# ES6 Tutorial: ES6 New Enhancements

## Agenda

0. Setup

1. var hoisting and function scope

2. const in ES6

3. Arrow function, functional programming in JS

4. Default value to function arguments

5. Rest operation in ES6 (aka variable argument methods in Java)

6. Spread operator in ES6

7. Spread operator with object literals

8. Destructuring array

9. Destructuring objects

10. String templates

11. for...of loop (used with iterables)

12. ES6 Classes

- Types of functions in a class

- Class inheritance

13. Set and Map in ES6

# 🧠 ES6 Tutorial – Topic 1

## 🧾 var, let, const — Scope, Hoisting, and Redeclaration

### ➤ Variable Declarations in JS

* Traditionally, JavaScript used var to declare variables.
* ES6 introduced let and const to overcome the **scoping and hoisting issues** of var.

### 🔍 Problem with var — Scoping Issue

#### 🔸 JavaScript uses ****function-level scope**** for var, unlike C/Java/C++ which are ****block-scoped****.

#### Example 1:

var flag = true;

if (flag) {

var fname = "rajeev";

}

console.log(fname); // ✅ Accessible — not block scoped

🧠 Our understanding from C/Java is that fname should be restricted to the block, but not in JS using var.

#### Example 2:

for (var i = 0; i < 10; i++) {

var fname = "rajeev";

console.log(fname + ": " + i);

}

console.log(i); // ✅ Still accessible

* var does not respect block scope.
* i leaks out of the loop block.

✅ **Conclusion**: var is **function-scoped**, not block-scoped.

### ✅ Advantage of let

* let introduces **block scope** — just like C/Java.
* Variables declared inside {} are not accessible outside.

if (flag) {

let fname = "rajeev";

console.log(fname); // ✅ Works

}

console.log(fname); // ❌ ReferenceError

### 🌀 Hoisting Issue with var

#### Example:

console.log(x); // Output: undefined

var x = 33;

* JavaScript **hoists** var declarations to the top (declaration only, not assignment).
* So, above code is interpreted as:

js

CopyEdit

var x;

console.log(x); // undefined

x = 33;

### ❌ let is ****not hoisted**** like var

console.log(x);

let x = 33; // ❌ ReferenceError: Cannot access 'x' before initialization

✅ Technically, let **is hoisted**, but it is in a **Temporal Dead Zone (TDZ)** from start of block to declaration line.

### 📛 Redeclaring variables

#### With var:

var greeting = "good morning";

var greeting = "good evening"; // ✅ No error

console.log(greeting); // Output: "good evening"

🧨 This may cause bugs when accidentally re-declaring variables.

#### With let:

let greeting = "good morning";

let greeting = "good evening"; // ❌ SyntaxError: Identifier has already been declared

✅ Safer, avoids silent bugs.

### ✅ Summary Table – var vs let

| Feature | var | let |
| --- | --- | --- |
| Scope | Function-scoped | Block-scoped |
| Hoisting | Yes (initialized as undefined) | Yes (in TDZ) |
| Redeclaration | Allowed | ❌ Not allowed |
| Use in Loops | Variable leaks outside | Confined to block |

### 📌 Example – var vs let inside a function

#### Using var:

function greetPerson(name) {

if (name === "rajeev") {

var greet = "hello programmer";

} else {

var greet = "hello person";

}

console.log(greet); // ✅ Works

}

✅ var can be redeclared, and the declaration is hoisted to the top of the function.

#### Hoisting example with var:

function greetPerson(name) {

if (name === "rajeev") {

greet = "hello programmer";

} else {

greet = "hello person";

}

var greet;

console.log(greet); // ✅ Works due to hoisting

}

#### Using let:

function greetPerson(name) {

if (name === "rajeev") {

let greet = "hello programmer";

} else {

let greet = "hello person";

}

console.log(greet); // ❌ ReferenceError

}

#### ✅ Solution:

function greetPerson(name) {

let greet;

if (name === "rajeev") {

greet = "hello programmer";

} else {

greet = "hello person";

}

console.log(greet); // ✅ Works

}

#### ❌ Wrong usage of let (temporal dead zone):

function greetPerson(name) {

if (name === "rajeev") {

greet = "hello programmer";

} else {

greet = "hello person";

}

console.log(greet);

let greet; // ❌ ReferenceError

}

### 🧪 What will this output?

var a = 1;

var b = 10;

if (a == 1) {

var a = 10;

let b = 20;

console.log(a); // 10

console.log(b); // 20

}

console.log(a); // 10 (because var is function scoped)

console.log(b); // 10 (original b, block b does not leak)

### 🎯 Redeclaration in summary

var a = 1;

var a = 10; // ✅ OK

let x = 1;

let x = 10; // ❌ SyntaxError

### 💡 Loop Variable Example

for (var i = 0; i < 10; i++) {

console.log(i);

}

console.log("=====" + i); // ✅ i is accessible

for (let i = 0; i < 10; i++) {

console.log(i);

}

console.log("=====" + i); // ❌ ReferenceError

### 🔒 const in ES6 – Immutable Binding

* Like final in Java or const in C++
* Must be initialized during declaration
* Cannot be reassigned
* For objects and arrays, reference is constant, values can change

const pi = 3.1415;

pi = 3.14; // ❌ TypeError

#### const with objects

const obj1 = { name: "raj" };

obj1.name = "rajeev"; // ✅ Allowed

obj1 = { name: "ravi" }; // ❌ TypeError

### ✅ Summary: const vs let vs var

| Feature | var | let | const |
| --- | --- | --- | --- |
| Scope | Function | Block | Block |
| Hoisting | Yes | Yes (TDZ) | Yes (TDZ) |
| Reassignable | Yes | Yes | ❌ No |
| Redeclarable | Yes | ❌ No | ❌ No |
| Must initialize? | No | No | ✅ Yes |

# 🧠 ES6 Tutorial – Topic 2

## 🔒 const in ES6 – Constant Declarations and Object Behavior

### 🔹 const = Constant Binding

* const creates **block-scoped** variables **just like** let, but **you cannot reassign** them.
* Think of it as similar to final in Java or const in C++.

