors rather than deep hierarchies.