### 🔹 Basic Example

const pi = 3.1415;

console.log(pi);

pi = 3.14; // ❌ TypeError: Assignment to constant variable

* ✅ pi is **read-only** and **must be initialized during declaration**.
* ❌ Cannot be re-declared or reassigned.

### 📌 Important Note

* const **does not make objects immutable**, it just makes the reference to the object constant.
* You can still **mutate** the internal properties.

### 🔹 Example: const with Objects

const obj1 = {

name: "raj"

};

obj1.name = "rajeev"; // ✅ Allowed

console.log(obj1.name); // "rajeev"

#### ❌ Trying to reassign the object reference:

obj1 = {

name: "ravi"

};

// ❌ TypeError: Assignment to constant variable

✅ You **can change** the **contents** of the object but **cannot reassign** the object reference.

### 🔹 Example: const in Arrays

const colors = ["Red", "Blue"];

colors.push("Green"); // ✅ Works

console.log(colors); // ["Red", "Blue", "Green"]

colors = ["Yellow"]; // ❌ TypeError

✅ Arrays declared with const can still be mutated.

### 🔍 Example Recap – let vs const mutation

let num = 1;

const num2 = 10;

num2 = 33; // ❌ TypeError: Assignment to constant variable

### 📌 Common Mistake

const ≠ deeply frozen objects.

To make an object completely immutable (deep freeze), you need:

Object.freeze(obj1); // Makes the object immutable (shallow freeze)

But it won't freeze nested objects. For deep freeze, you'd need a utility like:

function deepFreeze(obj) {

Object.freeze(obj);

Object.keys(obj).forEach(key => {

if (typeof obj[key] === 'object' && !Object.isFrozen(obj[key])) {

deepFreeze(obj[key]);

}

});

}

### ✅ Summary: When to use const?

* Use const **by default** for all variables that don’t need reassignment.
* Use let **only when** reassignment is necessary.
* Avoid var in ES6+ code.

### ✅ Final Notes:

| Feature | const |
| --- | --- |
| Scope | Block |
| Hoisting | Yes (in TDZ) |
| Must initialize | ✅ Yes |
| Redeclaration allowed? | ❌ No |
| Reassignment allowed? | ❌ No |
| Object mutation allowed? | ✅ Yes (but reference is fixed) |

# 🧠 ES6 Tutorial – Topic 3

## 🚀 Arrow Functions & Functional Programming in JavaScript

### 🔷 Arrow Functions – The Shorthand Syntax

ES6 introduced **arrow functions** as a more concise way to write function expressions.

#### 💡 Traditional function:

var a = function() {

return 10;

}

#### 💡 ES6 Arrow function:

var b = () => 10;

✅ Implicit return is allowed when there's only one expression (no {} needed).

#### 💡 Another example:

const adder = (a, b) => a + b;

### 🔸 Benefits of Arrow Functions

1. **Concise syntax**
2. **Implicit return (no need for return keyword)**
3. **No own** this **binding** (lexical this)
4. Ideal for **callback functions** and **functional programming**

### 🧪 Console Output

console.log(b); // b is a function, prints the function body

## 🔁 Functional Programming with JavaScript

Functional programming = Writing code using **pure functions**, **immutability**, and **data transformations**.

ES6 arrow functions make this style easier.

### 📊 Dataset Example:

const companies = [

{name: "Company One", category: "Finance", start: 1981, end: 2003},

{name: "Company Two", category: "Retail", start: 1992, end: 2008},

{name: "Company Three", category: "Auto", start: 1999, end: 2007},

{name: "Company Four", category: "Retail", start: 1989, end: 2010},

{name: "Company Five", category: "Technology", start: 2009, end: 2014},

{name: "Company Six", category: "Finance", start: 1987, end: 2010},

{name: "Company Seven", category: "Auto", start: 1986, end: 1996},

{name: "Company Eight", category: "Technology", start: 2011, end: 2016},

{name: "Company Nine", category: "Retail", start: 1981, end: 1989}

];

const ages = [33, 12, 20, 16, 5, 54, 21, 44, 61, 13, 15, 45, 25, 64, 32];

### 📍 forEach – Print all companies

#### 🔹 Traditional:

for(let i = 0; i < companies.length; i++) {

console.log(companies[i]);

}

#### 🔹 Functional:

companies.forEach(function(company) {

console.log(company);

});

#### 🔹 With Arrow:

companies.forEach(company => console.log(company));

### ✅ filter() – Returns a subset of data

#### 🔹 Ages 21 and above

const selectedAges = ages.filter(function(age) {

if(age >= 21) {

return true;

}

});

const selectedAges = ages.filter(age => age >= 21);

console.log(selectedAges);

#### 🔹 Filter Retail Companies

const retailCompanies = companies.filter(function(company) {

return company.category === "Retail";

});

const retailCompanies = companies.filter(company => company.category === "Retail");

console.log(retailCompanies);

#### 🔹 Companies from 1980s

const companies80 = companies.filter(company =>

company.start >= 1980 && company.start <= 1990

);

console.log(companies80);

#### 🔹 Companies lasting 10+ years

const companiesMore10 = companies.filter(company =>

(company.end - company.start) >= 10

);

console.table(companiesMore10);

### 🔁 map() – Transform data into new arrays

#### 🔹 Just company names

const companyNameArr = companies.map(company => company.name);

console.log(companyNameArr);

#### 🔹 Company name with duration

const companyNameArr2 = companies.map(company =>

`${company.name} [${company.start}--${company.end}]`

);

console.table(companyNameArr2);

### 🔀 sort() – Sort elements

#### 🔹 By start year

const sortedCompanies = companies.sort((c1, c2) =>

c1.start > c2.start ? 1 : -1

);

console.table(sortedCompanies);

### 🧮 Bonus: reduce()

You didn’t add it, but it fits here perfectly:

#### 🔹 Sum of all ages:

const ageSum = ages.reduce((total, age) => total + age, 0);

console.log(ageSum);

### ✅ Summary Table: Functional Utilities

| Method | Use Case |
| --- | --- |
| forEach | Looping through items |
| filter | Getting a subset |
| map | Transforming elements |
| sort | Ordering elements |
| reduce | Aggregating to single value |

# 🧠 ES6 Tutorial – Topic 4

## 🧩 Default Function Parameters in ES6

### 🔷 What are Default Parameters?

ES6 allows **default values** for function parameters, similar to Java and C++.

This makes your function definitions **more flexible and safer** by reducing the need to check for undefined inside the function.

### 🔸 Without Default Parameters

let getValue = function(a) {

console.log(a);

}

getValue(); // undefined

getValue(5); // 5

You had to manually assign defaults:

let getValue = function(a) {

a = a || 10;

console.log(a);

}

### 🔸 With ES6 Default Parameters

let getValue = function(a = 10) {

console.log(a);

}

getValue(); // 10

getValue(5); // 5

✅ Cleaner and more readable code.

### 🔸 Multiple Parameters with Defaults

let getValue = function(a = 10, b = 4) {

console.log(a, b);

}

getValue(); // 10 4

getValue(20); // 20 4

getValue(undefined, 12); // 10 12

⚠️ If you skip a parameter, use undefined explicitly to trigger the default.

### 💡 Use Cases

* Optional arguments
* Safer API design
* Cleaner fallback logic

### 🔍 Summary Table: Function Defaults

| Case | Output |
| --- | --- |
| getValue() | 10 4 |
| getValue(5) | 5 4 |
| getValue(undefined, 12) | 10 12 |
| getValue(7, undefined) | 7 4 |

### ✅ Best Practices

* Always put default parameters **after** non-default ones.
* Combine with **rest** or **destructuring** for powerful patterns.

function createUser({name = "Guest", role = "viewer"} = {}) {

console.log(name, role);

}

createUser(); // Guest viewer

# 🧠 ES6 Tutorial – Topic 5

## 🔁 Rest Operator (...args) – Variable Arguments in ES6

### 🔷 What is the Rest Operator?

The **rest operator** (...) allows you to represent an **indefinite number of arguments** as an array.

✅ Equivalent to:

* varargs in Java
* \*args in Python
* But **cleaner and safer** in JS

### 🔸 Pre-ES6: arguments Object

let displayColor = function() {

console.log(message);

for (let i in arguments) {

console.log(arguments[i]);

}

}

let message = "list of colors";

displayColor(message, "red", "black", "blue");

#### 🔻 Problems with arguments:

* Not a real array (can't map/filter easily)
* Doesn't play well with arrow functions
* Less readable

### 🔸 ES6 Rest Parameters: ...colors

let displayColors = function(message, ...colors) {

console.log(message);

console.log(colors); // ✅ An actual array!

for (let i in colors) {

console.log(colors[i]);

}

}

let message = "List of Colors";

displayColors(message, 'Red'); // Red

displayColors(message, 'Red', 'Blue'); // Red Blue

displayColors(message, 'Red', 'Blue', 'Green'); // Red Blue Green

✅ Now colors is an array holding all extra arguments.

### 🧪 Internal Behavior:

// Behind the scenes:

function show(...args) {

console.log(args); // all extra args as array

}

show(1, 2, 3); // [1, 2, 3]

### ⚠️ Only One Rest Parameter Per Function

function sum(a, ...nums, b) {

// ❌ SyntaxError: Rest parameter must be last

}

✅ The rest parameter must be the **last** one.

### 🔍 Summary Table: arguments vs ...rest

| Feature | arguments | ...rest |
| --- | --- | --- |
| Type | Array-like object | Actual Array |
| Works in arrow funcs | ❌ No | ✅ Yes |
| Easy to manipulate | ❌ No | ✅ Yes |
| Destructuring ready | ❌ No | ✅ Yes |

### ✅ Practical Use: Variadic Utility

const addAll = (...nums) => {

return nums.reduce((acc, n) => acc + n, 0);

}

console.log(addAll(1, 2, 3, 4)); // 10

# 🧠 ES6 Tutorial – Topic 6

## 📤 Spread Operator (...) in ES6

### 🔷 What is the Spread Operator?

The **spread operator** (...) **spreads** the elements of an array (or object) into **individual items**.

✅ Use cases:

* Pass array elements as individual arguments
* Clone or merge arrays/objects
* Expand iterable elements

### 🔸 Problem Without Spread

let colorArray = ['Orange', 'Yellow', 'Indigo'];

displayColors(message, colorArray);

// ❌ Passes entire array as one argument

### 🔸 Solution: Spread Operator

let colorArray = ['Orange', 'Yellow', 'Indigo'];

displayColors(message, ...colorArray);

// ✅ Passes each color as a separate argument

### 🧪 Example: Array Expansion

let arr1 = [1, 2, 3];

let arr2 = [...arr1, 4, 5];

console.log(arr2); // [1, 2, 3, 4, 5]

### 🔸 Copying Arrays

const original = [10, 20, 30];

const copy = [...original];

console.log(copy); // [10, 20, 30]

✅ Creates a **shallow copy**

### 🔸 Combining Arrays

const nums1 = [1, 2];

const nums2 = [3, 4];

const merged = [...nums1, ...nums2];

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

### 🔸 Spread in Function Calls

function add(a, b, c) {

return a + b + c;

}

let nums = [1, 2, 3];

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

### ⚠️ Spread ≠ Rest

| Feature | Spread | Rest |
| --- | --- | --- |
| Use case | Expanding elements | Gathering elements |
| Used in | Function call, literals, arrays | Function definition |
| Syntax | ...iterable | ...args |

### 🔍 Summary Table: Use Cases of Spread

| Use Case | Example |
| --- | --- |
| Function args | func(...arr) |
| Array clone | let copy = [...arr] |
| Array merge | let merged = [...a1, ...a2] |
| Object clone | let copy = {...obj} |
| Object merge | let merged = {...obj1, ...obj2} |

# 🧠 ES6 Tutorial – Topic 7

## 🧱 Spread Operator with Object Literals

### 🔷 Object Literals Before ES6

You had to **manually map variables to properties**:

let firstname = "rajeev";

let lastname = "gupta";

let person = {

firstname: firstname,

lastname: lastname

};

console.log(person.firstname); // rajeev

### 🔸 ES6 Enhancement – Property Shorthand

If **key and variable names** are the same, you can **omit** the key.

let firstname = "rajeev";

let lastname = "gupta";

let person = {

firstname,

lastname

};

console.log(person.firstname); // rajeev

console.log(person.lastname); // gupta

### 🔸 Returning Object Literals from Functions

function createPerson(firstname, lastname, age) {

let fullname = firstname + " " + lastname;

return { firstname, lastname, fullname };

}

let p = createPerson("rajeev", "gupta", 62);

console.log(p.firstname); // rajeev

console.log(p.fullname); // rajeev gupta

### 🔸 Function Shorthand in Objects

Define methods without the function keyword:

function createPerson(firstname, lastname, age) {

let fullname = firstname + " " + lastname;

return {

firstname,

lastname,

fullname,

isSenior() {

return age > 60;

}

};

}

let p = createPerson("rajeev", "gupta", 62);

console.log(p.isSenior()); // true

### 🔸 Spread Operator with Objects

✅ Introduced in ES2018 (still considered ES6+).

#### 🔸 Cloning an Object

const user1 = {

name: "Rajeev",

role: "Trainer"

};

const user2 = { ...user1 };

console.log(user2); // { name: "Rajeev", role: "Trainer" }

#### 🔸 Merging Objects

const obj1 = { a: 1 };

const obj2 = { b: 2 };

const merged = { ...obj1, ...obj2 };

console.log(merged); // { a: 1, b: 2 }

#### 🔸 Overriding Properties

const base = { role: "user", active: true };

const override = { role: "admin" };

const updated = { ...base, ...override };

console.log(updated); // { role: "admin", active: true }

⚠️ Order matters: properties in later objects **override** earlier ones.

### 🔍 Summary Table: Object Spread vs Array Spread

| Use Case | Array Spread | Object Spread |
| --- | --- | --- |
| Clone | [...arr] | {...obj} |
| Merge | [...a1, ...a2] | {...o1, ...o2} |
| Override | N/A | {...defaults, ...custom} |
| Shorthand Props | Not Applicable | { key } for { key: key } |

# 🧠 ES6 Tutorial – Topic 8

## 🎯 Destructuring Arrays

### 🔷 What is Destructuring?

Destructuring allows you to **unpack values** from arrays (or objects) into distinct variables — like tuple unpacking in Python or pattern matching in Scala.

### 🔸 Simple Array Destructuring

let employee = ["rajeev", "gupta", "Male"];

let [fname, lname, gender] = employee;

console.log(fname); // rajeev

console.log(lname); // gupta

console.log(gender); // Male

✅ Unpacks values **in order** into variables.

### 🔸 Default Values in Destructuring

What if a value is missing from the array?

let employee = ["rajeev", "gupta"];

let [fname, lname, gender = "Male"] = employee;

console.log(gender); // Male (default)

### 🔸 Skipping Elements

You can skip unwanted values using commas:

let employee = ["rajeev", "gupta", "Male"];

let [, , gender] = employee;

console.log(gender); // Male

### 🔸 Collecting Remaining Elements (Rest)

Use the **rest operator** to gather remaining elements into an array:

let employee = ["rajeev", "gupta", "Male", "Trainer", "Delhi"];

let [fname, ...rest] = employee;

console.log(fname); // rajeev

console.log(rest); // [ 'gupta', 'Male', 'Trainer', 'Delhi' ]

✅ Very useful in cases where only the first few elements matter.

### 🔍 Summary Table: Array Destructuring

| Pattern | Result |
| --- | --- |
| [a, b] = [1, 2] | a = 1, b = 2 |
| [a, , b] = [1, 2, 3] | skip 2nd value → a = 1, b = 3 |
| [a, b = 5] = [1] | a = 1, b = 5 (default) |
| [...rest] = [1,2,3] | rest = [1,2,3] |
| [x, ...rest] = [10, 20, 30] | x = 10, rest = [20, 30] |

### 🔔 Best Practices

* Use default values to prevent undefined.
* Combine with rest operator for flexible assignments.
* Always respect order when destructuring arrays.

# 🧠 ES6 Tutorial – Topic 9

## 🧩 Object Destructuring

### 🔷 What is Object Destructuring?

Object destructuring is a convenient way to extract multiple properties from an object and assign them to variables.

### 🟠 Original Way (Before Destructuring)

const msg = {

name: "rajeev gupta",

desi: "trainer",

hobby: "traveling",

social: {

twitter: "@rajeev\_gupta76",

facebook: "https://www.facebook.com/profile.php?id=100021806671318"

}

};

// Traditional way

const name = msg.name;

const desi = msg.desi;

const hobby = msg.hobby;

const twitter = msg.social.twitter;

console.log(name);

console.log(hobby);

console.log(twitter);

🔻 Problem: Code is repetitive and hard to read.

### 🔵 ES6 Object Destructuring (Cleaner)

const { name, desi, hobby, social } = msg;

console.log(name); // rajeev gupta

console.log(hobby); // traveling

console.log(social); // entire nested object

### 🔸 Nested Destructuring

You can destructure nested objects like social:

const {

name,

desi,

hobby,

social: { twitter, facebook }

} = msg;

console.log(twitter); // @rajeev\_gupta76

console.log(facebook); // https://facebook.com/...

### 🔸 Renaming Variables

Assign properties to different variable names:

const { name: fullName, desi: role } = msg;

console.log(fullName); // rajeev gupta

console.log(role); // trainer

### 🔸 Providing Default Values

const { company = "Busy Coder Academy" } = msg;

console.log(company); // Busy Coder Academy (default fallback)

### 🔍 Summary Table: Object Destructuring Features

| Feature | Syntax Example |
| --- | --- |
| Basic destructuring | const { name } = obj; |
| Nested destructuring | const { social: { twitter } } = obj; |
| Renaming | const { name: fullName } = obj; |
| Default values | const { age = 30 } = obj; |
| Skipping properties | Just omit them in the destructuring assignment |

# 🧠 ES6 Tutorial – Topic 10

## ✨ Template Literals (aka String Templates)

### 🔷 Problem with Old-Style String Concatenation (ES5)

const person = {

name: "rajeev gupta",

address: "delhi",

phone: "43544344444"

};

let strMsg = "my name is " + person.name + ": " + " my address is " + person.address;

😞 Hard to read and maintain. Breaks easily when adding variables or formatting.

### ✅ ES6 Template Literals – Backtick Syntax

let strMsg2 = `my name is ${person.name} and my address is ${person.address}`;

console.log(strMsg2);

### ✨ Multiline String Support (No \n Needed)

const strMsg3 = `

my name is ${person.name}

my address is ${person.address}

my phone is ${person.phone}

`;

console.log(strMsg3);

✅ Automatically preserves formatting, tabs, line breaks — **no** \n **or** \t **needed**.

### 🔸 Template Literal Can Call a Function Too

function bio(x) {

console.log(x);

}

const person = {

name: "rajeev gupta",

address: "delhi",

phone: "43544344444"

};

bio `

my name is ${person.name}

my address is ${person.address}

my phone is ${person.phone}

`;

✅ This works because **template literals can be tagged**, where the function (bio) receives the literal strings and expressions separately.

### 🔍 Summary Table: Template Literal Features

| Feature | ES5 | ES6 Template Literal |
| --- | --- | --- |
| String concatenation | "Hi " + name | `Hi ${name}` |
| Multiline strings | 'line1\nline2' | `line1\nline2` or raw lines |
| Embedded expressions | ❌ not possible | ✅ with ${} |
| Function tagging | ❌ | ✅ tag\Hello ${name}`` |

# 🧠 ES6 Tutorial – Topic 11

## 🔁 for...of Loop: Used with Iterables

### 🔷 Problem with for...in Loop (ES5)

let colors = ['Red', 'Blue', 'Green'];

for (let index in colors) {

console.log(colors[index]);

}

🔸 for...in is meant for **enumerating object keys**, not array elements.  
🔸 It iterates over **enumerable properties**, which may include inherited ones.

### ✅ ES6 Solution: for...of

Introduced in ES6, for...of is the right loop for **iterables** like arrays, strings, sets, maps, etc.

let colors = ['Red', 'Blue', 'Green'];

for (let color of colors) {

console.log(color);

}

✅ Cleaner  
✅ No indexing needed  
✅ Works only on actual values, not keys or indexes

### 🧵 Works on Strings Too

let letters = "ABC";

for (let letter of letters) {

console.log(letter);

}

Output:

A

B

C

### ⚠️ Difference Between for...in vs for...of

| Loop Type | Use Case | Iterates Over | Example |
| --- | --- | --- | --- |
| for...in | Object keys | Enumerable property keys | Objects (not arrays) |
| for...of | Iterable values | Values (from iterables) | Arrays, Strings, Maps, etc |

### 💡 Bonus: What are Iterables?

Iterables in JS include:

* Arrays
* Strings
* Maps
* Sets
* arguments object
* DOM collections (like NodeList)

You can use for...of on **any iterable object**.

# 🧠 ES6 Tutorial – Topic 12

## 🧱 ES6 Classes

### 🔷 What are JavaScript Classes?

* JavaScript classes are **syntactic sugar** over JavaScript's existing **prototype-based inheritance**.
* Before ES6, JS did not have a formal class keyword — developers used **constructor functions** and **prototypes**.
* ES6 introduced the class keyword to make OOP-style development easier and more familiar (especially for Java/ C# developers).

### 🔸 Basic Class Example

class Person {

greet() {}

}

let p = new Person();

console.log(p.greet === Person.prototype.greet); // true

✅ Under the hood, **classes are just functions**.  
❌ Classes are **not hoisted** like regular functions.

### 🔸 Class Hoisting Behavior

employee(); // Works (function hoisting)

function employee() {}

let p1 = new Employee(); // ❌ ReferenceError

class Employee {}

🔻 Function declarations are hoisted, **but class declarations are not**.

### 🟠 Prior to ES6: Constructor Function + Prototype

function Animal(type) {

this.type = type;

}

Animal.prototype.identify = function () {

console.log(this.type);

};

var cat = new Animal('Cat');

cat.identify(); // Cat

### ✅ Same in ES6 Using class

class Animal {

constructor(type) {

this.type = type;

}

identify() {

console.log(this.type);

}

}

let cat = new Animal('Cat');

cat.identify(); // Cat

typeof Animal is still "function" — just syntactic sugar!

console.log(typeof Animal); // function

### ⚠️ Class vs Custom Type Differences

| Behavior | Function Constructor | ES6 Class |
| --- | --- | --- |
| Hoisted | ✅ Yes | ❌ No |
| Called without new | ✅ Yes | ❌ Error |
| Constructor function syntax | function X() | class X {} |
| Prototype method definition | Explicit | Built-in with {} block |

js

CopyEdit

let dog = new Animal('Dog'); // Works

let duck = Animal('Duck'); // ❌ Error: must use 'new'

### 🔹 JavaScript Class Expressions

let Animal = class {

constructor(type) {

this.type = type;

}

identify() {

console.log(this.type);

}

};

let duck = new Animal('Duck');

console.log(duck instanceof Animal); // true

console.log(duck instanceof Object); // true

console.log(typeof Animal); // function

✅ Classes can be anonymous and used in expressions — just like functions.

### 🔹 Getter and Setter in Classes

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

get fullName() {

return this.firstName + ' ' + this.lastName;

}

set fullName(str) {

let names = str.split(' ');

if (names.length === 2) {

this.firstName = names[0];

this.lastName = names[1];

} else {

throw 'Invalid name format';

}

}

}

let mary = new Person('rajeev', 'Gupta');

console.log(mary.fullName); // rajeev Gupta

mary.fullName = 'Rajeev Gupta';

console.log(mary.fullName); // Rajeev Gupta

### 🔹 Static Methods

class Animal {

constructor(type) {

this.type = type;

}

identify() {

console.log(this.type);

}

static create(type) {

return new Animal(type);

}

}

var mouse = Animal.create('Mouse');

mouse.identify(); // Mouse

// mouse.create('Monkey'); // ❌ Error: mouse.create is not a function

🔸 Static methods are called on the class, not on instances.

## 🔷 Inheritance in ES6

JavaScript uses **prototype inheritance**, but ES6 class simplifies it.

### 🔹 Basic Inheritance Example

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log('walking on ' + this.legs + ' legs');

}

}

class Bird extends Animal {

constructor(legs) {

super(legs); // must call parent constructor

}

fly() {

console.log('flying');

}

}

let bird = new Bird(2);

bird.walk(); // walking on 2 legs

bird.fly(); // flying

### 🔹 Inheritance with Additional Properties

class Bird extends Animal {

constructor(legs, color) {

super(legs);

this.color = color;

}

fly() {

console.log('flying');

}

getColor() {

console.log(this.color);

}

}

let pigeon = new Bird(2, 'white');

console.log(pigeon.getColor()); // white

### 🔹 Shadowing Methods (Method Overriding)

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk(); // go walking

### 🔹 Calling Super Method (Base Class Method)

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

super.walk(); // base class method

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk();

// Output:

// walking on 4 legs

// go walking

### 🔷 Recap: Why Classes in ES6?

✅ Makes JavaScript look more like class-based languages (Java/C#)  
✅ Easier for OOP devs to transition  
✅ Encourages clean and modular code  
✅ Adds structure to prototype inheritance

# 🧠 ES6 Tutorial – Topic 13

## 🔗 Set, Map, WeakSet, and WeakMap in ES6

## 🔷 Why Set and Map in ES6?

Before ES6:

* JavaScript lacked native Set and Map data structures.
* Developers used **Objects** to emulate them.
* This had serious drawbacks:
  + Keys in objects can only be strings or symbols.
  + Duplicate values couldn’t be automatically managed.
  + Objects didn’t support key ordering.

🧪 ES6 introduced Set, Map, WeakSet, and WeakMap to solve these.

## 🔸 Set in ES6

### ✅ What is a Set?

A Set is a collection of **unique values**.  
It automatically removes **duplicates**.

const mySet = new Set();

mySet.add(1);

mySet.add(1); // duplicate, ignored

console.log(mySet.size); // 1

### 🧪 Adding Multiple Types

let obj1 = {};

let obj2 = {};

mySet.add("Hello");

mySet.add(42);

mySet.add(obj1);

mySet.add(obj2);

console.log(mySet.size); // 4 (all are unique)

✅ Set allows any type — strings, numbers, objects, arrays.

### 📌 Constructor with Array

let newSet = new Set([1, 2, 3, 4, 4, 4]);

console.log(newSet.size); // 4 — duplicates removed

### 🔄 Iterating Over a Set

for (let value of newSet) {

console.log(value);

}

Or using forEach:

newSet.forEach(value => console.log(value));

### 🧹 Other Set Methods

| Method | Description |
| --- | --- |
| add(value) | Adds a value |
| has(value) | Checks if value exists |
| delete(value) | Deletes a value |
| clear() | Removes all elements |
| size | Number of elements in the set |

js

CopyEdit

newSet.delete(1);

console.log(newSet.has(1)); // false

newSet.clear();

### 🔥 Builder Pattern

let chainSet = new Set().add("hello").add("world");

console.log(chainSet.size); // 2

## 🔸 Map in ES6

### ✅ What is a Map?

* A Map is a key-value pair structure.
* **Keys can be of any type** — not just strings.
* Maintains **insertion order**.

let myMap = new Map();

myMap.set("name", "rajeev");

myMap.set("job", "trainer");

console.log(myMap.get("name")); // rajeev

### 🧠 Problem with Object-as-Map in ES5

let myMap = Object.create(null);

let obj1 = {};

let obj2 = {};

myMap[obj1] = "World";

console.log(myMap[obj2]); // "World" — BAD! keys converted to [object Object]

❌ In Objects, keys are always strings. Even obj1 and obj2 get stringified.

### ✅ Maps Fix This

let myMap = new Map();

let obj1 = {};

let obj2 = {};

myMap.set(obj1, "World");

myMap.set(obj2, "Planet");

console.log(myMap.get(obj1)); // World

console.log(myMap.get(obj2)); // Planet

✅ Objects remain unique as keys.

### 📌 Initializing a Map from Array

let myMap = new Map([

["fname", "Chandler"],

["lname", "Bing"]

]);

console.log(myMap.get("fname")); // Chandler

### 🔄 Iterating Over Map

// keys only

for (let key of myMap.keys()) {

console.log(key);

}

// values only

for (let val of myMap.values()) {

console.log(val);

}

// key-value pairs

for (let [key, val] of myMap.entries()) {

console.log(`${key} → ${val}`);

}

Or use forEach:

myMap.forEach((value, key, callingMap) => {

console.log(`${key} → ${value}`);

});

### 🧹 Map Methods

| Method | Description |
| --- | --- |
| set(key, value) | Adds/updates entry |
| get(key) | Retrieves value |
| has(key) | Checks if key exists |
| delete(key) | Removes entry by key |
| clear() | Removes all entries |
| size | Number of entries in the map |

## 🧪 WeakSet and WeakMap

⚠️ These are like Set and Map but:

* Only work with **objects (not primitives)**.
* Do **not prevent garbage collection** (they are weakly held).
* Cannot be iterated or have .size.

### 🔸 WeakMap Example

let myMap = new WeakMap();

let ob1 = {};

myMap.set(ob1, "Hello World");

console.log(myMap.get(ob1)); // Hello World

ob1 = null; // reference is gone, GC will clear it

## ⚖️ Summary: Set vs Map vs WeakMap vs WeakSet

| Feature | Set | Map | WeakSet | WeakMap |
| --- | --- | --- | --- | --- |
| Keys/Values | Only values | key-value | only objects | object keys |
| Uniqueness | ✅ Yes | ❌ No | ✅ Yes | ❌ No |
| Iteration | ✅ Yes | ✅ Yes | ❌ No | ❌ No |
| Keys Types | Any value | Any type | Objects only | Objects only |
| GC-friendly | ❌ No | ❌ No | ✅ Yes | ✅ Yes |

## 🧪 Bonus: Use Cases

| Use Case | Recommended Structure |
| --- | --- |
| List of unique items | Set |
| Lookup table | Map |
| Cache without memory leak | WeakMap |
| DOM elements as keys | WeakMap / WeakSet |

# 🧠 ES6 Tutorial – Topic 14

## 🔸 ES6 Classes, Inheritance, Getters/Setters, Static Methods, and OOP in JavaScript

## 🔷 Introduction

Before ES6, JavaScript had no class keyword. Developers used **constructor functions** and **prototype inheritance** to emulate object-oriented programming.

ES6 introduced class syntax as **syntactic sugar** over the existing prototype-based system.

## 🔶 Basic Class Syntax

class Person {

greet() {

console.log("Hello!");

}

}

const p = new Person();

p.greet(); // Hello!

### ✅ Under the Hood

console.log(typeof Person); // "function"

console.log(p.greet === Person.prototype.greet); // true

* ✅ class declarations are **not hoisted** (unlike functions).
* ✅ Class methods are automatically added to prototype.

## 🔸 Old Way (Pre-ES6)

function Animal(type) {

this.type = type;

}

Animal.prototype.identify = function () {

console.log(this.type);

};

const cat = new Animal('Cat');

cat.identify(); // Cat

* identify() is shared across instances via prototype.

## 🔸 New Way (ES6 Class)

class Animal {

constructor(type) {

this.type = type;

}

identify() {

console.log(this.type);

}

}

const cat = new Animal('Cat');

cat.identify(); // Cat

✅ Cleaner, readable, and aligns with OOP concepts.

## ⚠️ Important Differences from Java

| Feature | JavaScript ES6 Class | Java Class |
| --- | --- | --- |
| Compilation | Dynamic, runtime | Compile-time |
| Inheritance | Prototype-based | Class-based |
| Access Modifiers | Not enforced (private via #) | Enforced (private, public) |
| this Binding | Dynamic unless arrow used | Static |
| Class Hoisting | ❌ Not hoisted | ✅ Yes |

## 🔸 Constructor Method

class Student {

constructor(name, age) {

this.name = name;

this.age = age;

}

}

## 🔸 Class Expressions

const Teacher = class {

constructor(subject) {

this.subject = subject;

}

};

const t = new Teacher("Math");

console.log(t.subject); // Math

✅ Classes can be anonymous and assigned to variables.

## 🔸 Getters and Setters

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

get fullName() {

return `${this.firstName} ${this.lastName}`;

}

set fullName(name) {

const [f, l] = name.split(" ");

this.firstName = f;

this.lastName = l;

}

}

const p = new Person("Rajeev", "Gupta");

console.log(p.fullName); // Rajeev Gupta

p.fullName = "Ravi Kumar";

console.log(p.firstName); // Ravi

## 🔸 Static Methods

class MathUtil {

static add(a, b) {

return a + b;

}

}

console.log(MathUtil.add(5, 7)); // 12

🧠 Static methods belong to the class, not to instances.

js

CopyEdit

const m = new MathUtil();

m.add(); // ❌ TypeError: m.add is not a function

## 🔸 Inheritance in ES6

### ✅ Basic Inheritance

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log(`Walking on ${this.legs} legs`);

}

}

class Bird extends Animal {

constructor(legs) {

super(legs);

}

fly() {

console.log("Flying");

}

}

const pigeon = new Bird(2);

pigeon.walk(); // Walking on 2 legs

pigeon.fly(); // Flying

### ✅ Inheritance with Additional Properties

class Bird extends Animal {

constructor(legs, color) {

super(legs);

this.color = color;

}

getColor() {

console.log(this.color);

}

}

const dove = new Bird(2, "White");

dove.getColor(); // White

### ⚠️ Shadowing Methods

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

console.log("Go walking");

}

}

const bingo = new Dog();

bingo.walk(); // Go walking

### 🔁 Calling Super Method

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

super.walk(); // call parent method

console.log("Go walking");

}

}

const rocky = new Dog();

rocky.walk();

// Walking on 4 legs

// Go walking

## 🔸 Class Constructor Rules

* Must call super() in subclass constructor **before** using this.
* Cannot call class without new.

let d = Animal("Duck"); // ❌ Error

## 🔸 Classes as First-Class Citizens

function factory(aClass) {

return new aClass();

}

const greeting = factory(

class {

sayHi() {

console.log("Hi");

}

}

);

greeting.sayHi(); // Hi

## 🔸 Singleton with IIFE

let app = new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting ${this.name}...`);

}

}("Awesome App");

app.start(); // Starting Awesome App...

## 🔁 Summary Table

| Concept | Syntax Example |
| --- | --- |
| Class Declaration | class MyClass {} |
| Constructor Method | constructor(args) {} |
| Instance Method | method() {} |
| Getters / Setters | get prop(), set prop(val) |
| Static Method | static method() {} |
| Inheritance | class Child extends Parent {} |
| Super Constructor | super(args) |
| Super Method Call | super.method() |
| Anonymous Class | const C = class {} |

### ✅ ****MCQs – ES6 Concepts****

#### 🔹 1–5: let, var, const, Scope & Hoisting

1. What is the output of the following code?

js

CopyEdit

console.log(x);

var x = 5;

A) 5  
B) undefined  
C) ReferenceError  
D) null  
**✅ Ans: B**

1. Which of the following keywords does **not** support block scoping?

A) let  
B) var  
C) const  
D) None of the above  
**✅ Ans: B**

1. What is the output?

{

let a = 10;

}

console.log(a);

A) 10  
B) undefined  
C) ReferenceError  
D) null  
**✅ Ans: C**

1. Which of the following statements is **true**?

A) let is hoisted and initialized as undefined  
B) var is block scoped  
C) const variables can be reassigned  
D) let prevents variable redeclaration  
**✅ Ans: D**

1. What happens when you declare a const object and modify one of its properties?

const obj = {name: "raj"};

obj.name = "ravi";

A) Error  
B) name becomes read-only  
C) Valid, no error  
D) Object becomes frozen  
**✅ Ans: C**

#### 🔹 6–10: Arrow Functions & Functional Programming

1. Which of the following is a valid arrow function?

A) const sum = (a, b) => return a + b;  
B) const sum = a, b => a + b;  
C) const sum = (a, b) => a + b;  
D) const sum(a, b) => a + b;  
**✅ Ans: C**

1. Which of the following is **not** true about arrow functions?

A) They have lexical this  
B) They support implicit return  
C) They are hoisted  
D) They are concise  
**✅ Ans: C**

1. Which functional method is used to transform elements in an array?

A) filter()  
B) map()  
C) reduce()  
D) sort()  
**✅ Ans: B**

1. What is the result of:

[1, 2, 3].map(n => n \* 2);

A) [1, 2, 3]  
B) undefined  
C) [2, 4, 6]  
D) Error  
**✅ Ans: C**

1. Which method accumulates a single result from an array?

A) filter()  
B) map()  
C) reduce()  
D) find()  
**✅ Ans: C**

#### 🔹 11–15: Default, Rest, Spread

1. What is the output?

function greet(name = "Guest") {

return `Hello, ${name}`;

}

greet();

A) Hello,  
B) Hello, undefined  
C) Hello, Guest  
D) Error  
**✅ Ans: C**

1. What does the rest operator do?

A) Merges arrays  
B) Converts arguments into an array  
C) Returns rest of the string  
D) None of the above  
**✅ Ans: B**

1. What is the output?

const nums = [1, 2, 3];

console.log(...nums);

A) [1,2,3]  
B) SyntaxError  
C) 1 2 3  
D) undefined  
**✅ Ans: C**

1. In which scenario do we typically use the spread operator?

A) Skipping parameters  
B) Combining or cloning arrays/objects  
C) Accessing object keys  
D) Creating closures  
**✅ Ans: B**

1. What is the result?

let a = [1, 2];

let b = [...a, 3];

console.log(b);

A) [1, 2, 3]  
B) [3]  
C) [undefined, 3]  
D) Error  
**✅ Ans: A**

#### 🔹 16–18: Destructuring & Template Literals

1. Which syntax extracts values from an object?

A) let {x} = obj;  
B) let x = obj{x};  
C) let x = {obj};  
D) let x = obj[x];  
**✅ Ans: A**

1. What is the output?

let [a = 10, b = 20] = [1];

console.log(a, b);

A) 10 20  
B) 1 20  
C) 1 undefined  
D) undefined undefined  
**✅ Ans: B**

1. Template literals use which syntax?

A) 'Hello ${name}'  
B) $(name)  
C) ${name}  
D) Hello ${name}  
**✅ Ans: D**

#### 🔹 19–22: for...of vs for...in

1. Which loop is best for iterating array values?

A) for...in  
B) forEach  
C) for...of  
D) for loop  
**✅ Ans: C**

1. What does for...in loop iterate?

A) Array values  
B) Object keys  
C) Set elements  
D) Map entries  
**✅ Ans: B**

1. What is the output?

for(let ch of "Hi") {

console.log(ch);

}

A) Error  
B) H i  
C) ["H", "i"]  
D) undefined  
**✅ Ans: B**

1. What is the output?

let colors = ['Red', 'Green'];

for(let index in colors){

console.log(index);

}

A) Red Green  
B) 0 1  
C) undefined  
D) Error  
**✅ Ans: B**

#### 🔹 23–27: ES6 Classes, Inheritance, Static, Getters

1. What happens if you call a class constructor without new?

A) Works normally  
B) Returns undefined  
C) TypeError  
D) SyntaxError  
**✅ Ans: C**

1. Which is true about ES6 classes?

A) They are hoisted  
B) They can be called without new  
C) They are syntactic sugar over functions  
D) They allow private inheritance  
**✅ Ans: C**

1. Which keyword invokes the parent constructor?

A) this  
B) super  
C) parent  
D) constructor  
**✅ Ans: B**

1. How do you declare a static method?

A) function static greet()  
B) greet static()  
C) static greet() {}  
D) class greet() {}  
**✅ Ans: C**

1. What is the purpose of a getter?

A) Returns class name  
B) Changes object  
C) Accesses properties like a method  
D) None  
**✅ Ans: C**

#### 🔹 28–30: Set, Map, WeakMap

1. Which of the following is **true** about Set?

A) Allows duplicate values  
B) Maintains insertion order  
C) Allows key-value pairs  
D) Is a weak collection  
**✅ Ans: B**

1. What is the output?

js

CopyEdit

let map = new Map();

map.set("name", "raj");

console.log(map.get("name"));

A) raj  
B) name  
C) undefined  
D) Error  
**✅ Ans: A**

1. Which of the following is true for WeakMap?

A) Keys must be strings  
B) Keys must be objects  
C) WeakMap has .size()  
D) It supports iteration  
**✅ Ans: B**